A Framework for Node-Level Fault Tolerance in Distributed Real-time Systems

Joakim Aidemark[1]
Volvo Car Corporation
Department of Safety Electronics
SE-405 31 Gothenburg, Sweden
jaidemar@volvocars.com

Peter Folkesson and Johan Karlsson
Chalmers University of Technology
Department of Computer Engineering
SE-412 96 Gothenburg, Sweden
{peterf, johan}@ce.chalmers.se

Abstract

This paper describes a framework for achieving node-level fault tolerance (NLFT) in distributed real-time systems. The objective of NLFT is to mask errors at the node level in order to reduce the probability of node failures and thereby improve system dependability. We describe an approach called light-weight NLFT where transient faults are masked locally in the nodes by time-redundant execution of application tasks. The advantages of light-weight NLFT is demonstrated by a reliability analysis of an example brake-by-wire architecture. The results show that the use of light-weight NLFT may provide 55% higher reliability after one year and almost 60% higher MTTF, compared to using fail-silent nodes.

1. Introduction

An important class of fault-tolerant computer systems is those used for real-time control of safety-critical applications. Classical examples are computerized flight control systems, known as fly-by-wire systems, which have been in use for more than a decade in commercial airplanes and even longer in military aircraft. The automotive industry has recently started development of sophisticated active safety systems and brake-by-wire systems. These systems are expected to reach the market within a few years.

The design principles of existing fault tolerant space and aviation systems have proven very successful, but are often too costly for emerging applications such as micro-satellites, unmanned air vehicles, and active safety systems for road vehicles. Thus, a current trend is to use distributed systems that are implemented using generic platforms, which allow development of different systems with varying dependability requirements. Examples of generic platforms are the Time-Triggered-Architecture [1] and GUARDS [2].

[1] This work was conducted while J. Aidemark was with Chalmers University of Technology

A distributed system consists of several computers, or nodes, that interact via a communication network. Fault-tolerance is achieved by executing critical programs redundantly on two or more nodes. The number of redundant nodes required depends on the failure modes of the nodes. If we use nodes that may deliver erroneous outputs without any error indication, we need majority voting to mask errors requiring at least $2f+1$ nodes to tolerate f node failures. Another approach is to use *fail-silent* nodes [1,3]. A fail-silent node produces either correct outputs, no outputs, or outputs that can be identified as erroneous by the receiver. This property makes it possible to use $f+1$ nodes to tolerate f node failures, which minimizes the number of the nodes in the system [4].

The use of *fail-silent* nodes is therefore an attractive solution for reducing the number of nodes in the system. However, the fail-silent property requires that the nodes are equipped with adequate internal error detection mechanisms. This increases the complexity and thereby the cost of the nodes. Optimizing the cost of fault-tolerant distributed systems therefore involves a trade-off between the complexity of the nodes and the number of redundant nodes required.

Systems that rely on fail-silent nodes generally shut down their operation for both transient and permanent faults. Thus, all faults regardless of their duration lead to node failures that must be handled at the system level by means of a distributed redundancy management protocol.

In this paper, we propose the use of node-level fault tolerance (NLFT) as a complement to system-level fault tolerance. The objective of NLFT is to mask errors at the node level in order to reduce the probability of node failures and thereby improve system dependability. To keep the cost of NLFT low, we propose an approach to node-level fault tolerance called light-weight NLFT in which only transient faults are masked at the node level. More precisely, light-weight NLFT corresponds to using nodes that (i) mask

the effects of most transient faults locally in the node and (ii) exhibit omission or fail-silent failures for all permanent faults and all those transient faults that cannot be masked by the node itself. Thus, the term light-weight refers to the fact that only a subset of the faults is tolerated at the node level. The main purpose of light-weight NLFT is to make systems more resilient to transient faults. Transient faults are much more common than permanent faults in digital systems and, because of technology scaling, the frequency of transient faults is expected to increase in future systems [5].

Tolerating transient faults at the node level is clearly an advantage in systems that use two fail-silent nodes, as it allows the system to survive transient faults also when one of the nodes have failed permanently. It also improves the robustness of the system when both nodes are affected by correlated or near-coincident transient faults. Tolerating transient faults at the node level may also reduce hardware costs, as fewer redundant (active or spare) nodes may be required to achieve a given level of system dependability.

The framework presented in this paper describes the principles for light-weight NLFT and proposes a set of error handling mechanisms suitable for a low-cost implementation. In addition, the advantages of light-weight NLFT are demonstrated by a reliability analysis of an example brake-by-wire architecture.

Our approach is based on a real-time kernel that operates on *commercial off-the-shelf* (COTS) processors. The key element for tolerating transient faults is a time redundancy approach called *temporal error masking*, or TEM [7]. In TEM, a critical task is executed twice and the two results are compared; if they do not match or if errors are detected by other mechanisms, an additional execution of the task is started to allow a majority vote on three results. To cope with the dynamic nature of TEM, the real-time kernel uses fixed-priority preemptive scheduling [6]. Technology trends in microprocessor technology have shown a tremendous increase in processing power in recent years. The resulting decline in the price of processing power has made time redundancy an attractive approach for achieving fault tolerance in real-time systems.

