IIETA International Information and Engineering Technology Association
Advancing the World of Information and Engineering

Journal Européen des Systèmes Automatisés
Vol. 57, No. 6, December, 2024, pp. 1713-1720
Journal homepage: http://iieta.org/journals/jesa

Watchdog Timer for Fault Tolerance in Embedded Systems

Ridha Mehalaine[1,2*], Meriem Djezzar[1,2], Djamal Nessah[1,2], Zineb Saiad[2], Asma Saidi[2]

[1] ICOSI Laboratory, University of Khenchela, Khenchela 40004, Algeria
[2] Department of Computer Science, University of Khenchela, Khenchela 40004, Algeria

Corresponding Author Email: r_mahaline@univ-khenchela.dz

https://doi.org/10.18280/jesa.570619

Received: 23 November 2024
Revised: 6 December 2024
Accepted: 13 December 2024
Available online: 31 December 2024

Keywords:
watchdog timer, embedded systems, scheduling, fault tolerance, EDF algorithm*

ABSTRACT

Embedded artificial intelligence encompasses a diverse range of technologies, from advanced algorithms to highly specialized computing systems. Intelligent embedded systems are playing an increasingly crucial role in various industries such as automotive, aerospace, healthcare, and IoT. When considering the place that intelligent embedded systems take in our daily lives, it is very important to understand how critical their security is. In order to ensure their high performance, energy efficiency, and robustness, it is imperative to ensure rigorous task scheduling. We are interested in the problem of hard real-time fault-tolerant scheduling for periodic and independent preemptive tasks. This paper focuses on proposing a fault-tolerant scheduling algorithm for these systems. By using the watchdog timer, which allows intelligent embedded systems to be more autonomous by detecting processor errors and adopting the Earliest Deadline First (EDF) algorithm to allow our system to respect time constraints. The objective is to improve reliability and efficiency by ensuring the execution of critical tasks despite the presence of faults. Designing and implementing a fault-tolerant scheduling algorithm for embedded systems is a crucial aspect in various industries. This helps to improve the reliability and security of intelligent embedded systems, which is essential to ensure the smooth operation of the system.

1. INTRODUCTION

In the era of ubiquitous digitalization, embedded systems play a fundamental role in our daily lives. Whether controlling vital medical devices, piloting autonomous vehicles, or managing critical infrastructures, these systems provide discrete intelligence that shapes our environment in invisible but profound ways. At the heart of this ubiquity are real-time embedded systems, specialized computing infrastructures designed to meet the requirements of strict time constraints. Most embedded systems are indeed strict real-time systems. This means that they must meet strict time constraints and deliver results within specific deadlines. Real-time embedded systems are extremely sensitive to faults.

A single fault can have catastrophic consequences on the proper functioning of these systems. This is why it is essential to implement fault tolerance mechanisms to ensure their reliability. When a fault occurs, it can disrupt the normal operation of the system. In some cases, this can even endanger the lives of people relying on these systems, as in the case of vital medical devices. The fault sensitivity of real-time embedded systems underlines the importance of their rigorous design and the use of advanced techniques to ensure their reliability and safety.

A necessary but not sufficient condition for the proper functioning of embedded real-time systems is the respect of time constraints throughout the life of the system. The occurrence of faults is inevitable, whatever the precautions taken (human error, malicious intent, hardware aging, natural disaster, etc.), which could lead to catastrophic consequences (loss of money, time or worse, human lives). An embedded system must be fault tolerant, in a way that it must be able to continue its operation despite the failure of a part of its hardware or system. Reliability is the probability that a system will be continuously in operation over a given period. Critical real-time embedded systems must thus cover an important property of safe operating systems which is reliability. The presence of techniques that ensure operational safety is vital in the design of these systems. Fault tolerance is one of the methods used in the literature to ensure the operational safety of embedded real-time systems.

Our goal in this work is to design a strict real-time system that can efficiently detect and recover from faults while maintaining the required reliability and performance. To minimize the risks, it is essential to implement fault detection and recovery strategies. This may include the use of redundancy, where multiple components or modules work in parallel to verify results and detect errors. To design safe systems, we rely on reliability and fault tolerance. Fault tolerance is an essential aspect of real-time embedded systems, it concerns the ability of a system to function correctly even in the presence of faults. This ensures continuity of operations and user safety. As for reliability, it aims to ensure that the system operates predictably and without errors, even under critical conditions. This involves the use of robust design techniques, extensive testing, and error detection and

correction mechanisms.

The goals of fault tolerance and reliability are to create reliable, stable, and secure systems that can meet the stringent requirements of the applications for which they are intended. We strive to implement fault-tolerant scheduling for distributed embedded systems with multiple processors. A scheduling algorithm with a reliability objective for independent periodic tasks is preemptive even in the presence of processor faults, which is our goal. The objective of this work is to propose an approach that meets the needs of fault tolerance using the watchdog timer.

2. RELATED WORK

In this section, we present some relevant works on fault tolerance in real-time embedded systems.

