POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Tesi di Laurea Magistrale in
Computer Science and Engineering
Ingegneria Informatica

A Quorum-based Approach for Hard Real-Time Distributed Systems

Author:
Matteo Villa

Student ID:
226190

Advisors:
Prof. Federico Reghenzani

Co-Advisors:
Tomas A. Lopez

Academic Year:
2024-25

To my family,
who have always supported me.

Abstract

Hard real-time distributed systems are increasingly deployed in safety-critical domains such as avionics, automotive, and industrial automation. Ensuring both correctness and availability in these systems poses significant challenges, since correctness also depends on meeting strict timing constraints, particularly in the presence of distributed components. While the literature provides several techniques to address hardware failures in such systems, fewer studies focus on Byzantine failures, and all of them rely on consensus-based mechanisms. This thesis investigates quorum-based approaches for hard real-time distributed systems, aiming to achieve fault tolerance while ensuring that task deadlines are met even under Byzantine failures. Unlike consensus-based methods, the proposed model enables failure detection and recovery without requiring task re-execution, thereby improving efficiency and maintaining determinism. Furthermore, we show how the quorum-based approach can also be extended to handle hardware failures. The model has been evaluated using OMNeT++, an open-source network simulator, integrated with the INET framework, which provides a comprehensive set of communication protocols, components, and network models. The experimental results highlight both the strengths and current limitations of the proposed solution, while also outlining directions for future improvements. Overall, this work contributes a novel foundation for research on fault-tolerant hard real-time distributed systems.

Keywords: hard real-time distributed systems, byzantine failures, hardware failures, quorum-based approach, load balancing, network calculus, time-sensitive networking, time-aware shaper

Abstract in lingua italiana

I sistemi distribuiti hard real-time sono sempre più diffusi in domini safety-critical come l'avionica, l'automotive e l'automazione industriale. Garantire sia la correttezza sia la disponibilità in questi sistemi rappresenta una sfida significativa, poiché la correttezza dipende anche dal rispetto di stringenti vincoli temporali, soprattutto in presenza di componenti distribuiti. Sebbene la letteratura offra diverse tecniche per gestire i guasti hardware in tali sistemi, pochi studi si concentrano sui guasti bizantini, e tutti si basano su meccanismi di consenso. Questa tesi indaga approcci basati su quorum per sistemi distribuiti hard real-time, con l'obiettivo di ottenere tolleranza ai guasti garantendo al contempo il rispetto delle deadline dei task anche in presenza di guasti bizantini. A differenza dei metodi basati sul consenso, il modello proposto consente la rilevazione e il recupero dai guasti senza richiedere la riesecuzione dei task, migliorando così l'efficienza e mantenendo il determinismo. Inoltre, mostriamo come l'approccio basato su quorum possa essere esteso anche alla gestione dei guasti hardware. Il modello è stato valutato tramite OMNeT++, un simulatore di rete open-source, integrato con il framework INET, che fornisce un insieme completo di protocolli di comunicazione, componenti e modelli di rete. I risultati sperimentali evidenziano sia i punti di forza sia le attuali limitazioni della soluzione proposta, delineando al contempo possibili direzioni per futuri miglioramenti. Nel complesso, questo lavoro contribuisce a gettare nuove basi per la ricerca su sistemi distribuiti hard real-time tolleranti ai guasti.

Parole chiave: sistemi distribuiti hard real-time, guasti bizantini, guasti hardware, approccio basato su quorum, bilanciamento del carico, network calculus, time-sensitive networking, time-aware shaper

Contents

1 | Introduction

1.1. Distributed Hard Real-Time Systems

Hard real-time distributed systems are a class of computing systems in which a set of interconnected nodes collaboratively execute tasks under strict timing constraints. In such systems, the correctness of operations depends not only on their logical outcomes but also on their completion within predefined deadlines. A missed deadline is considered a system failure, regardless of the functional correctness of the result. These systems are commonly employed in safety-critical domains such as aerospace, automotive, industrial automation, and medical devices, where deterministic behavior and high availability are essential. The distributed nature of these systems introduces additional complexity, including communication delays, fault propagation, and synchronization challenges among nodes, all of which must be accounted for to ensure predictable and reliable operation. To guarantee temporal correctness, hard real-time distributed systems rely on rigorous design methodologies, including precise scheduling, fault-tolerant architectures, and predictable communication protocols. Moreover, due to the increasing complexity and scale of modern applications, achieving both real-time guarantees and robustness against faults, such as hardware failures, software bugs, or network disturbances, has become a central concern in their development [15]. In addition, the integration of embedded systems with hard real-time systems has become increasingly common in safety-critical domains such as automotive, avionics, and industrial automation. Embedded platforms are typically responsible for executing control and monitoring tasks while operating under stringent resource constraints. The combination of embedded architectures with hard real-time requirements introduces a range of challenges, including limited computational capacity, constrained memory, and restricted communication bandwidth. Despite these limitations, such systems must still ensure predictable and deterministic behavior. Achieving timing correctness in this context demands precise scheduling strategies, robust fault-tolerance mechanisms, and communication protocols capable of guaranteeing bounded latency and jitter.

As a result, research in this field focuses on balancing timing predictability, system relia-

bility, and resource efficiency, often under stringent design constraints and in the presence of partial failures or uncertainties. This balance is critical to ensuring that the system can maintain correct and timely behavior under all anticipated operational conditions. To date, only network protocols for hard real-time distributed systems have been standardized, with *Time-Sensitive Networking* (TSN) and *Time-Triggered Ethernet* (TTEthernet) representing the most notable examples. In the literature, several approaches have been proposed to achieve fault tolerance in the presence of hardware failures, while preserving strict real-time guarantees. However, when considering Byzantine failures, where components may behave arbitrarily or maliciously, the body of research remains relatively limited. This indicates a need for further investigation into techniques capable of tolerating such failures in hard real-time distributed environments, particularly in safety-critical domains where both timing and correctness are paramount.

1.2. Contributions

The primary contribution of this thesis is the development of a conceptual model for managing Byzantine faults in hard real-time distributed systems. The proposed model is designed to serve as a foundation for future research and practical advancements in this domain, where real-time constraints and fault tolerance must coexist under strict determinism. This work aims to bridge the gap between existing fault-tolerance mechanisms and the stringent requirements of hard real-time systems by adapting and integrating known Byzantine fault-handling techniques within a time-critical distributed context. In doing so, the thesis highlights how such techniques can be effectively leveraged not only to address Byzantine behavior, but also to handle hardware faults and support fault-recovery in real-time environments. Furthermore, the proposed model seeks to maximize the advantages of these techniques, such as strong consistency and resilience, while simultaneously mitigating their limitations, notably their communication and computational overhead, whenever feasible. Special attention is given to maintaining a high degree of modularity and adaptability, allowing the model to be reused across various system architectures and to be combined with other complementary fault-tolerance strategies, thereby enhancing overall system availability and efficiency.

In summary, this thesis contributes a novel perspective on the integration of Byzantine fault tolerance in hard real-time distributed systems, laying the groundwork for more robust, flexible, and efficient real-time infrastructures.

1.3. Thesis Structure

Chapter 2 provides the necessary background and reviews the current state of the art. Chapter 3 introduces the proposed system model and presents its detailed analysis. Chapter 4 is organized into three key sections: the employed load-balancing technique (Section 4.1), the network scheduling strategy (Section 4.2), and the estimation of TAS cycle window durations (Section 4.3). Chapter 5 describes the testing methodology and discusses the obtained results. Finally, Chapter 6 draws the conclusions and outlines potential directions for future work.

2 | Background

In this chapter, we provide the necessary background and review the state of the art related to distributed hard real-time systems. We first introduce the fundamental concepts and technologies that form the basis of our work, and then analyze the most relevant approaches and solutions proposed in the literature. This overview serves to position our contribution within the broader research context and to highlight the existing gaps that the proposed model aims to address.

2.1. Network Standards

Distributed hard real-time systems require not only deterministic behavior at the local processing level, like in ordinary hard real-time systems, but also deterministic behavior in end-to-end communication. In recent years, standards such as *Time-Sensitive Networking* (TSN) and *Time-Triggered Ethernet* (TTEthernet) have introduced new mechanisms for managing time-critical traffic over Ethernet, enabling predictable communication in domains such as automotive, avionics, and industrial automation.

2.1.1. Time-Sensitive Networking (TSN)

TSN is a set of IEEE 802.1 standards that extend traditional Ethernet to provide bounded latency, low jitter, and high reliability, enabling its use in industrial and hard real-time applications. TSN achieves time determinism through several key mechanisms:

- **Time Synchronization (IEEE 802.1AS) [18]:** Ensure that all devices on the network share a common sense of time using a variant of the Precision Time Protocol (PTP).
- **Redundancy and Reliability (IEEE 802.1CB) [17]:** Allows seamless redundancy via frame replication and elimination.
- **Network scheduling:** TSN provides a set of standard techniques for scheduling time-critical traffic over Ethernet [19]. The primary TSN schedulers are:

- **Time-Aware Shaper (TAS – IEEE 802.1Qbv):** The Time-Aware Shaper is the cornerstone of TSN scheduling. It introduces Gate Control Lists (GCLs) on egress ports, which define precise time windows during which specific traffic classes are allowed to transmit. This enables time-triggered communication by isolating high-priority flows from lower-priority or interfering traffic.
- **Credit-Based Shaper (CBS – IEEE 802.1Qav):** The Credit-Based Shaper regulates the transmission of stream reservation (SR) traffic based on a credit mechanism. It ensures fair bandwidth allocation and limits burstiness while allowing co-existence with other traffic types.
- **Asynchronous Traffic Shaping (ATS – IEEE 802.1Qcr):** ATS is a rate-controlled scheduling mechanism that works without global time synchronization. It controls transmission time using local shaping queues and timers, ensuring predictable behavior in systems where time synchronization is difficult or expensive to achieve.

TSN is designed to support both time-triggered and event-triggered traffic, making it suitable for mixed-criticality systems, including automotive, robotics, and industrial control networks. For deterministic and real-time traffic, the Time-Aware Shaper (TAS) is generally more suitable, as several studies in the literature [3, 23] have demonstrated that the delay bounds achievable with Credit-Based Shaping (CBS) are often significantly higher and less predictable. This makes TAS a more appropriate choice for applications requiring stringent latency guarantees and bounded jitter.

2.1.2. Time-Triggered Ethernet (TTEthernet)

TTEthernet [8] extends Ethernet to provide fully deterministic communication with strict temporal guarantees, specifically targeting safety-critical domains such as avionics and space systems.

TTEthernet categorizes traffic into three classes:

- **Time-Triggered (TT):** Traffic is transmitted according to a static, globally synchronized schedule, ensuring zero congestion and collision-free transmission.
- **Rate-Constrained (RC):** Traffic has a bounded bandwidth and latency, suitable for periodic or low jitter streams.
- **Best-Effort (BE):** Traditional Ethernet traffic with no timing guarantees.

A key component is its Fault-Tolerant Synchronization Protocol, which ensures all network

nodes maintain a consistent global time base even in the presence of faults. The static scheduling of TT messages ensures determinism, while allowing coexistence with less critical traffic.

2.1.3. Comparison between TSN and TTEthernet

In [21] the authors presents a comparative study between Time-Sensitive Networking (TSN) and TTEthernet, with a focus on flow scheduling strategies and the factors contributing to communication delay in each approach. One of the main advantages highlighted for TSN is its ability to reserve bandwidth dynamically based on the actual demand of each flow, enabling a more stable and efficient transmission environment. Furthermore, while strict priority queuing (SPQ), a mechanism where higher-priority traffic is always served before lower-priority traffic, can lead to significant delays for low-priority flows during high traffic bursts, TSN addresses this limitation by incorporating credit-based shapers (CBS), which regulate the transmission of high-priority traffic and thereby reduce the starvation of lower-priority flows. Another key distinction lies in the scheduling model: TTEthernet relies on static time-triggered (TT) schedules that allocate fixed time slots to each TT frame at every node. In contrast, TSN employs a Time-Aware Shaper (TAS), which defines scheduling through time windows that specify whether a given traffic class, or a group of classes, is allowed to transmit at each point in time. This model provides greater flexibility and adaptability in response to configuration changes, positioning TSN as a more configurable and versatile solution for real-time communication networks.

2.2. Worst-Case Latency Analysis

In distributed hard real-time systems, computing worst-case latency is essential to guarantee that all timing constraints are met under all operating conditions. Several techniques have been developed to provide tight and analyzable upper bounds on end-to-end delays. One widely adopted approach for verifying the schedulability of real-time systems is *response-time analysis* (RTA) [1], originally developed for uniprocessor systems and later extended to distributed contexts. RTA computes the worst-case response time of a task by iteratively evaluating the time it takes from task release to task completion, taking into account the task's own execution time, interference caused by higher-priority tasks, and possible blocking due to shared resource access. The analysis guarantees that a task will always complete before its deadline if the computed response time is less than or equal to the specified timing constraint. In its classical form, RTA assumes a single processor with a fixed-priority preemptive scheduler and statically assigned task priorities. However, in

distributed real-time systems, tasks are often spread across multiple nodes and may involve inter-node communication. Consequently, RTA has been extended to accommodate these additional dimensions. Key contributions in this area include the work of Tindell and Burns [20], who extended RTA to support distributed systems by incorporating both local computation delays and communication latencies, such as those arising from CAN bus or TTP networks. Their framework models the network as a schedulable resource and includes the analysis of message transmission delays and synchronization constraints between distributed tasks. This distributed extension of RTA enables end-to-end timing analysis in systems where tasks are partitioned across processors and communicate via real-time networks, thus providing formal guarantees on the temporal behavior of complex, heterogeneous architectures. Another prominent method is *Network Calculus* [12], a formal framework based on min-plus algebra that models traffic flows and system components as arrival and service curves. This technique provides a compositional way to derive deterministic upper bounds on delay and congestion through network elements, such as switches and links. It is particularly effective for analyzing Time-Sensitive Networking (TSN), where traffic shaping and scheduling mechanisms like TAS or CBS are in place. In systems employing time-triggered architectures (e.g., TTEthernet or TSN with TAS), worst-case latency can be statically determined based on the global schedule. Here, latency computation reduces to extracting transmission offsets and evaluating flow-specific paths according to the predefined time slots. This method ensures full predictability but may require complex offline scheduling tools (e.g., SMT-based solvers) to guarantee feasibility and optimize latency.

Finally, some systems rely on formal verification techniques, where schedulability and timing correctness are proven through model checking or constraint solving. These methods offer high assurance but can be computationally expensive, especially in large-scale topologies or systems with dynamic behavior. In all these techniques, a central challenge remains balancing tightness of the latency bound with analytical tractability, especially in the presence of shared resources, contention, and mixed-criticality traffic.

In distributed hard real-time systems, the estimation of communication latency represents a fundamental challenge. Unlike soft real-time contexts, where occasional deadline misses can be tolerated, hard real-time systems impose strict temporal guarantees: every message must be delivered within a bounded and predictable time window. This requirement makes the analysis of end-to-end latency a critical component in the design and validation process. Latency in such systems arises from multiple sources, including message transmission over physical channels, queuing and scheduling delays at intermediate switches, and potential contention with concurrent traffic flows. The complexity of these

factors is further increased by the deterministic communication models required in safety-critical domains, such as avionics, automotive, and industrial automation, where worst-case bounds must be provably guaranteed. Traditional performance analysis techniques, which rely on average-case metrics, are insufficient in this context. Instead, methodologies such as network calculus, response-time analysis, and formal scheduling models are employed to derive safe upper bounds for message delays. These approaches allow system designers to capture the combined effects of channel capacity, packet size, hop count, and traffic shaping mechanisms, while ensuring that the guarantees provided are both conservative and applicable to the entire network.

2.2.1. Network Calculus Introduction

As previously stated in Section 2.2, *Network Calculus* is a mathematical framework based on min-plus and max-plus algebra, which provides deterministic guarantees on the performance of communication networks, that has become a widely adopted methodology for analyzing systems that require strict timing constraints, such as hard real-time distributed applications and Time-Sensitive Networking (TSN). The fundamental idea behind Network Calculus is to model traffic flows and network elements by means of bounding functions. On the traffic side, *arrival curves* characterize the maximum amount of data that can be injected into the network within any given interval. On the service side, *service curves* describe the minimum service a network element can guarantee to a flow. By combining these abstractions through algebraic operations, it becomes possible to derive safe upper bounds on critical performance metrics such as end-to-end delay, backlog, and output burstiness.