The real-time kernel and parts of the mechanisms proposed have previously been implemented and evaluated using fault injection. This includes implementation and evaluation of temporal error masking (TEM) for the Thor microprocessor [7], and an implementation and evaluation of a real-time kernel with extensive internal error detection and the support of TEM for the Motorola 68340 microcontroller [8].

The next section describes the design principles for light-weight NLFT and Section 3 demonstrates the usefulness of the approach by calculating and comparing the reliability for two versions of a brake-by-wire systems, one with and one without light-weight NLFT. The conclusions of this study and future work are given in Section 4.

2. Light-weight NLFT

In this section, we first discuss the distributed architecture considered and the redundancy concepts that may most favor the use of NLFT. The light-weight NLFT approach is then described.

2.1 Hardware architectures considered

For cost-effective implementation, we consider only distributed systems that employ single computer nodes (simplex configuration) or double computer nodes (duplex configuration), see Figure 1. A computer node in such a system conceptually consists of a host processor with memory (Host) and a network interface (NI). The duplex configuration execute in active replication to allow permanent faults to be tolerated.

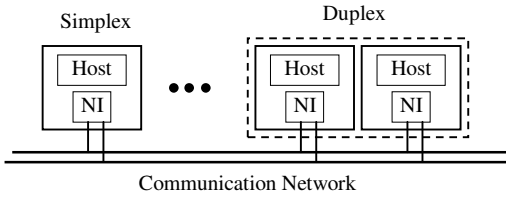

Figure 1. Distributed architecture

The network and the network interface are treated as a separate entity that we assume provides reliable transmission of messages. We assume that the communication protocol is time-triggered, or even more preferable, offers a mix of event- and time-triggered communication (such as provided by the FlexRay protocol [9]). Time-triggered scheduling is used for all critical messages, while support for event-triggered scheduling may be advantageous as it allows for fast handling of sporadic activities.

2.2 Objective and basic approach

Our light-weight NLFT approach is intended for improving the dependability of nodes by tolerating the majority of the errors caused by transient faults in the host. We consider a host processor based on a single COTS microprocessor, where a real-time kernel controls the execution of tasks. In our approach, we implement support for light-weight NLFT in the real-time kernel, thus allowing the programmer to focus on the application.

Implementing mechanisms for masking the effect of all transient faults may lead to unacceptable overheads. Hence, the objective of the light-weight NLFT approach is to mask the majority of the transient faults in critical tasks. Permanent faults and transient faults that cannot be tolerated must be handled at the system level. The following strategies are chosen for faults affecting critical tasks, non-critical tasks or the real-time kernel:

1. Critical tasks - The objective is to tolerate all transient faults occurring in critical tasks. If there is not enough time to recover from an error and meet the deadline, an omission failure is enforced, which must be handled at the system level
2. Non-critical tasks - If an error is detected during execution of a non-critical task, the task is shut down to allow continued operation of the remaining tasks. Hence, any interaction between critical tasks and non-critical tasks must be avoided so that a critical task is not affected by the failure of a non-critical task.
3. Real-time kernel - Errors detected during execution of the real-time kernel should result in the node becoming silent. Thus, recovery must be handled at the system level. This may be acceptable since the kernel's execution time typically represents only about 5% of a processor's total execution time [10].

Omission failures may be allowed in a duplex configuration since the partner node can provide the service. Omission failures may also be allowed in a simplex configuration if the system is able to use a previous value, a default value, or, as is the case for some control systems, withstand a certain delay in the delivery of the control signal without losing stability [11].

2.3 Error handling

The error detection mechanisms provided by modern microprocessors will detect many errors. However, the effects of certain faults may pass undetected. Therefore, additional error handling must be provided at the node-level.

Node-level fault tolerance may be realized using software-implemented techniques, which may be *application specific* or *systematic* (application independent). Application specific techniques utilize detailed information about the application to allow custom designed error handling, while systematic techniques rely on the use of duplication in space or time. Systematic techniques generally incur higher overhead than application specific approaches, but are simpler to use for the system designer since they do not require application specific knowledge. Moreover, separating the error handling mechanisms from the application generally reduces the complexity of the system [12].

In our approach, systematic techniques are used to tolerate faults in critical tasks, while specific techniques, such as assertions and range checks, are used to detect errors affecting the kernel. In this paper, we do not discuss the error handling for the kernel further - an experimental study of a prototype kernel supporting node-level fault tolerance can be found in [8]. Examples of error handling mechanisms suitable for low-cost implementation of light-weight NLFT are given in Table 1. Note that we rely on a combination of hardware and software error handling techniques to achieve NLFT.