Reghenzani et al. [1] presented the state-of-the-art scientific work analyzing the Software-Implemented Fault Tolerance SIFT mechanisms and their real-time scheduling. It presents an extension of the model based on resource allocation functions, which allows a more accurate representation of the failure probabilities for each task. Then, they presented how to calculate the probability that a job is affected by a failure and the resulting impact on the failure requirement. Using this joint temporal fault model can improve the satisfaction of failure and scheduling requirements.

Kumar et al. [2] studied the various fault tolerance techniques that are used in many distributed real-time systems. The paper focuses on the types of faults occurring in the system, fault detection techniques and recovery techniques used. Thet explained how these methods are applied to detect and tolerate faults in various distributed real-time systems. The fault has to be detected by applying a reliable fault detector followed by a recovery technique. Many fault detection techniques are available but it is necessary to apply a proper fault detector. An unreliable fault detector may commit errors by mistakenly suspecting a correct process or trusting a failed process.

Manimaran and Murthy [3] proposed an algorithm to schedule dynamically arriving real-time tasks with resource-based fault tolerance requirements and primary backup in a multiprocessor system. The tasks are assumed to be non-preemptive and each task has two copies (versions) that are mutually exclusive in space as well as in scheduling, to handle permanent processor failures and to achieve better performance, respectively. According to the simulation results, the proposed algorithm tolerates more than one fault at a time and employs performance improvement techniques.

Ramanathan and Shin [4] proposed an active replication-based approach to solve the problem of delivering critical messages before their deadline in strict real-time embedded systems in the case of processor or communication link failures at a lower cost. They use a distributed architecture with a hexagonal mesh topology and a hypercube topology. The idea is to duplicate each message at least twice depending on its criticality and the number of processors and communication links it has to traverse. Then, the messages are broadcast on different routes to reduce the cost of retransmission.

Chevochot and Puaut [5] presented another approach to tolerating faults in distributed hard real-time systems. They developed a replication tool called HYDRA, which allows to integration of active, passive, or hybrid replication into the scheduling algorithm. This approach aims to tolerate both transient and permanent faults of a physical component, using sites composed of processors, memories and clocks. Sensors are exposed to timing and functional errors, while actuators are assumed to be reliable. This is a very promising solution to ensure the reliability of distributed hard real-time systems.

Despite the existence of several approaches to solve fault tolerance in real-time embedded systems, most of these approaches do not deal with processor faults; and those that deal with this type of fault, use in their context, aperiodic tasks to simplify their approach, unlike real-time embedded systems that generally use periodic tasks. The use of data redundancy or physical resources as a solution to solve the fault problem in these approaches can lead to the non-compliance with time constraints in strict real-time embedded systems. Our approach is based on the proposal of a fault-tolerant and not negatively affecting scheduling algorithm in terms of time and energy constraints, which uses periodic tasks in a strict real-time embedded system based on the use of the watchdog timer for the detection of processor faults.

3. EMBEDDED SYSTEMS

Embedded systems play an important role in our daily lives. These systems are ubiquitous in many devices and equipment, ranging from consumer electronics to complex industrial applications. The integration of these systems in specific environments allows dedicated tasks to be accomplished efficiently and often transparently for the end user. Their role is essential in sectors such as automotive, aerospace, healthcare, telecommunications and many others.

According to the language of Molière, the word embedded is derived from the verb "to embark" which means "to put something on board a ship, a plane or a vehicle an embedded system can be defined as: "A stand-alone electronic and computer system, which is dedicated to a very specific task" [6].

An embedded system is a computer system whose calculation means are embedded in the controlled process. Embedd computing means implies, in addition to space constraints (size, weight, shape), energy consumption and heat dissipation constraints. In addition, the power supply for the computing elements is embedded (batteries, fuel, etc.), and/or ambient (solar panels, etc.) [7].

An embedded system (ES) is a specialized computer system that constitutes an integral part of a larger system or a machine. Typically, it is a system on a single processor and whose programs are stored in ROM. A priori, all systems that have digital interfaces (watch, camera, car, etc.) can be considered as ES. Some ES have an operating system and others do not because all their logic can be implemented in a single program [8].

An embedded system is an autonomous combination of hardware and software (electronics plus computing) dedicated to generally carrying out a specific task in interaction with its environment and respecting often severe constraints such as: energy consumption, weight, reliability, response time, cost, etc. [9].

We can distinguish two categories of embedded systems: autonomous systems and embedded systems.

An autonomous system: corresponds to an autonomous device containing intelligence that allows it to interact directly with the environment in which it is placed. These include

mobile phones, electronic personal diaries or GPS.

An embedded system: (often invisible to the user) is a coherent set of computer components (hardware and software), a device that gives it the ability to fulfill a set of specific missions. It is an underlying physical system with which the software interacts and controls [10].

