A Cascading Redundancy Approach for Dependable Real-Time Systems

Hüseyin Aysan, Radu Dobrin, and Sasikumar Punnekkat
Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden
{huseyin.aysan, sasikumar.punnekkat, radu.dobrin}@mdh.se

Abstract

Dependable real-time systems typically consist of tasks of multiple criticality levels and scheduling them in a fault-tolerant manner is a challenging problem. Redundancy in the physical and temporal domains for achieving fault tolerance has been often dealt independently based on the types of errors one needs to tolerate. To our knowledge, there had been no work which tries to integrate fault tolerant scheduling and multiple redundancy mechanisms. In this paper we propose a novel cascading redundancy approach within a generic fault tolerant scheduling framework. The proposed approach is capable of tolerating errors with a wider coverage (with respect to error frequency and error types) than time and space redundancy in isolation, allows tasks with mixed criticality levels, is independent of the scheduling technique and, above all, ensures that every critical task instance can be feasibly replicated in both time and space.

1 Introduction

Redundancy in the physical, temporal, information and analytical domains is the key for achieving fault tolerance and due to the wealth of research in this domain, a rich set of techniques has been successfully used in many critical applications. Similarly, due to the inherent criticality of real-time systems, several researchers have been trying to incorporate fault tolerance into various real-time scheduling paradigms. However, we have not so far come across any works that integrate these two domains of research. In this paper we propose a novel approach of cascading redundancy which brings out the synergetic effect of an appropriate combination of time and space redundancy techniques within a fault-tolerant scheduling framework.

Incorporating fault tolerance into various real-time scheduling paradigms has been addressed by several researchers. In [12], Han et.al. presented an approach to schedule primary and alternate versions of tasks to provide fault tolerance. In [11], the authors presented a method for guaranteeing that the real-time tasks will meet the deadlines under transient faults, by resorting to reserving sufficient slack in queue-based schedules. Pandya and Malek [19] showed that single faults with a minimum inter-arrival time of largest period in the task set can be recovered if the processor utilization is less than or equal to 0.5 under rate monotonic (RM) scheduling. Burns et. al. [6, 20] provided exact schedulability tests for fault tolerant task sets under specified failure hypothesis for fixed priority scheduling, which can guarantee task sets with even higher utilizations. Lima and Burns [16] extended this analysis in case of multiple faults, as well as for the case of increasing the priority of a critical task's alternate upon fault occurrences. While the above works have advanced the field of fault tolerant scheduling within specified contexts, each one has some shortcomings, e.g., restrictive task and fault models, non-consideration of task criticality, complex online mechanisms, and scheduler modifications which may be unacceptable from an industrial perspective.

On the other hand, static space redundancy techniques such as N-modular redundancy (NMR) have been used in safety and mission critical applications, often in the well-known form of triple-modular redundancy (TMR)[18]. The key attractions of the approach are low overhead and fault masking abilities, without the need for backward recovery [15]. The disadvantages include redundancy cost (in terms of hardware resources) and single point failure mode of the voter. Traditionally, voters are constructed as simple electronic circuits so that a very high reliability can be achieved. Distributed voters have also been employed to take care of the single-point failure mode in highly critical systems [17].

Replicated nodes' output values can vary slightly, resulting in a range (or a set) of values which should be considered as correct to avoid consistency issues [5], later addressed by *inexact voting strategies* [14]. This phenomenon is also observed in time domain due to, e.g., clock drifts, node failures, processing and scheduling variations at node level, and communication delays. Most of the existing voting strategies, however, focus solely on tolerating variations in the value domain by assuming tight synchronization [13]. On the other hand, the use of loose synchronization is an

attractive alternative due to low overheads, requiring, however, specifically designed asynchronous voting algorithms to compensate for the timing variations.

Real-time adaptations of majority voting techniques were proposed by Ravindran et. al., [21], and Shin et. al., [22], to overcome the problem of waiting for late or omitted replica outputs, where voting is performed among a quorum or a majority of outputs received, rather than waiting for all the outputs. Recently, we have proposed a technique [2] that additionally tolerates early timing failures of replica nodes.

In a system with no redundancy, or in a redundant system with tight synchronization, the deviation in output delivery times of a task is less than or equal to the maximum permissible deviation (MPD) that is equal to its *feasibility window*, i.e., the time interval between its release time and deadline, minus its best case execution time ($BCET$) (Figure 1 (a)). However, in loosely synchronized redundant systems, local clocks on the processing nodes are allowed to drift by a specified value (δ), which can, in certain scenarios, result in delivery of replica outputs that are farther apart from each other than they are designed for with respect to time (Figure 1 (b)), hence, causing system failures.

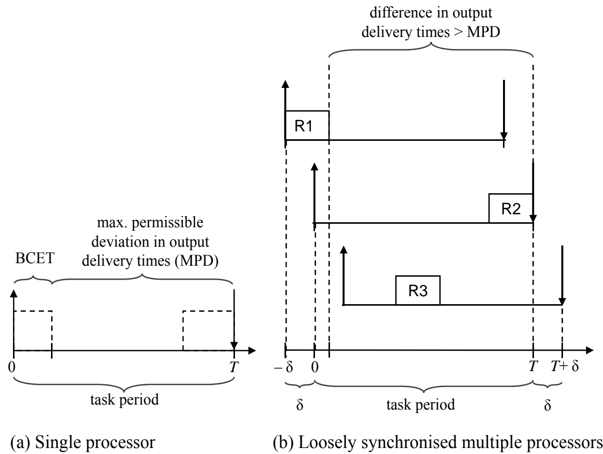


Figure 1. Replica outputs' timing issue

In this paper we propose a cascading redundancy framework capable of tolerating errors with a wider coverage (with respect to error frequency and error types) than time and space redundancy in isolation, handles tasks with mixed criticalities, and, above all, ensures that every critical task instance can be feasibly replicated in both time and space, independent of the RT scheduling policy. Hence, the research presented in this paper is significantly different from our previous works [2, 8] due to the following original contributions:

1. the composition of the time and space redundancy techniques in a mutually aware manner
2. the ability to handle a wider range of task criticalities
3. the applicability to any real-time scheduling paradigm

The rest of the paper is organized as follows: the system and error model are presented in Sections 2 and 3. In Sections 4 and 5 we propose time and space redundancy techniques for RT systems which we build upon in the cascading redundancy approach proposed in Section 6. The paper is concluded in Section 7.

2 System model

We assume a distributed real-time architecture consisting of processing nodes, sensors and communication media. Each node has its own clock allowed to drift from the correct time (i.e., seen by a perfect observer) by a maximum permissible deviation δ. This deviation can be handled by relatively inexpensive clock synchronization algorithms implemented in software (as opposite to expensive tight clock synchronization implementations).

On each processing node P_n, a periodic task set, $\Gamma_n = \{\tau_1, \tau_2, \ldots\}$, is allocated where each task represents a real-time thread of execution with a specified criticality. Each task τ_i has a period $T(\tau_i)$, a deadline $D(\tau_i)$, a known best case execution time $BCET(\tau_i)$ and a known worst case execution time ($WCET$) represented by $C(\tau_i)$. We assume tasks deadlines equal to their periods. Execution of error detection or error handling mechanisms such as sanity checks and re-execution of failed computations are considered as a part of the computation stage.

The criticality of a task could be seen as a measure of the impact of its correct (or incorrect) functioning on the overall system behavior, and also indicates the error model the system is designed to tolerate. Without loss of generality, in this paper, we assume the following four criticality levels:

1. **Non-critical tasks:** can be shed if needed without adversely affecting the system performance.
2. **Critical tasks:** are guaranteed sufficient time to re-execute (or run an alternate action) upon error occurrences.
3. **Highly-critical tasks:** are replicated over a set of processor nodes and their outputs are voted on a standalone voter.
4. **Ultra-critical tasks:** are both replicated over a set of processor nodes and guaranteed sufficient time to re-execute as well.

Each critical and ultra-critical task τ_i has an alternate task $\bar{\tau}_i$ with a worst case execution time $\bar{C}(\tau_i) \leq C(\tau_i)$ and a deadline $\bar{D}(\tau_i) = D(\tau_i)$. This alternate can typically be a

re-execution of the same task, a recovery block, an exception handler or an alternate with imprecise computations.

Let $\Gamma_n^{nc}, \Gamma_n^{c}, \Gamma_n^{hc}$ and Γ_n^{uc} represent the subsets of non-critical tasks, critical tasks, highly-critical tasks and ultra-critical tasks out of the original task set on node P_i respectively, so that $\Gamma_n = \Gamma_n^{nc} \cup \Gamma_n^{c} \cup \Gamma_n^{hc} \cup \Gamma_n^{uc}$. We use $\bar{\Gamma}_n^{c}$ and $\bar{\Gamma}_n^{uc}$ to represent the subsets of critical and ultra-critical alternate tasks respectively. We assume that the total utilization of the critical, highly-critical and ultra-critical tasks together with all the alternates on a processing node can be as much as 100%:

$$U(\Gamma_n^{c}) + U(\Gamma_n^{hc}) + U(\Gamma_n^{uc}) + U(\bar{\Gamma}_n^{c}) + U(\bar{\Gamma}_n^{uc}) \leq 1$$

We also assume that the delays introduced by error detection mechanisms, voting mechanisms as well as communication delays between the processing node and the voter are included in the task worst case execution requirements. These assumptions imply that, during error recovery, the execution of non-critical tasks cannot be permitted as it may result in overload conditions. We assume that the scheduler has adequate support for flagging non-critical tasks as unschedulable during such scenarios, along with appropriate error detection mechanisms for the errors that can be tolerated by time redundancy.

3 Error model and error recovery strategy

Our approach builds upon the failure concepts originally introduced in [1, 4]. For the sake of readability, we denote the ith replica of a replicated task r in the system by r_i. The output delivered by replica r_i is specified by two domain parameters, viz., value and time, together with their admissible deviations:

$$\textit{Specified output for } r_i = < v_i^*, t_i^*, \sigma, \delta >$$

where v_i^* is the correct value, t_i^* is the correct time *interval* seen by a perfect observer (whose length is equal to (T(r_i) - BCET(r_i))), during which the output must be delivered, $[v_i^* - \sigma, v_i^* + \sigma]$ is the admissible value range and $[t_i^{*start} - \delta, t_i^{*end} + \delta]$ is the admissible time interval for output delivery as per system specifications. An output delivered by r_i is denoted as:

$$\textit{Delivered output from } r_i = < v_i, t_i >$$

where v_i is the value and t_i is the time point at which the value was delivered. We define the output generated by replica r_i as *incorrect in value domain* if

$$v_i < v_i^* - \sigma \text{ or } v_i > v_i^* + \sigma$$

and *incorrect in time domain* if

$$t_i < t_i^{*start} - \delta \; (early\; timing\; failure)$$

or if

$$t_i > t_i^{*end} + \delta \; (late\; timing\; failure).$$

Furthermore, since replica outputs are only comparable as long as they are released and completed within a time interval whose length is equal to the replicas' period, we define the output generated by replica r_i as *early compared to* r_j (equivalently, we say that r_j is *late compared to* r_i), if

$$t_i < t_j - T(r) + BCET(r).$$

Our goal is to develop a cascading fault tolerance mechanism to enable both a wide error coverage and efficient resource usage in dependable real-time systems composed of tasks with various criticality levels. Hence, the possible redundancy configurations are:

1. no redundancy
2. time redundancy
3. space redundancy
4. time and space redundancy

We assume that the value errors caused by a large variety of transient and intermittent hardware faults can effectively be tolerated by a simple re-execution of the affected task, whilst the value errors caused by software design faults could be tolerated by executing an alternate action such as recovery blocks or exception handlers. Both situations could be considered as executions of another task (either the primary itself or an alternate) with a specified computation time requirement. On the other hand, in addition to all types of errors that can be tolerated by a time redundancy mechanism, value errors caused by permanent hardware faults and timing errors can be tolerated by a space redundancy mechanism. If ultimate dependability is desired, using both approaches together will provide recovery from a wide range of errors, as well as from an increased number of error occurrences, with the obvious additional cost. The error type coverage achieved by each technique is shown in Figure 2.

Our approach relies on the following set of basic assumptions (to a large extent based on [9]):

- A1 non-faulty task replicas produce values within a specified admissible value range after each computation block
- A2 non-faulty task replicas produce values within a specified admissible time interval after each computation block
- A3 incorrect replica outputs do not form (or contribute in forming) a consensus
- A4 the voting mechanism does not fail, as being designed and implemented as a highly reliable unit

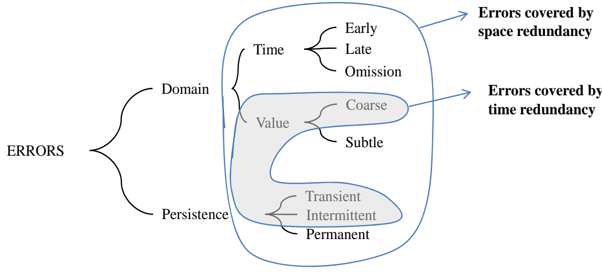


Figure 2. Coverage of error types achieved by time and space redundancy

4 Time redundancy

An approach to maximize the fault tolerance capability in FPS has been presented in [8]. In this paper we propose a scheduler independent approach, suitable for, e.g., FPS, EDF or table driven scheduling (TD), to enable time redundancy *for every critical and ultra-critical task instance*, executed along with highly-critical as well as non-critical tasks.

4.1 Overview

Our goal is to, first, derive feasibility windows for each task instance $\tau_i^j \in \Gamma$ to reserve the slack necessary to the re-execution of every critical task instance. Then, we derive scheduler dependent attributes to ensure task executions within their new feasibility windows. However, our task model consists of critical, highly-critical, ultra-critical, as well as non-critical tasks. While executing non-critical tasks in the background can be a safe and straightforward solution, in our approach we aim to provide non-critical tasks a better service than background scheduling. Hence, depending on the criticality of the original tasks, the new feasibility windows we are looking for differ as:

1. *Fault Tolerant* (FT) feasibility windows for critical, highly-critical and ultra-critical task instances

2. *Fault Aware* (FA) feasibility windows for non-critical task instances

While the FT feasibility windows represent time intervals in which critical, highly-critical and ultra-critical task instances need to execute and complete to ensure the time redundancy, the derivation of FA feasibility windows has two purposes: 1) to prevent non-critical task instances from interfering with critical ones, i.e., to cause any critical task instance to miss its deadline, while 2) ensure the non-critical a short response time. Since the size of the FA feasibility windows depend on the size of the FT feasibility windows,

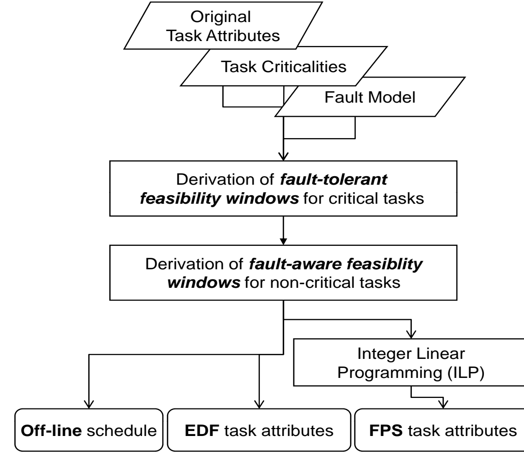


Figure 3. Time redundancy - overview

in our approach we first derive FT-feasibility windows and then FA feasibility windows.

At this point we need to make sure that the underlying scheduler will actually schedule the tasks within their derived FT/FA feasibility windows. While the new deadlines can be used directly under EDF, or an off-line scheduler can easily allocate the tasks to feasible time slots and store the information in a scheduling table, in FPS a further priority assignment is required. Later on, we present a method to derive FPS attributes that guarantees task execution within their FT/FA deadlines under FPS.

Obviously, the maximum utilization of the critical, highly-critical and ultra-critical tasks and alternates can never exceed 100%. Hence, during error recoveries, the non-critical task executions cannot be permitted as it may result in overload conditions. We assume that the scheduler has adequate support for flagging non-critical tasks as unschedulable during such scenarios, in addition to appropriate error detection mechanisms in the operating system. An overwiew of the proposed methodology is presented in Figure 3.