Table 1. Examples of error handling suitable for implementing light-weight NLFT

Hardware techniques	**Description**
CPU hardware exceptions	CPU run-time error detection mechanisms
Error correcting codes (ECC)	Detects and corrects errors in memories
Memory management unit (MMU)	Detects memory accesses outside the task's allowed memory area
Software techniques provided by kernel	**Description**
Temporal error masking (TEM)	Detects and tolerates computation errors caused by, e.g. transient faults in data registers, adders or multipliers
Execution time monitoring	Detects timing violations for individual tasks
Data integrity checks and end-to-end error detection	Detects errors in internal data structures and errors in input and output data

2.4 Hardware techniques and timing checks

Current state-of-the-art COTS microprocessors provide extensive error detection mechanisms (EDMs) such as illegal op-code detection, address range checking and error correcting codes (ECC) on memories and caches. Often, they also provide a *memory management unit* (MMU), which supports fault confinement between tasks or between tasks and the kernel. This simplifies fault tolerance, as we only need to consider recovery of the affected task. To ensure that a task does not execute for too long, which may prevent other tasks from executing, an *execution time monitor* may be used. For example, *budget timers* [2] may be used to monitor the execution time of individual pre-emptive tasks. Such a mechanism allows the action taken when an error is detected, to be decided for each tasks, e.g. conduct a recovery of the affected task.

2.5 Temporal error masking

To support transparent error handling in a real-time kernel, temporal error masking (TEM) is used. In TEM, the kernel executes all critical tasks twice and compares the results in order to detect errors. A third execution is started if an error is detected by the comparison or by any hardware or software EDM. This allows the kernel to mask errors by conducting a majority vote on three results. To ensure that the recovery of an erroneous task does not lead to any deadline violations, sufficient slack must be provided in the schedule, see Section 2.8.

Figure 2 shows our basic model for a critical task, which is assumed to be executed in a periodic *read input - compute - write output* loop. The input data are received first from input devices or from other tasks. The input data are then processed and the results sent to actuators or to other tasks in the system at the end of the loop.

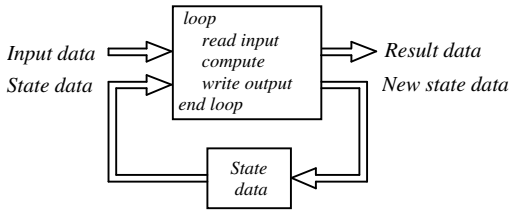

Figure 2. Task model

Figure 3 shows three different scenarios using TEM: in fault-free operation (i), a critical task, T, is executed two times (denoted T^1 and T^2) and a comparison is made to detect errors. As the results match, a third copy does not have to be executed and the time may be used by other tasks. In (ii) an error is detected by the comparison and a third copy of the task, T^3, is then executed. The results of the three copies are checked by a majority vote. If the majority voter detects two matching results, these are accepted as a valid result of the task. Otherwise, no result is delivered, which leads to an omission failure.

In (iii), an error is detected by a hardware mechanism or another node level mechanism. The affected copy, T^2, is then terminated and a new copy, T^3, is started immediately. The new copy will use time reclaimed from the terminated copy as well as time from any available slack. A comparison is made to confirm that the results match before a result is delivered. For errors detected by CPU EDMs, the task's CPU state context, e.g. the program counter (PC) and stack pointer (SP) etc., is restored to the initial conditions from information stored in the task control block in the kernel. The reason for restoring the complete context is that errors detected by hardware exceptions often originate from faults in the CPU registers. For example, in [8] we showed that an *illegal instruction* exception may occur as a result of faults in the PC register and that *address* and *bus* exceptions are often triggered by faults in the SP register. When errors are detected by the comparison of a task, we assume that the error only affects the data computations; new copies can therefore be started without restoring the CPU context. Scenario (iv) is similar to scenario (iii), but the fault occurs in copy T^1.

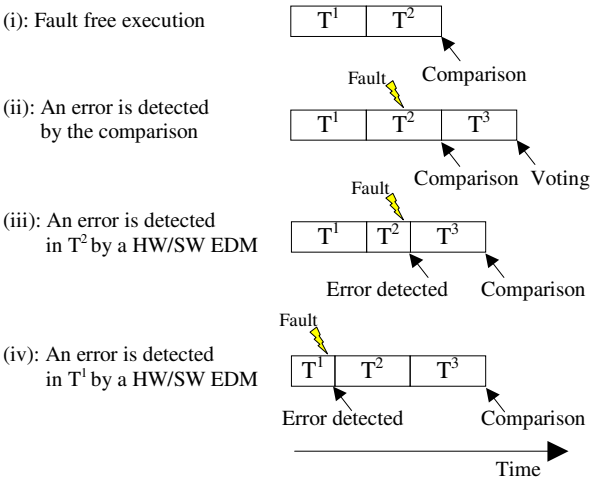

Figure 3. Error detection and recovery using temporal error masking

The kernel always checks the deadline of the task after an error is detected to determine whether it is possible to execute an additional task copy and still meet the deadline. Even if additional time is reserved in the schedule to handle recovery, enough time may not be available, e.g. because more faults than anticipated occurs. In this case, no result is delivered and an omission failure occurs. If time is available, a new copy is started. The task result is delivered and the state data are only updated when two matching results have been produced. Errors that are repeated for some time are considered to be caused by permanent faults. In this case, the node is shut down for off-line diagnosis to establish whether a transient or a permanent fault caused the error. For transient faults, the node may be re-integrated.