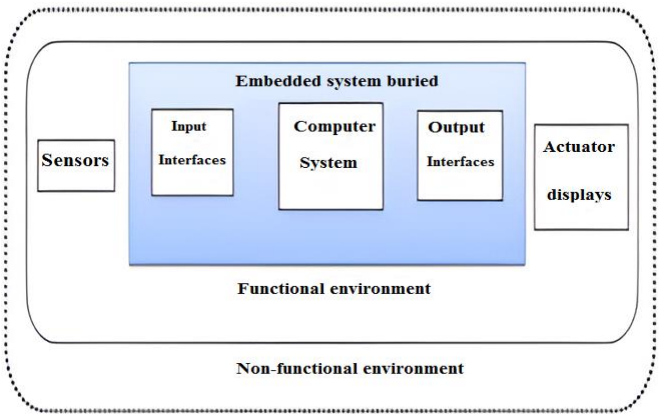

Figure 1. Embedded system [11]

As shown in Figure 1, an embedded system is built around a computer system that receives information from sensors and interacts with the environment using actuators and/or displays. The sensors measure the physical quantities characteristic of the environment in order to determine its current state. This information is converted and processed by the computer to produce a result based on the state of the environment. This result is converted and transmitted to the actuators to bring the environment into the expected state. The environment of an embedded system is composed of two parts:

The functional environment: it refers to the environment that is in direct interaction with the embedded system. This can be another system, an industrial process to be controlled or an individual.

The non-functional environment: it refers to the environment outside the embedded system that is not controlled by it. This environment will impose constraints and usage parameters (for example temperature or humidity level) which must be taken into account when designing the system [11].

3.1 The characteristics of embedded system

The main characteristics of embedded system are:

Real time: these systems are subject to time constraints. They must interact with their environment at a speed that is imposed by the latter. This therefore induces response time requirements. An embedded system is generally a real-time system.

Critical safety: A system failure can lead to a human, ecological or financial disaster. Embedded systems are often critical. Indeed, as such a system acts on a physical environment, the actions it performs are irremediable [12].

Limited resources: embedded software has limited resources, whether for reasons of weight, volume, or energy consumption.

Autonomy: autonomy is necessary when human intervention is impossible, but also when human reaction is too slow or insufficiently reliable. Embedded systems must generally fulfill their mission for long periods without human intervention.

Security and reliability: security in the sense of resistance to malicious acts, and reliability in the sense of continuity of service, is often linked to the issue of embedded systems.

Indeed, the criticality of embedded systems requires guaranteeing an appropriate level of reliability and security. Safety studies must be conducted throughout the system development cycle. These studies allow for better control of risks and reliability. Weak points are thus highlighted and allow designers to specify reconfiguration strategies before the real prototype phase and real tests. Safety studies must be conducted as early as possible in the design phase, in order to reduce costs and the number of prototypes required for system validation [12].

3.2 Embedded system architecture

Centralized architecture: It is composed of a calculator that can be single-processor or multiprocessor with shared memory and a set of sensors and actuators, all connected to the calculator. This type of architecture leads to “star” type wiring that is often significant and costly [13].

Multi-calculator architecture: This architecture is composed of a set of calculators that are not connected to each other. The application is fragmented and each computer implements a set of functionalities. Here, the main interest is to allow a rapprochement between the calculators and the transducers. The reduction in wiring thus obtained not only limits costs, but also increases the reliability of the system: if a processor fails, not all the functionalities are out of service, the shorter cables are less sensitive to external electromagnetic disturbances. The design of this type of system is simplified because it comes down to creating several simpler real-time systems. The main disadvantage of this architecture is not being able to guarantee overall consistency in the behavior of all of these systems that do not communicate with each other [13].

Weakly distributed architecture: The most commonly used bus is the CAN bus, which has become a standard in automotive applications. Thanks to these new buses, architectures that we call “weakly distributed” have appeared in embedded real-time applications. They are made up of a set of computers connected to each other by a bus. Compared to the multi-computer approach, the gain is undeniable: it is possible to control the overall behavior of all the computers and it is possible to share sensors between the computers [14].

3.3 Real-time embedded systems and their classification

Hard real-time: Strictly constrained real-time systems have deterministic behavior. In this case, all constraints must be strictly respected. Indeed, systems with hard temporal constraints only tolerate strict time management in order to maintain the integrity of the service provided. Failure to respect the constraints can cause catastrophic consequences [15]. This type of system is common in applications affecting public safety. Examples include nuclear station control systems, railway control systems, and computer-assisted medicine [16].

Soft real time: This class of system is less demanding with regard to compliance with all temporal constraints. This means that non-compliance with temporal constraints is tolerated by the system without this having catastrophic consequences. Systems with flexible or soft temporal constraints accept variations in data processing, we then speak of Quality of

Service. This means that these are systems where quality is appreciated by the human senses in the form of a service and that a low probability of not respecting temporal limits can be tolerated. This is the case for multimedia systems and applications (telephony, video, etc.) [16].

Mixed-constraint systems: These are systems composed of two types of tasks (strictly constrained tasks and softly constrained tasks). Consequently, a subset of tasks emerges that must imperatively respect temporal constraints and another subset of tasks whose evaluation criterion is the minimization of temporal errors [16].

3.4 Classification of real-time scheduling algorithms

Scheduling algorithms have the mission of finding, at any given time during the execution of the system, a hardware component for the highest priority software component. The way to perform this assignment allows these algorithms to be classified into:

Offline and online algorithms: A scheduling algorithm is offline if it constructs the complete scheduling sequence of all tasks before execution. This is well suited to periodic task systems. These algorithms allow the design of predictive systems, since the temporal constraints can be verified and validated even before the system is put into operation. A scheduling algorithm is online if it constructs the scheduling sequence of all tasks during the execution of the application. Online algorithms are more robust with respect to Worst Case Execution Time WCET overruns. This is well suited for sporadic and aperiodic task systems [17].