4.2 Derivation of FT- and FA feasibility windows

The first part of our approach is the derivation of FT and FA feasibility windows. Our approach first derives new FT deadlines for the primary versions of the critical and ultra-critical task instances, so that in case of a critical task error, an alternate version of that instance can be executed before its original deadline. As highly-critical tasks do not have alternates, their FT deadlines are equal to the original deadlines. Then, FA deadlines are derived for the non-critical task instances, so that the provided fault tolerance for the critical ones is not jeopardized.

We illustrate the derivation of FT and FA feasibility windows by using a simple example consisting of three tasks with parameters described in table 1.

Task	T	WCET	criticality
A	3	1	non-critical
B	4	1	critical
C	12	3	ultra-critical

Table 1. Example: task set attributes

Derivation of FT deadlines: The aim of this step is to reserve sufficient resources for the executions of the critical and ultra-critical task alternates in the schedule. Our primary goal is to provide re-execution guarantees for every critical task instance. Thus, we calculate the latest possible start of execution for critical and ultra-critical task alternates by using the concept of earliest deadline last (EDL) scheduling policy [7]. Specifically, we select the set of critical, highly-critical and ultra-critical tasks Γ_n^c, Γ_n^{hc} Γ_n^{uc} calculate FT-deadlines for each of them when scheduled by EDL together with their alternates $\bar{\Gamma}_n^c, \bar{\Gamma}_n^{hc}$ on every node n. The FT deadlines of a critical or ultra-critical task instances is equal to the latest start time of its alternate, while the FT deadline of the highly-critical task instances (that do not have alternates) are equal to their original deadlines. In this way, we reserve sufficient resources for each critical and ultra-critical task alternate, assuming that the cumulative processor utilization of the primaries and their alternates does not exceed 100%.

Lemma 4.1. *A schedule produced by EDL on a set of periodic tasks with deadlines equal to their periods and executions equal to their WCET, is identical to the schedule produced by EDF on the same task set, but in reversed order, i.e.,at any instant* $t \leq LCM$

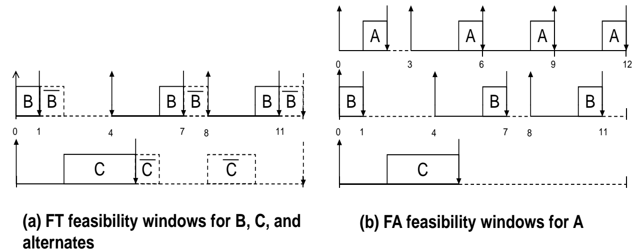
$$f_{EDF}(t) = f_{EDL}(LCM - t)$$

Proof. The proof is a straightforward from the results provided in [7], based on the idle time calculation under EDF and EDL. □

The FT feasibility windows for B and C are presented in figure 4(a). The dashed boxes represent the latest possible execution of the alternates.

Derivation of FA deadlines: We aim to provide FA deadlines to non-critical task instances to protect critical, highly critical and ultra-critical ones from being adversely affected. As a part of recovery action upon errors, the underlying fault tolerant on-line mechanism checks if there is enough time left for the non-critical task instances to complete before their new deadlines. If not, these instances are not executed.

To derive the FA deadlines, we repeat the process used in the derivation of FT deadlines, on the set of non-critical tasks, Γ_n^{nc}, but *in the remaining slack* after the critical highly-critical and ultra-critical task primaries are scheduled to execute as late as possible *within their FT feasibility windows*, as derived in the previous step. The main reason is that interference between primary tasks executions "as late as possible" and non-critical tasks may result in situations where primary tasks may miss their deadlines, when the total task utilization exceeds 100%, i.e., due to the execution of critical or ultra-critical task alternates. Hence, the primaries need to be guaranteed that their latest possible execution windows are free from interference with non-critical task. A benefit of this approach is that the non-critical tasks can be provided a better service than, e.g., background scheduling as they are allowed to execute before the critical ones. Note that we do not need to take into account the interference between non-critical tasks and critical or ultra-critical alternates, as the first ones are shed by the scheduler during the execution of the critical alternates. The derived FA feasibility window for A is shown in figure



(a) FT feasibility windows for B, C, and alternates

(b) FA feasibility windows for A

Figure 4. FT and FA deadlines derivation

4(b). In some cases, we may fail finding *valid* FA deadlines for *some* non-critical task instances. We say that a FA deadline, $D_{FA}(\tau_i^j)$, is *not valid* if $D_{FA}(\tau_i^j) - est(\tau_i^j) < C(\tau_i^j)$, i.e., if the processor utilization demand exceeds the available resources. This scenario could occur since the task set consists now of tasks with deadlines less than periods, i.e., due to the newly derived FT feasibility windows. In these cases, we keep the original deadline and, in the following subsections, we present scheduler specific solutions towards handling these non-critical tasks.

4.3 EDF scheduling

EDF can feasibly execute all primaries and alternates up to 100% utilization if the deadlines are equal to the periods. However, in our task model, only the alternates and the

highly critical tasks have FT deadlines equal to their periods, while the FT/FA deadlines of the rest of the tasks are less than periods.