2.6 Data integrity checks and end-to-end error detection

Data used for the computations in the task must not only be protected during the actual computation, but also before and after the computation. This is often referred to as *end-to-end* error detection [4]. We assume that the memory is protected from direct faults using ECC, see Table 1. Furthermore, static data such

as program code and constants may be saved in read only memory and may therefore not be erroneously overwritten. However, additional protection is needed to ensure the integrity of the input data, state data and result data. Faults occurring in the CPU affecting the input data or state data during the computations of the individual copies may be detected by the comparisons, while faults occurring when data are stored to memory may cause data to be overwritten or to be written to an erroneous location in memory. There are a number of techniques that can be used to detect such errors. The simplest is to duplicate the data and conduct a comparison before it is used to reveal discrepancies. To protect larger data structures, it may be more effective to generate CRC checksums for the data.

For a duplex configuration, errors that are detected may be handled by exhibiting an omission failure, since the partner node provides the full service. Recovery of input data may be conducted by simply obtaining new data in the next cycle, and eventual state data may be recovered by obtaining the partner node's state data. For a simplex configuration, errors in the state data may be handled in different ways depending on the application. For example, triplication of data may be employed to mask the effect of faults, or the node may just be shut down for a fail-safe system.

2.7 Control flow errors

Faults may cause *control flow errors*, i.e. deviation from the correct execution order, and thus cause failures. The MMU provided by the processor may detect control flow errors as the task's address range is bounded. If the control flow error causes the task to execute too long, the error may also be detected by the execution time monitoring mechanism. In addition, TEM will also detect control flow errors within a task if the error affects the result and the comparison/voting is executed correctly. There is, however, a small risk that a control flow error may cause the execution to by-pass the comparison/voting by erroneously jumping directly to the code that writes the checked/voted result to main memory or to an output device. Specific checks should be provided to avoid that such control flow errors pass undetected.

2.8 Real-time requirements

The TEM approach relies on event-triggered fault handling, since recovery (through execution of a third copy of the affected task) should only be initiated when an error is detected. The kernel therefore uses *fixed priority* (FP) pre-emptive scheduling [6].

In fixed priority scheduling, the priority of each task is determined before run-time. FP scheduling allows both periodic and sporadic task executions where the task with the highest priority is always allowed to execute first. In our kernel, priority assignments are made on the basis of the *criticality* of the task. The criticality of a task relates to the consequences of a failure of the task, e.g. a brake request is assigned a higher priority than a diagnostic request. Since the scheduling is pre-emptive, a task is suspended whenever another task with higher priority requires access to the CPU.

To ensure that critical tasks meet their deadlines also in the presence of errors, we assume that a fault-tolerant scheduling algorithm supporting fixed priority scheduling is employed. Fault-tolerant scheduling [6] guarantees that all tasks meet their deadlines, even in the presence of a specified number of faults. To allow a failed task to re-execute without causing other tasks to miss their deadlines, extra time (slack) must be reserved *a priori* and be accounted for in a schedulability test. The amount of extra time needed depends on the number and type of faults anticipated.

3. Dependability analysis

The ability to recover quickly may significantly increase the dependability of systems. This is exemplified in this section through a dependability analysis of a distributed brake-by-wire (BBW) architecture, which is a typical example of the kind of safety-critical distributed systems considered here. In particular, we examine the dependability of the BBW architecture using conventional fail-silent computer nodes, as compared to using nodes with light-weight NLFT.

3.1 Brake-by-wire system

A distributed architecture for a brake-by-wire system can be implemented according to Figure 4. The brake pedal is connected to a central unit (CU). The central unit handles the all-embracing control, distributing the correct brake force to each wheel node (WN). The control algorithms in the individual wheel nodes then ensure that the requested brake force is applied to the respective wheel in the most favorable way.

The architecture considered consists of one duplex configuration for the central unit and a simplex configuration in conjunction with a brake actuator for each wheel. A simplex configuration is used for the wheel nodes in order to reduce equipment costs. This configuration may be used since the driver can still brake the vehicle if one wheel node fails. That is, the system can change to a *degraded functionality mode* where the brake force is distributed to the remaining

fault-free wheel nodes after a node failure. However, the efficiency of the brakes will be substantially degraded. Thus it is desirable to reduce the probability of entering the degraded functionality mode.

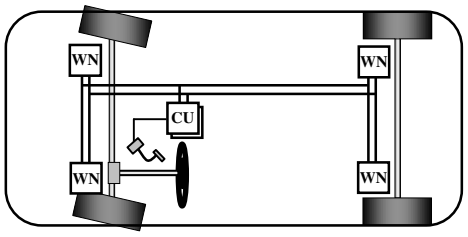

Figure 4. Example of a distributed BBW architecture

3.2 Reliability Modeling

We evaluate the reliability of the brake-by-wire system in this study. The reliability, $R(t)$, of a system is the probability that the system is operating correctly throughout a specific time interval, given that the system was operating correctly at the start of that interval.

In the following, the BBW architecture will be studied with respect to both full and degraded functionality. Full functionality mode refers to a requirement that all wheel nodes and one central unit node must function correctly; otherwise a system failure has occurred. In the degraded functionality mode, the requirement is that at least three wheel nodes and one central unit node must function correctly; otherwise a system failure has occurred. To simplify the analysis, only the nodes of the BBW system are included. Thus, failures of the actuators, sensors and communication busses are not considered.

