Scheduling Replica Voting in Fixed-Priority Real-Time Systems

Pietro Fara
Scuola Superiore Sant'Anna, Pisa, Italy

Gabriele Serra
Scuola Superiore Sant'Anna, Pisa, Italy

Alessandro Biondi
Scuola Superiore Sant'Anna, Pisa,Italy

Ciro Donnarumma
Rete Ferroviaria Italiana S.P.A., Rome, Italy
Scuola Superiore Sant'Anna, Pisa, Italy

Abstract

Reliability and safety are mandatory requirements for safety-critical embedded systems. The design of a fault-tolerant system is required in many fields (e.g., railway, automotive, avionics) and redundancy helps in achieving this goal. Redundant systems typically leverage voting techniques applied to the outputs produced by tasks to detect and even tolerate failures.

This paper studies the integration of distributed voting protocols in fixed-priority real-time systems from a scheduling perspective. It analyzes two scheduling strategies for implementing voting. One is attractive and friendly for software developers and based on suspending the task execution until the replica provides the data to be voted. The other one is inspired by the Logical Execution Time (LET) paradigm and requires introducing additional tasks in the system to accomplish voting-related activities. Queuing and delays introduced by inter-replica communication interfaces are also analyzed.

Experimental results are finally presented to compare the two strategies, showing that LET-inspired voting is much more predictable and hence more suitable than the other strategy for fixed-priority real-time systems.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks; Computer systems organization → Real-time systems

Keywords and phrases Real-time systems, safety-critical systems, voting, redundancy, fault-tolerance, logical execution time

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2021.13

1 Introduction

Embedded computing systems have become more and more pervasive in our lives: they are used to fulfill evermore functions, a lot of which are related to the safety of people and the surrounding environment. Indeed, embedded systems are nowadays widely present in avionic, railway, automotive, and military applications in a way that their failures could lead to catastrophic consequences. As these systems are related to our safety, they are commonly called safety-critical embedded systems.

In most application domains there exist a lot of regulations to which a safety-critical system must comply [11, 12]. Such regulations mandate the use of certain techniques to improve the *reliability* and the *safety* of a system. These techniques can be mainly classified into two categories: *fault avoidance* (also known as fault intolerance) and *fault tolerance*. Fault avoidance techniques aim at drastically reducing by design the probability of failure. This approach is generally not viable for complex systems because, even by performing a

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 13; pp. 13:1–13:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

meticulous design, it could be impossible to eliminate all the internal sources of faults, so that the system will eventually experience a failure. On the other hand, *fault tolerance* techniques aim at making a system capable of properly react to faults, avoiding that they lead to a failure of the functionality offered by the system [31]. Redundancy is a widespread approach to build fault-tolerant systems. Redundant systems are built by several subsystems, called *replicas*, that perform the same computations over time. The replication allows the detection and/or the masking of a fault through the *voting* (i.e., comparison) of the results computed by all replicas. A redundant architecture is said to be *r-out-of-n* (with $r \leq n$) if it is built by n replicas, r of which have to properly work to make the whole system failure-free [30].

The *2-out-of-2* architecture is the most used redundant architecture in the railway domain. It is an architecture that enables the detection of faults: if the results provided by the two replicas are different, a fault is detected and the system goes into a fail-safe state (e.g., shutdown). As described by Shenghua and Li [29], in the railway domain, the *2-out-of-2* architecture is employed in a hierarchical framework where the entire system with two replicas is further replicated to build a *1-out-of-2* system of systems. In normal conditions, only the primary system provides the output to the external environment but, as soon as the voting in the primary system detects a fault, the primary is shut down and the secondary system (in *hot standby*) takes over. As a result, this architecture is capable of masking faults increasing system availability.

Another architecture used in many domains, such as avionics, is the *2-out-of-3* (also known as Triple Modular Redundancy) [17, 32]. It implements *majority-voting* and is known to be capable of providing both fault-detection and fault-masking [31]: if one replica is faulty (i.e., its output is different by the other two), a fault will be detected and the system output will continue to rely on the outputs of the other two replicas.

Even though several other redundancy schemes have been proposed, most of them are based on the same idea: replicated systems perform the same computations on the same inputs, sending through a communication network their results to be voted. Voting can be either centralized or distributed. In the former case, voting is implemented on a centralized node that collects and votes all the results provided by the replicated subsystems. In this case, the voter itself is clearly a single-point-of-failure. In the latter case, each replica has its voter, either implemented with a hardware component or with a software algorithm, and votes its data against the one produced by the other replicas.

In this work, we focus on distributed voting implemented with software techniques, which is a more and more widespread approach (e.g., in the railway domain) to achieve flexibility and contain cost in realizing fault-tolerant systems.

The implementation of distributed voting requires dealing with the transmission of data among replicas, the waiting and synchronization among replicas, and the execution of the voting protocol itself. These aspects clearly impact on the timing properties of real-time tasks and call for the investigation of different strategies to suitably schedule all voting-related activities.

1.1 This work

Informed by experience in safety-critical software for the railway industry, in this work we analyze and compare two different strategies for scheduling voting-related activities under 2-out-of-2 redundancy. The first one corresponds to a case that is particularly attractive and friendly for software developers: data is transmitted among replicas whenever they are produced by the tasks and each task waits for the reception of the data sent by the other replica by suspending its execution (e.g., by using a classical condition variable). When tasks are resumed, the data to be voted is available and they can proceed with the execution of the voting protocol and then complete it. The second one is a new approach proposed in this work inspired by the Logical Execution Time (LET) [19, 28] paradigm where voting is delayed at the end of the tasks' periods and delegated to dedicated tasks. Although the first approach may be preferable by software developers, this work shows that it introduces several sources of unpredictability that make it particularly challenging to be analyzed from a worst-case perspective.

In summary, this work makes the following contributions:

- It provides a response-time analysis for real-time tasks under two strategies for scheduling voting-related activities, one of the two being novel and proposed in this work.
- It provides an analysis of queuing effects and worst-case transmission delays introduced during inter-replica communications.
- It compares the two strategies by means of an experimental evaluation.

To the best of our records, this is the first work that analyzes in detail the timing properties of distributed voting protocols implemented upon a fixed-priority real-time system with periodic multitasking. Software engineers from the railway industry collaborated in this work. Since this work only addresses how voting operations are scheduled, other aspects such as fault detection and recovery strategies are not discussed as they depend on the target application and the adopted voting protocol.

Paper structure. The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 presents the system model by considering both tasks and inter-replica communication. Section 4 formalizes the behavior of the two voting strategies. Section 5 analyzes queuing effects and delays in inter-replica communications. Section 6 provides response-time analysis under the two voting strategies. Section 7 presents the experimental results and Section 8 concludes the paper.

2 Related Work

Several works in the literature studied fault-tolerant systems from both a hardware and software perspective.

Davies et al. [14] proposed a hardware-level solution, called *Synchronization Voting*, for achieving inter-replica synchronization in a redundant system, overcoming the need for a common external clock, which is a source of common-mode failures. Their approach consists of using a set of synchronizer modules (one for each replica) that, by exchanging mutual feedback, allow replicas to correct for their inevitable drift. McConnel et al. [27] continued this work by presenting voter designs for different signaling conventions (transition, level, and pulse). These papers present elegant solutions to implement inter-replica synchronization and voting at the hardware-level, but they do not consider the effects of multitasking on the replicated systems, where the voting has to be implemented on the outputs produced by tasks. Eris et al. [16] focused on railway systems (with 2-out-of-2 redundancy) with diverse programming. Their approach allows the voter to move the system from a safe state toward a less safe state only when all replicas agree. They also analyzed the effects of the synchronization issues (i.e., race conditions) on the railway signaling protocols by proposing a solution based on a centralized voter acting as a replica coordinator. Again, multitasking has not been considered.

Some real-time scheduling strategies, aimed at improving the system resilience against transient faults, have been proposed by Kim and Shin [21], and Kwak and Kim [24]. They are based on executing different copies of the same task at different times so that the probability

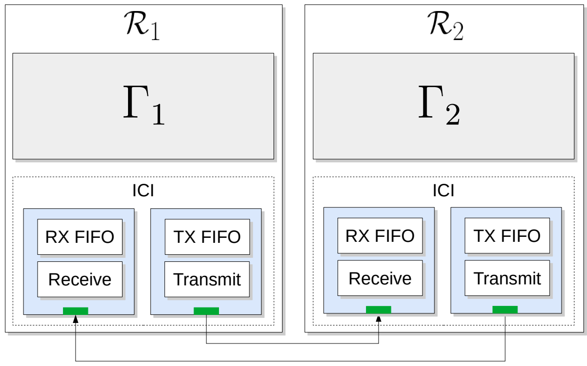

Figure 1 An overview of the system architecture.

that a common transient fault affects all of them is reduced. Back et al. [1] proposed TL-NMR, a task-level *N Modular Redundancy* schema, which allows the execution of several copies of tasks in parallel upon multiprocessor platforms scheduled by Global Fixed-Priority. The authors provided an algorithm that allows selecting the number of copies for each task along with a schedulability test based on the response-time analysis. However, these papers, focused only on the schedulability of the tasks' copies, without considering issues related to inter-replica synchronization and the impact on the scheduling of voting protocols. Another work that improves the fault-tolerance in the presence of environmentally-induced faults is due to Gujarati et al. [18]. The authors proposed an algorithm, along with a suspension-free model of its real-time implementation (based on the Liu and Layland task model), that allows a distributed real-time system to solve the *Interactive Consistency* problem in the presence of Byzantine faults. The authors also provided a detailed real-time-aware reliability analysis of the proposed solution.