Theorem 4.2. *All critical task primaries and alternates, $\tau_i^j \in \{\Gamma_n^c \cup \Gamma_n^{hc} \cup \Gamma_n^{uc} \cup \bar{\Gamma}_n^c \cup \bar{\Gamma}_n^{uc}\}$, complete before their FT deadlines, when scheduled by EDF.*

Proof. We prove the lemma by contradiction. Let assume that there exists a time interval in the schedule during which the cumulative processor demand [3] exceeds the length of the interval, i.e., $\exists\ L \geq 0$ such that $L < \sum_{i=1}^{n}(\lfloor \frac{L-D_i}{T_i} + 1\rfloor)C_i$. Here we have two cases:

1. the task set does not contain non-critical tasks. We have used the EDL to derive the latest finishing time for primary tasks, which are equal to the latest start time of the alternates. If L exists under EDF, then the same interval exists for EDL as well but located in a symmetric location in the schedule (lemma 4.1). Hence, a task set with an utilization less than or equal to 100% would not be schedulable under EDF, which contradicts the EDF schedulability bound.

2. the task set contains non-critical tasks with valid FA deadlines as well. In this case, FA deadlines are derived in the slack after scheduling the critical primaries. Hence, critical task primaries are guaranteed an execution window equal to their WCET. Moreover, non-critical tasks are shed by the scheduler during the execution of the critical alternates, hence the alternates are guaranteed to be scheduled before their deadlines.

□

Theorem 4.3. *In error free scenarios, all non-critical tasks with valid FA deadlines $\tau_i^j \in \Gamma_n^{nc}$ complete before their deadlines, when scheduled by EDF.*

Proof. If the algorithm finds any time interval in which the utilization demand exceeds the length of the interval, the non-critical tasks are assigned non-valid FA deadlines. If no such interval is found, the non-critical tasks are assigned valid FA deadlines. Hence, the non-critical tasks with valid FA deadlines tasks are schedulable by EDF together with the critical ones. □

In the case of the non-critical tasks with non-valid FA deadlines, the safest approach is to not schedule them at all, as the primary goal is to guarantee the feasible execution of all critical primaries and alternates. On the other hand, an on-line server [23] could be used to accommodate the execution of these tasks in cases the load permits, e.g., during error free scenarios or/and tasks execute less than WCET.

An example of an EDF schedule is presented in figure 5. Here, C and the last two instances of B are hit by errors.

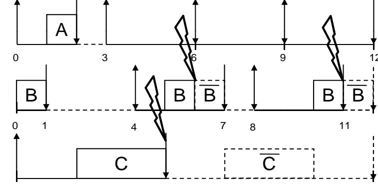


Figure 5. EDF Schedule under errors

4.4 Table driven scheduling

An arbitrary off-line scheduler is used to allocate tasks to slots within their derived FT/FA feasibility windows. As, according to our assumptions, the total utilization, including non-critical tasks, will exceed 100%, a feasible off-line allocation of the non-critical tasks will not be possible. However, in error free scenarios, the slots allocated to critical or highly-critical alternates will be unused due to the inflexibility of the off-line scheduling paradigm. Thus, the scheduling table could be derived to provide, e.g., a dual entry for the slots allocated to the alternates, i.e., in case the execution of the alternates is not required, non-critical execution can be scheduled instead. At the same time, existing approaches to add flexibility to off-line scheduling can be used as well [10]. The schedulability of the task set in table driven scheduling is proved by construction.

4.5 FPS

Here, our goal is to provide tasks with FPS attributes that guarantee the task executions within their derived FT/FA feasibility windows. As, in the general case, FPS has a lower schedulability bound than EDF, in our approach, we assign FPS task attributes to match the EDF priority ordering, thus, guaranteeing the feasible re-execution of every critical, highly critical and ultra-critical tasks, together with alternates, under the specified assumptions. We do so by analyzing the task set with new deadlines and derive priority relations for each point in time at which at least one task instance is released, based on their earliest deadlines. In the case of the non-critical tasks with non-valid FA deadlines, we make sure that the priority assignment mechanism will assign the non-critical tasks a background priority, i.e., lower than any other critical task, and any other non-critical task with a *valid FA deadline*.

As the priority assignment reconstructs the EDF schedule, when solving the derived priority inequalities it may happen that different instances of the same task need to be assigned different priorities. These cases cannot be expressed directly with fixed priorities and are the sources for *priority assignment conflicts*. The algorithm detects such situations, and circumvents the problem by splitting a task

into its instances. Then, the algorithm assigns different priorities to the newly generated "artifact" tasks, the former instances. Key issues in resolving the priority conflicts are the number of artifact tasks created, and the number of priority levels. Depending on how the priority conflict is resolved, the number of resulting tasks may vary, i.e., based on the size of the periods of the split tasks. Our algorithm minimizes the number of artifact tasks by using ILP for solving the inequalities to find the priorities and the splits that yield the smallest number of FT FPS tasks. The new task's instances comprise all instances of the original tasks. A complete description of the method together with an example of the ILP problem formulation, as well as an evaluation of the approach in comparison with the fault tolerant version of rate monotonic algorithm, can be found in [8].

Theorem 4.4.

- *All critical task primaries and alternates, $\tau_i^j \in \{\Gamma_n^c \cup \Gamma_n^{hc} \cup \Gamma_n^{uc} \cup \bar{\Gamma}_n^c \cup \bar{\Gamma}_n^{uc}\}$, complete before their FT deadlines, when scheduled by FPS.*
- *In error free scenarios, all non-critical tasks with valid FA deadlines $\tau_i^j \in \Gamma_n^{nc}$ complete before their deadlines, when scheduled by FPS.*

Proof. As the priority relations reflect the EDF scheduling policy, based on which the ILP solver provides absolute priority values, the theorems 4.2 and 4.3 are valid here as well. □

The task set scheduled according to FPS under the worst case error occurrence scenario is presented in figure 6. The artifact tasks, e.g., B1, B2, etc, were created from the original task instances to resolve the priority assignment conflicts. For example, B1 was created from the first instance of B and so forth. Note that the large number of splits resulting in 9 tasks is due to the high fault tolerance requirements, i.e., one re-execution per task instance, as well as the high processor utilization of the original task set.