Given a traffic flow, its *arrival curve* $\alpha(t)$ provides an upper bound on the cumulative data that can arrive within any time interval of length t. Formally, if $A(t)$ denotes the cumulative arrival function, then:

$$A(t) - A(s) \leq \alpha(t-s), \quad \forall\, 0 \leq s \leq t. \tag{2.1}$$

Typical choices for $\alpha(t)$ include leaky-bucket models, where the arrival curve takes the affine form

$$\alpha(t) = \sigma + \rho t \tag{2.2}$$

with burst parameter σ and rate ρ.

On the network side, a system (e.g., a link or a switch) is abstracted by a *service curve* $\beta(t)$, which characterizes the minimum service guaranteed to the flow. If $D(t)$ denotes

the cumulative departure function, then the service guarantee can be expressed as:

$$D(t) \geq (A \otimes \beta)(t) \tag{2.3}$$

where $\otimes$ denotes the min-plus convolution. A common form of service curve is the rate-latency function:

$$\beta(t) = R \cdot (t - L)^+ \tag{2.4}$$

with guaranteed rate R and latency L, where $(x)^+ = \max\{0, x\}$

The power of Network Calculus lies in combining these two abstractions. Given an arrival curve $\alpha(t)$ and a service curve $\beta(t)$, one can derive deterministic bounds on the worst-case backlog and delay. In particular, the worst-case delay bound D_{max} is obtained as the maximal horizontal deviation between the arrival and the service curves:

$$D_{max} = \sup\{\tau \geq 0 \mid \alpha(t) \leq \beta(t + \tau),\ \forall t \geq 0\} \tag{2.5}$$

For the common case of a token-bucket constrained flow, where $\alpha(t) = \sigma + \rho t$, and a system providing a rate-latency service curve $\beta(t) = R \cdot (t - T)^+$, the maximal delay can be explicitly computed as:

$$\boxed{D_{max} = L + \frac{\sigma}{R - \rho}, \quad \text{if } \rho < R.} \tag{2.6}$$

Here, σ is the initial burst size, ρ is the sustained arrival rate, R is the guaranteed service rate, and L is the intrinsic system latency.

The interpretation of this expression depends on the type of flow:

- **Stable flow ($\rho < R$):** The system can absorb the burst and serve the ongoing traffic. The maximal delay is finite and given by the formula above.
- **Flow at the limit ($\rho = R$):** The system serves traffic exactly at the rate it arrives. Any non-zero burst leads to an unbounded delay, making the system effectively unstable.
- **Unstable flow ($\rho > R$):** The arrival rate exceeds the service rate. Backlog and delay grow without bound, and the system cannot guarantee finite latency.

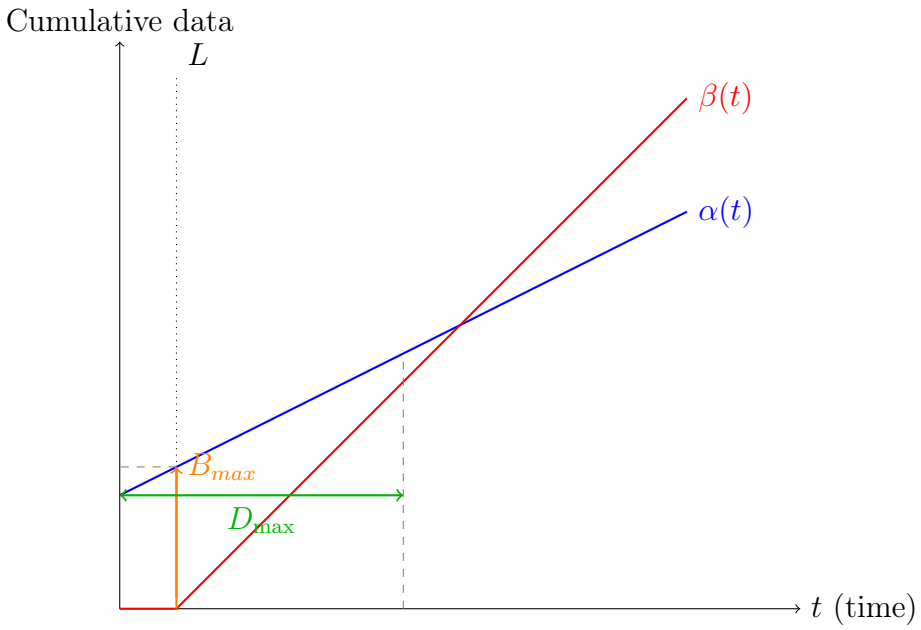


Figure 2.1: Geometrical Representation of the worst-case latency D_{max} in presence of the intrinsic system latency L.

Similarly, the backlog bound is obtained as the maximal vertical deviation:

$$B_{max} = \sup_{t \geq 0} \{\alpha(t) - \beta(t)\}. \tag{2.7}$$

In this way, Network Calculus provides a systematic method for deriving safe upper bounds on message latency across the network, thus enabling timing guarantees under worst-case conditions. Unlike stochastic or simulation-based approaches, Network Calculus focuses on worst-case guarantees, making it particularly suitable for safety-critical domains. Moreover, its compositional nature allows for modular analysis: the end-to-end performance of a flow can be obtained by successively applying service curve models across all the nodes and links along its path.

In Figure 2.1, we can observe a graphical representation of the curves $\alpha(t)$ and $\beta(t)$, together with the geometric interpretation of B_{max}, which corresponds to the maximum vertical distance between the two curves along the ordinate axis.

Regarding the geometric interpretation of D_{max}, since it is less evident in this figure, we refer to Figure 2.2, where the initial delay L has been excluded from the service curve $\beta(t)$. In this case, we can notice that the maximum horizontal distance between the two curves corresponds to D_{burst}, which is the latency introduced solely by the initial burst σ, while the abscissa of the intersection point between the two curves corresponds to D'_{max},

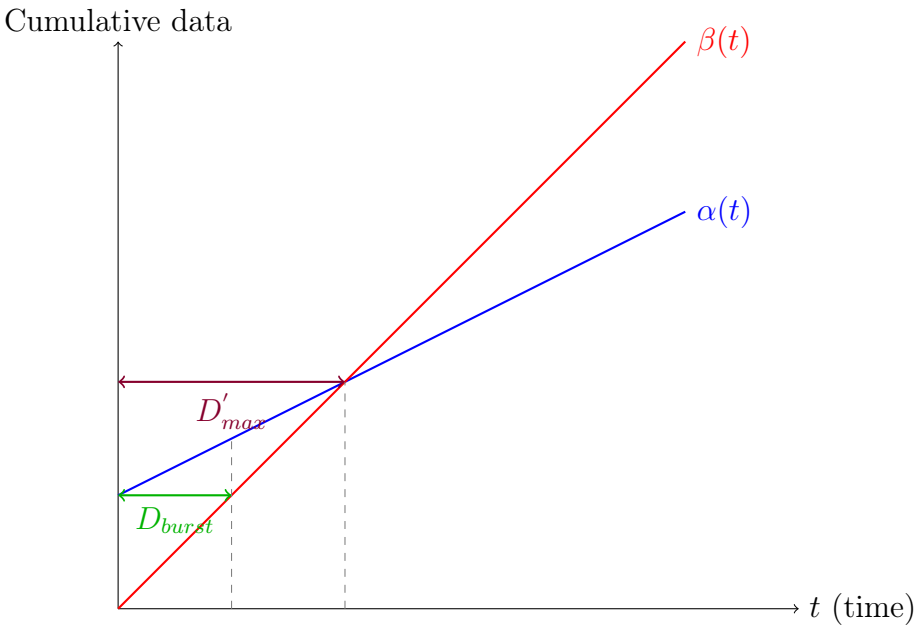


Figure 2.2: Geometrical Representation of worst-case latency D'_{max} excluding the intrinsic system latency L.

that is, the maximum delay determined by both the initial burst σ and the arrival rate ρ. Formally, D'_{max} can be defined as follows:

$$D'_{max} = \frac{\sigma}{R - \rho} \tag{2.8}$$

Consequently, the overall maximum delay can be obtained by simply adding L to D'_{max}, so:

$$D_{max} = L + D'_{max} \tag{2.9}$$

Worst-Case Latency Analysis for TAS using Network Calculus

In [22], the authors address the challenge of computing worst-case latency bounds in Time-Sensitive Networking (TSN), with a particular focus on networks that implement the Time-Aware Shaper (TAS) as specified by the IEEE 802.1Qbv standard. These networks are increasingly adopted in industrial and automotive systems not only for their ability to guarantee time-deterministic communication, but also due to their reliance on Ethernet technology, which is both widely standardized and supported by a broad range of existing hardware and tools. This dual advantage makes TSN a compelling choice for

safety-critical applications requiring predictable timing and interoperability within heterogeneous systems. The authors propose a formal method based on *deterministic Network Calculus* to derive tight upper bounds on end-to-end latency for time-triggered traffic. Their technique models the behavior of TAS through arrival and service curves, taking into account essential features such as cycle durations, guard bands, and gate control lists (GCLs), which regulate access to output queues based on priority and time windows.

The analysis demonstrates that Network Calculus can be effectively applied to perform rigorous worst-case timing analysis in TSN architectures with time-triggered scheduling, providing the formal guarantees required in safety-critical hard real-time systems. This study provided valuable insights into how worst-case latency can be computed within our model, given that our approach leverages a Time-Aware Shaper (TAS) to guarantee deterministic communication in the network. Moreover, since each task in our model is periodic and the configuration is computed off-line, the derivation of the arrival and service curves required by Network Calculus becomes straightforward. As a result, Network Calculus emerges as both a valid and well-suited analytical technique for our system.

2.3. Failure Model in Distributed Hard Real-Time Systems

According to the literature [10, 15] there are 3 main types of failure in distributed hard real-time systems:

- **Hardware failures** refer to permanent or transient malfunctions of physical components such as processors, memory units, sensors, or actuators. These failures typically manifest as fail-stop or crash failures, where a node ceases to operate and stops producing outputs. In safety-critical applications (e.g., avionics, automotive), such failures can compromise system-level timing guarantees if not detected and handled within bounded time.
- **Network failures** encompass communication-related faults such as packet loss, message corruption, link disconnection, and network partitions. In hard real-time systems, the temporal behavior of message delivery is as critical as its correctness. A delay in message transmission, even if eventually successful, is treated as a failure if it exceeds the system's deadline constraints.
- **Byzantine failures** are among the most challenging in distributed systems, especially in hard real-time contexts. These failures occur when a component behaves arbitrarily, such as sending conflicting, incorrect, or even malicious data to different

parts of the system. Unlike crash failures, Byzantine faults are non-detectable by simple timeout or omission checks, and they can lead to inconsistent system states, which are unacceptable in safety-critical domains. The challenge is compounded by the requirement to resolve such inconsistencies within strict timing bounds.

2.4. Fault-Tolerance Techniques in Distributed Hard Real-Time Systems

In the design of hard real-time distributed systems, ensuring reliability and fault tolerance is of paramount importance due to the critical nature of timing constraints and the potential impact of system failures. Various fault-tolerance mechanisms have been developed to address these challenges, each with specific trade-offs in terms of complexity, resource consumption, and applicability to real-time constraints.

Redundancy and Replication

Redundancy and replication are key techniques used to enhance reliability and availability. Redundancy refers to the addition of extra components, such as hardware modules or execution time, beyond the minimum required, allowing the system to tolerate faults by switching to backup resources. Replication, a specific form of redundancy, involves maintaining multiple identical instances of tasks or data across different nodes to ensure consistency and fault recovery. While both aim to improve fault tolerance, replication focuses on duplicating state or computation, whereas redundancy more broadly covers any surplus resource used to handle failures.

Checkpointing

Checkpointing is another widely used technique, especially for fault recovery. It consists of periodically saving the state of a process such that, in the event of a fault, execution can resume from a previously saved consistent state. While this technique introduces some runtime overhead, it provides an effective mechanism to restore correct operation without a full system restart, and can be optimized through careful timing and placement strategies.

Quorum-Based Approach

Quorum-based approaches leverage agreement among a subset of system components to tolerate faults and maintain consistency. In such schemes, a result or action is considered valid only if approved by a majority (or a qualified subset) of the participating

components.

Consensus

Consensus protocols aim to achieve agreement among distributed processes even in the presence of failures. In the standard formulation of the consensus problem, each process proposes a value, and the system must satisfy three core properties:

- **Termination:** Every correct (non-faulty) process eventually decides on some value.
- **Agreement:** No two correct processes decide on different values.
- **Validity:** If all correct processes propose the same value, then any decided value must be that one.

This form of consensus is sufficient for many distributed applications. However, in certain critical scenarios, especially those involving early decisions by faulty processes, a stronger condition is required: *Uniform Consensus*. Uniform consensus strengthens the agreement condition by requiring that all processes—whether correct or faulty—must decide on the same value if they decide at all. More precisely:

- **Uniform Agreement:** If any process (correct or faulty) decides on a value, then no other process (correct or faulty) can decide on a different value.

This distinction is subtle but crucial in systems where faulty processes may produce outputs before crashing. While standard consensus tolerates such behavior as long as correct processes remain consistent among themselves, uniform consensus demands global consistency of decisions, even from processes that may later become faulty.

2.5. State of the Art

Due to the constraints introduced by real-time systems, such the presence of deadlines, the limited amount of resources, or the need to reduce the resource consumption, over time many different approaches and studies have been presented in order to adapt those techniques to real-time distributed systems.

2.5.1. Bounded-time recovery

Bounded-time recovery (BTR) [4], rather than trying to mask the symptoms of a fault with massive redundancy, detects faults at runtime and enables the system to recover from them, by transferring tasks to other nodes that are still working correctly. The

key concept behind is that, when a fault does occur, there is a brief period of instability during which the system can produce incorrect outputs. However, as the authors stand, many cyber-physical systems (CPS) have physical properties, such as inertia or thermal capacity, that limit the rate at which the state of the system can change; thus, a very brief outage is often acceptable, as long as its duration can be bounded. The authors also introduce *Cascade*, a scheduling algorithm used to create a set of static task schedules used to handle different failure modes. Cascade take into account also the priority of the tasks, so high priority tasks are guaranteed to be schedule before,even at the cost, in case of multiple failures, of omit low priority ones. In this way Cascade also implements a static scheduler to handle mixed-criticality tasks. A *Mixed-Criticality Scheduler* addresses the challenge of executing tasks with different levels of criticality, while ensuring that all hard real-time constraints are met for the most critical tasks under all operational conditions. So in the event of system overloads, or faults, the scheduler must guarantee the execution of high-criticality tasks, possibly at the expense of low-criticality ones. BTR well address the problem of hardware failure, but it does not directly handle the byzantine one.

2.5.2. TERCOS and Multi-Component Architecture

There are also solution based on distributed task scheduling and replication. TERCOS [13] is a technique integrated with a fixed-priority-based scheduling algorithm, which makes use of the primary-backup scheme to tolerate permanent hardware failures. The innovating part is that it can also terminate the execution of active backup copies when corresponding primary copies are successfully completed, in order to enhance the task schedulability in fault-free scenario. A similar, but different, technique is presented in [5], where the authors designed an heuristic used to obtain a static, distributed and fault-tolerant schedule, by simply scheduling K additional replicas for each task, where K is the number of admissible failures, and statically calculates the main replica after each failure, in order to minimize the execution time. As already said, these two approaches are similar, both solve the hardware failure problem, their comparison is a perfect way to better understand the trade-off problem introduced by hard real-time distributed systems. TERCOS aim to optimize the resource consumption, sacrificing the level of replication, and consequently the number of admissible failures. While the second one, in order to handle the required number of failures, turns out to be less efficient in the resources usage.

2.5.3. TSNsched

Time-Sensitive Networking (TSN) aims to provide bounded latency and high reliability over Ethernet networks by leveraging time-aware scheduling mechanisms such as the

Time-Aware Shaper (TAS). However, generating valid TAS schedules is computationally complex, especially for large-scale networks with a mix of unicast and multicast flows, tight timing constraints, and intricate topologies. Manual configuration is infeasible in practice due to the combinatorial nature of the problem and the strict timing requirements involved. To address this challenge in [16] is presented *TSNsched*, a constraint-based tool that automates the synthesis of feasible and efficient TAS schedules. The tool transforms the scheduling problem into an *Satisfiability Modulo Theories* (SMT) formulation solvable via Z3.

The input parameters for TSNsched consist of the network topology, as well as the properties of switches, end devices, and the flows traversing the network. Based on these inputs, the tool generates a comprehensive set of constraints that characterize the scheduling problem and encodes them for the SMT solver Z3. Z3 then attempts to find a solution that satisfies all specified constraints. If the solver is unable to find a feasible schedule given the defined constraints, TSNsched issues a warning indicating that the problem is unsatisfiable. Conversely, when a solution exists, the tool extracts the relevant variable assignments from the solver's output, including:

- The start time and duration of the scheduling cycle for each switch.
- The start time and duration of time windows allocated for priority traffic transmission.
- The priority level assigned to packets at each hop along their routes.
- The initial transmission time of the first packet of each flow.