Bernat et al. [4, 5] presented a real-time fault-tolerant architecture capable of handling transient overload conditions through the firm real-time task model. The proposed architecture comprises multiple replicated subsystems, each executing a copy of the same task set, and a dedicated processor for the voting called *Redundancy eXecutive* (RX). Whenever a task finishes its execution, it sends the computed results to the RX and suspends its execution. As soon as the RX has collected enough replicas' results (some replicas could be failed), executes the voting protocol and sends the voted output back to the tasks' copies, allowing them to resume their computation. Similar to our work, the authors also provided a detailed schedulability analysis based on the response-time analysis. They consider every contribution to the tasks' execution time, such as the communication time spent into the results exchanging and executing the voting algorithm on the RX subsystem. These works have several limitations. First, they consider one scheduling scheme for voting only. Second, they rely on a dedicated subsystem to execute the voting algorithm, introducing a higher system cost and requiring to deal with the RX subsystem's potential faults. Third, they do not provide any experimental evaluation.

From the perspective of voting protocols, researchers consolidated several algorithms such as the ones presented in [2, 3, 7, 8, 14, 25, 33].

3 System model

This work considers a 2-out-of-2 redundant system with two replicas $\mathcal{R}_1$ and $\mathcal{R}_2$. Each replica $\mathcal{R}_k$ consists in a uni-processor platform that executes a set $\Gamma_k = \{\tau_1^k, \ldots, \tau_n^k\}$ of n periodic tasks. Each periodic task τ_i^k is characterized by a worst-case execution time (WCET) C_i^k, a release period T_i^k, and a relative deadline $D_i^k \leq T_i^k$. Tasks are scheduled according to fixed-priority preemptive scheduling. The set of higher-priority tasks with respect to τ_i^k that execute on the same replica $\mathcal{R}_k$ is denoted by $hp(i,k)$.

Each periodic task τ_i^1 running in the primary is associated with a corresponding periodic task τ_i^2 running in the secondary and the two tasks form a *replica pair* $r_i = \{\tau_i^1, \tau_i^2\}$. The tasks in a replica pair share the same period and deadline, i.e., $T_i^1 = T_i^2$ and $D_i^1 = D_i^2, \forall i = 1, \ldots, n$. Given a replica $\mathcal{R}_k$, the other replica is referred to as $\mathcal{R}_{or(k)}$, where $or(k) = (k+1) \bmod 2$. The clocks of the two replicas are *synchronized* so that the release of the periodic tasks in each replica pair is synchronized. The WCET of the tasks can be different from replica to replica, i.e., C_i^1 can be larger or shorter than C_i^2 for some pairs of tasks τ_i^1 and τ_i^2.

Inter-replica communication and voting. The two replicas are connected via two wired *inter-replica communication interfaces* (ICI): one for sending the data from $\mathcal{R}_1$ to $\mathcal{R}_2$, and one for sending data from $\mathcal{R}_2$ to $\mathcal{R}_1$. An overview of the system architecture is shown in Figure 1. For instance, the ICI can be realized with serial peripheral interfaces (SPI) for transmitting data and digital lines connected to general-purpose input/output (GPIO) for the synchronization signals.

Data transmission via the ICI occurs by acting on memory-mapped device registers. The ICI provides an output (resp., input) buffer organized as a first-in-first-out (FIFO) queue of Q elements, each of size b bytes. The ICI also provides synchronization signals to notify events among replicas (e.g., the completion of a computation). The minimum read/write rate in accessing such registers is denoted by β (in bytes per time unit), while the maximum one is denoted by $\overline{\beta}$. The minimum transmission rate guaranteed by the ICI is denoted by α (in bytes per time unit). The minimum read/write rate to access memory is γ[1]. For instance, this means that a task that intends to send x bytes via one of the ICI spends **(i)** at most x/γ time units to read the data to be sent from memory, **(ii)** at least $x/\overline{\beta}$ time units and at most x/β time units of its computation time to fill the ICI queue with data, and **(iii)** that such data will be transmitted to the other replica in at most x/α time units.

Periodic tasks may produce *vital outputs*, i.e., data that are critical for the system. Both the tasks τ_i^1 and τ_i^2 of each replica pair produce the same set of vital outputs. Before the completion of each job, each periodic task has to *vote* its vital outputs (if any) with the corresponding task of its replica pair. Voting is implemented via a distributed voting protocol [26] that exchanges data via the ICI.

Both the tasks in a replica pair r_i exchange M_i data packets with a fixed size of b bytes that contain the data to be voted.

Tasks that intend to send data via an ICI that has its queue full, busy-wait until at least one slot in the queue becomes empty. In reception, the ICI can either operate in *polling* mode or in *interrupt* mode. In the former case, tasks receive packets by actively sampling the ICI queue, possibly wasting processor cycles if the queue is empty. In the latter case,

[1] The authors acknowledge that memory write times are generally shorter than read times. A common rate γ has been considered just for the sake of simplicity as it does not particularly affect the results of this paper.

the ICI notify receipt of packets through interrupts. The ICI are programmed to raise one interrupt every time a packet is received. The corresponding *interrupt service routine* (ISR) is in charge of reading the packets in the queue, by acting on the ICI device registers, and copying them into a memory buffer shared with the task interested by the packet (interrupts that are raised while the ISR is pending are ignored and the corresponding packets are processed by the same). Each ISR introduces an overhead of at most σ^{ISR} time units due to the management of the ISR activation and completion (i.e., this overhead does not include the time required to process packets).

The time that τ_i^k spends to perform computations lasts at most E_i^k time units. This parameter does not account for packet transmissions and receptions and the execution of the voting protocol. The transmissions performed by each job of the tasks consist of copying the vital outputs from memory into the transmission registers of the ICI. For tasks of the replica pair r_i such transmissions take at most $VT_i = M_i \cdot PT$ time units, where $PT = \left(\frac{b}{\gamma} + \frac{b}{\beta}\right)$. The maximum time needed to receive the packets of the tasks of r_i and store them in a shared-memory buffer, to be later consumed by the voting protocol, is denoted by $VR_i = M_i \cdot PT$.

The utilization of the ICI, intended as the amount of bytes transmitted per time unit in the long run, is defined as $U^{\text{ICI}} = \sum_{i=1}^{n}(M_i b)/T_i$. To avoid dealing with cases in which the ICI is overutilized, which clearly makes the system not feasible, we require $\alpha > U^{\text{ICI}}$ and $\beta > U^{\text{ICI}}$.

After the two tasks in a replica pair r_i exchanged the data to be voted, a voting protocol can be executed, which takes at most VP_i time units.

The data transmission is performed by using one of the ICI in a mutually-exclusive manner. To this end, each task may have to acquire and release a lock before and after transmitting each packet, respectively. The immediate priority ceiling (IPC) locking protocol is adopted. The case in which all tasks have to vote data is equivalent to a resource shared by all tasks: hence, under the IPC protocol, the critical sections to access the communication interface are equivalent to non-preemptive sections.

4 Voting implementations

This work is focused on analyzing and comparing two schemes to schedule the execution of the voting protocol and the related data transmissions.

The first one, named *passive waiting*, is an approach that can be implemented with a minimal impact on general-purpose programming paradigms, as it corresponds to the case in which a task sequentially performs the following three operations: (i) compute, (ii) wait for the other replica to complete by self-suspending its execution, and (iii) execute the voting protocol. Note that passive waiting can be implemented with classical semaphores and condition variables. The second one is inspired by the Logical Execution Time (LET) paradigm and requires introducing additional tasks in the system.

The following rules characterize the behavior of each of the considered scheduling schemes:

- *Transmission Rule*: it defines how data transmission is performed among replicas.
- *Reception Rule*: it defines the behavior of the replica that receives the data.
- *Waiting Rule*: it defines how a task τ_i^k has to wait for the corresponding task $\tau_i^{or(k)}$ in the other replica.
- *Voting Rule*: it defines how the voting protocol is executed.