The reliability of the BBW architecture is calculated using the SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator) tool [13]. SHARPE allows various models such as fault trees, reliability block diagrams (RBD) and Markov models to be specified and dependability measures to be obtained.

3.2.1. Description of nodes. In our analysis, we consider both *permanent* and *transient* faults. A permanent fault occurs at a specific time and remains in the system requiring the faulty component to be either repaired or replaced. A transient fault occurs at a specific time and exists only for a limited period of time in the system.

The fault rate (λ) of a node refers to the occurrence rate of activated faults in the node, i.e. faults that generate errors in the node. Faults whose effects are overwritten or latent are not included in the fault rate.

Two types of nodes are considered; fail-silent computer nodes (called FS nodes), and nodes with light-weight NLFT (called NLFT nodes). Their *intended* behavior in presence of errors is as follows:

FS nodes: If an error is detected by one of the node's EDMs, then the node exhibits a fail-silent failure, i.e. the node immediately stops producing results and is excluded from the distributed system. The node is automatically restarted, and a diagnostic program establishes whether the failure was caused by a transient or a permanent fault. If the node is found to fault-free by the diagnostic test, the node is re-integrated into the distributed system.

NLFT nodes: Transient faults and their corresponding errors can be handled in three ways: i) the error is masked by TEM, ii) the error is detected and an omission failure occurs or iii) the error is detected and a fail-silent failure occurs. An omission failure occurs if there is not enough time to re-execute a task a third time before the task's deadline, or if three different results are produced in TEM. A fail-silent failure occurs if an error is detected during execution of the kernel. Such failures are handled in the same way as in the case of FS nodes.

Non-covered errors, i.e. errors that escape all EDMs, may cause both the FS nodes and NLFT nodes to deviate from their intended behaviors. We make the pessimistic assumption that all non-covered faults lead to a system failure of the entire BBW system.

3.2.2. Basic assumptions and notations. We assume that faults occurring in one computer node are statistically independent of faults occurring in other computer nodes. We also assume that the fault rate and the repair rate are constant over time, i.e. the time to failure and the time to repair are exponentially distributed. All nodes are assumed to have the same complexity and exposure to the environment, and thereby the same fault rate. The repair (recovery) action is assumed to be fault-free. Correlated faults are not considered. A correlated fault occurs when a single fault affects more than one component at one point in time. Neither is repair of permanent faults considered. The following notations are used in the models:

λ_P	Permanent fault rate
λ_T	Transient fault rate
C_D	Error detection coverage, i.e. the conditional probability that an error is detected given that a fault has occurred.
P_T	Given that an error caused by a transient fault is detected, this denotes the probability that the system can mask the effect of the fault using TEM
P_{OM}	Given that an error caused by a transient fault is detected, this denotes the probability that a node exhibit an omission failure

P_{FS}	Given that an error caused by a transient fault is detected, this denotes the probability that a node exhibits a fail-silent failure
μ_R	Repair rate for restart. This refers to the time required for a node to restart and reintegrate into the distributed system after a fail-silent failure
μ_{OM}	Repair rate for omission failures. This refers to the time required for a node to reintegrate into the distributed system after an omission failure

3.2.3. BBW System. A hierarchical approach similar to [14] is used to construct the reliability model of the BBW architecture. Figure 5 shows the fault tree model that represents the overall system. The basic components of the fault tree are the central unit and the wheel node subsystem that consists of the four wheel nodes. A failure of any subsystem results in a system failure. The hierarchical approach used allows the various parts of the system to be assessed separately and reliability bottlenecks to be identified.

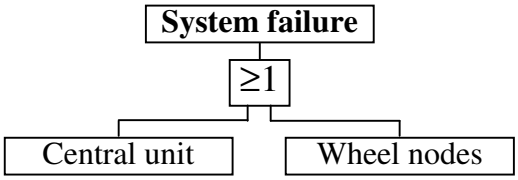

Figure 5. Fault tree model for the BBW system

3.2.4. Central Unit Subsystem. The central unit with two FS nodes can be modeled as a continuous-time Markov model according to the state transition diagram in Figure 6. The model consists of four states:

State	Description
0	Both computer nodes are working correctly
1	One of the computer nodes is affected by a permanent fault and is permanently down. The other node continues to provide service
2	One of the computer nodes is affected by a transient fault and is temporary down. The other node continues to provide service
F	Failure. Two computer nodes are shut down. Either due to a failure of two nodes, or an undetected error in one node

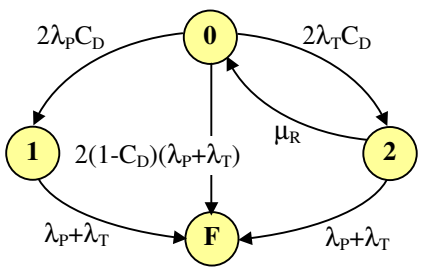

Figure 6. State transition diagram for the central unit with FS nodes

When NLFT nodes are used, the effect of a transient fault is tolerated with the probability of P_T, an omission failure occurs with the probability of P_{OM}, or a fail-silent failure occur with the probability of P_{FS}. The state transition diagram for the central unit with NLFT nodes is shown in Figure 7. The model consists of five states:

State	Description
0	Both computer nodes are working correctly
1	One of the computer nodes is affected by a permanent fault and is permanently down. The other node continues to provide service
2	One of the computer nodes is affected by a transient fault and is temporary down. The other node continues to provide service
3	One of the computer nodes is affected by a transient fault and produces an omission failure. The other node continues to provide service
F	Failure. Two computer nodes are down. Either due to a failure of two nodes, or an undetected error in one node

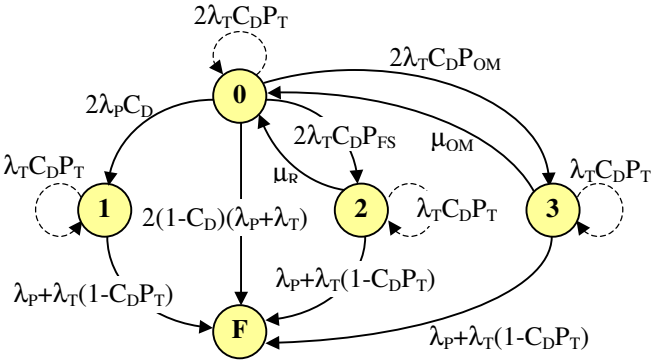

Figure 7. State transition diagram for the central unit with NLFT nodes

3.2.5. Wheel Node Subsystem. When the full functionality mode is considered, the failure of any of the wheel nodes can cause the wheel node subsystem to fail. An RBD model of the system with FS nodes is shown in Figure 8.

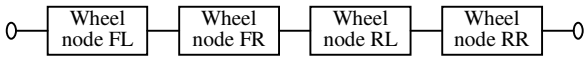

Figure 8. RBD for the wheel node subsystem with full functionality mode and FS nodes

Figure 9 shows the state transition diagram for the wheel node subsystem for degraded functionality mode when FS nodes are used. In contrast to the full functionality mode, the degraded functionality mode also allows re-integration of failed nodes since the system can operate when only three wheel nodes are working. The model consists of four states:

State	Description
0	All four computer nodes are working correctly
1	One of the computer nodes is affected by a permanent fault and is permanently down. The other nodes continue to provide their service
2	One of the computer nodes is affected by a transient fault and is temporary down. The other nodes continue to provide their service
F	Failure. Two computer nodes are shut down. Either due to a failure of two nodes, or an undetected error in one node

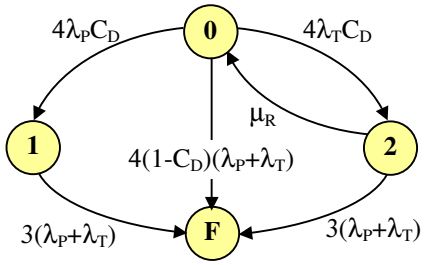

Figure 9. State transition diagram for the wheel node subsystem with degraded functionality mode and FS nodes

Figure 10 shows the state transition diagram for the wheel node subsystem with full functionality mode and NLFT nodes. State 0 represents the fault-free state where all four wheel nodes are working correctly or transient faults occur that are masked by TEM. The transition from state 0 to state F occurs in the case that a wheel node is affected by a permanent fault or, that a wheel node is affected by a transient fault that cannot be masked by TEM.

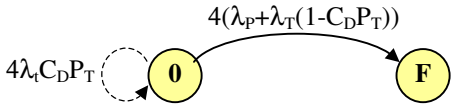

Figure 10. State transition diagram for the wheel node subsystem with full functionality mode and NLFT nodes

Figure 11 shows the state transition diagram for the wheel node subsystem when NLFT nodes and degraded functionality is considered. The model consists of five states:

State	Description
0	All four computer nodes are working correctly
1	One of the computer nodes is affected by a permanent fault and is permanently down. The other nodes continue to provide their service
2	One of the computer nodes is affected by a transient fault and is temporary down. The other nodes continue to provide their service
3	One of the computer nodes is affected by a transient fault and produces an omission failure. The other nodes continue to provide their service
F	Failure. Two computer nodes are shut down. Either due to a failure of two nodes, or an undetected error in one node

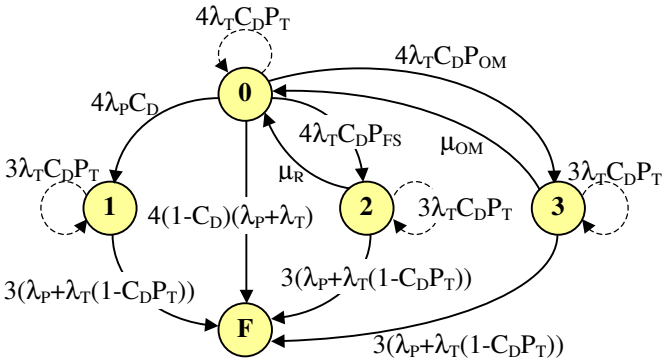

Figure 11. State transition diagram for the wheel node subsystem with degraded functionality mode and NLFT nodes

3.3 Parameter assignment

Before the results can be derived from the models presented, the parameter values must be assigned. In general, fault rates and repair rates are not easy to obtain as they depend on many factors, e.g. the underlying hardware, the software implementation and the operating environment. Nevertheless, as the objective is to compare the different approaches rather than deriving actual reliability measures, the following values may be acceptable.