Thus, TSNsched takes as input a deterministic scheduling problem for high-performance periodic network traffic, defined by performance criteria such as maximum latency and jitter per packet per flow, and outputs the scheduling configurations for all ports of the TSN switches within the given network topology.

2.5.4. Byzantine-Resilient Real-Time Reliable Broadcast

Until now, we have examined various fault-tolerant approaches for addressing hardware failures in hard real-time distributed systems. Conversely, well-established solutions such as Paxos [11] and Raft [14] have been proposed to handle Byzantine failures. However, to the best of our knowledge, these techniques have not yet been adapted to the context of hard real-time systems. Alternative approaches have been explored using different mechanisms.

Byzantine-Resilient Real-Time Reliable Broadcast (RT-ByzCast) [9] addresses the challenge of designing a *real-time Byzantine Reliable Broadcast* (RTBRB) protocol capable of withstanding network uncertainties, faults, and adversarial attacks. The authors begin by demonstrating that implementing real-time Byzantine Reliable Broadcast (RTBRB) is impossible under classical assumptions, such as separating failure detection from broadcast logic and relying solely on traditional failure detectors—even perfect ones. To overcome this, the paper introduces RT-ByzCast, an algorithm that integrates failure detection directly into the broadcast mechanism. RT-ByzCast operates by using digital signatures, which are aggregated within a sliding time window to validate message consistency. Each process that participates in the broadcast signs the message it receives before forwarding it to others. These signatures are not simply collected, but rather aggregated within a sliding time window, which is a bounded period during which the system collects evidence of agreement on a message. This mechanism serves two key purposes. First, it allows processes to verify that a sufficient number of distinct, authenticated processes have received and validated a message. Second, the use of a time window ensures that this verification is not open-ended, the agreement must be reached within a predefined temporal bound, which is crucial for satisfying real-time constraints. The accumulation of signatures enables each process to construct a verifiable certificate showing that a message was endorsed by a quorum of participants within the time window. Only upon collecting such a certificate will a process deliver the message. This prevents malicious nodes from injecting or modifying messages without detection, and also ensures that message delivery is both fault-resilient and time-bounded. Furthermore, each process is equipped with a self-crashing mechanism: if a node detects that it cannot guarantee reliable delivery within the timing bounds (due to message loss or suspected misbehavior), it voluntarily terminates its own participation to prevent inconsistency and preserve the correctness of the broadcast. By tightly coupling timing, failure detection, and message validation, RT-ByzCast ensures that all correct processes either deliver the same message within a bounded time or crash to prevent further inconsistencies. This makes it particularly suitable for cyber-physical systems or safety-critical applications where both reliability and real-time guarantees are essential. Even if the algorithm provides a reliable end result in real-time, it is not well suited for hard real-time systems, since, standing to the authors, it allows transient timing violations.

2.5.5. Interactive Consistency in Real-Time Distributed Systems

Interactive Consistency in Real-Time Distributed Systems (In-ConcReTeS) [6] is a key-value datastore, which addresses the problem of achieving *interactive consistency* in distributed real-time systems in the presence of Byzantine faults. Interactive consistency is a fundamental problem in distributed systems, requiring all non-faulty processes to agree on the same set of values initially proposed by each process, despite the presence of faulty or malicious actors. This property is crucial for safety-critical applications such as avionics or industrial automation, where consistent global state knowledge is a prerequisite for deterministic system behavior. To solve this problem under real-time constraints, the authors propose a protocol that operates under a *partially synchronous system model*, where communication delays are eventually bounded, and where the system relies on a *time-triggered communication schedule*. Specifically, it assumes that each node transmits during a predefined communication slot, and that messages are authenticated using *digital signatures*, ensuring origin integrity and non-repudiation. The protocol is structured into three communication rounds:

- **Round 1 – Dissemination:** Each node sends its own initial value, along with a digital signature, to all other nodes. This ensures authenticity and prevents impersonation.
- **Round 2 – Echo:** Each node forwards to others the signed values it received in the first round. This propagation allows all correct nodes to observe what others have received, introducing redundancy that is useful to detect Byzantine behavior.
- **Round 3 – Resolution:** Each node processes the information received in the previous rounds and decides, for each sender, which initial value to consider valid. The decision rules are designed to tolerate up to f Byzantine faults, ensuring that correct values from non-faulty nodes are consistently accepted, while values from faulty nodes are either consistently rejected or flagged as uncertain.

Thanks to its combination of deterministic communication, cryptographic authentication, and structured redundancy, In-ConcReTeS succeeds in solving interactive consistency in environments with strict timing constraints, making it a practical and robust solution for embedded real-time distributed systems affected by Byzantine faults. However, when applied to hard real-time distributed systems, In-ConcReTeS faces several critical challenges. The protocol requires three rounds of communication among all nodes. In each round, messages must include digital signatures and, in certain cases, replicas of

messages received from other nodes. In hard real-time distributed systems, where both bandwidth and processing time are severely constrained, this overhead poses significant challenges. Consequently, meeting the stringent timing budgets allocated to each task becomes difficult, potentially compromising the system's real-time guarantees.

2.5.6. Fast Failure Detectors

An approach to the consensus problem is presented in [7], where the authors introduce *Fast Failure Detectors* (Fast FDs) a class of failure detectors characterized by worst-case detection times that may be significantly shorter than the maximum interprocess message delay D. Building upon this concept, they propose *Fast Uniform Consensus* (Fast UC), a solution to the real-time Uniform Consensus problem. The *Uniform Consensus* (UC) problem is a fundamental agreement challenge in distributed systems, requiring that all processes, whether correct or faulty (if they decide before crashing), agree on the same value. This extends the classic consensus definition by reinforcing the agreement condition. Let's call t the maximum number of process crashes tolerated, the authors demonstrate that the worst-case termination time of Fast UC is, for most practical system configurations and typical values of t, less than or equal to D, making it a compelling candidate for real-time consensus. However, despite the efficiency of this failure detection approach, the proposed solution does not address failure recovery, nor does it support load balancing or task re-scheduling in the presence of faults.

2.5.7. Hydra

Another possibility is presented in [2], where a task replication tool, named Hydra, is used to obtain fault-tolerant tasks, through the replication of parts of their code. The system model is composed by N sites interconnected by a network and each of them is also connected to a sensor and an actuator. Furthermore each task is represented as a Directed Acyclic Graph (DAG), in which vertices correspond to computational units and edges represent precedence relationships and data dependencies among them. A set of attributes can be specified for each task, such resource sharing between them, time constraints, their placement and required replication strategy. Hydra enhances fault tolerance in real-time tasks by replicating selected portions of their code, a process performed off-line through task graph transformations. Given a set of subgraphs (patterns) and associated transformation schemes, Hydra generates multiple copies according to a specified replication level. These replicas are interconnected using fault-tolerance building blocks, such as consensus or state backup mechanisms. The framework supports various replication strategies, including active, passive, semi-active, and temporal replication, offering flexibility in both

error detection and recovery. As stated by the authors, there is a possible cause of non-determinism not handled by Hydra. This source of non-determinism comes up in case of execution in a different order on different sites of replicated computations which modify some common resource and can be solved by the scheduling algorithm. Also this approach has the same criticality of the technique presented in section 2.5.6.

To ensure the correctness of the replication strategies implemented by Hydra, the underlying system must conform to the assumed fault model, which includes fail-silent nodes, bounded message omissions, and reliable actuators. To achieve this a run-time support layer must be introduced. The run-time support must include monitoring mechanisms to detect timing faults, such as deadline violations and deviations from expected task arrival patterns. Furthermore, it has to capture hardware exceptions and verify implementation-specific invariants to identify value faults at the processor level. Detected errors result in stopping the corresponding node, thus maintaining the fail-silent model. Although Hydra operates at a different level of abstraction, relying on a dedicated run-time support for distributed real-time systems, its methodology exhibits several conceptual similarities with the approach proposed in this work. Notably, both strategies employ task-level redundancy as a means of tolerating Byzantine failures and explicitly integrate task scheduling across computational nodes into the overall fault-tolerance framework.

2.6. Limitations of the State of the Art

As discussed in Section 2.5.2, a clear trade-off exists between resource consumption and the level of replication. TERCOS exemplifies this trade-off by limiting the degree of replication to enhance resource efficiency, while still providing a basic fault recovery mechanism through the selective termination of active backup copies. Conversely, the approach proposed in [5] emphasizes fault tolerance by statically replicating each task K times to tolerate up to K faults. However, it lacks any recovery mechanism, leading to consistently high resource utilization regardless of actual fault occurrences. In this context, *Bounded-Time Recovery* (BTR) offers a compelling middle ground. It supports greater fault tolerance and recovery flexibility by adapting resource usage based on task priorities and system requirements. Unlike the aforementioned methods, BTR enables the dynamic reallocation of tasks upon fault detection. Nevertheless, this dynamic reallocation is governed by a set of static task schedules that are precomputed off-line prior to system deployment. This allows BTR to combine real-time fault detection and recovery capabilities with predictable system behavior, thereby achieving a balance between resource usage and fault tolerance in hard real-time distributed systems.

It is important to note, however, that none of these approaches provide a mechanism for addressing Byzantine failures. While they effectively manage permanent hardware faults and offer fault recovery strategies, they fall short in scenarios involving arbitrary or malicious faults. Furthermore, although some of the approaches discussed address Byzantine failures in distributed real-time systems, they often do so in isolation, without incorporating explicit fault-recovery mechanisms for hardware-level faults, whether permanent or transient. These studies typically focus on consensus or replication strategies designed to tolerate arbitrary faults, while overlooking the physical reliability of the system components. In contrast, the approach proposed in this work explicitly integrates recovery mechanisms for both permanent and transient hardware faults, alongside Byzantine fault tolerance. This dual-layered fault management significantly enhances overall system dependability, ensuring correct operation not only in the presence of arbitrary or malicious behavior but also in the event of physical component failures.

So it can be concluded that, although numerous studies have addressed fault tolerance in hard real-time distributed systems, there remains a significant opportunity to investigate and enhance consensus and, in particular, quorum-based techniques in this context.

2.6.1. Consensus vs Quorum-Based Approach

Consensus-based approaches ensure that all non-faulty processes agree on a single decision value, thereby providing strong consistency guarantees. In the case of uniform consensus, this agreement extends even to faulty processes that decide before crashing. Such guarantees are particularly useful in safety-critical applications, where inconsistencies between nodes may lead to hazardous outcomes. However consensus may result overly conservative and, in certain real-time scenarios, unnecessarily strict. On the other hand, quorum-based approaches offer a more flexible and potentially more time-efficient alternative. Rather than requiring all correct processes to agree on a single value, these techniques rely on the assumption that a subset (quorum) of nodes is sufficient to determine the system's state or to validate an output. This allows for partial agreement and the possibility of proceeding with computation even if some nodes fail or behave erroneously. Quorum-based mechanisms can be tailored to tolerate both crash faults and Byzantine faults, depending on the quorum size and the redundancy of replicated state. Importantly, in hard real-time systems, quorum-based strategies can reduce the latency associated with decision-making by allowing the system to continue operation as long as a quorum responds within the time constraints. For instance, in a system composed of three nodes executing the same task, if only one node produces an incorrect output, a consensus-based approach would discard the result entirely. In contrast, a quorum-

based strategy could identify and correct the faulty output without requiring full task re-execution. This distinction becomes critical in hard real-time systems, where time constraints may preclude the possibility of recomputation.

3 | System Model

In this chapter, we describe the adopted model, outlining the assumptions made regarding the model architecture, followed by a detailed explanation of its operation. The model architecture is composed by an arbitrary number of nodes, each of which runs a variable number of tasks. The nodes are connected by a network.

Task Model:

Each task is periodic and has a constant deadline equal to its period. We assume that all deadlines are harmonic with respect to each other, meaning that each longer deadline is an integer multiple of the shorter one(s). A task has both active and passive replicas, where with task replica we mean an instance of that task, which runs on a node. Active replicas are the one that are actually executed, while a passive replica is the backup instance, which will substitute an active one, while it is down. Furthermore a quorum size and compute time has to be specified for each task. The quorum size represents the number of task active replicas that have to run over the nodes, in order to achieve a quorum. All tasks that are part of a quorum communicate with all other active members. The compute time is the required time needed to complete the execution of the task. We call *state*, the result produced by a task at the end of its execution. The computation of the next state may depends on the previous N ones, in this case every task has to store the last N required states. We also call *current state* the last produced state of a task. Once a state is produced, a message containing it will be sent to the other active members of the quorum. Since the architecture doesn't implement a multicast, multiple copies of the same message will be sent through using a unicast. We assume that the complete communication of those messages has to be considered as atomic.

Node Model:

A node is a physical component, which sequentially runs a variable number of active, or passive, task replica and it can run only one between an active or passive replica of each task. The task execution order is defined by a static scheduling criterion, which is common for every node in the model, which will be described in Chapter 4.2. We assume that

every node has a limited amount of resources, where the most relevant resource for our model is the compute speed. Since we assume that the tasks are executed sequentially, we can omit concurrency problems between node's running tasks. The computation of a task can, and most likely, will be stopped and resumed later, so every node must implements a preemption mechanism. We also assume that all nodes in the system are homogeneous, so they provide the same computational and communication capabilities.

Network Model:

The proposed model can handle any type of network topology, provided that the network is TSN-based, each switch implements TAS, and every node is reachable from all other nodes, even if not through a direct link. In future work, this criterion could be extended by adapting the model to CBS as well; however, at present, we consider TAS to provide stronger guarantees compared to CBS, as previously discussed in Section 2.1.1. Naturally, different topologies will affect the performance and outcomes of the model in different ways. Furthermore, since the computation of TAS transmission windows relies on network calculus, the estimation of the maximum worst-case latency must also be adapted accordingly (for further details, see Chapter 4.3). Finally, regarding communication channels, we assume that they are full-duplex and, for the sake of simplicity, that all channels operate at the same transmission rate.

Failure Model:

As previously discussed in Section 2.3, there are three main types of failures in distributed hard real-time systems: Byzantine failures, hardware failures, and network failures. Since the primary focus of this thesis is to design a model capable of employing a quorum-based approach to handle Byzantine failures and to leverage its properties in order to also perform failure detection and recovery in the case of both transient and permanent hardware failures on the nodes, the model explicitly assumes the possibility of such failures occurring. By contrast, as mentioned in Section 1.1, standardized techniques already exist for managing network failures. Therefore, this type of failure is not considered within the proposed model.

Let Q_i and T_i denote, respectively, the quorum size and the task period of the i^{th} task. We assume that, during each T_i, each task may incur at most $\lfloor Q_i/2 \rfloor$ hardware and/or byzantine failures, of which only one can be a hardware failure. Byzantine failures are directly corrected by the quorum, since the majority of the voters agree on the proposed value. On the other hand, the model guarantees the correctness of the results produced in presence of maximum one hardware failure per time. In case of multiple persistent fail-

ures, and in line with the aforementioned assumption, the system will undergo a *graceful degradation*. Specifically, as long as at least one member of the quorum of each task remains available, the system will continue to produce results, though it can no longer guarantee their correctness.

3.1. System Model Description

For the time being, it is assumed that each node stores, from the very beginning, the following information:

- the set of active task replicas to be executed; these are the tasks that the node must execute under fault-free conditions.
- the set of passive tasks to be executed; these represent the backup tasks that will be activated whenever a node executing the corresponding active replica fails.
- for each active task, the node must also be aware of the quorum size associated with that task, as well as the other active members of the quorum.
- for each active task, the node must also store a map that associates each active node in the quorum with the node executing its corresponding backup replica. This aspect will be clarified further in Chapter 4.1.

That being said, the model guarantees correct operation as long as the quorum of each task is complete and all constraints defined in Section 3 are satisfied. A complete quorum is defined as the case in which the cardinality of the union set of the non-faulty active replicas and the passive replicas, which have taken over in place of the corresponding faulty active replicas, of the i^{th} task is equal to Q_i. We define T_c as the *Network Cycle*, that is, the duration of the TAS cycle, as follows:

$$T_c = \min(T_i)$$

Each TAS cycle is divided into two windows:

- **Quorum agreement window**, during which nodes execute tasks and transmit the agreement messages required for quorum formation.
- **State recovery window**, during which recovery messages are transmitted in case a failure has occurred.