4.1 Passive waiting

Under passive waiting tasks are composed of three serialized phases: **(i)** an execution phase (E), in which the task computes the data to be voted; **(ii)** a transmission phase (VT) where vital outputs are transmitted to the other replica; and **(iii)** a final phase where the voting protocol (VP) is executed. The reception of packets is handled by ISRs (the ICI are used in interrupt mode).

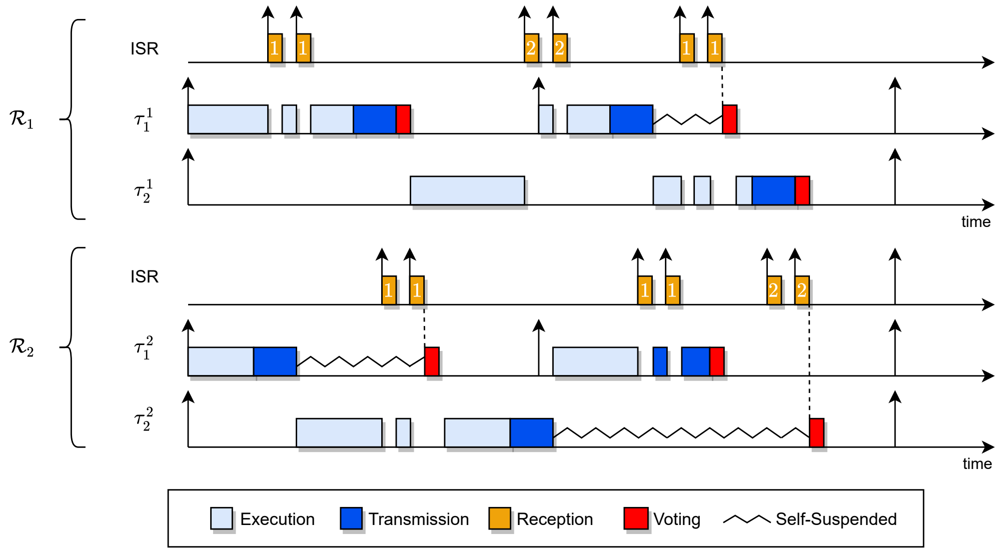

Figure 2 Example schedule of two replica-pairs under passive waiting.

Transmission rule. When completing its computations, each task τ_i^k transmits M_i packets of data to be voted on by the other replica. For each packet to transmit, the task first acquires the lock on the communication interface, then transmits the packet, and finally releases the lock.

Reception rule. Whenever a replica $\mathcal{R}_k$ receives a data packet, an ISR is executed by preempting any task in execution in $\mathcal{R}_k$, i.e., the ISRs run at the highest priority level and are not affected by the locking of the communication interface as two independent ICI are used for transmission and reception. The ISRs perform the operations specified in Section 3. For each task τ_i^k, when the last of the M_i packets sent by $\tau_i^{or(k)}$ (i.e., from the other replica $\mathcal{R}_{or(k)}$) is received, the ISR that handles the packet notifies τ_i^k that all its data is ready in a shared-memory buffer to be voted on.

Waiting rule. When a task τ_i^k completes its transmission phase it *self-suspends* its execution until all the M_i packets sent by the other replica task $\tau_i^{or(k)}$ are received and processed by the corresponding ISRs. The self-suspension is skipped if all M_i packets have already been received and processed by ISRs.

Voting rule. The voting protocol is executed after all the M_i packets have been received and processed by ISRs, i.e., after the eventual self-suspension enforced by the voting rule. The task terminates after the execution of the voting protocol.

An example schedule under passive waiting is illustrated in Figure 2. In this example, two replica-pairs are needing to vote two packets each. The first job of τ_1^1, according to the waiting rule, does not experience any suspension as it already received all the packets from the other replica when it becomes ready to vote. On the other hand, the first job of τ_1^2 completes its execution and transmission phases before τ_1^1, so it self-suspends until the delivery of the second packet. As soon as the ISR of replica R_2 handles the last packet, τ_1^2 is awakened to execute the voting protocol. The behaviors of the second jobs of the previous tasks are dual: task τ_1^2 completes without any suspension, instead, τ_1^1 self-suspends to wait for the other replica. Note that, at its release time, the second job of τ_1^2 is blocked by τ_2^2 because the latter acquires the lock on the ICI to transmit a packet.

Note that with this approach the ICI queues may contain packets of different tasks at the same time. Indeed, some task τ_i^k can start sending packets and then be preempted by another task τ_j^k that sends its packets, and so on. As such, packets must contain the identifier of the sender task to be correctly dispatched by ISRs in the other replica.

4.2 LET-inspired voting

The underlying idea of this scheduling scheme is to get rid of both the waiting times and the any-time data transmission of the preceding scheme by confining all voting-related activities in predefined time intervals.

Together with the task set Γ_k, each replica $\mathcal{R}_k$ serves the execution of a set $\Upsilon_k = \{\upsilon_1^k, \ldots, \upsilon_n^k\}$ of *voting tasks*, one for each task τ_i^k, each of them executing at the same priority equal to a value higher than the priority of any task in Γ_k. Voting tasks are executed with the same period of the corresponding (regular) task, i.e., $T_i^{k,V} = T_i^k, \forall i, \forall k$. Tasks communicate with their corresponding voting tasks via shared-memory buffers. A task completes as soon as it finishes its computations, leaving the data to be voted in a shared-memory buffer. Then, the voting-related activities are delegated to the corresponding voting task υ_i^k, which is synchronously activated with τ_i^k. Note that, being υ_i^k executed at a higher priority than τ_i^k, it always executes before τ_i^k. As such, each j-th job of the voting task υ_i^k accomplishes the voting-related activities for the *preceding job*, i.e., the $(j-1)$-th one, of τ_i^k. Voting tasks are synchronously-released among replicas and executed in the same order on both replicas (voting tasks are selected according to their identifier whenever they are simultaneously pending). The execution of the voting tasks is also synchronized among replicas, meaning that a rendez-vous point is provided at their completion so that each voting task υ_i^k finishes together to $\upsilon_i^{or(k)}$. The latter synchronization is implemented by means of the synchronization signals offered by the ICI.

Voting tasks access the ICI in polling mode (no ICI-related ISRs are present under this voting scheme). The voting tasks perform the transmission and reception of packets in inverse order on the two replicas, as stated by the following rules.

Transmission rule. After completing their computations, the tasks terminate their execution by leaving the packets to be transmitted in memory buffers shared with their corresponding voting tasks. The transmission is then delegated to the voting tasks. On replica $\mathcal{R}_1$, the voting task υ_i^1 of τ_i^1 transmits M_i packets to $\mathcal{R}_2$ as soon as it is activated. On replica $\mathcal{R}_2$, the voting task υ_i^2 of τ_i^2 transmits M_i packets to $\mathcal{R}_1$ after it received the packets sent by υ_i^1.

Receiving rule. On replica $\mathcal{R}_1$, the voting task υ_i^1 of τ_i^1 receives (in polling mode) M_i packets sent from $\mathcal{R}_2$ after it transmitted its packets. On replica $\mathcal{R}_2$, the voting task υ_i^2 of τ_i^2 receives (in polling mode) M_i packets from $\mathcal{R}_1$ as soon as it is activated.

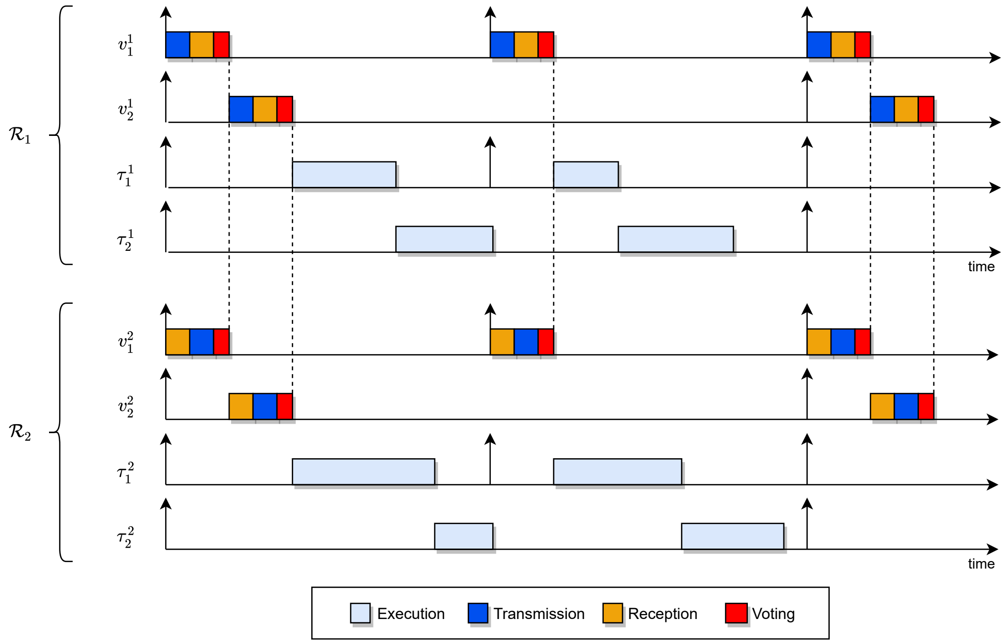

Figure 3 Example schedule under LET-inspired voting for a system with two replica pairs.