Exact and approximate algorithms: Offline and online algorithms that always find an optimal solution for the real-time scheduling problem, of course if this solution exists, are part of the class of exact algorithms. However, in the general case this problem is NP-hard and of exponential complexity, and to solve it in polynomial time, we adopt heuristic algorithms that seek solutions that are as close as possible to the optimal solution [17].

Single-processor/multi-processor: The scheduling is of the single-processor type if all the tasks can only be executed on a single processor. If several processors are available in the system, the scheduling is multi-processor [18].

This subsection briefly explains the principles of the different scheduling algorithms on a single-processor and multiprocessor architecture.

Rate-monotonic (RM) scheduling algorithm: The RM algorithm is a static period-based algorithm, which assigns the highest priority to the task with the smallest period. The use of periodicity as a scheduling criterion limits the applicability of this algorithm to periodic tasks with on-demand deadlines. Using this algorithm for other types of tasks does not provide any guarantee of meeting deadlines [19].

"Inverse Deadline" (ID) or "Deadline Monotonic" (DM): The "inverse deadline" algorithm is a static algorithm where the highest priority task is the one with the smallest deadline. Note that, compared to the Rate-monotonic algorithm, being based on the notion of deadline, this algorithm applies as well to other task models as those of tasks with deadlines on requests [19].

"Least Laxity First" (LLF): At time t and when the task is executed alone, the laxity of a task represents its maximum delay compared to its deadline to (re)start its execution. The "least laxity first" algorithm assigns, at time t, the highest priority to the task with the lowest laxity.

"Earliest Deadline First" (EDF): The EDF algorithm assigns, at time t, the highest priority to the task with the closest deadline. EDF* is used for where among the tasks with the same deadline, the one that arrives first will be elected.

An embedded system must be fault tolerant, in a way that it must be able to continue its operation despite the failure of a part of its software or hardware, and this is to justify the reliability of this system. The slightest failure of a critical embedded system can cause catastrophic consequences, so even in the presence of faults the embedded system must be delivered the service correctly to avoid the consequences.

4. FAULT TOLERANCE

In embedded and distributed real-time systems, compliance with time constraints throughout the life of the system is crucial. Given the potentially catastrophic consequences (loss of human lives, time, or money) that a fault could cause in a critical real-time system, incorporating techniques to ensure operational safety is essential when designing these systems. Fault tolerance is a method used to ensure the dependability of embedded real-time systems; it allows systems to provide the expected service even in the presence of faults. Critical real-time systems must therefore cover an important property of safe operating systems, which is reliability.

4.1 Operational safety

The dependability represents the ability of a system to deliver a service (its behavior as perceived by its user(s)) in which one can have justified confidence. A user is another system (human or physical) that interacts with the system considered [20].

4.2 The dependability tree

Dependability mainly manipulates three concepts: Attributes by which dependability is assessed, Hindrances by which dependability is affected, Means by which dependability is improved. Figure 2 summarizes the main notions of dependability [17].

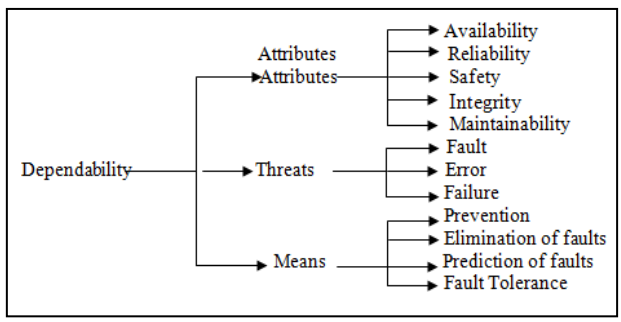

Figure 2. Embedded dependability tree [17]

To prevent system failures despite the presence of faults, this is equivalent to breaking the chain that leads from fault to failure. Fault tolerance is implemented by error detection and system recovery [21].

4.3 Principles of fault tolerance

The objectives are the subsequent identification of faults

with a view to their elimination or prevention, to avoid the propagation of the error to other components, to prevent the occurrence of a failure caused by the error. The parameters of the detection are the latency and the coverage rate. The coverage rate is the percentage of errors detected. The latency is the delay between the production and the detection of the error. Error detection is said to be concomitant when it is carried out during the normal execution of the service. Conversely, it is said to be preventive when it can be carried out during a suspension of the service. "Efficiency" means that failures are detected quickly and with acceptable accuracy. Concurrent detection techniques use redundancy at the information or component level, or temporal or algorithmic redundancy. The most commonly used forms are as follows.

Doubling and comparison: processing units are duplicated and their results are compared.

Error detection codes: they introduce redundancy into the representation of information. A watchdog timer (WDT) is a counter that counts down from a predefined value to zero at a fixed rate. WDT detects processor failures that can occur for various reasons. The idea is that tasks should periodically reset the WDT before it expires, by writing a specific value to a register or calling a function. In this way, the WDT acts as a watchdog that checks if the processor is working as expected.