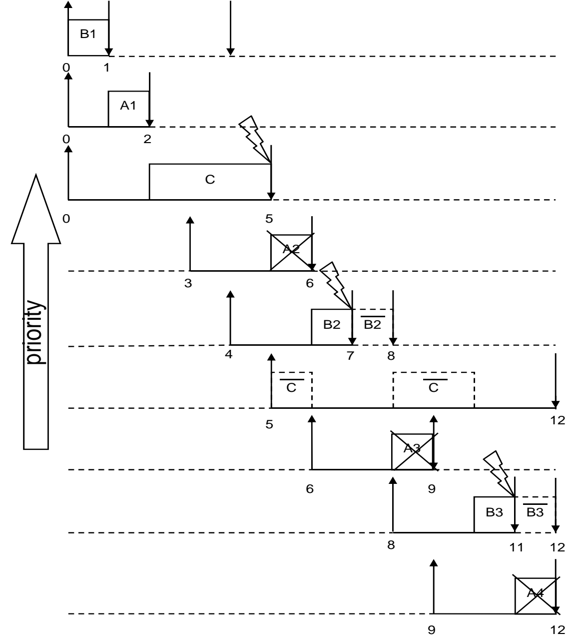


Figure 6. FPS schedule under errors

5 Space redundancy in real-time systems

In this section we propose a voting strategy that explicitly considers failures in both value and time domain. The correctness of our method relies on a number of conditions regarding the permissible number of replica failures:

$\mathcal{C}1$ The number of timely replica outputs should be greater than or equal to the minimum number of replica outputs required for consensus in time domain (M_t):

$$N - F_t \geq M_t$$

where N is the total number of replica tasks, F_t is the number of replica outputs with incorrect timing, and M_t is either majority ($M_t \geq \lceil \frac{N+1}{2} \rceil$) or plurality ($M_t \leq N/2$) in time domain.

$\mathcal{C}2$ In order to achieve a consensus in value domain, the number of replica outputs with correct values should be greater than or equal to the number of minimum number of replicas required to achieve consensus in value domain (M_v). However, since in the general case timeliness is a precondition for value correctness, i.e., since we want to compare only relatively timely replica outputs and we cannot wait for late outputs, the replica outputs forming the consensus in value domain needs to be free from any type of errors:

$$N - F \geq M_v$$

where F is the number of replica outputs with incorrect timing and/or value, and M_v is either majority ($M_v \geq \lceil \frac{N+1}{2} \rceil$) or plurality ($M_v \leq N/2$) in value domain.

Our goal is twofold:

1. always deliver the correct value within the correct time interval, if the conditions $\mathcal{C}1$ and $\mathcal{C}2$ hold
2. signal the disagreement, otherwise.

5.1 Method

In order to reach an agreement on the correct replica output, the voter needs to receive a number of matching values

which are also delivered not far apart from each other with respect to time, i.e., replica outputs need to form a consensus in both time and value domains. The first step is to ensure that the voter detects and delivers the agreed value (if obtained) within $[t^{*start} - \delta + BCET(r), t^{*end} + \delta]$. To do so, M_t out of N replica outputs need to be delivered within the absolute time interval of $[t^{*start} - \delta + BCET(r), t^{*end} + \delta]$. Furthermore, those outputs must also be received within a relative time interval, equal to MPD (referred to as *feasible voting window* henceforth).

The maximum number of sets, consisting of M_t consecutive replica outputs each (out of the N replicas), is $N - M_t + 1$. Since the consensus in time domain can be reached in any of these sets, a separate feasible voting window needs to be initiated upon receiving each of the first $N - M_t + 1$ replica outputs. We keep track of the feasible voting windows by using simple countdown timers. Once an agreement in time domain is obtained, then the values are voted. If an agreement in value domain is not obtained within a particular feasible voting window, the process continues with subsequent feasible voting windows, until agreement in both time and value domains can be achieved, or violations of $\mathcal{C}1$ or $\mathcal{C}2$ are detected.

Depending on the real-time application characteristics, a value produced by a replica may be considered *valid* or *invalid* for the purpose of voting, in case it is produced early, i.e., *all* received values vs. *all timely* received values are voted in value domain. We illustrate this voting dilemma by using the scenario illustrated in Figure 7. Let us assume an airbag control system, where a collision sensor is replicated in five different nodes that produce one out of two values periodically, e.g., value a in case of a collision detection, or value b otherwise. If a collision is detected at a time $t \leq t_1$ let us assume that the consensus has to be formed before t_6 in order to inflate the airbag within a correct time interval. In our example, the first two values are detected as *relatively early* compared to the last three, and the last three are identified as *timely* among themselves. However, in this case, even an early value has to be taken into consideration in the voting since an early collision detection is still a valid output with respect to the value domain. Thus, the output has to be voted upon receiving the last value at time t_5, among *all* values, i.e., a, a, a, b, and b, resulting in an output a at time (t_5). In this case, the condition $\mathcal{C}2$ becomes:

$\mathcal{C}2$ The number of failure-free replica outputs, excepting the ones with early timing errors must be greater than or equal to the minimum number of replica outputs required to achieve consensus in value domain (M_v):

$$N - (F - F_t^e) \geq M_v$$

where F is the number of replica outputs with incorrect timing and/or value and F_t^e is the number of replica outputs with early timing failure.