Waiting rule. None: when a task has finished its execution phase it terminates.

Voting rule. The voting protocol is executed by voting tasks after they completed both the packet transmission and reception. When the voting protocol terminates, the voting tasks busy waits until the corresponding voting task on the other replica sends a signal through the ICI synchronization line to notify the completion of the voting protocol.

The behavior of this LET-inspired scheduling scheme for voting is illustrated in Figure 3. Note that, since voting tasks are synchronously released together with their corresponding regular tasks and have a higher priority, voting is guaranteed to occur before starting executing the next job of regular tasks. In this way, voting is still *logically* occurring in the temporal context given by the period of the tasks that generate the data to be voted.

5 Inter-replica communication

This section deals with the analysis of inter-replica communications employing the ICI. Two problems are addressed. First, since under passive waiting packets can be sent at any time and that the communication is asynchronous (the ICI works in interrupt mode), packets of different tasks can be enqueued together in the ICI queues. This makes the worst-case transmission delay experienced by the packets of a certain task particularly challenging to be bounded, especially if considering the additional delays introduced by the waiting for the emptying of the queue. For this reason, we derive an analysis to ensure that the ICI queues are never full, hence getting rid of these additional delays by construction. Subsequently, we also provide a bound on the maximum delay introduced by the ICI.

5.1 Queuing analysis

We begin by bounding the amount of data sent within arbitrary time windows.

▶ **Lemma 1.** *In any time window of length t, the tasks can provide in the ICI queue at most $g(t)$ bytes of data, where*

$$g(t) = \min\left\{\sum_{i=1}^{n}\left\lceil\frac{t+T_i}{T_i}\right\rceil M_i b,\ \overline{\beta}t\right\}. \tag{1}$$

Proof. In any time window of length t a periodic task in r_i can release at most $\lceil (t+T_i)/T_i \rceil$ jobs (e.g., see [9], Ch. 5). Each job of the tasks in r_i sends at most M_i packets, each of size b bytes. Hence the first term in the minimum of Eq. (1). Note that the amount of data the tasks can send within a time window is also limited by the maximum rate with which the ICI queue can be filled, which is given by $\overline{\beta}$. Hence the lemma follows. ◀

The above lemma can then be used to derive a safe condition under which the ICI queues are never full.

▶ **Lemma 2.** *No task can find the ICI queues full if*

$$\forall t > 0, \quad g(t) - \alpha t \leq Qb. \tag{2}$$

Proof. Assume by contradiction that at a certain time instant t_1 a task finds an ICI queue full. Let $t_0 < t_1$ be the latest time at which the ICI queue has been empty and let $t = t_1 - t_0$. It holds that $(t_0, t_1]$ is an interval of length t in which the ICI has always been busy with packets to transmit to the other replica. Let $x(t)$ be the amount of bytes issued by the tasks to be provided in the ICI queue in $(t_0, t_1]$. Note that during this interval the ICI must have sent at least αt bytes: hence, if the queue is full at time t_1 it holds that $x(t) - \alpha t > Qb$.

By Lemma 1, in any time window of length t the cumulative amount of bytes provided in the ICI queue is bounded by $g(t)$. Hence, $g(t) \geq x(t)$, which implies $g(t) - \alpha t > Qb$. This contradicts Eq. (2). Hence the lemma follows. ◀

Note that Lemma 2 does not consist in a practical test as any possible value of t shall be checked. This issue is solved below by limiting the test to a finite number of check-points.

▶ **Lemma 3.** *Lemma 2 holds also if* $\forall t \in \Phi, \quad g(t) - \alpha t \leq Qb$, *where*

$$\Phi = \bigcup_{i=1}^{n} \{kT_i + \epsilon \leq t^*, k = 0, 1, 2, \ldots\} \cup \{\psi\} \tag{3}$$

with

$$t^* = \frac{2\sum_{i=1}^{n} M_i b}{\alpha - \sum_{i=1}^{n} \frac{M_i b}{T_i}}, \qquad \psi = \left\{t \leq t^* \ \mid \ \sum_{i=1}^{n}\left\lceil\frac{t+T_i}{T_i}\right\rceil M_i b = \overline{\beta}t\right\}, \tag{4}$$

and $\epsilon > 0$ arbitrarily small.

Proof. We prove the lemma by showing that function $g(t) - \alpha t$ can be maximal only for values $t \in \Phi$. First note that the minimum of two functions is upper bounded by the upper bound of one of the two functions. Hence $g(t) \leq G(t) = \sum_{i=1}^{n}\left(\frac{t+T_i}{T_i} + 1\right) M_i b$.

Note that both $G(t)$ and αt are two lines with slope $U^{\text{ICI}} = \sum_{i=1}^{n}(M_i b)/T_i$ and α, respectively. Recall that $\alpha > U^{\text{ICI}}$ (see Section 3). Therefore $G(t)$ and αt intersect and, from their intersection on, we have $g(t) \leq G(t) \leq \alpha t$ and hence also $g(t) - \alpha t \leq 0$.

The intersection occurs for the value t^* such that $G(t^*) = \alpha t^*$ and can be computed by solving the latter equality with respect to t^*, hence getting the expression at the left of Eq. (4). Therefore, for values of $t > t^*$ function $g(t) - \alpha t$ cannot be maximal.

If $g(t) = \sum_{i=1}^{n} \left\lceil \frac{t+T_i}{T_i} \right\rceil M_i b$ note that function $g(t) - \alpha t$ can be maximal only for those values of t that correspond to a step of the ceiling term of $g(t)$. The values are of the form $t = kT_i + \epsilon$ with k being a non-negative integer and $\epsilon > 0$ arbitrarily small. Conversely, if $g(t) = \overline{\beta} t$, being both the latter function and αt monotonic increasing, function $g(t) - \alpha t$ can be maximal only for those values of t for which at $t' = t + \epsilon$ (when $\alpha \leq \overline{\beta}$) or $t' = t - \epsilon$ (when $\alpha > \overline{\beta}$), with $\epsilon > 0$ arbitrarily small, it holds $g(t') \neq \overline{\beta} t$. These values of t must be an intersection between the two components that define $g(t)$, which are those of the set ψ at the right of Eq. (4). Hence the lemma follows. ◀

5.2 Delay analysis

▶ **Definition 4.** *The ICI-related delay Δ^{ICI} is an upper bound on the maximum time that can elapse from the time a packet is stored in the ICI queue by the sender task to the time the packet is available to be read from the ICI queue at the receiver.*

In the following the ICI-related delay is studied with queuing theory for networks [6] [23]. Under this approach, the ICI-related delay is decomposed as

$$\Delta^{\text{ICI}} = d_{\text{prop}} + d_{\text{trans}} + d_{\text{proc}} + d_{\text{queue}},$$

where d_{prop} is the propagation delay, d_{trans} is the transmission delay, d_{proc} is the processing delay at the receiver, and d_{queue} is the queuing delay. We proceed by individually bounding the above delay components.

Propagation delay. This delay corresponds to the physical propagation of the data along the wires that connect the two replicas. Clearly, it depends on both the technology used to realize the ICI and the wire length, as well as other physical properties such as the wire material. For instance, a typical SPI has a propagation delay of 5 ns/m [22], which is hence mostly negligible in an integrated system with short wiring. Hence $d_{\text{prop}} \approx 0$.

Transmission delay. This delay is simply bounded by the minimum guaranteed transmission rate α of the ICI as $d_{\text{trans}} \leq b/\alpha$.

Processing delay. This delay corresponds to the time taken by the ICI peripheral to make a packet available to be read from the ICI queue after it has been received. For instance, for SPI it is typically in the order of a very few microseconds (e.g., see [34]) and is hence mostly negligible. Thus $d_{\text{proc}} \approx 0$.

Queuing delay. This delay corresponds to the maximum time some data can remain in the ICI queues before being actually transmitted. In order to bound this delay component, the maximum number of packets that can be enqueued in the ICI queues at any time must be bounded first.

▶ **Lemma 5.** *The ICI queues never contain more than Q^{MAX} packets, where*

$$Q^{MAX} = \max_{t \in \Phi} \left\{ \left\lceil \frac{g(t) - \alpha t}{b} \right\rceil \right\} \tag{5}$$

and Φ is defined as in Lemma 3.