Plausibility or structured data checks: assertions are inserted into the code to verify types, indices, values, etc.

Temporal and execution checks: a "watchdog" monitors response times or execution progress [13].

Compensation: requires that the system state has sufficient redundancy to allow its transformation into an error-free state. It is transparent to the application because it does not require re-executing part of the application (restart), nor executing a dedicated procedure (continuation). It can for example be achieved by replicating components and performing a majority vote on the results. Another way to proceed is to use error-correcting codes or more generally fault-tolerant algorithms. It can be noted that the compensation method does not require specific error detection since it performs the error detection itself. A compensation method can serve as an error detector, while the reverse is not true. Indeed, compensation requires greater redundancy to be able to correct the error. For example, in terms of components, two components are sufficient to detect an error, but at least three will be necessary to correct it [21].

4.4 Fault tolerance techniques

Fault tolerance methods are based on two classes of techniques:

-Treating faults.

-Treating errors.

Fault treatment: In this case, the fault tolerance algorithm aims to prevent faults from being activated. It involves at least two steps which are fault diagnosis and fault inactivation

Fault diagnosis: determines the causes of the error in terms of location and nature.

Fault inactivation: prevents faults from being activated again (by making them passive) [9].

Error treatment: In this case, the fault tolerance algorithm consists of detecting the existence of an incorrect state (error), then replacing the incorrect state with a correct state that complies with the specifications. In all cases, redundancy is the sole principle used to treat errors, there are three forms of redundancy.

Hardware redundancy: includes hardware components added to the system to support fault tolerance (e.g., using an available processor if one of the executing processors fails).

Software (or information) redundancy: includes all programs and instructions that are used to support fault tolerance (e.g., using two implementations of the same module).

Time redundancy: consists of allowing additional time to complete the execution of tasks to support fault tolerance (e.g., executing a module again later) [9].

The main objective of fault-tolerant scheduling algorithms (EDF fault-tolerant) is to study processor allocation strategies in the presence of faults, to propose new improvement methods for scheduling and to choose one that significantly decreases the execution time of the algorithm without degrading the system performance.

5. RESULTS AND DISCUSSIONS

In the domain of real-time systems, where timeliness and predictability are paramount, scheduling algorithms play a crucial role in ensuring efficient and reliable task execution. An algorithm, Earliest Deadline First* (EDF*), has been proposed that is distinguished by its simplicity and efficiency in prioritizing tasks based on their impending deadlines. This chapter embarks on a practical journey to realize EDF*, by implementing a multiprocessor EDF* scheduler using the Python programming language.

Our approach delves into the intricacies of scheduling tasks across multiple processors, unraveling the challenges and opportunities that arise in this dynamic environment. We will address the complexities of managing a constantly changing set of tasks, synchronizing execution across multiple processors, and protecting the system from timing faults. By addressing these challenges, we will develop robust strategies to ensure scheduler resilience. Redundancy and migration techniques will be employed to protect critical tasks from processor failures, while scheduling mechanisms will dynamically adapt to changing system conditions.

To evaluate the effectiveness of our approach, we conclude this paper by simulating a case study with the results obtained in detail, we first present an error detection technique which is the Watch dog-timer.

Task Representation: Tasks in a scheduling system are defined by several parameters that determine their behavior and importance in the scheduling process. Here is an explanation of these parameters.

Task ID: A unique identifier assigned to each task to distinguish it from other tasks.

Arrival Date (DA): the task arrival date (creation date or possibly the date a task transferred by another processor is received). It is now possible to schedule this task.

Execution Time (TE): This is the time required for the execution of the task. It is determined by simulations or by a thorough study of the source code before execution.

Deadline (DI): it represents the instant at which the execution of a task must be completed to respect the time constraint.

Period (PI): it represents the time between two consecutive creations of a task.

Figure 3 shows that the proposed approach expresses the different steps to follow to solve the problem of fault tolerance for real-time embedded systems. We have chosen the

independent periodic tasks. For each arrival of a task in the system and before inserting it into the queue, we must calculate the feasibility to ensure the existence of a real-time schedule for all of these tasks.

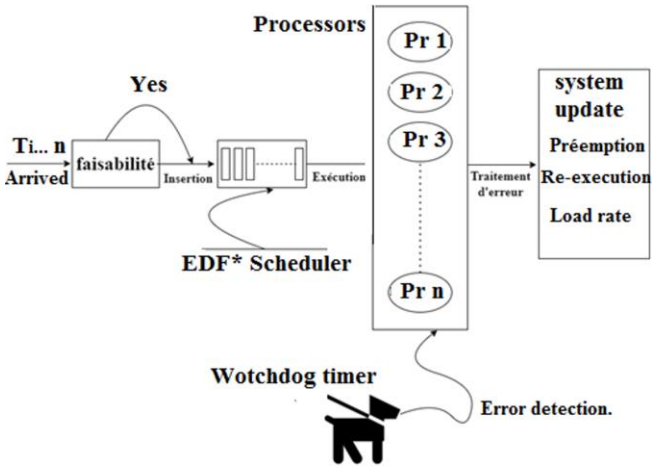

Figure 3. The proposed approach

According to reference [22], the feasibility test to execute the tasks is composed of two conditions:

Necessary feasibility (Feasibility used in practice): is calculated by summing the ratios between the execution time of each task and its period, then dividing this sum by the number of available processors. This gives a measure of resource utilization relative to the periodicity constraints of the tasks.

$$Necessary\ Feasibility = \frac{\sum_{task} \frac{Task\ execution\ time}{Task\ period}}{Number\ of\ processors} \quad (1)$$

Sufficient feasibility: is calculated by summing the ratios between the execution time of each task and its deadline, then dividing this sum by the number of available processors. This gives a measure of the leeway available to each task relative to its time constraints.

$$Sufficient\ Feasibility = \frac{\sum_{task} \frac{Task\ execution\ time}{task\ dedline}}{Number\ of\ processors} \quad (2)$$

Once the feasibility conditions of the tasks have been verified, they are placed in a queue and then ordered using the EDF* algorithm.

Algorithm: EDF scheduling algorithm

Input: List of tasks tasks, number of processors num_processors, simulation period simulation_period, optional processor stops.
Output: Timeline for each processor
1. Initialize task_list with metadata for each task
2. Initialize processor_timelines for each processor
3. for t in range(simulation_period) do
4. for each proc_id in processor_stops do
5. t≥processor_stops[proc_id] processor_timelines [proc_id][t] ← −1
6. available_tasks ← {task | task.capacity > 0}
7. Sort available_tasks by deadline
8. for each proc_id in range(num_processors) do
9. processor_timelines[proc_id][t] = −1 continue
10. current_task ← processor_timelines[proc_id][t − 1]
11. current_task≠ −1 processor_timelines[proc_id][t] ← current_task
12. current_task.capacite ← current_task.capacite − 1
13. for each proc_id in range(num_processors) do
14. processor_timelines[proc_id][t] = −1 and available_tasks≠ Ø
task ← available_tasks.pop(0)
15. processor_timelines[proc_id][t] ← task.id
16. task.capacity ← task.capacity − 1
17. for each task task in task_list do
18. task.capacite = 0 task.capacite ← task.initial_capacite
19. task.deadline ← task.deadline + task.period
20. return processor_timelines.

Calculating the LCM of periodic tasks: Calculating the LCM (Least Common Multiple) of periodic tasks is essential to ensure synchronized and efficient execution in various systems, such as real-time systems, communication networks, and industrial control systems. By determining the LCM, it is ensured that all tasks execute at regular intervals, avoiding conflicts and of each task: The period of a task represents the time required to complete a complete execution cycle.

List the periods of all tasks: Write down the periods of all the periodic tasks that you want to consider.

Determine the prime factorization of each period: Decompose each period into its prime factors.

Identify the highest power of each prime factor: For each prime factor present in one of the periods, identify the highest power of that prime factor appearing in all periods.

Multiply the identified prime factors: Multiply the identified prime factors, raised to their respective highest powers, to obtain the lcm.

Process faults: There are several types of system faults (code faults, network faults, communication faults, etc.) we cannot treat them all at once. The fault that we will consider here is the failure of a processor. This can be due to a processor delay or a complete failure. If a task does not complete before the watchdog timer expires, this may indicate a fault in the processor on which it was running

Fault tolerance mechanism with watchdog timer: WatchDog: or the watchdog, is an integrated circuit used to ensure that the system does not get stuck at a particular stage in the processing it performs. It is a protection generally intended to restart the system in the event that a defined action is not executed within a given time. The i.MXL provides a WatchDog with a granularity of 0.5 seconds, the allowed interval for a test period is between 0.5 seconds up to 64 seconds [14]. In the context of fault tolerance for our EDF* scheduling system, we will implement a watchdog timer mechanism to detect potential faults during the execution of tasks. Then, we will consider a specific fault, namely a processor failure, which can be caused by a processor delay or a complete failure. Here is how the process could be implemented:

Error detection: In a strict real-time system that uses the EDF* scheduling algorithm: To manage tasks according to their earliest deadlines. The watchdog timer is particularly useful to ensure that tasks are executed within the given deadlines and that no timeout occurs.

System recovery: When an error is detected, the watchdog timer triggers a corrective section such as a system reset or taking action to correct the error. In a multiprocessor system, a test task that has been monitored by WDT encounters an

error or has been executed late, it is possible to redirect this task to another free processor so that it can be executed correctly from its deadline, so the tasks must be distributed among the available processors.

In case of fault detection, we will consider two solutions:

Task Immigration: The task currently running will be migrated to another available and functional processor.

Preemption: If migration is not possible or if it is not desirable, another solution is to preempt a task with a deadline as far away as possible to free the processor for the pending task.

Scheduling after fault detection: Once the tasks are in the global queue, we will use a star classification based approach to decide the execution. This involves calculating the load rate of each processor. The processor with the lowest load rate will be chosen to execute the task.

6. SIMULATION

In this section, we propose a case study to evaluate the simulation and to show the effectiveness of the proposed approach. We first present the description of the proposed tasks. Then we present the execution results.