The benefit of this observation is that the number of replicas required to mask a given number of failures (in time and value domain) can be significantly reduced, compared to traditional NMR approaches, as replica outputs failed in one domain may still be used to reach consensus in the other.

On the other hand, let us assume that the same Figure 7 illustrates an altitude sensor in an airplane, replicated by five nodes to read and output the altitude periodically to the voter, where data freshness may be a more desirable aspect. As the correct window of time for the output is the same as described in the previous example, the only relevant values to be taken into consideration by the voter are a, b, and b corresponding to the time points t_3, t_4, and t_5 respectively. Hence, the output produced in this scenario at time t_5 is b.

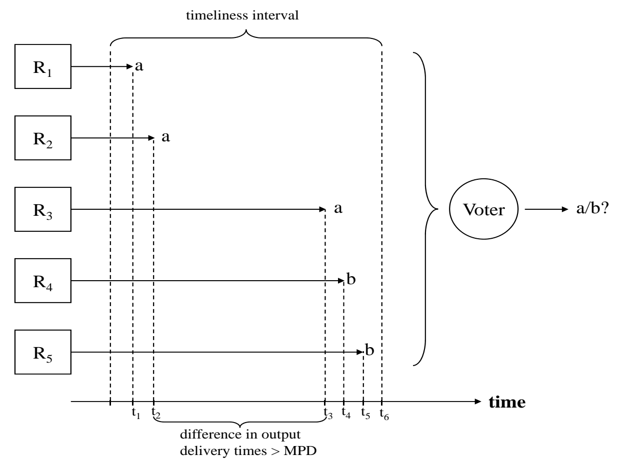


Figure 7. Voting dilemma

Upon finding a feasible voting window, the decision on whether the early generated replica outputs are used in value voting or not, results in two cases:

Case 1 *Early and timely outputs are considered valid.* If a consensus among all received values exists, it is delivered as the correct output.

Case 2 *Only timely outputs are considered valid.* If a consensus among the *timely* received values exists, it is delivered as the correct output.

If no agreement in value domain is reached within a feasible voting window, the process continues with the subsequent window. If the end of last feasible voting window is reached, or all replica outputs are received without reaching an agreement on the values, disagreement is signalled to the nodes indicating a violation of $\mathcal{C}1$ or $\mathcal{C}2$. The complete description of the methodology, as well as an algorithm enabling voting in time and value is presented in [2].

6 Cascading redundancy

The cascading redundancy approach addresses the different error and cost models of tasks with various criticality by allowing the following configuration levels for each predetermined criticality:

1. no redundancy for *non-critical tasks*
2. only time redundancy for *critical tasks*
3. only space redundancy for *highly-critical tasks*
4. both time and space redundancy for *ultra-critical tasks*

In this section, we present an algorithm (Algorithm 1) which enables the use of cascading redundancy in real-time systems. The algorithm is executed upon every task instance release, regardless of their criticality. Highly-critical and ultra-critical tasks are replicated, and the replica outputs are voted by the voting mechanism implemented on a a stand-alone node, to ensure correctness in both value and time. For the sake of readability, in our description each highly-critical and ultra-critical task is assumed to have a dedicated stand-alone voter. In a practical implementation, however, the voter tasks can execute on a single dedicated voting node. In this case, the worst case response times of the voter tasks need to be accounted for in the primary task worst case execution times, in the off-line time redundancy approach.

Algorithm 1: Cascading redundancy

```
input : output value of τi, type of τi
output: consensus value for τi or
        indication of error (trigger for time redundancy) or
        indication of disagreement (trigger for emergency)
1  while output value of τi is not received do wait;
2  type ← type of τi
3  value ← value of τi
4  switch type do
5     case critical
6        if DetectError(value) then
7           signal error detected;
8        else
9           output value;
10       end
11       break;
12    case highly-critical
13       SendToVoter(value);
14       while voter-output is not received do wait;
15       if voter-output is disagreement then
16          signal disagreement;
17       else
18          output consensus;
19       end
20       break;
21    case ultra-critical
22       if DetectError(value) then
23          signal error detected;
24          while output value of τ̄i is not received do wait;
25       end
26       SendToVoter(value);
27       while voter-output is not received do wait;
28       if voter-output is disagreement then
29          signal disagreement;
30       else
31          output consensus;
32       end
33       break;
34    otherwise
         // (non-critical)
35       break;
36    end
37 end
```

Upon receiving identical requests or inputs, replicas start their executions on separate processors whose clocks are allowed to drift from each other by a maximum deviation. When the highly-critical task replicas complete their executions, the outputs are sent to the stand-alone voter. For ultra-critical tasks, before sending the outputs to the voter, error detection for transient coarse value errors and re-executions of primaries or executions of alternates are performed upon error occurrences. We assume that the deviation in message transfer times from the replicas to the voter is assumed to be bounded by using reliable communication techniques. Furthermore the delays introduced by error detection mechanisms, voting mechanisms, as well as communication delays between the processing node and the voter are included in task executions for the purpose of schedulability analysis. Upon receiving the the required replica values, the voter starts executing the voting algorithm and sends the agreed value back to the processing nodes, or signals the non-existence of a correct output before the latest deadline among the tasks whose outputs are subject to voting. In some cases, however, the voter output may be delivered after one, or several task deadlines. This scenario can occur in cases when the last required replica output is provided close to its deadline, while the first replicas are executing on nodes drifting ahead of the clock, thus, with deadlines before the latest admissible voter output. In these cases the schedules on nodes which are drifting ahead of time are rolled back to the end of the replica task period, with adequate checkpointing mechanisms. This action will ensure that the subsequent tasks will process the consensus value delivered by the voter and, in the worst case, force the nodes executing such tasks to drift from the correct time by maximum δ, which is still admissible with respect to the real-time specifications.