The rate of permanent faults, $\lambda_P = 1.82 \cdot 10^{-5}$ faults per hour, is obtained from [15], where the fault rate of a computer node in a distributed brake-by-wire system for heavy duty trucks is derived using MIL-HDBK-217 standard. The computer node consists of a 32-bit processor with memory, communication interface, power IC, bus driver and bus connections.

The rate of transient faults, λ_T, is assumed to be ten times higher than the rate of permanent faults, i.e. $\lambda_T = 1.82 \cdot 10^{-4}$ faults per hour. Recent studies indicate that the proportion of transient faults will become even higher in future microcontrollers and memories [5]. Transient faults are handled differently depending on whether faults affect the application tasks or the real-time kernel. It is stated in [10] that about 5% of the CPU time is used by a real-time kernel; thus we assume that $P_{FS} = 0.05$. Furthermore, the results from fault injection experiments with our light-weight NLFT kernel [7] suggest that we may assume 90% of the faults to be tolerated ($P_T = 0.9$), and that 5% of the transient faults result in omission failures ($P_{OM} = 0.05$).

An error detection coverage of 99% ($C_D = 0.99$) is assumed in Section 3.4 and varied in Section 3.4.1. The repair rate, μ_R, includes time for restarting the node, checking whether there was a permanent fault and then

reintegrating the node. In [16], a distributed system based on the time-triggered TTP/C protocol executing a brake-by-wire algorithm was evaluated employing heavy-ion fault injection. The system was composed of five nodes, where one cycle consisted of 2048 TDMA rounds and one TDMA round took approximately 20 ms. The estimated time necessary to restart the operating system and reintegrate one computer node was 1.6 seconds. Assuming that hardware reset of a computer node conducting a diagnostic test to ensure that no permanent faults exist would be in the order of 1.4 seconds, a total repair time of 3 seconds is used (μ_R = $1.2 \cdot 10^3$ repairs per hour). The repair time for omission failures is assumed to take at most 1.6 seconds ($\mu_{OM} = 2.25 \cdot 10^3$ repairs per hour).

3.4 Results

The results of the reliability analysis for the complete BBW system over one year are shown in Figure 12.

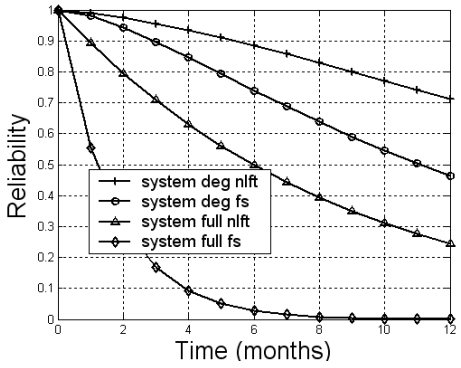

Figure 12. Reliability of the BBW system

As expected, the reliability for degraded functionality mode is higher than for full functionality mode. With regard to degraded functionality with NLFT nodes, the reliability increases by 55% (from 0.45 to 0.70) after one year as compared to FS nodes. The reliability may also be expressed as the system's mean time to failure (MTTF), i.e. the expected operation time before first failure. As concerns degraded functionality mode, the MTTF increases by almost 60% (1.2 year to 1.9 year) when NLFT nodes are used.

Figure 13 shows the reliability of the various subsystems with respect to both full and degraded functionality. The main reliability bottleneck is the wheel node subsystem.

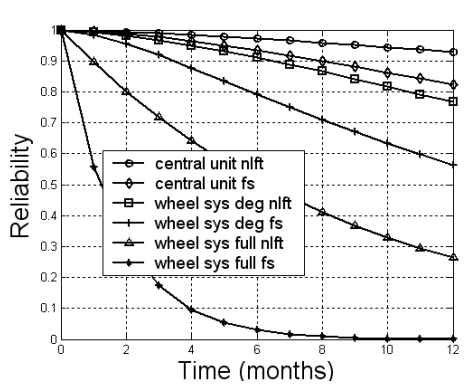

Figure 13. Reliability of the subsystems

3.4.1. Effect of varied error detection coverage and transient fault rate. The highest reliability for the BBW system is obtained when degraded functionality is considered. In this subsection, we show the reliability for this mode after five hours, for different values of the error detection coverage and the fault rate. The results are given in Figure 14, where the reliability of the system for increasing transient fault rates is shown.

The results show that the coverage has a significant influence on the reliability. The fault rate has a negligible impact as long as the fault rate is much smaller than the repair rate. However, as seen in the figure, the reliability improvements of using NLFT increase for higher fault rates.