In Figure 3.1, we can observe the task state diagram. The behavior of each task during

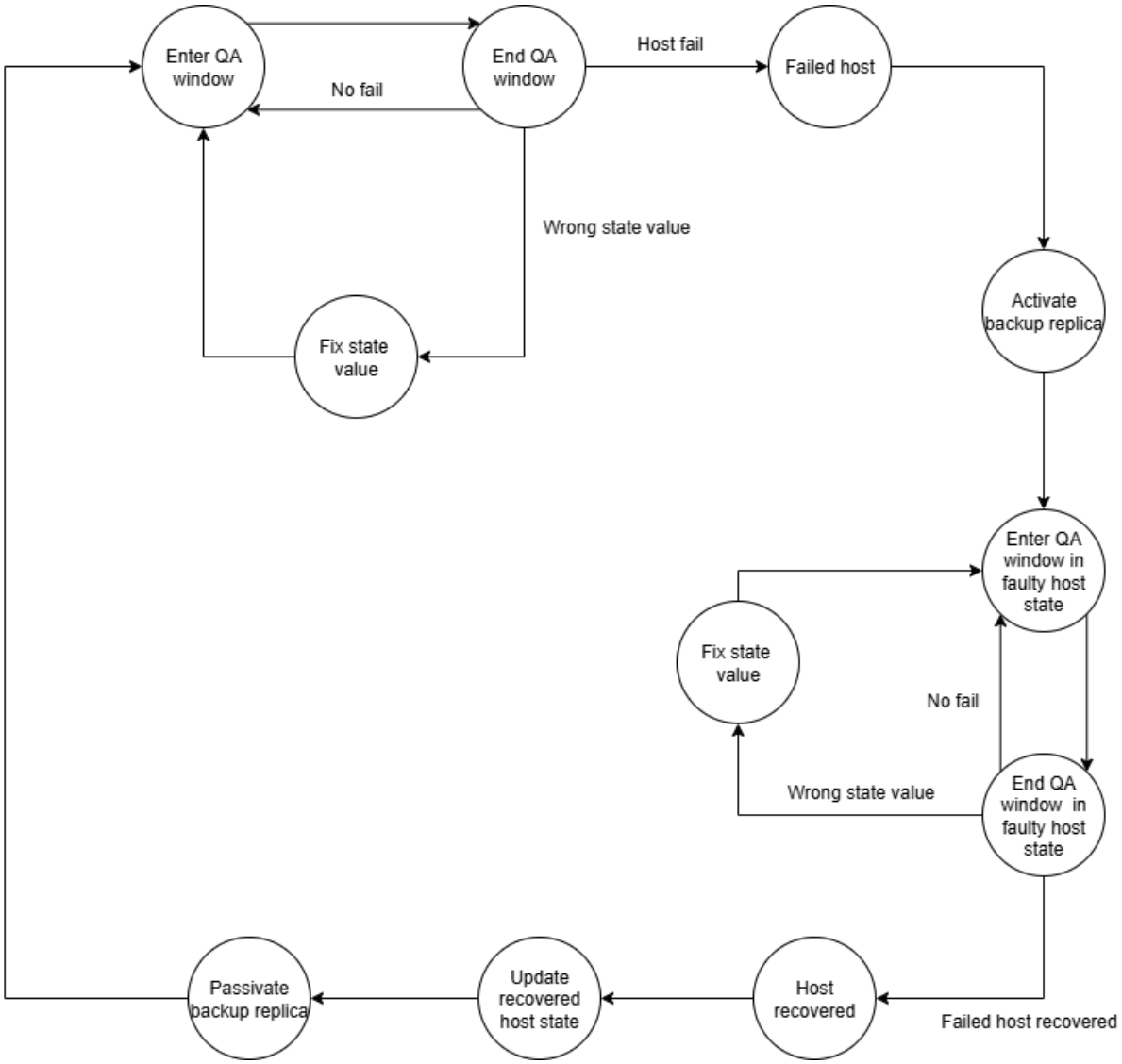


Figure 3.1: Task State Diagram.

the two windows will be described in detail in Sections 3.1.1 and 3.1.2.

Both agreement and recovery messages are considered high-priority traffic. They belong to the hard real-time traffic category, whose role is to guarantee the correctness and availability of the system (high-priority traffic will be further detailed in Sections 3.1.1 and 3.1.2). In addition, the model assumes the presence of low-priority traffic, referred to as *best-effort* traffic. This class includes all messages that are not required to ensure the correct functioning of the system and therefore must be treated as secondary. The TAS Gate Control List (GCL) allows best-effort traffic to be transmitted throughout the entire cycle. This design choice enables low-priority traffic to utilize idle periods of the network, that is intervals in which no high-priority messages are being transmitted, thus maximizing overall network utilization. Since the priority of hard real-time traffic is

strictly higher than that of best-effort traffic, the latter does not cause conflicts; instead, its transmission is simply delayed until the channels are free from high-priority traffic.

3.1.1. Quorum Agreement Window

At the beginning of the quorum agreement window, each node starts computing sequentially all active tasks (both active replicas and backup replicas that have been activated) scheduled on it. Each task is executed for a certain period of time, referred to as the *Time Slice*. At the end of this interval, if the task has been completed, the node sends a quorum agreement message containing the result produced by the task. Otherwise, the computation is interrupted (to be completed later) and the node proceeds to the next task, if any.

Let i denote the i^{th} task, where C_i is the compute time and N_T^i represents the maximum number of T_c periods required for completing task i, defined as follows:

$$N_T^i = \frac{T_i}{T_c}$$

The model provides two possible definitions of the time slice, denoted respectively as S_i and S, which are computed as follows:

$$S_i = \frac{C_i}{N_T^i} \tag{3.1}$$

$$S = \max(S_i) \tag{3.2}$$

Equation 3.1 defines the *Fit Time Slice*, a fitted time slice for each task, which aims at completing the execution of task i within exactly N_T^i periods. Using this type of time slice avoids wasting time between the execution of tasks, thus increasing the number of tasks schedulable on each node. However, depending on the system under consideration, this method may be more complex to implement. On the other hand, Equation 3.2 defines a maximum time slice, equal for all tasks. This approach may be simpler to implement, depending on the context. In this case, each task is guaranteed to be completed within N_T^i periods, or even fewer. Nonetheless, since some tasks may be allocated more time than actually required, this approach reduces the schedulability of tasks on the nodes.

As previously mentioned, once a task has been completed, after at most N_T^i periods, $Q_i - 1$ identical messages containing the produced result are sent to the other members of

the quorum. The transmission of these messages is assumed to be atomic, since the model does not implement a multicast mechanism. In addition, each agreement message also contains a *Sequence ID* field, which is an integer incremented after every transmission of a task's agreement traffic. This allows each quorum to detect whether one of its members is proposing an outdated state. Because the quorum agreement phase involves both task execution and the transmission of their corresponding messages, it is assumed that the duration of the quorum agreement window is sufficiently large to allow each node to complete its computations and the network to deliver all produced messages. The sizing of the quorum agreement window will be analyzed in detail in Chapter 4.3. At the end of the agreement phase, each task replica performs a check on the values produced locally and those received from the other quorum members, and acts accordingly:

- If the proposed value differs from the one voted by the quorum majority, the incorrect current state is corrected, thereby handling the Byzantine fault.
- If no message has been received from one of the quorum members, then, given the assumption of reliable channels, that member is considered faulty. In this case, the task is considered as faulty and the recovery procedure must be initiated.
- If one of the proposed values has its corresponding Sequence ID lower than the others, this indicates that the message was sent by a faulty node that has become operational again. In this case, a recovery procedure must also be performed.

3.1.2. State Recovery Window

As introduced in Section 3.1, the state recovery window is intended to manage the recovery of faulty tasks and, consequently, is used only in the event of a hardware failure. In fault-free situations, no high-priority messages are transmitted, and therefore best-effort traffic fully occupies the network for the entire duration of the window. We now proceed to describe the behavior of the system in the presence of a hardware failure. The model defines three types of high-priority messages, namely:

- **State Recovery Message:** a message containing all the information required for a task to be properly updated. In particular, it consists of the list of all past states necessary to compute the new current state.
- **Host Fail Message:** a message notifying the corresponding passive replica that it must be activated, in order to replace the active replica that has encountered a failure, and it also acts as a state recovery message.
- **Host Recovered Message:** the dual of the host fail message. It notifies the

previously activated backup replica that it must return to a passive state. As can be easily inferred, this type of message is used only in the case of a transient node failure.

Once the message types employed by the model have been defined, we can describe how they are used.

At the beginning of the recovery phase, each task that has detected the crash of one of its quorum members sends a host fail message to the node executing the corresponding passive replica of the task. The recipient node is determined according to the mapping previously introduced in Section 3.1. At this point, each backup task receives $Q_i - 1$ host fail, where Q_i denotes the quorum size of the i^{th} task, and the -1 accounts for the member of the quorum that has failed. In this way, the backup replica is updated and also learns the identities of the other quorum members with which it must communicate. At this stage, the system is considered to be in the *Faulty Host State*. Consequently, no additional hardware failures can be tolerated, and both the workload distribution across nodes and the direction of network traffic are adapted in order to preserve correct operation. During the faulty host state, best-effort traffic is likely to be further reduced, so as to leave sufficient bandwidth for hard real-time traffic. Moreover, in the case of a transient hardware failure, the crashed node eventually becomes operational again, along with all its active task replicas, which resume sending agreement messages (during the designated agreement window) containing an outdated state (smaller Sequence ID). In this way, each task belonging to the corresponding quorum realizes that the replica has recovered. Consequently, during the first available recovery window, quorum members send both a state recovery message to the recovered member and a host recovered message to the backup replica that had temporarily replaced it. Once this process has been completed for all active tasks of the recovered node, the system exits the faulty host state and returns to its initial operational mode. As in the case of the agreement window, the recovery window must also be properly dimensioned in order to ensure that all transmitted messages are correctly delivered to their destination within the required time.

3.1.3. Transient Failure Task Synchronization

As discussed earlier, when a previously crashed node becomes operational again, it resumes the execution of its active tasks. However, it is not possible to know in advance whether and when a node will recover. Depending on the recovery time, the reactivation of a node may lead to a desynchronization among the replicas of the tasks belonging to a quorum. To better illustrate this situation, let us consider a task, denoted as $Task_1$,

whose quorum is composed of three replicas executed on the nodes H_1, H_2, H_3. In this example, we set $N_T^1 = 2$, which, as previously recalled, denotes the number of periods required by $Task_1$ to produce a new state, together with its corresponding time slice S_1. Let t_1 denote the time at which node H_1 crashes, and t_2 the time at which it recovers. Depending on the value of t_2, the two different behaviors can be observed in Figures 3.2a and 3.2b, where the rectangle in red represents the time slice in which the first part of the task is calculated, while the one in green is the time slice in which the task execution is completed, and, consequentially, its result is sent over the network.

In Figure 3.2a we can see that the node H_1 becomes operational again after H_2 and H_3 have already transmitted their current state, but before they start computing the next state. In this case, when H_1 resumes operation it remains synchronized with the other quorum members, and all replicas continue to transmit the results of $Task_1$ within the same period. Conversely, in Figure 3.2b, we observe that at the time H_1 recovers, nodes H_2 and H_3 have already partially computed $Task_1$. As a result, in the following period, H_1 will begin executing $Task_1$, while H_2 and H_3 will complete their computation and transmit the produced result. In this case, the quorum members are desynchronized, and if this situation is not properly handled, the desynchronization will persist.

For this reason, we introduce, when necessary, a waiting time denoted as T_w, before the transmission of agreement messages by a recovered node. Let T_x and t denote, respectively, the number of elapsed T_c periods and the elapsed time, where T_x is defined as:

$$T_x = \left\lceil \frac{t}{T_c} \right\rceil$$

We define T_w as the additional number of T_c periods that the recovered task must wait before transmitting its agreement messages. It can be computed as follows:

$$T_w = T_x \bmod N_T^i$$

Thus, when a node becomes operational again, each of its tasks must wait T_w additional periods before transmitting their agreement messages. Returning to the previous example, we obtain the scenario depicted in Figure 3.3, where node H_1 remains in the wait state for one period T_c, represented by the blu rectangle, before resuming agreement transmission.

3.1.4. Node Crash Analysis

Now, we proceed to analyze how the system behaves depending on the moment at which a node failure occurs. This can happen either during the agreement window or during the recovery window. In the latter case, the analysis is considerably simpler, since the faulty node will not transmit any messages until it becomes operational again; therefore, the other quorum members will detect its absence within the period of the respective task T_i. On the other hand, if the failure occurs during the agreement window, two possible cases may arise:

- The failure occurs before the node transmits the messages of a task.
- The failure occurs after the node has transmitted the messages of a task.

In the first case, the quorum will be able to perform failure detection within the same T_i, whereas in the second case, the detection requires the subsequent T_i, since the other quorum members will already have received the vote from the faulty node and will therefore still consider it as operational.

Example

To clarify this behavior, let us assume four nodes, namely H_1, H_2, H_3, H_4, and two tasks, $Task_1$ and $Task_2$, with quorum sizes $Q_1 = Q_2 = 3$, and the following configuration:

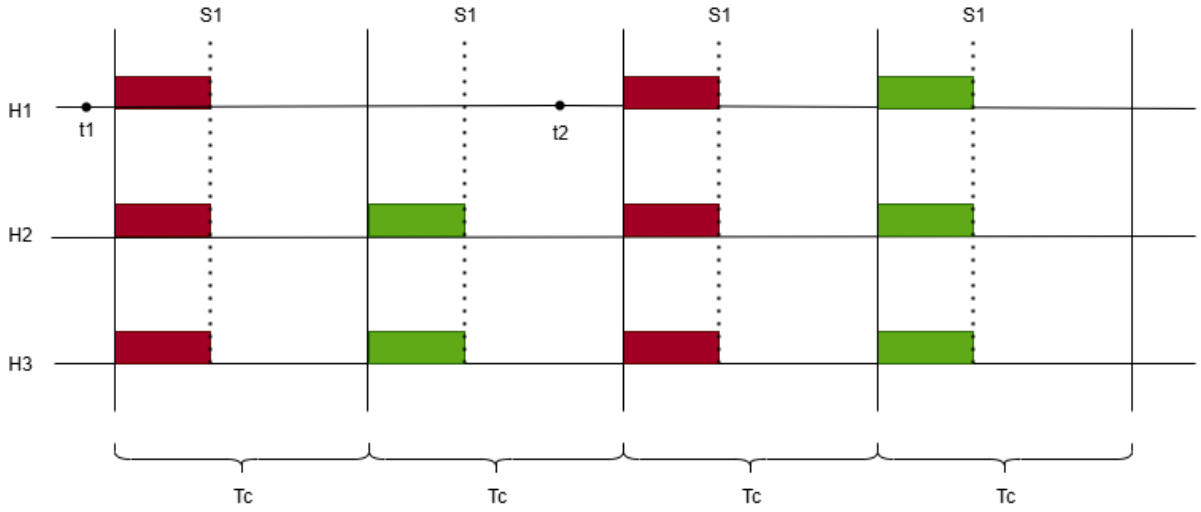
- H_1 and H_3 actively execute $Task_1$ and $Task_2$.
- H_2 actively executes $Task_2$ and passively replicates $Task_1$.
- H_4 actively executes $Task_1$ and passively replicates $Task_2$.

Figure 3.4 illustrates the initial configuration. For simplicity, we do not consider the network topology, since it is irrelevant for this analysis. What matters, as assumed by the model, is that all nodes are able to communicate with each other. Moreover, we assume $T_1 = T_2 = T$, and that both tasks compute and produce one result per period T_c.

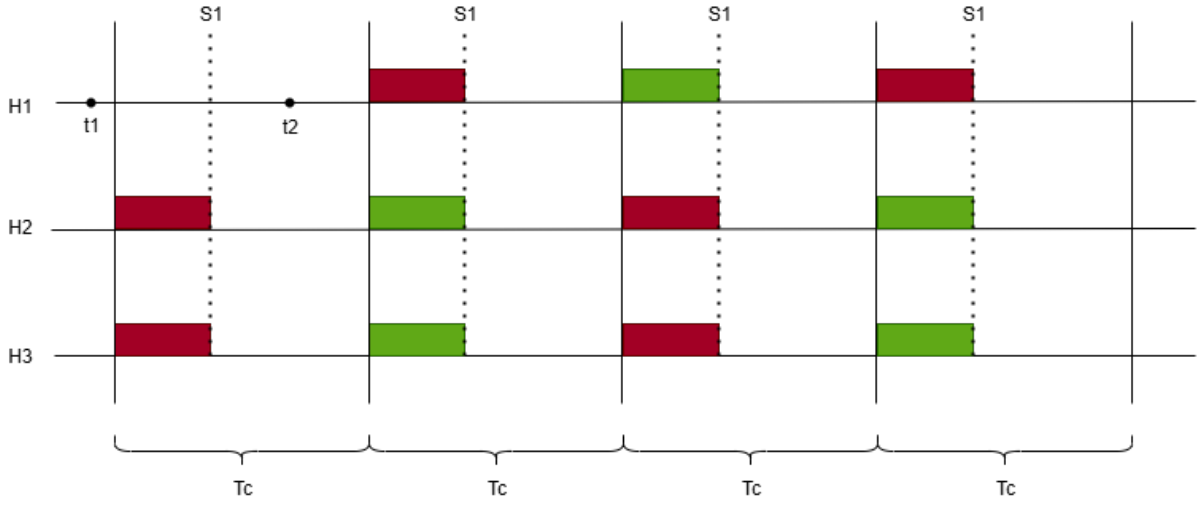
Now, assume that H_3 is the faulty node, and that it crashes after sending the messages related to $Task_1$, but before those of $Task_2$, as shown in Figure 3.5a, where blue arrows represent the traffic of $Task_1$ and red arrows represent the traffic of $Task_2$. From Figure 3.5b, we can observe that the quorum of $Task_2$ has already detected the failure of node H_3 and, consequently, activated the corresponding backup replica on node H_4. However, this is not the case for the quorum of $Task_1$, where nodes H_1, H_2 are still attempting to communicate with H_3. Finally, in Figure 3.5c, we see that during the third period T_c, the quorum of $Task_1$, having detected the failure of H_3 at the end of the agreement window

of the second period T_c, adapts by activating the corresponding passive replica on H_2.