As the replicas of a task need to agree on an output value, the only feasible order for performing cascading redundancy is first time redundancy followed by space redundancy. Whenever a consensus is achieved for a replicated critical task, any uncompleted replica instances, or their alternate actions becomes obsolete for the purpose of voting and, thus, can be shed. This can allow for, e.g., improving the service to non-critical tasks. In Figure 8, neither the

ultra-critical task B on Node 3 nor its alternate is executed since the voter reports that majority has been formed before B starts executing on that node.

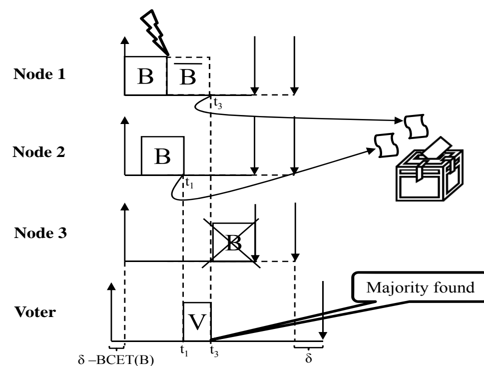


Figure 8. Cascading redundancy - example

7 Conclusions

In this paper we have presented a fault tolerant scheduling framework that enables the use of both time and space redundancy in dependable real-time systems. We have proposed a cascading redundancy approach that is capable of tolerating errors with a wider coverage (with respect to error frequency and error types) than either time or space redundancy alone, is independent from the underlying scheduler, can cope with tasks of mixed criticality levels, and guarantees the feasible time and space replication of every critical task instance while fully utilizing the resources.

Future work will include task migration aspects together with on-line adaptations, in order to optimize the schedulability of the lower critical tasks, e.g., non critical, in error free scenarios.

References

[1] A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability. *Research Report N01145, LAAS-CNRS*, 2001.

[2] H. Aysan, S. Punnekkat, and R. Dobrin. VTV – a voting strategy for real-time systems. *IEEE Pacific Rim International Symposium on Dependable Computing*, 2008.

[3] S. K. Baruah, R. R. Howell, and L. E. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2:301–324, 1990.

[4] A. Bondavalli and L. Simoncini. Failure classification with respect to detection. *IEEE Workshop on Future Trends in Distributed Computing*, 1990.

[5] S. S. Brilliant, J. C. Knight, and N. G. Leveson. The consistent comparison problem in N-version software. *IEEE Trans. on Software Engineering*, 15(11):1481–1484, 1989.

[6] A. Burns, R. I. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. *Euromicro Real-Time Systems Workshop*, 1996.

[7] H. Chetto and M.Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, 1989.

[8] R. Dobrin, H. Aysan, and S. Punnekkat. Maximizing the fault tolerance capability of fixed priority schedules. *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2008.

[9] P. Ezhilchelvan, J.-M. Helary, and M. Raynal. Building responsive TMR-based servers in presence of timing constraints. *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2005.

[10] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. *Real-time Systems Symposium*, 1995.

[11] S. Ghosh, R. Melhem, and D. Mosse. Enhancing real-time schedules to tolerate transient faults. *Real-Time Systems Symposium*, 1995.

[12] C.-C. Han, K. G. Shin, and J. Wu. A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults. *IEEE Trans. Computers*, 52(3):362–372, 2003.

[13] H. Kopetz. Fault containment and error detection in the time-triggered architecture. *International Symposium on Autonomous Decentralized Systems*, 2003.

[14] J. Lala and R. Harper. Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE*, 82(1):25–40, 1994.

[15] J.-C. Laprie. Dependable computing and fault-tolerance: Concepts and terminology. *International Symposium on Fault-Tolerant Computing, ' Highlights from Twenty-Five Years'.*, 1995.

[16] G. Lima and A. Burns. Scheduling fixed-priority hard real-time tasks in the presence of faults. *Lecture Notes in Computer Science*, pages 154–173, 2005.

[17] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *Journal of Research and Development*, 6:200–209, 1962.

[18] J. V. Neuman. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, pages 43–98, 1956.

[19] M. Pandya and M. Malek. Minimum achievable utilization for fault-tolerant processing of periodic tasks. *IEEE Transactions on Computers*, 47(10), 1998.

[20] S. Punnekkat, A. Burns, and R. I. Davis. Analysis of checkpointing for real-time systems. *Real-Time Systems*, 20(1):83–102, 2001.

[21] K. Ravindran, K. Kwiat, A. Sabbir, and B. Cao. Replica voting: a distributed middleware service for real-time dependable systems. *International Conference on Communication System Software and Middleware*, 2006.

[22] K. Shin and J. Dolter. Alternative majority-voting methods for real-time computing systems. *IEEE Transactions on Reliability*, 38(1):58–64, 1989.

[23] M. Spuri and G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. *Real-time Systems Symposium*, 1994.