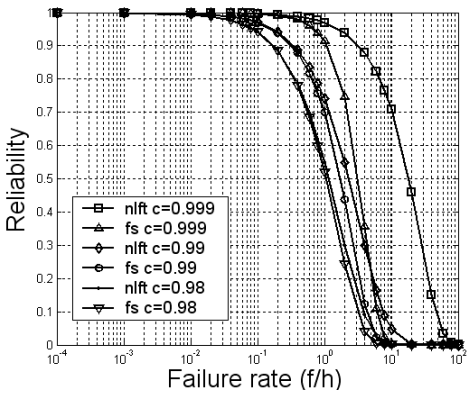

Figure 14. Reliability after five hours for varying error detection coverage and transient fault rate

4. Conclusions and future work

This paper proposes the use of node-level transient fault tolerance (NLFT) for improving the dependability of distributed systems. Especially, we present an approach called light-weight NLFT that aims at masking the majority of transient faults locally in the node. For permanent and transient faults that cannot be masked, the node must exhibit omission or fail-silent failures, which simplifies error handling at the system-level.

The paper suggests a number of error handling mechanisms based on experiences from previous studies, where a real-time kernel and parts of the mechanisms proposed have been implemented and evaluated using fault injection [7, 8]. In addition, reliability calculations are made on an example brake-by-wire application to demonstrate the advantages of the approach. The results shows that the reliability may increase by 55% after one year, and the MTTF increases almost 60% when light-weight NLFT nodes are used, compared to using nodes that are fail-silent.

Further work includes implementation and evaluation for the full set of error handling proposed in this paper to verify that the approach is viable, and to estimate the total coverage and overhead figures. Additional work also includes investigation of how to ensure replica determinism in replicated nodes and how to maintain consistency in replicated nodes in case of omission failures. For example, the study of protocols such as FlexRay [9] that may facilitate fast recovery of state data with low communication overhead through special requests to the partner node in the event-triggered part of the protocol, while also guaranteeing the delivery of critical data transmitted in the pre-allocated time slots in the time-triggered part of the protocol.

5. Acknowledgements

This work was partially supported by ARTES and the Swedish Foundation for Strategic Research (SSF), and the Saab Endowed Professorship in Robust Real-time Systems. We would like to thank Jonny Vinter at Chalmers University and Dr. Örjan Askerdal at Volvo Car Corporation for their many valuable suggestions to this work.

6. References

[1] H. Kopetz and G. Bauer, "The Time-Triggered Architecture", *Proceedings of the IEEE*, vol. 91, 2003, pp. 112-26.

[2] D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabejac, and A. Wellings, "GUARDS: A Generic Upgradable Architecture for Real-time Dependable Systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, 1999, pp. 580-599.

[3] D. Powell, "Distributed Fault Tolerance: Lessons from Delta-4", *IEEE Micro*, vol. 14, 1994, pp. 36-47.

[4] Kopetz, H., *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Boston: Kluwer Academic, 1997.

[5] R. C. Baumann ”Soft Errors in Commercial Integrated Circuits”, *International Journal of High Speed Electronics and Systems*, Vol. 14, No. 2, 2004, pp. 299-309

[6] Burns, A., and Wellings, A., *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time Posix*, third ed. Harlow: Addison-Wesley, 2001.

[7] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "Experimental Evaluation of Time-redundant Execution for a Brake-by-wire Application", *Proc. of International Conference on Dependable Systems and Networks*, Washington, DC, USA, 2002, pp. 210-216.

[8] J. Aidemark, P. Folkesson, and J. Karlsson, "Experimental Dependability Evaluation of the Artk68-FT Real-time Kernel", *International Conference on Real-Time and Embedded Computing Systems and Applications*, Göteborg, Sweden, 2004.

[9] FlexRay Communications System Specifications Version 2.0, www.flexray.com June 2004.

[10] Labrosse, J. J., *MicroC/OS-II : The Real-Time Kernel*, second edition, Lawrence: R&D, 1999.

[11] G. Heiner and T. Thurner, "Time-Triggered Architecture for Safety-related Distributed Real-time Systems in Transportation Systems", *Proceedings of the 28th International Symposium on Fault Tolerant Computing*, Munich, Germany, 1998, pp. 402-407.

[12] Poledna, S., *Fault-tolerant Real-time Systems: The Problem Of Replica Determinism*. Boston, Mass, Kluwer Academic Publishers, 1996.

[13] R. A. Sahner and K. S. Trivedi, "Reliability Modeling using SHARPE", *IEEE Transactions on Reliability*, vol. R-36, 1987, pp. 186-93.

[14] D. Chen, S. Dharmaraja, D. Chen, L. Li, K.S. Trivedi, R.R. Some, A.P. Nikora, “Reliability and Availability Analysis of the JPL Remote Exploration Experimentation System”, *Proc. of International Conference on Dependable Systems and Networks*, Washington, DC, USA, 2002, pp. 337-342.

[15] Claesson, V., *Efficient and Reliable Communication in Distributed Embedded Systems*, Ph.d thesis, Chalmers University of Technology, Göteborg, Sweden, 2002.

[16] Sivencrona, H., *On the Design and Validation of Fault Containment Regions in Distributed Communication Systems*, Ph.d thesis, Chalmers University of Technology, Göteborg, Sweden, 2004.