This analysis reveals that the assumption made in Section 3, namely that each task can tolerate at most $\lfloor Q_i - 1 \rfloor$ failures within each of its periods T_i, is not entirely accurate. In fact, in this case, if a Byzantine failure had occurred in the quorum of $Task_1$ on one of the two nodes H_1, H_4 during the first period T_c, it would still have been correctly handled, despite the fact that a total of two failures occurred within the same period. This observation suggests a refinement of the failure-handling assumption: the system can tolerate at most $\lfloor Q_i - 1 \rfloor$ failures per task, not during each of its periods T_i, but within the same period T_i in which failure detection effectively takes place. Returning to our example, if the Byzantine failure were to occur not during the first period T_c but in the second, we would indeed face a critical situation outside the scope of the assumed failure model, since at the end of the second period there would be two conflicting values proposed within the quorum. A further observation from this analysis is that, in the case of transient failures, the system will, in the worst case, exit the faulty host state after at most $2 \cdot \max(T_i)$. This is because, as mentioned earlier, detecting a hardware failure requires at most $2T_i$ for a quorum, and it is possible that the faulty node executes the task with the largest T_i. It is important to emphasize that these refinements are relevant considerations, but they do not undermine the correctness guarantees provided by the proposed model.



(a) Synchronized Case.



(b) Desynchronized Case.

Figure 3.2: Transient Failure Task Synchronization.

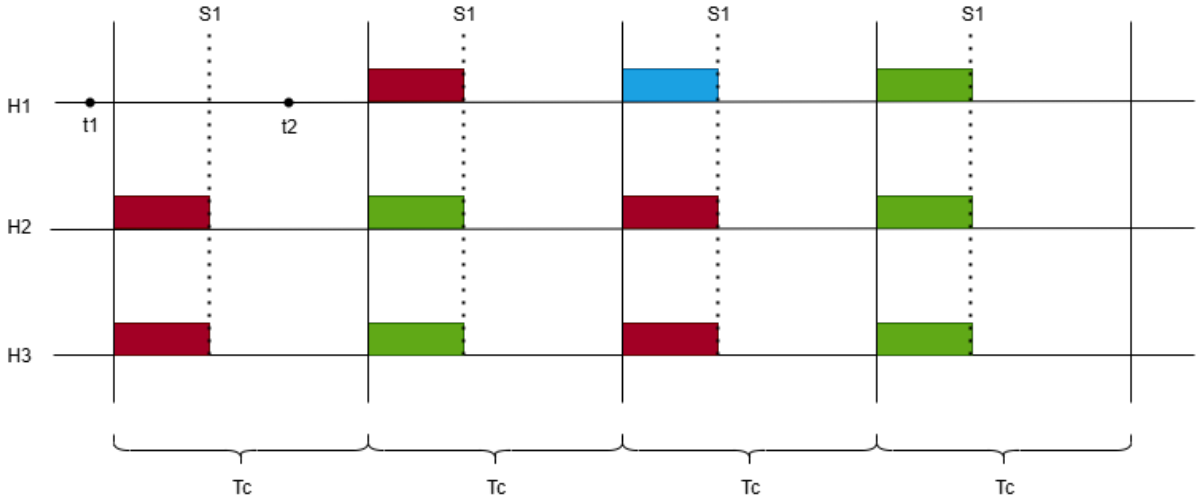


Figure 3.3: Desynchronized Case Fixed.

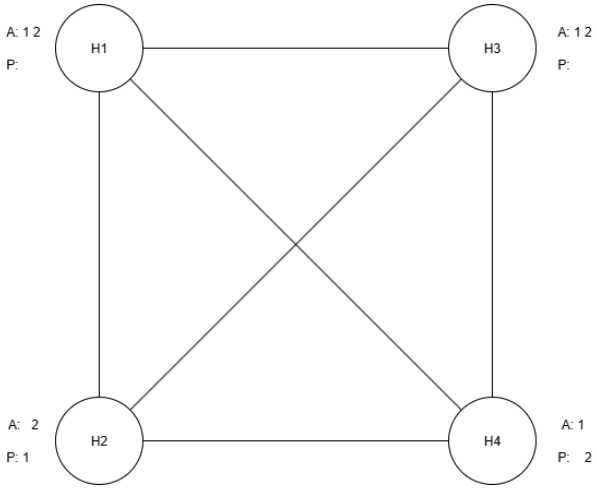
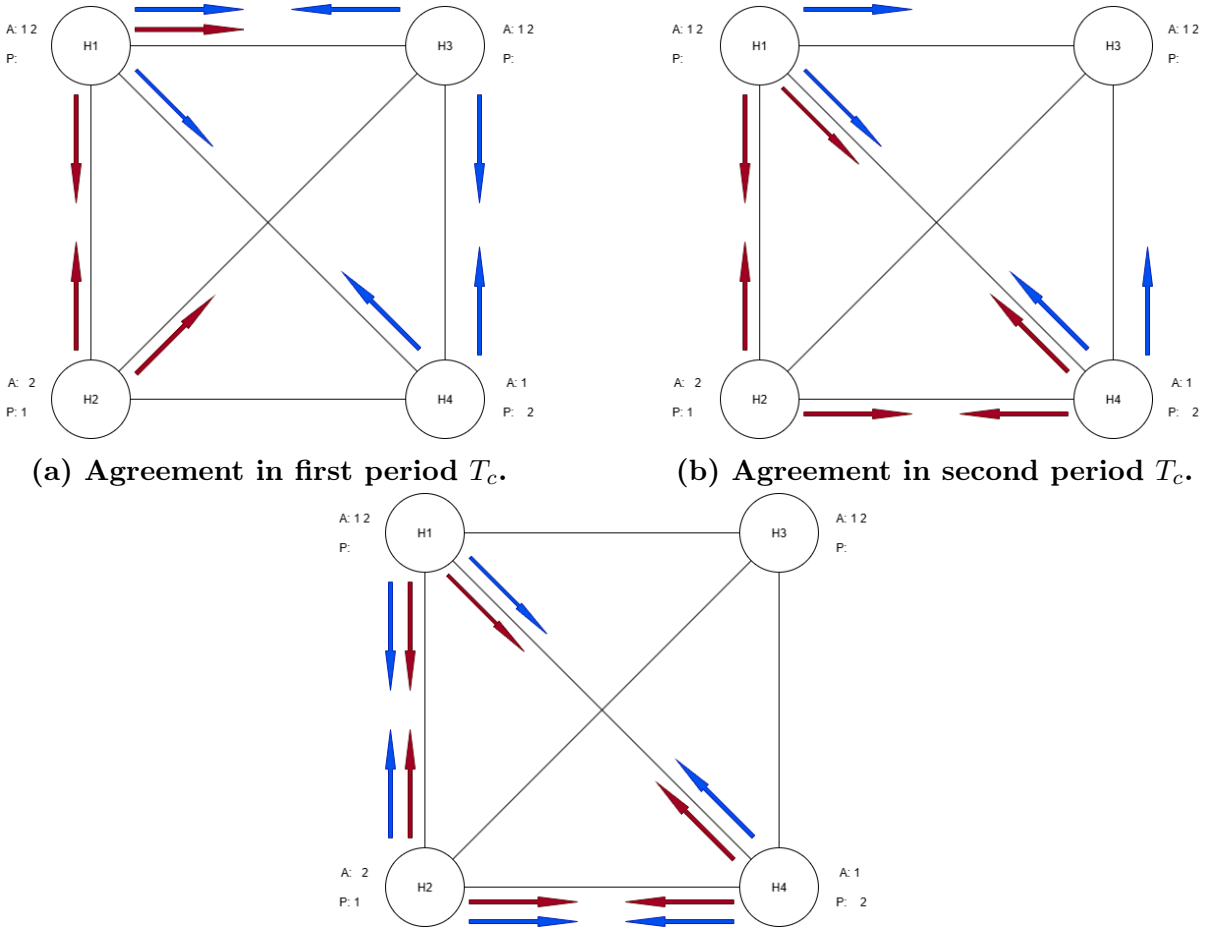


Figure 3.4: Initial Configuration.



(a) Agreement in first period T_c.

(b) Agreement in second period T_c.

(c) Agreement in third period T_c.

Figure 3.5: Example of a possible communication flow under hardware failure of H_3.

4 | Scheduling

This chapter is organized into three main sections. In Section 4.1, we present the load balancing algorithm adopted in this work. Section 4.2 discusses the network scheduling approach, highlighting the key aspects that must be considered for the design of a scheduling algorithm suitable for the proposed model. Finally, Section 4.3 illustrates how Network Calculus has been employed to estimate the duration of the quorum agreement and state recovery windows.

4.1. Load Balancing

The concept of load balancing refers to the fair and efficient distribution of workload (tasks) among the various nodes of the system, with the primary goal of ensuring that all deadlines are met. The main objective is to prevent certain nodes from becoming overloaded, which could result in missed deadlines, while others remain underutilized. Moreover, proper load balancing allows the system, in the presence of hardware failures affecting one or more nodes, to redistribute the workload across the remaining operational nodes without compromising the respect of critical timing constraints, thereby contributing to the construction of a fault-tolerant system.

4.1.1. Static vs Dynamic Load Balance

Load balancing can be approached using two main strategies: static and dynamic.

Static load balancing is based on the offline assignment of tasks to nodes, determined during the design phase. This approach has the advantage of guaranteeing predictability and ease of analysis, since the allocation remains fixed over time and can be verified through deterministic techniques such as schedulability analysis or, as in our case, network calculus. Furthermore, the absence of task reallocation mechanisms during execution reduces overhead and facilitates system certification, which is a crucial aspect in safety-critical domains such as avionics and automotive. However, the main drawback of this approach is its limited flexibility: workload variations, hardware failures, or unforeseen

execution conditions may compromise the satisfaction of deadlines, as the system is unable to adapt dynamically.

Dynamic load balancing, by contrast, allows tasks or messages to be reassigned during execution, adapting to failures or sudden variations in workload. This approach improves fault tolerance and enables better utilization of resources, as the system reacts to actual runtime conditions rather than relying on a predefined static scenario. However, the cost of such flexibility is a significant overhead, stemming from the monitoring of node states, the communication required to negotiate reassignments, and the time spent migrating tasks. In hard real-time systems, this overhead complicates the formal guarantee of deadline compliance, reducing the overall predictability of the system's behavior.

4.1.2. Static Load Balance Tool

Considering the observations discussed in Section 4.1.1 and since our model relies on a quorum-based approach, which is known to be less resource-efficient compared to other techniques, in order to maximize system determinism while minimizing communication and monitoring overhead, we have chosen to adopt a *static load balancing* strategy. To achieve this, we developed a simple load balance tool, implemented in Python, whose purpose is to compute the initial allocation of tasks to nodes, as well as the various configurations required for the system to adapt to failures of different hosts.

Input:

- the number of nodes in the system (N).
- the number of tasks to be executed (N_t).
- for each task, the corresponding compute time required for its completion (C_i).
- for each task, the corresponding deadline/period (T_i).
- for each task, the size of the quorum to which it must belong (Q_i).

Output (provided for each node):

- the list of active replicas of the tasks assigned to it,
- the list of passive replicas of the tasks assigned to it,
- a mapping that associates $task_i$ with a $node_B$. This mapping is used when $task_i$ detects that one of the members of its quorum has failed, and consequently it must contact the corresponding backup replica located on $node_B$.

The Algorithm

As introduced in the previous sections, exactly as for the quorum agreement window, the first step consists of computing the period T_c, the fit time slice S_i, and the common time slice S, which are defined as follows:

$$T_c = \min(T_i)$$

$$S_i = \frac{C_i}{N_T^i}, \quad S = \max(S_i)$$

where N_T^i denotes the maximum number of periods T_c required by task i to complete, computed as:

$$N_T^i = \frac{T_i}{T_c}$$

At this point, for each task we compute the respective utilization, that is, the fraction of a node's resources (in percentage) consumed by a replica of the task. This value depends on the chosen time slice and can therefore be obtained in two different ways, depending on whether S_i or S is adopted.

In the case of S_i, we obtain the utilization U_t^i, defined as:

$$U_t^i = \frac{S_i}{T_c}$$

while, in the case of S, the utilization U_t is computed as:

$$U_t = \frac{S}{T_c}$$

Once this preliminary phase is completed, the actual algorithm begins. The underlying idea is to first assign the Q_i active replicas of each task to the available nodes and, once this phase is completed, proceed with the assignment of the corresponding backup replica. In order to maintain a well-balanced workload distribution across the nodes, both active and backup replicas must be allocated incrementally to those nodes with the greatest amount of available resources.

Algorithm 4.1 Assign Task Active Replicas

```
1: for each t ∈ task_list do
2:     if node_list.size() ≥ t.quorum_size then
3:         for j = 1 → task.quorum_size do
4:             node_list[j].active_tasks.append(t)
5:             node_list[j].utilization ← node_list[j].utilization + t.utilization
6:         end for
7:         reorder node_list by increasing utilization
8:     else
9:         Error: not enough nodes available
10:    end if
11: end for
```

To achieve this, as we can see from Algorithm 4.1, we define two lists: one containing all the tasks, denoted `task_list`, and one containing all the nodes, denoted `node_list`. The algorithm then assigns Q_i replicas of the task at the head of `task_list` to the first Q_i nodes of `node_list`. If, for any task i, it is not possible to allocate all Q_i replicas to different nodes, then the algorithm raises an error, indicating that the given number of nodes N is insufficient, and terminates. This leads to the first constraint of the model:

$$N > \max(Q_i) \tag{4.1}$$

Constraint 4.1 states that N must be sufficiently large to ensure that all Q_i active replicas of each task can be assigned to distinct nodes (recall that a node can execute at most one replica, either active or passive, of a specific task), together with at least one passive replica, which is required to guarantee fault tolerance in the presence of a hardware failure. Once a replica of a task is assigned to a node, part of the node's resources is consumed. Consequently, the parameter representing the utilization of the node j must be updated by adding the corresponding task utilization. After all Q_i replicas of a task have been successfully assigned, the list `node_list` is reordered in ascending order based on the respective utilization. This step ensures that the nodes with the largest amount of available resources are always the first to receive a new replica, thus maintaining a balanced workload distribution across nodes. This procedure is repeated for every task in `task_list`.

Algorithm 4.2 Assign Task Backup Replicas

```
for each tintask_list do
    first_node ← first node ∈ node_list | t ∉ node.active_tasks
    if first_node ≠ None then
        first_node.backup_tasks.append(t)
        first_node.utilization ← first_node.utilization + t.utilization
    end if
    reorder node_list by increasing utilization
end for
```

The second step, as shown in Algorithm 4.2, consists in assigning the passive replicas of the tasks. For each task in the task list, the algorithm selects the first node that does not already contain the task among its active replicas. If such a node exists, the task is appended to its backup task list, and the node's utilization is incremented by the utilization of the assigned task. After each assignment, the node list is re-ordered based on utilization, ensuring that subsequent tasks are preferentially placed on less loaded nodes. This approach guarantees that backup replicas are consistently distributed across nodes, avoiding conflicts with active replicas.

At the end, the algorithm produces a table similar to the one shown in Table 4.1, where the input parameters are $N = 5$, $N_t = 3$, $Q_1 = Q_2 = Q_3 = 3$, and each task is assumed to have identical C_i and T_i values (for explanatory purposes, the actual values are not relevant in this example). Table 4.1 shows, for each node, the active task replicas to be executed, denoted without a superscript, as well as the backup replicas, indicated with the superscript P.

Table 4.1: Example of the final configuration table.