Proof. Assume by contradiction that at a certain time instant t_1 there are more than Q^{MAX} packets in an ICI queue. Let $t_0 < t_1$ be the latest time at which the ICI queue has been empty and let $t = t_1 - t_0$. Similarly as argued in the proof of Lemma 2 this implies $g(t) - \alpha t > Q^{\text{MAX}} b$, which in turn also implies $\left\lceil \frac{g(t)-\alpha t}{b} \right\rceil > Q^{\text{MAX}}$. By Lemma 3, function $g(t) - \alpha t$ can be maximal only for values $t \in \Phi$, hence Eq. (5) also gives the maximal value of $\left\lceil \frac{g(t)-\alpha t}{b} \right\rceil$ that must be both equal and larger to Q^{MAX}. This is a contradiction. The lemma follows. ◀

The maximum time a packet can be delayed while being in the queue is guaranteed not to be larger than the cumulative transmission time of all the preceding packets in the queue, which can be at most $Q^{\text{MAX}} - 1$. Hence

$$d_{\text{queue}} \leq (Q^{\text{MAX}} - 1) \cdot b/\alpha. \tag{6}$$

6 Response-time analysis

This section focuses on bounding the worst-case response time of tasks under both passive waiting and LET-inspired voting.

6.1 Passive waiting

Following Section 4, besides its regular execution, which lasts at most E_i^k time units, each task also executes the transfer of the data to be voted into the ICI registers and the voting protocol, which last at most VT_i and VP_i time units, respectively, on both replicas. Hence, the cumulative WCET of task τ_i^k is given by

$$C_i^k = E_i^k + VT_i + VP_i. \tag{7}$$

The analysis of tasks under passive waiting is split into two parts. First we bound the *partial response time* of a task, which is defined as the response time up to the copy into the ICI registers of the data to be voted, i.e., just before the start for the waiting of the other replica. Subsequently, the response time of the whole task is bounded as a function of the partial response time.

Up to the partial response time, task τ_i^k can be delayed by (i) its own execution and the transfer of the data to be voted into the ICI registers, which can last at most $E_i^k + VT_i$ time units, (ii) the interference generated by high-priority tasks, (iii) the blocking time generated by low-priority tasks, and (iv) the interference generated by the ISRs (which run at higher priorities). We proceed by bounding these components individually.

Note that, under passive waiting, tasks behave as self-suspending tasks [13]. As such, high-priority interference can be bounded utilizing a state-of-the-art result provided that the WCET bound of Equation (7) is used.

▶ **Lemma 6.** *Under passive waiting, the high-priority interference generated to a job of task τ_i by high-priority tasks in any interval of length t is bounded by*

$$I_i^{k,hp}(t) = \sum_{\tau_j^k \in hp(i,k)} \left\lceil \frac{t + \overline{R_j^k} - C_j^k}{T_j} \right\rceil \cdot C_j^k,$$

where $\overline{R_j^k}$ is an upper bound on the response time of τ_j^k.

Proof. Follows by [13] (Theorem 1). ◀

Now, we proceed by bounding the non-preemptive blocking generated by low-priority tasks because of the locking of the ICI.

▶ **Lemma 7.** *Under passive waiting, a job of task τ_i can be blocked at most twice, one before its partial response time and one after, and each time by at most PT time units.*

Proof. Due to the transmission rule under passive waiting (Sec. 4.1), a task can lock one of the ICI to transmit a packet, hence entering a non-preemptive section that can delay a higher-priority task. As the lock is released after the packet is stored in the ICI registers, the non-preemptive section can last at most PT time units. Tasks can be prevented from execution due to non-preemptive blocking (i) at their release, and (ii) when resuming their execution after self-suspensions, which occurs after their partial response time. Case (ii) can happen only once as tasks suspend once to wait for the completion of the replica task. Hence the lemma follows. ◀

▶ **Lemma 8.** *Let $\overline{PR}_j^k$ be an upper bound on the partial response time of task τ_j^k. Under passive waiting, the interference generated to a job of task τ_i^k, in any interval of length t, by ISRs that handle packets for τ_j^k is bounded by*

$$I_{i,j}^{k,ISR}(t) = \left\lceil \frac{t + \overline{PR}_j^{or(k)} + \Delta^{ICI}}{T_j} \right\rceil \cdot M_j \cdot (\sigma^{ISR} + PT).$$

Proof. Consider an arbitrary time interval $[0, t]$ and a replica $\mathcal{R}_k$. ISRs are activated by packets sent by jobs of tasks running in the other replica $\mathcal{R}_{or(k)}$. Each job of task $\tau_j^{or(k)}$ in $\mathcal{R}_{or(k)}$ can send at most M_j packets, each requiring PT time units to be read by ISRs in $\mathcal{R}_k$. Each job of task $\tau_j^{or(k)}$ can also activate at most M_j ISRs in $\mathcal{R}_k$, one per packet sent, each introducing an overhead of at most σ^{ISR} time units. Overall, the total ISR-related workload generated by a job $\tau_j^{or(k)}$ is bounded by $M_j \cdot (\sigma^{\text{ISR}} + PT)$.

Now, note that tasks can send packets only before the occurrence of their partial response time. Hence, a job of task $\tau_j^{or(k)}$ released before time $-(\overline{PR}_j^k + \Delta^{\text{ICI}})$ cannot activate an ISR in $\mathcal{R}_k$ during $[0, t]$ as its packets would have already been sent and transmitted before the beginning of the interval. Hence, only jobs of $\tau_j^{or(k)}$ released in interval $[-(\overline{PR}_j^k + \Delta^{\text{ICI}}), t]$ may activate ISRs in $[0, t]$. This means that there are most $\left\lceil \frac{t+\overline{PR}_j^{or(k)}+\Delta^{\text{ICI}}}{T_j} \right\rceil$ jobs of $\tau_j^{or(k)}$ that can activate ISRs in $\mathcal{R}_k$ during $[0, t]$. Hence the lemma follows. ◀

Bounds on contributions (i)-(iv) mentioned above are hence now available. Following classical response-time analysis, a bound on the worst-case partial response time PR_i^k of each task τ_i^k can then be computed as the least positive fixed point of the recurrence:

$$PR_i^k = E_i^k + VT_i + I_i^{k,\text{hp}}(PR_i^k) + PT + \sum_{j=1}^{n} I_{i,j}^{k,\text{ISR}}(PR_i^k). \tag{8}$$

Note that Equation (8) uses the interference bound of Lemma 8, which in turn requires the knowledge of an upper bound on the partial response time $\overline{PR}_j^k$ that is to be computed by Equation (8), hence introducing a circular dependency. This issue can be solved with a typical refinement algorithm for response-time bounds starting from a safe value (e.g., see [10]), such as the task deadline.

It is now possible to bound the total response time of the tasks by bounding the worst-case response time of the execution of the voting protocol.

▶ **Lemma 9.** *After at most*

$$J_i^k = \max\{PR_i^k, PR_i^{or(k)}\} + \Delta^{ICI} + Q^{MAX} \cdot (\sigma^{ISR} + PT) \tag{9}$$

time units from the task release, the voting protocol of task τ_i^k is ready to start executing.

Proof. Given the task behavior under passive waiting specified in Section 4.1, the voting protocol of task τ_i^k can start executing only after that (i) all its M_i packets have been sent to the other replica, and (ii) all packets sent by the replica task $\tau_i^{or(k)}$ have been received and handled by ISRs.

Let us consider times related to the release of τ_i^k. At time $\max\{PR_i^k, PR_i^{or(k)}\}$ both τ_i^k and $\tau_i^{or(k)}$ have sent their packets by definition of partial response time. The last packet sent by $\tau_i^{or(k)}$ will take at most Δ^{ICI} to be transmitted to $\mathcal{R}_k$. Hence, at time $\max\{PR_i^k, PR_i^{or(k)}\} + \Delta^{\text{ICI}}$ all packets sent by $\tau_i^{or(k)}$ must already have been received by $\mathcal{R}_k$. When the last of such packets is received it may still be the case that there are some other packets ahead in the ICI queue to be processed: by Lemma 5, they can be at most $Q^{\text{MAX}} - 1$ and each of them can take at most $(\sigma^{\text{ISR}} + PT)$ time units to be processed as discussed in the proof of Lemma 8. At most other $(\sigma^{\text{ISR}} + PT)$ time units are needed to process the last packet sent by $\tau_i^{or(k)}$. Hence the lemma follows. ◀

The above lemma allows studying the execution of the voting protocol of each task τ_i^k as a sub-task with jitter J_i^k whose completion corresponds to the completion of τ_i^k.