In our case study we assume that the system is composed of 08 periodic tasks (as shown in Table 1) and 03 processors; each task described with a 4-tuple: Ti = (DA, TE, Di, Pi).

Table 1. Tasks description

Tasks	Arrival Date	Execution Time	Deadline	Period
Task 1	0	2	7	15
Task 2	0	2	10	10
Task 3	0	1	6	5
Task 4	1	1	9	6
Task 5	3	1	12	15
Task 6	2	2	11	6
Task 7	3	1	8	10
Task 8	2	1	10	5

The approach begins in the first step to calculate the feasibility of task execution with the time constraints for each task; and to achieve this goal we must first calculate the overall period of all tasks Ptot = PPCM (15,10.5, 6, 15, 6,10.5) =30 At time t=0 the 03 tasks T1, T2, T3 have arrived in the system, If the execution of the tasks is feasible, we must insert the tasks into the queue and ordered by the proposed algorithm. Tasks T1, T2, T3 are waiting to use the available processors.

Task T3 used processor P1 from date t=0 with TE=1, Task T1 used processor P2 from date t=0 with TE=2, Task T2 used processor P3 from date t=0 with TE=2. In date t=1 it is the arrival of task T4 in the system, this event the system will check if there are free processors, in our system processor P1 is free so task T4 used processor P1 with TE=1. In date t=2, it is the arrival of tasks T6, T8, at this moment all processors are free. T8 used processor P1 and task T6 used processor P2Note: (DI(T8)=10 < DI(T6)=11) this is the justification for using processors in order T8 → P1 and T6 → P2. In the date t=3 it is the arrival of the tasks T7, T5, this event the two processors P1 and P3 are free at this time the task T5 uses the processor P1 with TE=1 and T7 uses the processor P3 with TE=1. t=5 it is the arrival of the task T3 for the period n°2, T3 uses the processor P3 (we choose the oldest processor used) with a TE=1 until t=6 (termination of the execution of the task T1), by the same principle T4 uses the processor P1 and T8 uses the processor P2.

From the date t=8 until the end of the timeline all the tasks have arrived to complete their execution according to their period, they will be ordered by the algorithm and each time chooses the oldest processor used and its lowest load rate The execution results of the tasks on the 3 processors are graphically represented in Figure 4. As the Figure 5 shows the feasibility and utilization rate of each processor.

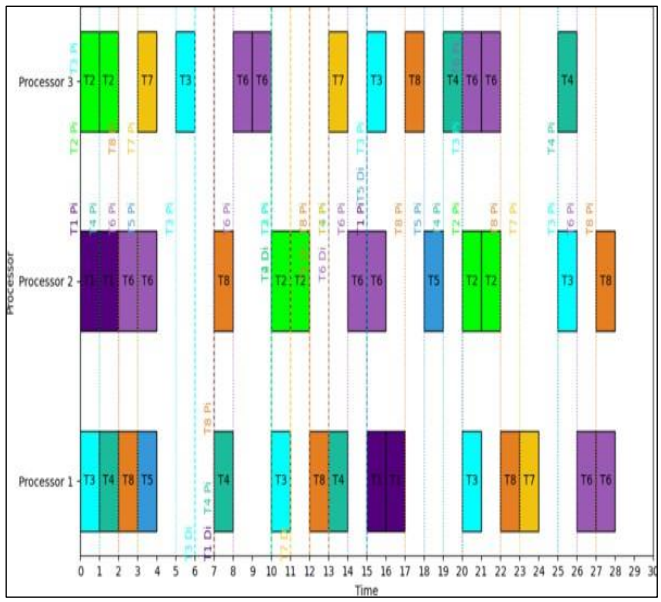

Figure 4. Tasks scheduling

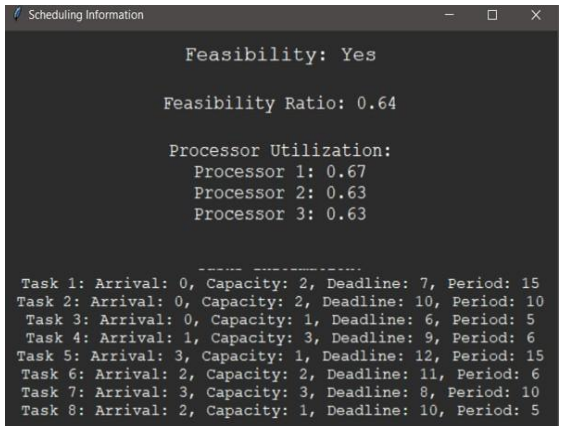

Figure 5. Feasibility results and load rate

7. CONCLUSION

This paper has contributed to the advancement of knowledge in the field of the dependability of distributed embedded systems (DES). The results obtained, both theoretically and practically, will allow DES designers and developers to implement safer and more reliable systems, thus meeting the increasing requirements of critical applications. With the results of our proposed approach, a significant improvement for the reliability of embedded systems by fault tolerance that allows to give justified confidence to the system despite the presence of processor faults. This makes the application of our proposed approach crucial in critical embedded systems and which can generate catastrophic results with the slightest system failure such as the e-marked systems

that exist in means of transport (airplane, vehicle, train...) and devices used in medicine. Our research opens the way to several interesting perspectives in the field of distributed embedded systems and fault tolerance. By further exploring the integration of fault tolerance mechanisms in DES, we could consider studying the impact of different scheduling strategies on the reliability and performance of the systems. In addition, analyzing the efficiency of the EDF algorithm in more complex scenarios or exploring new fault detection and recovery techniques could be promising research avenues to enhance the dependability of DES. Communication between tasks is very important in intelligent embedded systems, which opens the way for future research that uses dependent tasks.