Host1	Host2	Host3	Host4	Host5
1	1	1	1^P	
	2^P	2	2	2
3	3		3^P	3^P

We have thus obtained a possible configuration that assigns the required number of active and passive replicas of each task to the nodes in the network, ensuring that the system can tolerate a hardware failure of any node. However, one final verification is still required. Indeed, our model assumes that each node has a limited amount of resources, which in this case constrains the number of operations it can execute per unit of time. Therefore, it is necessary to verify that each node can sustain the overall workload imposed

by the replicas (both active and passive) of the tasks it must execute. Let U_n denote the available utilization of each node, defined as:

$$U_n = \frac{T_c - T_{sr} - D_{max}}{T_C}$$

where T_c is the period, T_{sr} is the duration of the state recovery window, and l_{max} is the worst-case network latency (both T_{sr} and D_{max} will be analyzed in detail in Chapter 4.3). From this, we can derive the second constraint imposed by the model, where U_n^{jk} represents the utilization of node j in case of failure of node K:

$$\max(U_n^{jk}) \leq U_n \tag{4.2}$$

Constraint 4.2 states that, in order for the system to support a given configuration, there cannot exist a pair of nodes $\langle j, k \rangle$, with $j \neq k$, such that the sum of the utilizations U_t^i of the tasks executed by node j in the event of a failure of node k exceeds the available utilization U_n of that node.

For the sake of completeness, it is worth noting that the proposed load balancing and distribution algorithm, since it is not the primary focus of this work, is not optimal. In fact, the algorithm identifies only one of the possible configurations, which is not necessarily the best in terms of workload distribution, both under failure-free conditions and in the presence of failures. This limitation arises also because the impact of each node failure on the overall workload distribution is not explicitly taken into account.

4.2. Network Scheduling

Network scheduling refers to the set of techniques and mechanisms used to determine the transmission order and timing of messages over a communication network. Its goal is to ensure predictable and efficient utilization of shared communication resources, while respecting application-specific timing and reliability constraints. In the context of distributed hard real-time systems, network scheduling plays a fundamental role. Such systems rely on the timely and deterministic exchange of messages to guarantee the correct coordination of tasks distributed across multiple nodes. Any violation of communication deadlines may compromise not only system performance but also functional correctness and safety. Therefore, effective network scheduling is essential to provide bounded delays, prevent congestion, and ensure that critical messages are delivered within their deadlines, thereby enabling the system to meet stringent real-time guarantees.

4.2.1. Static vs Dynamic Network Scheduling

Network scheduling can be classified into two primary paradigms: *Static Scheduling* and *Dynamic Scheduling*.

Static Network Scheduling

Static network scheduling involves determining the schedule a priori at compile time, based on complete knowledge of task parameters such as execution times, periods, and deadlines. The principal advantage of this approach lies in its predictability and low runtime overhead, which is essential to guarantee that all hard real-time deadlines are met. Since the schedule is fixed, worst-case execution times (WCET) can be thoroughly analyzed, increasing the system's determinism and facilitating formal verification. However, such rigidity carries significant drawbacks, including lack of flexibility, the system cannot adapt to changes or variations in workload or task execution times, and inefficient resource utilization, especially when tasks deviate from their expected behavior or when supporting aperiodic tasks.

Dynamic Network Scheduling

In contrast, dynamic network scheduling makes decisions at runtime, adapting to the current system state, such as task arrivals, varying execution times, or communication delays. This flexibility allows dynamic schedulers, like Earliest Deadline First (EDF) or Least Laxity First (LLF), to more effectively utilize resources and handle aperiodic or unpredictable workloads, dynamically adjusting task priorities and offering better response under variable conditions. The trade-off, however, is increased runtime complexity and overhead, which can undermine timing guarantees. Moreover, the system becomes less predictable, making it more challenging to ensure that all deadlines are met under worst-case scenarios.

4.2.2. Network Scheduling Description

At this point, we proceed to present the network scheduling strategy adopted for our model. It should be emphasized that the proposed model does not mandate the use of a specific network scheduling policy. Any suitable scheduling technique can be adopted, provided that it is adapted to remain consistent with the framework introduced. The scheduling strategy presented in this work is not necessarily optimal; however, it highlights the key aspects that must be considered in order to evaluate which technique best fits the proposed model. Furthermore, it is particularly straightforward to implement, making

it a practical reference for demonstrating and reasoning about the impact of scheduling choices within the system.

Key Aspects

Given the considerations introduced in Section 4.2.1, we have decided to adopt a static network scheduler, which is recalculated whenever a node needs to activate its assigned passive replicas. This choice has been made in order to preserve the highest possible level of system determinism. Moreover, the rescheduling step is required because passive task replicas are not taken into account in the initial computation; therefore, when they are activated, a recalculation becomes necessary to correctly schedule the newly introduced replicas.

As already discussed, in distributed hard real-time systems the most critical resource to be managed is time, and this also applies to our case. In our model, the two main factors influencing the temporal dimension are the compute time of the tasks, or, more specifically, the task time slice S_i, and the transmission time (latency) tt_i of the messages sent over the network by each task. Consequently, in order to construct an optimal network scheduling strategy for the proposed model, both quantities must be taken into account. Moreover, as specified in Section 3, each task produces a result and generates network traffic only once its computation is completed. Therefore, as long as no task has finished execution, the network does not carry any hard real-time traffic. This observation highlights that, in the proposed model, the network scheduler should also determine the order of task execution, such that tasks with smaller task time slices S_i are executed first. In this way, hard real-time traffic starts circulating in the network as early as possible. Hence, in our case, the network scheduler also partially fulfills the role of a task scheduler on the nodes.

Implementation

After analyzing the key aspects we can now present the network scheduling strategy that has been implemented. The network scheduling strategy adopted in this work is based on a ratio-driven criterion. For each task i, we consider its task time slice S_i and the transmission time tt_i of the messages it generates upon completion. The scheduler assigns higher priority to tasks with a smaller ratio:

$$K_i = \frac{S_i}{tt_i} \tag{4.3}$$

Tasks are then ordered in ascending order of K_i, so that those with relatively low com-

Figure 4.1: Simple Network.

putation cost compared to their transmission demand are executed first. The underlying motivation for this criterion is to align task execution with the characteristics of network communication. By prioritizing tasks that are associated with more costly transmissions relative to their computation time, the scheduler enables network traffic to be generated and injected into the network as early as possible. This design choice reduces the likelihood of network congestion, promotes a more efficient overlap between computation and communication, and minimizes end-to-end latency.

This scheduling policy is not guaranteed to be optimal in the general case, since it does not account for global system constraints such as interference patterns across different nodes. Nevertheless, it captures the two essential aspects that significantly impact timing behavior in our model. By explicitly incorporating both factors into the scheduling decision, the criterion provides an effective heuristic that balances processing and communication delays. Furthermore, the approach is straightforward to implement, as it relies only on local knowledge of task execution times and message sizes, without requiring complex global coordination.

In summary, while the ratio-based scheduling criterion does not ensure optimality under all conditions, it highlights the critical trade-off between computation and communication. This makes it a useful reference for understanding the impact of scheduling choices in the proposed model and offers a simple yet effective policy that can be employed or adapted depending on system requirements.

Analysis

We can now proceed to analyse the proposed network scheduling strategy and show, through examples, how it improves network management and utilization. For reasons of simplicity and clarity, in these examples we consider only two nodes: a source node (the one actually analyzed in the examples) and a destination node, directly connected to each other, as shown in Figure 4.1. The conclusions drawn from this analysis remain valid regardless of whether or not this simplification is applied.

First Example

This example considers a single node that executes three tasks, namely $\{task_1, task_2, task_3\}$, with the following characteristics (where U denotes a generic time unit):

- $task_1$:
  - $S_1 = 1\,\mathrm{U}$
  - $tt_1 = 0.5\,\mathrm{U}$
- $task_2$:
  - $S_2 = 1\,\mathrm{U}$
  - $tt_2 = 1\,\mathrm{U}$
- $task_3$:
  - $S_3 = 1\,\mathrm{U}$
  - $tt_3 = 1.5\,\mathrm{U}$

All three tasks have the same task time slice and we assume each produces a result in every period T_c. As shown in Figure 4.2, at time $t = 4\,\mathrm{U}$ the quorum-agreement window closes; according to the model, all messages produced by the tasks must arrive before that deadline, otherwise a *missed deadline* occurs, which will most likely lead to a system failure. Analysing the first case (Figure 4.2a), where a simple FIFO (first-in, first-out) policy is used, we observe that although the messages from $task_1$ and $task_2$ reach their destination on time, the same does not hold for $task_3$. In fact, the traffic from $task_3$ would arrive at $t = 4.5\,\mathrm{U}$, resulting in a missed deadline for $task_3$ and thus a potentially critical system instability. In Figure 4.2b, where the proposed network scheduling policy is applied, all task traffic is delivered by $t = 3.5\,\mathrm{U}$, i.e. well before the end of the quorum-agreement window. A closer inspection, however, reveals that this ordering causes congestion on the communication channel for the traffic generated by $task_3$ and $task_2$. Indeed, when $task_2$ becomes ready to transmit its result, it must wait for the channel to complete the transmissions initiated by $task_3$, introducing an additional delay before $task_2$ (and subsequently $task_1$) can send their messages. Finally, in Figure 4.2c we account for the latencies introduced by channel congestion and observe that, despite these additional delays, all tasks still manage to produce and deliver their messages before the quorum-agreement window closes, i.e. approximately at $t = 4\,\mathrm{U}$.

We can therefore conclude that, for tasks sharing the same task time slice S_i but with different transmission times tt_i, the proposed network scheduler significantly increases the

number of tasks that can be scheduled successfully within the system. This improvement stems from the fact that the scheduler prioritizes tasks whose traffic requires larger transmission time tt_i, thereby reducing wasted idle intervals on the network. For instance, in the FIFO example of Figure 4.2a, the network remains idle for 0.5 U between the completion of $task_1$'s transmission and the start of $task_2$'s transmission; the proposed scheduler eliminates such gaps and better overlaps computation and communication.

Second Example

In this second example we again consider a single node executing three tasks $\{task_1, task_2, task_3\}$, but now with the following characteristics:

- $task_1$:
  - $S_1 = 1.5\,\mathrm{U}$
  - $tt_1 = 1\,\mathrm{U}$
- $task_2$:
  - $S_2 = 1\,\mathrm{U}$
  - $tt_2 = 1\,\mathrm{U}$
- $task_3$:
  - $S_3 = 0.5\,\mathrm{U}$
  - $tt_3 = 1\,\mathrm{U}$

In this scenario the tasks have different task time slices but identical transmission times. The same assumptions from the previous example apply: each task produces a result every period T_c, and the quorum-agreement window closes at $t = 4\,\mathrm{U}$ (see Figure 4.3). At first glance (Figure 4.3a) it appears that all tasks complete their computation and transmissions before the quorum deadline. However, applying the same reasoning as in the previous example reveals a channel congestion affecting the traffic of $task_2$ and $task_3$. Consequently, $task_3$ must wait before it can transmit its produced messages. This leads to the situation depicted in Figure 4.3b, where the messages of $task_3$ would actually arrive at approximately $t = 4.5\,\mathrm{U}$, so after the quorum-agreement window has closed. This constitutes a *missed deadline*, with the same critical consequences discussed in the previous example. By contrast, when our proposed network scheduler is applied (Figure 4.3c), all tasks meet the timing constraint imposed by the quorum window. The scheduler effectively eliminates the congestion and ensures timely delivery of all messages.

We can therefore conclude that, even when tasks exhibit different task time slices S_i but equal transmission times tt_i, the proposed network scheduling policy yields a marked improvement in resource utilization. In this case the improvement is attributable to the fact that the scheduler prioritizes tasks with smaller S_i, thereby anticipating the time at which the network begins to carry hard real-time traffic and reducing the initial idle interval during which the network would otherwise remain unused.

Transmission Time Definition

We now refine the notion of the transmission time tt_i of a task i. Let R denote the channel capacity (which we assume to be identical for all channels), and let M_i be the total size of the messages generated by task i. We define the latency contribution on a single channel, denoted as l_i, as:

$$l_i = \frac{M_i}{R}$$

If we call h_i the number of hops required by the traffic of task i to reach its destination, then the transmission time of task i can be expressed as:

$$tt_i = l_i \cdot h_i \tag{4.4}$$

which represents the end-to-end latency induced by the task's traffic across h_i links. However, Equation 4.4 is valid only under the assumptions of no congestion on the network and uniform hop count h_i for all the packets generated by task i. Since the accurate estimation of end-to-end latency in a distributed hard real-time network is a complex problem requiring a dedicated analysis, in this work we deliberately exclude such aspects from the definition of transmission time for the purpose of network scheduling. Accordingly, we redefine the transmission time as the latency on the first outgoing channel, so the link directly attached to each node:

$$tt_i = l_i \tag{4.5}$$

This choice ensures that the scheduler captures the same aspects already considered in the illustrative examples presented earlier, while issues related to congestion and hop count are managed instead at the network design stage (see Chapter 4.3). We recall that M_i is the aggregate size of the traffic generated by task i. Let P_i denote the size of a single packet produced by task i, and Q_i the quorum size of task i. Then:

$$M_i = P_i \cdot (Q_i - 1) \tag{4.6}$$

Combining Equations 4.5 and 4.6, we obtain the effective definition of the transmission time used in the scheduling procedure:

$$tt_i = \frac{P_i \cdot (Q_i - 1)}{R} \tag{4.7}$$

Finally, by plugging Equation 4.7 into the scheduling criterion originally given in Equation 4.3, we derive the final expression for the ranking ratio K_i, namely the metric adopted for ordering the replicas of tasks on the nodes:

$$K_i = \frac{S_i}{\frac{P_i \cdot (Q_i - 1)}{R}} \tag{4.8}$$

4.3. TAS Windows Sizing

We now explain how we applied Network Calculus to estimate the size of the two windows required for *quorum agreement* and *state recovery*. At this stage, we already have a configuration that assigns the active and passive replicas of the tasks to the network nodes, the static scheduling of tasks on the nodes, and all task-related parameters. In order to properly dimension the two windows, we first need to estimate the *state recovery window*. Based on this result, we can then derive the size of the *quorum agreement window*.

4.3.1. State Recovery Window Estimation

During the state recovery window, different types of messages are exchanged, which can be grouped into two possible traffic flows:

- **Host Fail**: when the system detects a hardware failure, all task replicas belonging to the affected quorums transmit a *Host Fail* message.
- **Host Recovered**: when the system detects that the faulty node has become operational again, all task replicas belonging to the affected quorums transmit two messages, namely *State Recovery* and *Host Recovered*.

It is worth noting that the *Host Fail* and *State Recovery* messages essentially serve the same function, i.e., updating the state of the receiving node. Consequently, these two messages have the same packet size. Therefore, to correctly estimate the size of the state

recovery window, we need to consider the worst-case scenario, namely the traffic flow that generates the largest network load: the *Host Recovered* flow.

Let P_{sr}^{i} denote the size of the state recovery packet generated by task i, P_{base} denote the size of the host recovered packet (which is constant for all tasks), and M_{sr} the total state recovery traffic entering a switch. Since this traffic is ready to be transmitted at the very beginning of the state recovery window, we can directly consider it as an input burst at the switch. Hence, we obtain $\sigma = M_{sr}$ and, consequently, a rate $\rho = 0$. In this way, we have already identified the arrival curve $\alpha(t)$, and we can proceed to compute the worst-case maximum latency on the backbone, $D_{max}^{'}$, as:

$$D_{max}^{'} = \frac{\sigma}{R}$$

The next step is to compute the intrinsic system latency L. To this end, we first calculate l_{max}, i.e., the maximum latency required for the transmission of the two messages belonging to the *Host Recovered* flow across a single network link:

$$l_{max} = \frac{\max(P_{sr}^{i}) + P_{base}}{R}$$

Finally, we compute L as:

$$L = l_{max} \cdot n_c$$

where n_c represents the number of links that the traffic must traverse from the source to the destination, excluding the backbone. At this point, we can compute the length of the state recovery window, T_{sr}, as:

$$T_{sr} = L + D_{max}^{'}$$

4.3.2. Quorum Agreement Window Estimation

The length of the quorum agreement window, T_{qa}, can be derived directly from the state recovery window:

$$T_{qa} = T_c - T_{sr}$$

Before validating this result, we need to estimate the worst-case packet latency. This step is required to verify whether T_{qa} is sufficiently large to accommodate both the execution of all tasks and the transmission of their corresponding packets. As in the state recovery analysis, we apply Network Calculus under a worst-case assumption, i.e., we consider that all tasks transmit their generated traffic within the same network cycle. However, in this case, packets are not all available at the beginning of the quorum agreement window, since tasks are executed sequentially on the nodes. For this reason, unlike Section 4.3.1, both the burst σ and the rate ρ must be computed.