▶ **Theorem 10.** *The response time of task τ_i^k is bounded by $J_i^k + R_i^k$, where R_i^k is the least positive fixed point of the following recurrence:*

$$R_i^k = VP_i + PT + I_i^{k,hp}(R_i^k) + \sum_{\substack{j=1 \\ j\neq i}}^{n} I_{i,j}^{k,ISR}(R_i^k). \tag{10}$$

Proof. Task τ_i^k completes when the execution of the voting protocol completes. The latter lasts at most VP_i time units and can be delayed by (i) non-preemptive blocking, (ii) the execution of high-priority tasks, and (iii) the execution of ISRs. By Lemma 7, non-preemptive blocking is no larger than PT time units. By Lemma 6, high-priority task interference is bounded by $I_i^{k,hp}(t)$. Note that only ISRs that handle packets of other tasks $\tau_j^k \neq \tau_i^k$ can interfere with the execution of the voting protocol as the latter becomes eligible for execution only when all packets of τ_i^k have been received. Hence, by Lemma 8, the last term in Eq. (10) bounds the ISR interference.

Due to the fact that all the phenomena that can delay the execution of the voting protocol are safely bounded, by standard response-time analysis the least positive fixed point of Eq. (10) bounds the largest amount of time the execution of the voting protocol can take to complete from the time it becomes ready to execute. Therefore, after recalling Lemma 9, $J_i^k + R_i^k$ is a safe response time and the theorem follows. ◀

6.2 LET-inspired voting

Under the LET-inspired scheduling scheme for voting, the tasks compute their results and terminate without undertaking any voting-related activity. Therefore, the WCET of each task τ_i^k can be computed as just $C_i^k = E_i^k$.

Conversely, the voting tasks (i) receive the data produced by the other replica, (ii) transmit the data to the other replica, (iii) execute the voting protocol, and (iv) finally wait for the completion of the corresponding voting task on the other replica. As specified in

Section 4.2, voting tasks are synchronized among replicas: being synchronously released and synchronously terminated, the execution of the voting tasks running on the two replicas perfectly overlaps in time. This allows bounding the WCET of the voting tasks as follows.

▶ **Theorem 11.** *The WCET of voting task* υ_i^k *is bounded by*

$$C_i^{k,V} = 2\left(VT_i + \frac{M_i b}{\alpha} + VR_i\right) + VP_i. \tag{11}$$

Proof. Following the behavior specified in Section 4.2, voting tasks execute the transmission and reception of packets in different orders. We then separately study the voting tasks on the two replicas. On $\mathcal{R}_1$, υ_i^1 first executes the transmission and then the reception. The time taken to perform these operations is due to (i) the actual copies to and from the ICI device registers and (ii) the eventual busy waiting either because the ICI queue is full during transmission or because the ICI queue is empty during the reception. Contribution (i) can be at most $VT_i + VR_i$ time units.

Since voting tasks are synchronously executed on the two replicas and the transmission and reception phases are performed in inverse orders on the two replicas, when υ_i^1 is transmitting packets υ_i^2 can continuously make progress in receiving them, and vice versa. Furthermore, since the completion of voting tasks is synchronized among replicas, the ICI queues are guaranteed to be empty whenever the voting tasks are activated. Hence, during the execution of υ_i^1 and υ_i^2 only packets related to replica pair r_i can be present in the ICI queues. This means that υ_i^2 can take at most $\frac{M_i b}{\alpha} + VR_i$ time units to receive the packets sent by υ_i^1 and υ_i^2 can take at most $\frac{M_i b}{\alpha} + VT_i$ time units to transmit its packets to υ_i^1. These terms bound the corresponding waiting times experienced because the ICI queues are either full or empty. Hence, contribution (ii) is bounded by $2\frac{M_i b}{\alpha} + VR_i + VT_i$.

Finally, since the reception is performed in polling mode, when υ_i^1 starts executing the voting protocol υ_i^2 must already have transmitted its packets, otherwise the reception phase of υ_i^1 would not be completed. Hence, υ_i^1 can either execute the voting protocol for at most VP_i time units or wait for the completion of just the execution of the voting protocol in $\mathcal{R}_2$, which lasts anyway at most VP_i time units. Overall, υ_i^1 can execute for at most $(VT_i + VR_i) + (2\frac{M_i b}{\alpha} + VR_i + VT_i) + VP_i$ time units, hence matching Eq. (11).

Now, let us consider υ_i^2. For the same reasons discussed above, this task can wait at most $\frac{M_i b}{\alpha} + VT_i$ time units during the reception of packets performed at the beginning of the task. At the time t^* at which υ_i^2 received all packets it is guaranteed that υ_i^1 has completed its transmission phase. Hence, υ_i^2 cannot wait for more than the time υ_i^1 can take to complete its reception phase and the execution of the voting protocol, which is bounded by $\frac{M_i b}{\alpha} + VR_i + VT_i + VP_i$ as discussed above. From t^* on, υ_i^2 can also executes its transmission phase and voting protocol for no more than $VT_i + VP_i$ time units. Hence, the total time υ_i^2 can take from t^* to its completion, either busy waiting or executing, is bounded by $\max\{\frac{M_i b}{\alpha} + VR_i + VT_i + VP_i, VT_i + VP_i\} = \frac{M_i b}{\alpha} + VR_i + VT_i + VP_i$. Hence, the total execution time of υ_i^2 is bounded again by Eq. (11). The theorem follows. ◀

With the above lemma in place, it is now possible to bound the worst-case response time of the tasks as follows. The worst-case response time of task τ_i^k is bounded by the least positive solution of the following recurrence:

$$R_i^k = C_i^k + I_i^{k,\mathrm{hp}}(R_i^k) + I_i^{k,\mathrm{V}}(R_i^k),$$

where $I_i^{k,\mathrm{hp}}(R_i^k)$ is a bound on the interference generated by high-priority tasks and $I_i^{k,\mathrm{V}}(R_i^k)$ is a bound on the interference generated by voting tasks.

▶ **Lemma 12.** *It holds* $I_i^{k,hp}(R_i^k) = \sum_{\tau_j^k \in hp(i,k)} \left\lceil \frac{R_i^k}{T_j^k} \right\rceil \cdot C_j^k$.

Proof. Under LET-inspired voting, tasks τ_i^k behave as regular periodic tasks (note that no suspensions are involved). Thus, the lemma follows from standard response-time analysis for periodic tasks under preemptive fixed-priority scheduling [20]. ◀

▶ **Lemma 13.** *It holds* $I_i^{k,V}(R_i^k) = \sum_{\upsilon_j^k \in \Upsilon_k} \left\lceil \frac{R_i^k}{T_j^{k,V}} \right\rceil \cdot C_j^{k,V}$.

Proof. Voting tasks have a higher priority than any task in Γ_k, hence they all generate high-priority interference to τ_i^k. They are also periodically activated and execute as standard periodic tasks. Hence the lemma follows as for Lemma 12 provided that the WCET bound of Theorem 11 is used. ◀

6.3 Discussion

As it can be noted from the above sections, the analysis of voting with passive waiting is much more challenging than the one under LET-inspired voting due to the various sources of unpredictability introduced by that scheme. In addition, passive waiting requires the analysis of packet queuing presented in Section 5 to deal with any-time packet transmissions.

On the other hand, passive waiting is relatively simple to implement from the perspective of the programmer and does not require introducing additional tasks in the system. Furthermore, it introduces limited priority inversion related to voting: indeed, a high-priority task can be delayed by voting-related activities of low-priority tasks only by the transmission of one packet and the reception of packets by means of ISRs.

Conversely, LET-inspired voting does not require the packet queuing analysis since the voting data is communicated in precise time intervals during which the interested voting tasks are synchronously executed on both replicas. Nevertheless, this approach tends to introduce larger priority inversion because all voting-related activities are executed by LET tasks at the highest priority. Hence, the whole transmission and reception of packets as well as the voting protocol of a low-priority task can interfere with the execution of a high-priority task.

7 Experimental results

This section reports the results of an experimental evaluation that was conducted to compare those two voting scheduling strategies studied in this paper.

Workload generation. Given a target task set utilization U and a number of tasks n, N task sets have been generated with the Emberson et al.'s generator [15], which was configured to randomly select the task periods in the range $[T^{\min}, T^{\max}]$ with log-uniform distribution. The task sets Γ_1 and Γ_2 of the two replicas were then generated as follows. For each replica pair r_i, one replica was randomly selected to be the slower in executing it, say $\mathcal{R}_k$, then $E_i^{or(k)}$ was set to the WCET value obtained by the task generator and $E_i^k = E_i^{or(k)} \cdot \xi$, where ξ was randomly selected in $[\xi^{min}, 1]$ with uniform distribution. Note that, since the WCET provided by the task generator is used to control the worst-case duration of the execution phase of tasks (parameter E_i^k), the utilization U used to control the generation refers to the maximum per-replica utilization *without voting-related activities.* A random number $\lfloor p^{\text{vital}} \cdot n \rfloor$ of tasks, with p^{vital} randomly chosen in $[0.6, 0.8]$ with uniform distribution, were selected to be vital in each replica pair, and hence to require voting. For each vital replica pair, the number of packets M_i was randomly generated in $[0, M^{\max}]$ with uniform distribution.