REFERENCES

[1] Reghenzani, F., Guo, Z., Fornaciari, W. (2023). Software fault tolerance in real-time systems: Identifying the future research questions. ACM Computing Surveys, 55(14s): 306. https://doi.org/10.1145/3589950

[2] Kumar, A., Yadav, R.S., Ranvijay, A.J. (2011). Fault tolerance in real time distributed system. International Journal on Computer Science and Engineering, 3(2): 933-939.

[3] Manimaran, G., Murthy, C.S.R. (1998). A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis. IEEE Transactions on Parallel and Distributed Systems, 9(11): 1137-1152. https://doi.org/10.1109/71.735960

[4] Ramanathan, P., Shin, K.G. (1992). Delivery of time-critical messages using a multiple copy approach. ACM Transactions on Computer Systems, 10(2): 144-166. https://doi.org/10.1145/128899.128902

[5] Chevochot, P., Puaut, I. (1999). Scheduling fault-tolerant distributed hard real-time tasks independently of the replication strategies. In Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No.PR00306), Hong Kong, China, pp. 356-363. https://doi.org/10.1109/RTCSA.1999.811280

[6] Chkouri, M.Y. (2010). Modélisation des systèmes temps-réel embarqués en utilisant AADL pour la génération automatique d'applications formellement vérifiées. Doctoral dissertation, Université Joseph-Fourier-Grenoble I.

[7] Kamni, S. (2023). Un framework d'aide au déploiement et à la personnalisation des systèmes temps réel: Application aux autopilotes de drones. Doctoral dissertation, ISAE-ENSMA Ecole Nationale Supérieure de Mécanique et d'Aérotechique-Poitiers.

[8] Z. Mammeri. Introduction Aux Systèmes Embarqués Et Temps Réel, Cours – Module ASTRE IRIT - UPS – Toulouse, 2010. https://pdfcoffee.com/chapitre1-pdf-pdf-free.html.

[9] Bachir, M. (2011). Tolérance aux fautes des systèmes temps-réel embarqués basée sur la redondance. Doctoral dissertation, Université de Batna 2.

[10] Meliouh, A. (2021). UML et model-checking pour la modelisation et la verification des systemes embarques. Doctoral dissertation, Université de Mohamed Kheider Biskra.

[11] Solet, D. (2020). Systèmes embarqués temps réel fiables et adaptables. Doctoral dissertation, Universite DE Nantes.

[12] Ghenai, A. (2015). Évaluation de la fiabilité des systèmes embarqués dès la phase de conception par réseaux de Pétri temporels étendus. Doctoral dissertation, Thèse de Doctorat en Sciences de L'Université Constantine 2.

[13] Ridha, M. (2023). Tolérance aux fautes pour les systemes embarques distribues intelli- gents. Doctoral dissertation, Ecole National Supérieur D'informatique.

[14] Oubadi, S. (2014). Optimisation de l'ordonnancement/allocation dans les systèmes embarqués distribués temps réel. Doctoral dissertation, Oum-El-Bouaghi.

[15] Xu, J., Parnas, D.L. (1991). On satisfying timing constraints in hard-real-time systems. ACM SIGSOFT Software Engineering Notes, 16(5): 132-146. https://doi.org/10.1145/123041.123066

[16] BENDIB, S.S. (2019). Méthodologie de conception de systemes embarques temps reel. Doctoral dissertation, Université de Batna 2.

[17] Arar, C. (2016). Redondance logicielle pour la tolérance aux fautes des communications. Doctoral dissertation, Université de Batna 2.

[18] Bounabi, C., Boutekkouk, F. (2013). Optimisation des performances dans les systèmes embarqués distribués. e Master dissertation, Université Larbi Ben M'Hidi Oum El Bouaghi.

[19] Vîlcu, D.M.R. (2004). Systèmes temps réel embarqués: Ordonnancement optimal de tâches pour la consommation énergétique du processeur. Doctoral dissertation, Université Paris XII – Val de Marne.

[20] Laprie, J.C. (2004). Sûreté de fonctionnement des systèmes: Concepts de base et terminologie: Sûreté de fonctionnement. REE. Revue de L'électricité et de L'électronique, (11): 95-105. https://doi.org/10.3845/ree.2004.109

[21] Besseron, X. (2010). Tolérance aux fautes et reconfiguration dynamique pour les applications distribuées à grande échelle. Doctoral dissertation, Institut National Polytechnique de Grenoble-INPG.

[22] Mehalaine, R., Boutekkouk, F. (2020). Energy consumtion reduction in real time mult, processor embedded systems with uncertain data. In Artificial Intelligence and Bioinspired Computational Methods, Springer, Cham. https://doi.org/10.1007/978-3-030-51971-1_4