Let M_{qa} denote the total quorum agreement traffic entering a switch. Let t_{first} be the first instant at which at least one task i completes execution and becomes ready to transmit packets. The traffic generated at this time is denoted by M_{burst}, which corresponds to the initial burst:

$$\sigma = M_{burst}$$

The remaining traffic, which defines the rate, is:

$$M_{rate} = M_{qa} - M_{burst}$$

and thus:

$$\rho = M_{rate}$$

With these parameters, the worst-case latency on the backbone, D'_{max}, can be computed as:

$$D'_{max} = \frac{\sigma}{R - \rho}$$

Next, we estimate the maximum latency on a single link, l_{max}, and the intrinsic system latency L. Let P^i_{qa} be the size of the quorum agreement packet generated by task i. Then:

$$l_{max} = \max(P^i_{qa})$$

$$T = l_{max} \cdot n_c$$

The overall worst-case delay is therefore:

$$D_{max} = L + D'_{max}$$

Finally, let t_{last} denote the latest completion time among all tasks. For the quorum agreement window to be sufficiently large, the following condition must hold:

$$T_{qa} \geq D_{max} + t_{last} \tag{4.9}$$

4.3.3. Network Configuration

In Sections 4.3.1 and 4.3.2, we applied *Network Calculus* to estimate latencies and timing bounds of the modeled distributed system. However, the results are not unique, as they strongly depend on the *network topology* and the placement of nodes within it. The position of the nodes determines the routing of packets and, consequently, the distribution of flows across the network links. As a result, variations in the network configuration can significantly affect both congestion levels and transmission delays, directly influencing the computed bounds. In addition, it is important to account for the fact that the switches within the network may introduce different delays on the backbone links. Since the channel is full-duplex, for the worst-case analysis we only need to consider the largest of these delays. Given these considerations, in order to derive a valid worst-case upper bound independent of the specific network configuration, we evaluated all possible node placements in the network. For each configuration, we applied *Network Calculus* as described previously, and we also computed T'_c, defined as the estimated total time required by that configuration to complete all tasks and transmit all packets during both windows.

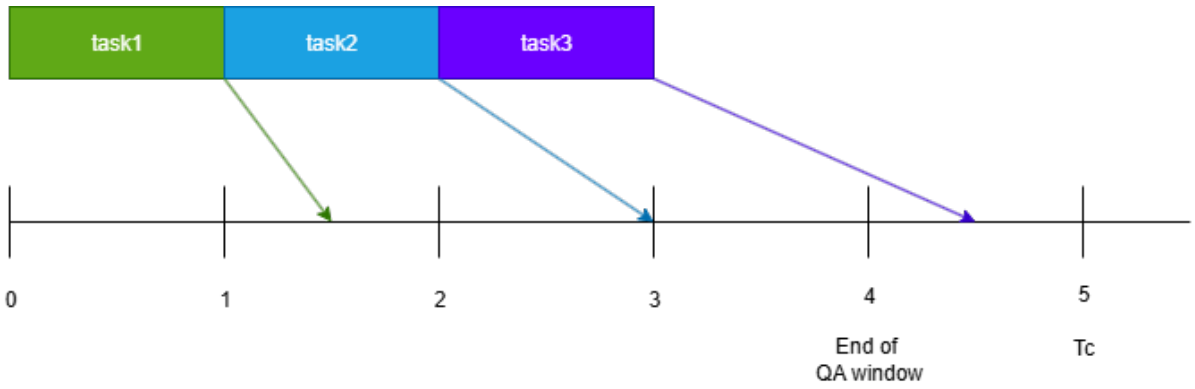
Let D_{qa}, D_{sr}, and t_{last} denote, respectively, the maximum backbone latency in the quorum agreement window, the maximum backbone latency in the state recovery window, and the latest completion time of a task. Then, we obtain:

$$T'_c = D_{qa} + D_{sr} + t_{last}$$

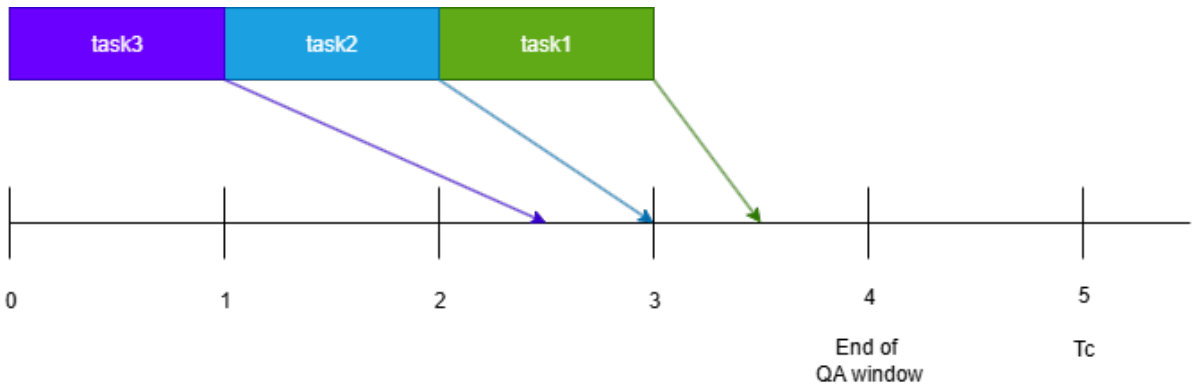
After calculating this value for each possible network configuration, we selected as the upper bound for the length of the quorum agreement window T_{qa} and the state recovery window T_{sr} the values estimated from the configuration with the largest T'_c.

4.3.4. Node Crash Specification

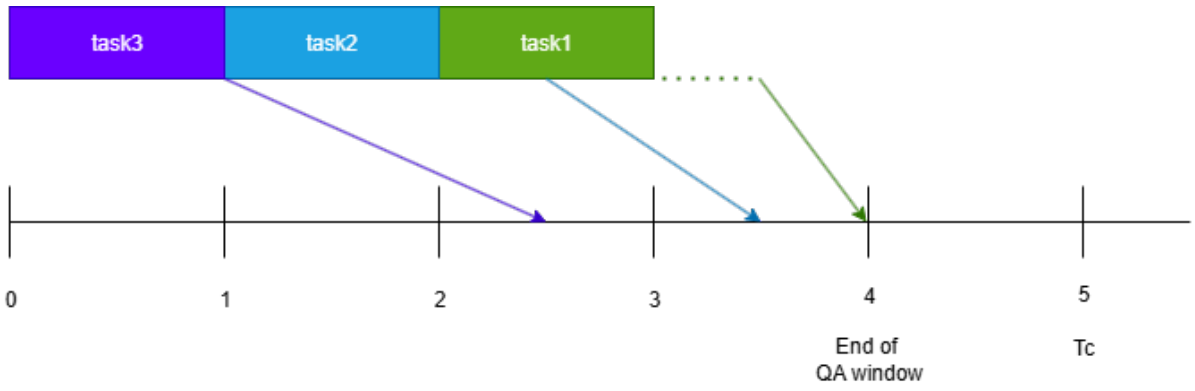
Another aspect to consider is that the model assumes that, in the case of a transient hardware failure, the faulty node will eventually recover and resume transmitting packets during the quorum agreement window. As a result, until the backup replica is deactivated, the traffic load on the network will temporarily increase. This implies that the calculation of the upper bound for the quorum agreement window T_{qa} must also take into account this additional traffic. Furthermore, the estimated bounds may vary depending on which node experiences the hardware failure. If a node with a higher workload fails and subsequently recovers, the system must redistribute a larger portion of tasks, which leads to higher resource utilization on the remaining nodes and, potentially, additional network load. For these reasons, we explicitly simulated the failure and subsequent recovery of each node during the estimation of the quorum agreement window. Also in this case, we selected the largest window size obtained across all simulations, thereby deriving a worst-case upper bound that remains valid under any possible failure scenario.



(a) Unscheduled Tasks.

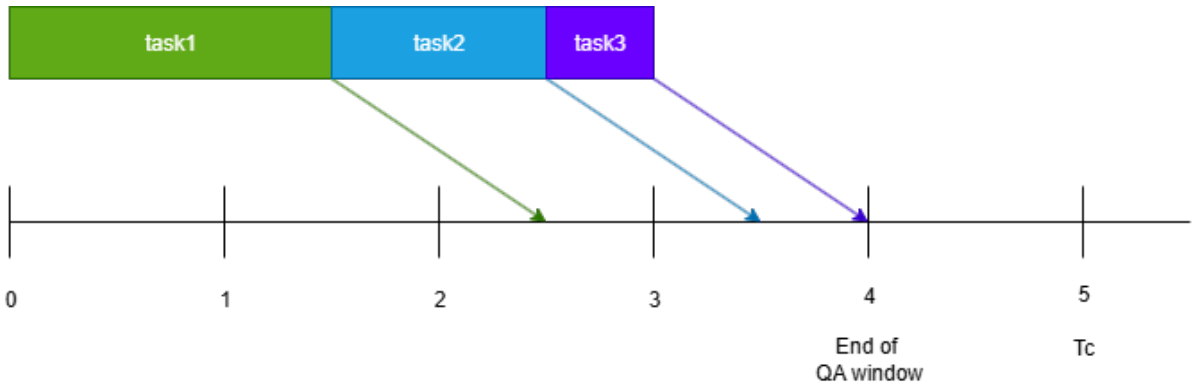


(b) Scheduled Tasks.

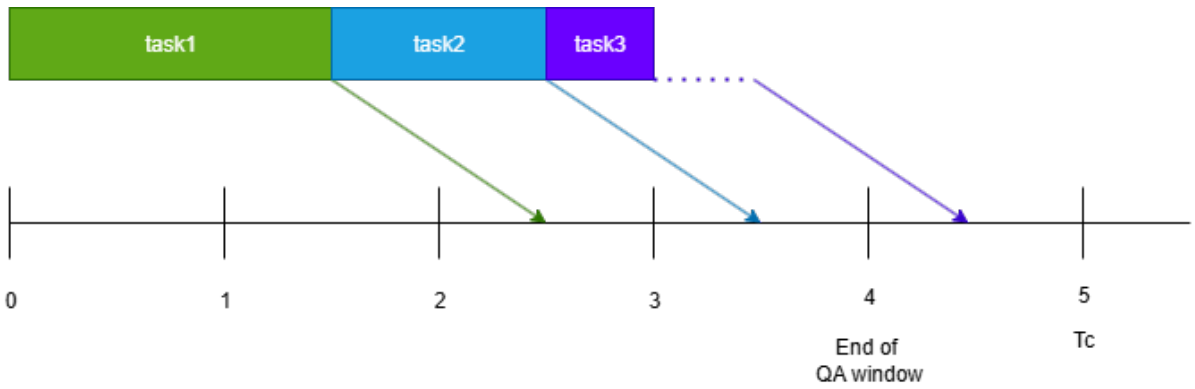


(c) Scheduled Tasks in Practice.

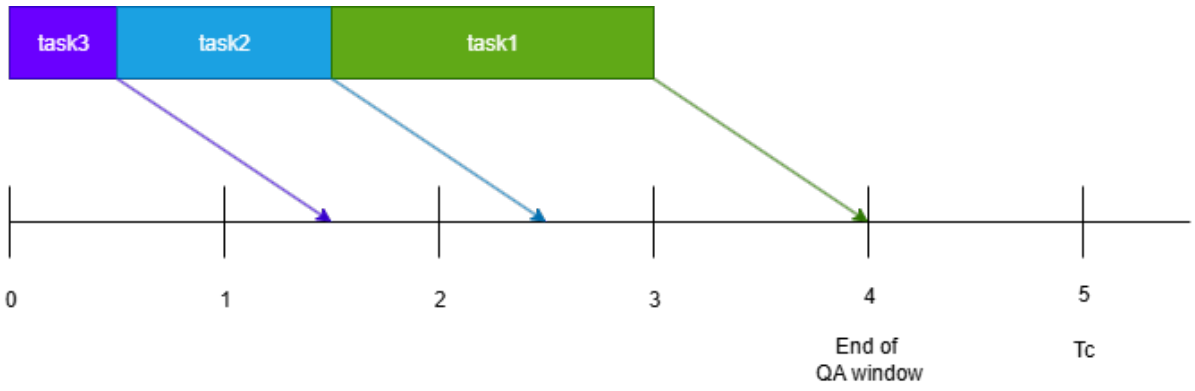
Figure 4.2: Network scheduling example showing tasks with equal time slices (rectangles) and different transmission times (arrows).



(a) Unscheduled Tasks.



(b) Unscheduled Tasks in Practice.



(c) Scheduled Tasks.

Figure 4.3: Network scheduling example showing tasks with different time slices (rectangles) and equal transmission times (arrows).

5 | Testing and Results

For the testing phase, we relied on the *OMNeT++* simulation framework, a widely used open-source tool for modeling communication networks and distributed systems, together with the *INET* library, which offers protocol implementations and network models for diverse scenarios. This combination allowed us to reproduce the behavior of the distributed system under realistic networking conditions, configuring switches, hosts, and TSN-based communication links. The modularity of INET and the event-driven accuracy of OMNeT++ were instrumental both in validating the analytical results and in evaluating system performance under different configurations.

5.1. Simulation Input

In order to evaluate the performance of the proposed model, it is essential to clearly define the set of input parameters used during the simulation and testing phase. In this section, we present the main categories of inputs considered, including system-level parameters and network-level parameters. The main input parameters that influence the behavior and performance of the model are the following:

- N: the number of nodes in the network.
- N_t: the number of tasks to be executed.
- C_i: the compute time of each task.
- T_i: the period/deadline of each task.
- Q_i: the quorum size required for each task.
- P^i_{qa}: the size of the quorum agreement packets generated by each task.
- P^i_{sr}: the size of the state recovery packets generated by each task.
- R: the communication rate of the network links.

In addition, based on these inputs, the following values must also be computed:

- T_c: the duration of the network cycle.
- S_i: the task time slice allocated to the execution of each task within a network cycle.

It is worth noting that the model requires several input parameters, each of which may impact system performance to a different extent. Simulating the model's behavior by varying each of these inputs, as well as their combinations, would result in a highly complex and computationally expensive process. For this reason, some of the input values were fixed or set to meaningful constants, chosen according to the application of the model in realistic systems. Consequently, we considered the following assumptions:

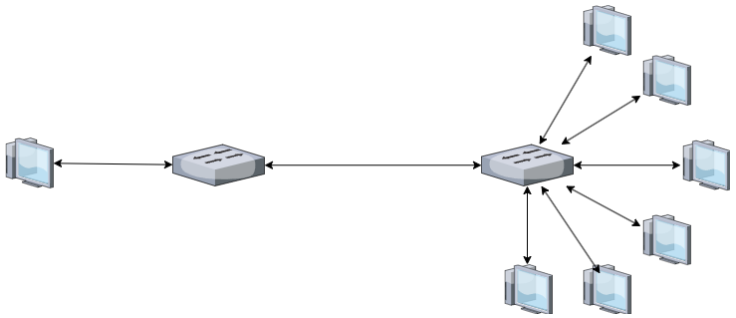
- C_i and T_i were set so that the task time slice S_i corresponds to a percentage of the network cycle (this aspect will be clarified in Section 5.3).
- $Q_i = 3$, the minimum quorum size.
- $P^i_{qa} = P^i_{sr} = 1500B$, corresponding to the standard Ethernet frame size.
- $R = 100$ Mbps.

With these simplifications, simulations can be performed by varying only N, N_t, and S_i. This allows us to more easily identify the limits of the model and analyze the results obtained during the testing phase. It is also important to specify that the number of tasks N_t depends on the number of nodes N. In fact, as the number of nodes increases, both the total computational capacity and the distribution of the workload change. As a result, the maximum number of tasks supported by the model grows with the number of available nodes.

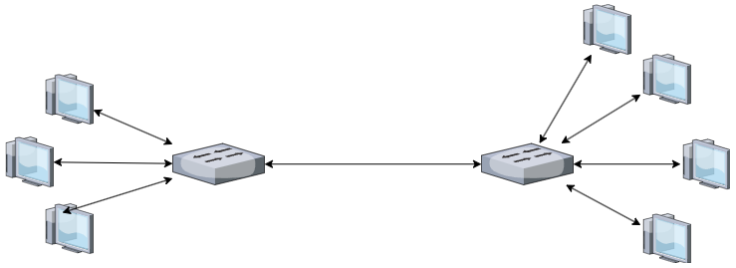
5.2. Configuration

As already discussed in Sections 4.3.3 and 4.3.4, both the network topology and the failure of nodes affect the estimation of the quorum agreement and state recovery windows. This leads us to conclude that these aspects must also be considered in the testing phase. Indeed, as the number of nodes increases, the number of possible network configurations grows rapidly. However, many of these configurations may not be realistic. For instance, Figure 5.1a shows a highly unbalanced distribution, with six out of seven nodes placed on the right star topology, while only one node is placed on the left. Such a configuration is unlikely to occur in a real-world system. By contrast, Figure 5.1b shows a balanced configuration, which is more representative of realistic systems.