Table 1 Nominal setting of the parameters that control the workload generation.

Parameter	Value	Description
N	500	Number of task sets
n	10	Number of tasks per replica
p^{vital}	[0.6, 0.8]	Vital task ratio for each replica pair
T^{min}	5000	Task minimum period (μs)
T^{max}	500000	Task maximum period (μs)
ξ^{min}	0.85	Minimum faster-replica speed coefficient
M^{max}	5	Maximum number of packets sent by a task
α	15	ICI bandwidth (MB/s)
b	16	Number of bytes per packet
Q	8	ICI queue size (packets)
γ	13.24	Minimum read/write rate to access memory (MB/s)
β	13.24	Minimum read/write rate to access device registers (MB/s)
$\overline{\beta}$	56.47	Maximum read/write rate to access device registers (MB/s)
σ^{ISR}	2	ISR overhead (μs)
λ^{VP}	50	Time required to vote a packet of data (μs)

Parameters VR_i and VT_i were computed accordingly as a function of M_i. The WCET of the voting protocol was also generated as $VP_i = \lambda^{\text{VP}} \cdot M_i$ where $\lambda^{\text{VP}} \geq 0$ is another parameter that control the generation. For non-vital replica pairs we set $VP_i = VT_i = VR_i = 0$. All tasks were assigned implicit deadlines (i.e., $D_i = T_i$).

To configure the device register and memory access rates β, $\overline{\beta}$, and γ we took the Xilinx Ultrascale+ SoC (considering the Cortex-A cores running at 1.2 GHz) as a reference platform, from which we obtain respectively, 725, 170, and 170 clock cycles by profiling. The configuration of other parameters that are not mentioned above is varied in the experiments presented next and, whenever mentioned, is kept fixed to the nominal setting reported in Table 1.

Experiments. A first experiment was conducted by varying the utilization without voting U and testing $N = 500$ task sets per utilization value. The results under four representative configurations are reported in Figure 4. The plots report the schedulability performance of the proposed analysis techniques for voting with passive waiting (Section 4.1) and LET-inspired voting (Section 4.2), as well as for the system without voting activities (used as a reference upper bound of the schedulability performance). These results were obtained under the setting reported above each plot, where the parameters that are not mentioned were set to the nominal configuration of Table 1.

As it can be noted from the plots, LET-inspired voting always outperforms passive waiting. Passive waiting is strongly penalized in the presence of short ICI queues (see Fig. 4(a) vs. Fig. 4(b)) due to the queuing analysis, while LET-inspired voting is almost insensitive to the ICI queue size as expected. The performance of both approaches degrades as the number of packets sent by tasks increases (see Fig. 4(c) vs. Fig. 4(d)), but LET-inspired voting is capable of guaranteeing much better schedulability performance than passive waiting as M^{max} increases.

Another experiment was conducted to study the dependency of the schedulability performance of the two approaches as a function of other parameters different than U. The results are reported in Figure 5, where 500 task sets have been tested for each value of the varied parameters. Figure 5(a) illustrates the dependency of the schedulability performance on the minimum task period T^{min} in a condition of high system load (U= 0.9). This figure clearly

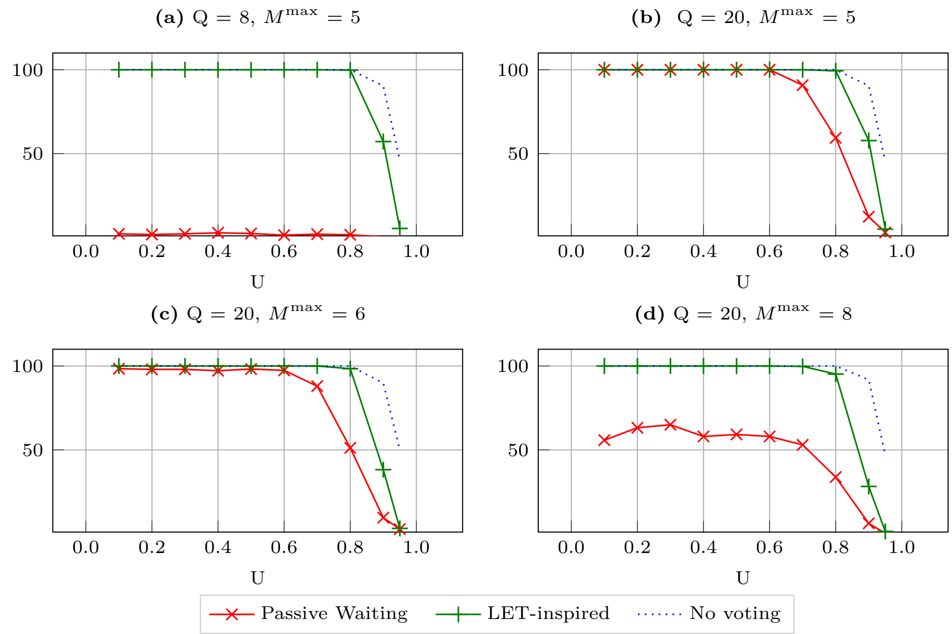

Figure 4 Schedulability ratio (y-axis of the plots) as a function of the voting-unrelated utilization U used to control the task set generation under four representative configurations.

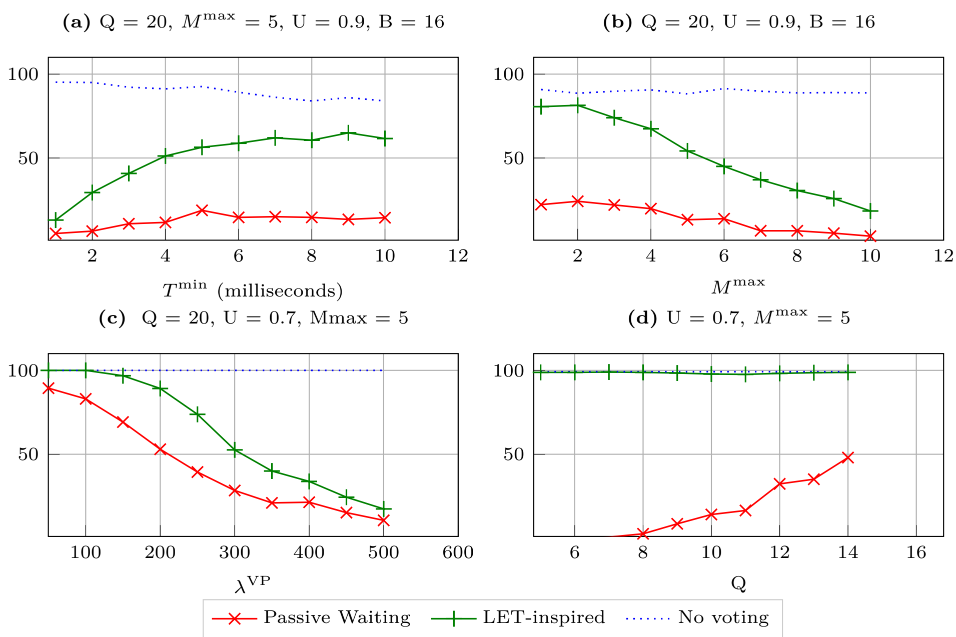

Figure 5 Schedulability ratio (y-axis of the plots) as a function of $T^{\min}$, $M^{\max}$, λ^{VP}, and Q under four representative configurations.

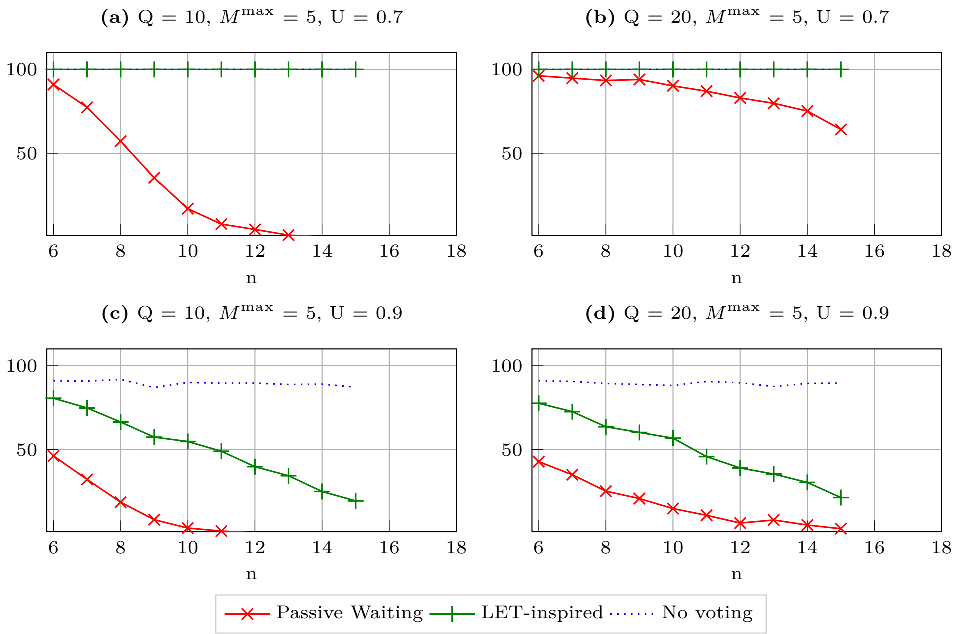

Figure 6 Schedulability ratio (y-axis of the plots) as a function of n under four representative configurations.

shows that LET-inspired voting is penalized in the presence of very short task periods due to the priority-inversion generated by voting tasks discussed in Section 6.3. Figures 5(b) and 5(c) show how the performance of both approaches degrades as either $M^{\max}$ or λ^{VP} increases, and that the gap between the two reduces for large values of these parameters. Finally, Figure 5(d) confirms that passive waiting exhibits very poor performance in the presence of short ICI queues and that LET-inspired voting is insensitive to this parameter. The last experiment was conducted to assess how the schedulability ratio of both approaches varies as a function of the number of tasks in the tested task sets. Figure 6(a) shows that the performance of passive waiting quickly degrades by increasing the number of tasks while LET-inspired voting is not affected by the size of the task set. Figure 6(b) reports the results under the same configuration of Figure 6(a) but considering larger ICI queues: in this case, the performance of passive waiting definitively improves but is still lower than the one of LET-inspired voting. Furthermore, Figure 6(c) and Figure 6(d) show that the performance of both approaches decreases as the number of tasks increases at high utilization ($U = 0.9$). Nevertheless, LET-inspired voting always outperforms passive waiting in all the tested cases.

8 Conclusion and future work

This paper studied two scheduling strategies for distributed voting protocols in 2-out-of-2 redundant real-time systems, namely passive waiting (based on task self-suspensions to wait for the other replica) and LET-inspired voting. Both queuing and delays related to inter-replica communication interfaces have been studied. Response-time analysis for real-time tasks under the two strategies has been presented. The pros and cons of the two scheduling strategies have also been discussed. The two strategies have been experimentally compared in terms of schedulability performance. The experimental results revealed that LET-inspired voting is always preferable to passive waiting, exhibiting even a 100% performance gap in the presence of short packet queues of inter-replica communication interfaces. In other configurations with longer queues, LET-inspired voting is also capable of scheduling up to five more times task sets than passive waiting.

Future work should investigate the possibility of improving the analysis of passive waiting, both in terms of packet queuing and response times, and on the design of improved scheduling strategies that can better control the priority inversion introduced by LET-inspired voting.

References

1 Jaemin Baek, Jeonghyun Baek, Jeeheon Yoo, and Hyeongboo Baek. An n-modular redundancy framework incorporating response-time analysis on multiprocessor platforms. *Symmetry*, 11(8):960, 2019.

2 Julian M Bass. *Voting in real-time distributed computer control systems*. PhD thesis, University of Sheffield, 1995.

3 H Benítez-Pérez, G Latif-Shabgahi, HA Thompson, S Bennett, PJ Fleming, and JM Bass. Integration and comparison of fdi and fault masking features in embedded systems. *IFAC Proceedings Volumes*, 32(2):7712–7717, 1999.

4 Guillem Bernat, Jose Miro-Julia, and Julian Proenza. A technique to analyze the tolerance to transient overloads of a fault-tolerant real-time system. In *Proceedings 1997 High-Assurance Engineering Workshop*, pages 221–226. IEEE, 1997.

5 Guillem Bernat, Jose Miro-Julia, Julian Proenza, et al. Fixed priority schedulability analysis of a distributed real-time fault tolerant architecture. In *PDPTA*, pages 479–487, 1997.

6 Dimitri Bertsekas and Robert Gallager. *Data Networks (2nd Ed.)*. Prentice-Hall, Inc., USA, 1992.

7 DM Blough and GF Sullivan. Voting using predispositions. *IEEE Transactions on reliability*, 43(4):604–616, 1994.

8 Douglas M Blough and Gregory F Sullivan. A comparison of voting strategies for fault-tolerant distributed systems. In *Proceedings Ninth Symposium on Reliable Distributed Systems*, pages 136–145. IEEE, 1990.

9 B. Brandenburg. Scheduling and locking in multiprocessor real-time operating systems. In *Ph.D. dissertation, The University of North Carolina at Chapel Hill*, 2011.

10 D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo. Partitioned fixed-priority scheduling of parallel tasks without preemptions. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 421–433, 2018. doi:10.1109/RTSS.2018.00056.

11 EN CEI. Cei en 50126-1. *Railway Applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS). Part 1: Generic RAMS Process*, 2019.

12 EN CEI. Cei en 60730-1. *Automatic electrical controls - Part1: General requirements*, 2019.

13 J. Chen, G. Nelissen, and W. Huang. A unifying response time analysis framework for dynamic self-suspending tasks. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 61–71, 2016. doi:10.1109/ECRTS.2016.31.

14 Daniel Davies and John F. Wakerly. Synchronization and matching in redundant systems. *IEEE Computer Architecture Letters*, 27(06):531–539, 1978.

15 P. Emberson, R. Stafford, and R.I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–11, July 2010.

16 Oytun Eriş, Uğur Yıldırım, Mustafa S Durmuş, Mehmet T Söylemez, and Salman Kurtulan. N-version programming for railway interlocking systems: Synchronization and voting strategy. *IFAC Proceedings Volumes*, 45(24):177–180, 2012.

17 Saurabh Gohil, Aravind Basavalingarajaiah, and Varadharajan Ramachandran. Redundancy management and synchronization in avionics communication products. In *2011 Integrated*

Communications, Navigation, and Surveillance Conference Proceedings, pages C3–1. IEEE, 2011.

18 Arpan Gujarati, Sergey Bozhko, and Björn B Brandenburg. Real-time replica consistency over ethernet with reliability bounds. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 376–389. IEEE, 2020.

19 T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003. doi:10.1109/JPROC.2002.805825.

20 M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, January 1986. doi:10.1093/comjnl/29.5.390.

21 Hagbae Kim and Kang G Shin. Sequencing tasks to minimize the effects of near-coincident faults in tmr controller computers. *IEEE transactions on computers*, 45(11):1331–1337, 1996.

22 Thomas Kugelstadt. Extending the spi bus for long-distance communication. *Analog Applications Journal*, 2011. URL: https://www.ti.com/lit/an/slyt441/slyt441.pdf.

23 J.F. Kurose and K.W. Ross. *Computer Networking: A Top-Down Approach*. Pearson Education, Limited, 2010. URL: https://books.google.it/books?id=2hv3PgAACAAJ.

24 Seong Woo Kwak and Byung Kook Kim. Task-scheduling strategies for reliable tmr controllers using task grouping and assignment. *IEEE Transactions on Reliability*, 49(4):355–362, 2000.

25 G Latif-Shabgahi, JM Bass, and S Bennett. Complete disagreement in redundant real-time control applications. *IFAC Proceedings Volumes*, 31(4):223–228, 1998.

26 G Latif-Shabgahi, Julian M Bass, and Stuart Bennett. A taxonomy for software voting algorithms used in safety-critical systems. *IEEE Transactions on Reliability*, 53(3):319–328, 2004.

27 Stephen R McConnel and Daniel P Siewiorek. Synchronization and voting. *IEEE Transactions on Computers*, 100(2):161–164, 1981.

28 P. Pazzaglia, D. Casini, A. Biondi, and M. Di Natale. Optimal memory allocation and scheduling for dma data transfers under the let paradigm. In *58th Design Automation Conference (DAC)*, 2021.

29 Dai Shenghua and Li Yishi. Research on 2-out-of-2 multiplying 2 redundancy system used in high-speed train. In *2011 IEEE International Conference on Computer Science and Automation Engineering*, volume 2, pages 483–486. IEEE, 2011.

30 Martin L Shooman. *Reliability of computer systems and networks*. Wiley Online Library, 2002.

31 Daniel Siewiorek and Robert Swarz. *Reliable computer systems: design and evaluatuion*. Digital Press, 2017.

32 Daniel P Siewiorek and Priya Narasimhan. Fault-tolerant architectures for space and avionics applications. *NASA Ames Research http://ic. arc. nasa. gov/projects/ishem/Papers/Siewi*, 2005.

33 Zhijun Tong and Richard Y Kain. Vote assignments in weighted voting mechanisms. *IEEE Transactions on Computers*, 40(5):664–667, 1991.

34 Xilinx. Zynq-7000 soc: Dc and ac switching characteristics - ds191, 2018. URL: https://www.xilinx.com/support/documentation/data_sheets/ds191-XC7Z030-XC7Z045-data-sheet.pdf#G1940899.