It is also straightforward to conclude that an unbalanced configuration, such as the one depicted in Figure 5.1a, increases message latency. This occurs because almost all



(a) Unbalanced Node Disposition.



(b) Balanced Node Disposition.

Figure 5.1: Examples of Node Dispositions.

the traffic is routed through a single switch, resulting in a heavily loaded output gate on the backbone link and, consequently, a higher risk of congestion. For these reasons, we decided to focus on balanced configurations in the testing phase. Nevertheless, the quorum agreement and state recovery windows were dimensioned according to the most critical case, as explained in Section 4.3.3, ensuring that any other balanced configuration would still be supported.

5.3. Simulation and Result Analysis

As specified in Section 2.6, in the literature we were not able to identify studies addressing hard real-time distributed systems using a quorum-based approach. For this reason, we do not have reference results from other works to compare against our own. Consequently, the objective of the testing phase is to identify the limitations of the proposed model and, based on these, to analyze its performance, highlight its strengths and weaknesses, and understand how it can be improved. To this end, we performed two main types of simulations:

- **Low Load:** simulations where the task time slice S_i of each task corresponds to approximately 5–10% of the network cycle T_c.
- **High Load:** simulations where the task time slice S_i of each task corresponds to

approximately 25–30% of the network cycle T_c.

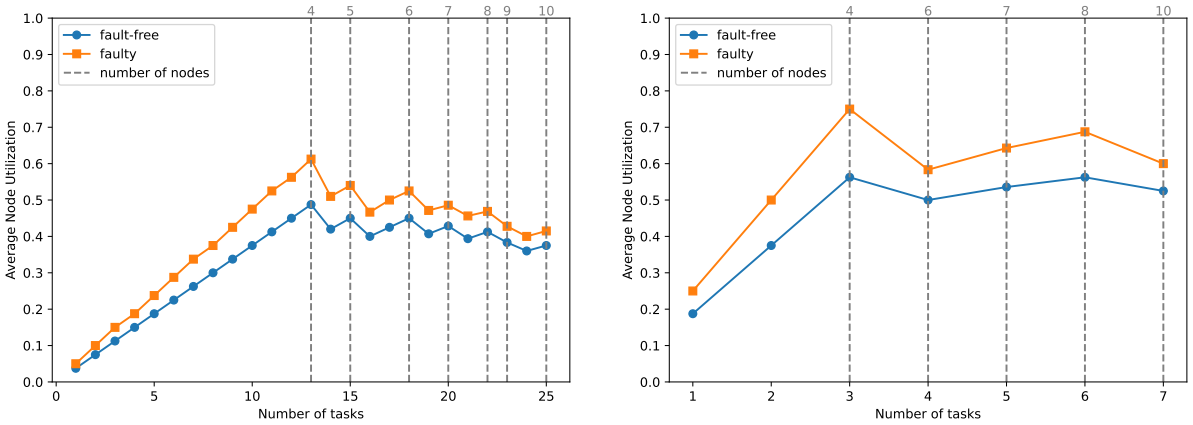
Moreover, in order to reproduce the worst-case scenario, all simulations were configured so that each task is executed and completed in every network cycle (thus, in every cycle, all tasks generate traffic to be transmitted over the network). In this way, the system is stressed with the maximum load foreseen by the model. As introduced in Section 5.1, we set $Q_i = 3$ for every task. This choice was made to identify the maximum number of tasks supported by the model. Finally, all simulations were carried out both in the presence and absence of hardware failures, allowing us to analyze how the system performance varies under normal and critical conditions.

It is worth noting that the number of tasks supported by the system directly depends on the number of nodes in the network. This explains the oscillating trend observed in the plots: as the number of nodes increases, the overall capacity of the system to handle a larger number of tasks grows accordingly. In the presented results, the x-axis is expressed in terms of the number of tasks, while the number of nodes is indicated by vertical dashed lines, which highlight the points corresponding to the different network scenarios.

Average Node Utilization

From the plots in Figure 5.2, it can be observed that the trend is approximately linear within each interval where the number of nodes remains constant. This, combined with the fact that the difference in utilization between the fault-free and faulty scenarios is not particularly large, indicates that the system as a whole achieves a well-balanced distribution of the task workload. On the other hand, the node utilization is relatively low. Moreover, once the first maximum load of tasks supported by the given number of nodes is reached ($N = 4$ in both plots), utilization tends to decrease as the number of tasks further increases. Regarding the plot in Figure 5.2b, this behavior is mainly due to the fact that tasks have a particularly demanding time slice. Consequently, introducing an additional task would leave no available time for transmitting the generated messages. However, if lighter tasks were introduced instead, these would still be supported. In contrast, the situation depicted in Figure 5.2a is more critical. Here, the trend is explained by the fact that, as the number of tasks scheduled on each node increases, the duration of the state recovery window also grows. As a result, less time remains available during the quorum agreement window for task execution.

This aspect becomes clearer when analyzing Figures 5.3, which highlight the ratio between the duration of the quorum agreement window and the network cycle. In particular, Figure 5.3a shows that, in the case of low-load tasks, this ratio starts at around 95% and



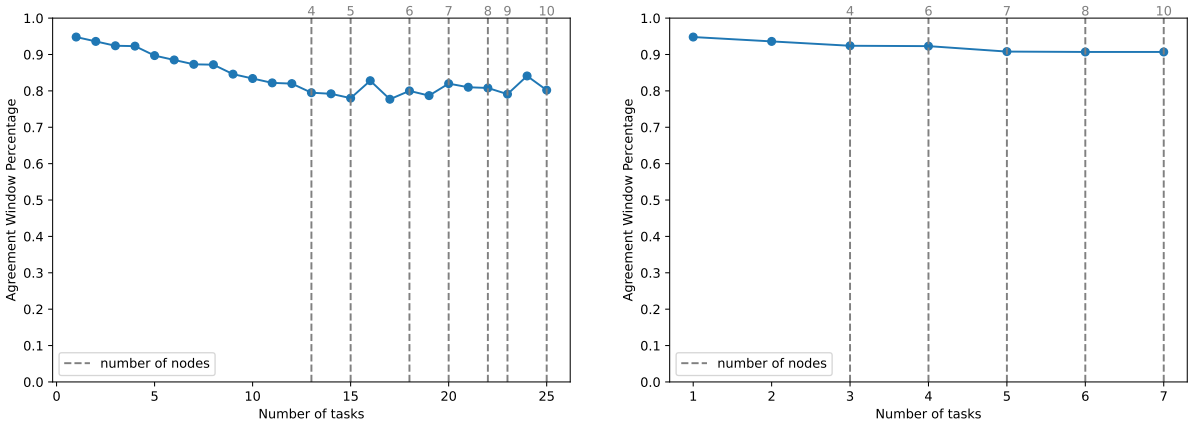
(a) Low load condition. **(b) High load condition.**

Figure 5.2: Comparison of the average node utilization, in the presence and absence of faults, for high and low load conditions.

gradually decreases to about 80%. However, this alone does not fully explain the low node utilization, especially once the number of tasks exceeds 20, where utilization stabilizes around 45%, even though the quorum agreement window still accounts for approximately 80% of a network cycle.

To better understand this phenomenon, we need to analyze the system behavior shown in Figures 5.4 and 5.5. These plots highlight the ratio between the effectively required and estimated duration of the quorum agreement and state recovery windows. From Figure 5.5, it can be immediately observed that Network Calculus is able to estimate the duration of the state recovery window with high accuracy, as the ratio consistently remains above 94%, reaching up to 98%. Naturally, this consideration only applies in the presence of a hardware failure, since in fault-free scenarios the state recovery window is not used at all.

The situation is different in the case of the quorum agreement window, as shown in Figure 5.4. Here, the estimation becomes less accurate as the number of tasks increases, with the ratio dropping to around 80% (again, in the presence of failures), corresponding to an overestimation of roughly 20%. This can be explained by two main factors. First, the duration of the quorum agreement window is not estimated directly but rather derived from the state recovery window, which naturally introduces some extra time that, however, is insufficient to accommodate additional tasks. Second, the estimation of the maximum latency provided by Network Calculus plays a critical role. Recall that Network Calculus is applied to estimate the worst-case latency during the quorum agreement window, in order to verify whether its duration is sufficient to guarantee the execution of all tasks



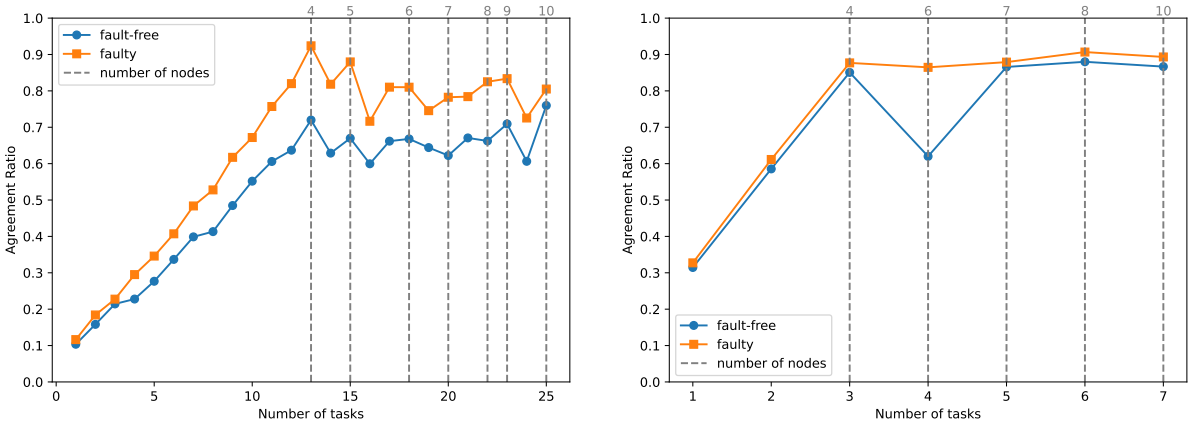
(a) Low load condition. **(b) High load condition.**

Figure 5.3: Trend of the quorum agreement window duration as a percentage of the network cycle, for low and high load conditions.

and the transmission of their generated traffic. The low ratio therefore indicates that the maximum latency is overestimated, which in turn restricts the number of tasks that the system can support.

Average Network Utilization

Figure 5.6 reports the average network utilization for both low-load and high-load tasks. It can be immediately observed that both plots exhibit the same alternating trend already seen in the case of node utilization. This behavior is once again explained by the increase in the number of nodes: as new nodes are added, tasks are redistributed and, more importantly, each node contributes with its own communication channel. Consequently, the average channel utilization decreases whenever a new node is introduced into the network. Another relevant aspect is that, also in this case, utilization tends to decrease as the number of tasks increases, which is consistent with the previous results. Indeed, the ratio between the number of tasks and the number of nodes gradually decreases, and therefore the average number of tasks scheduled on each node also decreases. As a result, each channel is required to transmit fewer packets within a network cycle. A further common feature, particularly visible in the high-load case shown in Figure 5.6b, is that overall network utilization remains rather low. This indicates that the model achieves efficient message exchange while still providing fault-tolerance guarantees. In addition, it implies that more network resources are available for transmitting low-priority traffic, which in turn improves the overall efficiency and usability of the system. Finally, as expected, the gap between the curves with and without failures is significant, especially in the low-load case shown in Figure 5.6a. This is due to the fact that, when the system



(a) Low load condition. **(b) High load condition.**

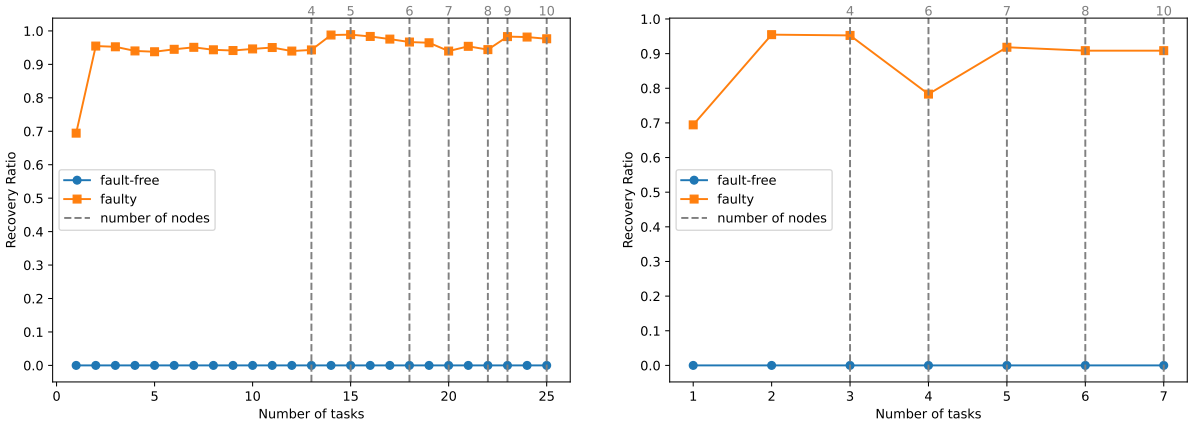
Figure 5.4: Comparison of the ratio between the actual and estimated duration of the quorum agreement window, in the presence and absence of faults, for high and low load conditions.

is in a faulty state, network traffic increases: on the one hand because the state recovery window is used, and on the other because, once the faulty node resumes operation, it restarts transmitting the results produced by its scheduled tasks.

System Utilization

The plots in Figure 5.7 illustrate the trends of the average system utilization. As in the previous results, the alternating behavior of the curves is again observed, which is due to variations in the number of nodes in the network. It is also worth noting that, at the relative maxima under faulty conditions, even though the system reaches the maximum theoretical load supported by the given configuration, the system utilization never reaches 100% and, in fact, never exceeds approximately 80%. This characteristic is primarily explained by the fact that we are analyzing the average system utilization as perceived by each node; in other words, the average time each node requires to complete all tasks and deliver all generated traffic across the network, both within the quorum agreement window and the state recovery window. Consequently, in the presence of a fault, it is expected that some nodes (a minority) will experience an increase in resource consumption, while others may remain unaffected.

To better understand this aspect, we can refer to the plots in Figure 5.8, which report the maximum system utilization observed across all nodes. Here, it can be seen that under faulty conditions the values reach approximately 80–90%. However, the 100% threshold is never reached. As in the case of node utilization, this is due to the fact that Network Calculus tends to overestimate maximum latencies. Therefore, in practice, the



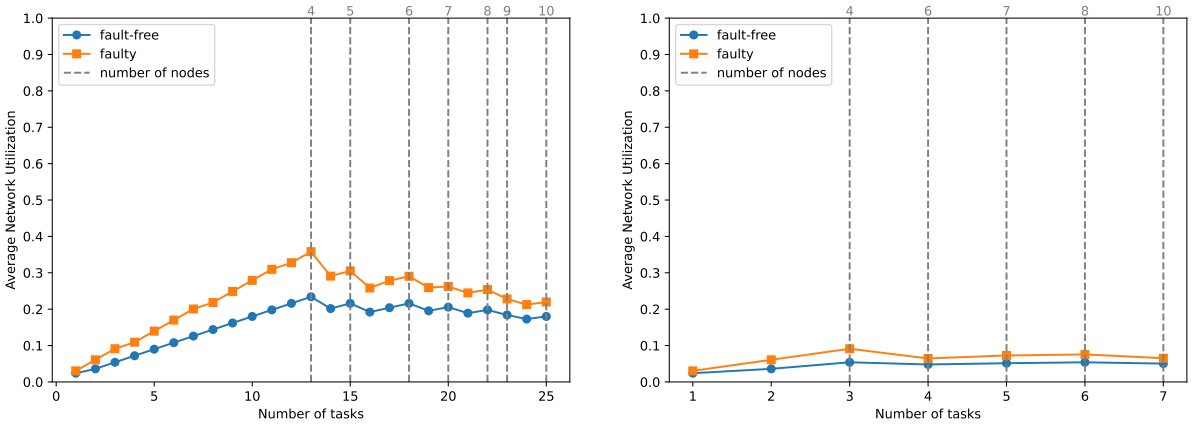
(a) Low load condition. **(b) High load condition.**

Figure 5.5: Comparison of the ratio between the actual and estimated duration of the state recovery window, in the presence and absence of faults, for high and low load conditions.

system does not actually operate at its maximum sustainable load, contrary to what is suggested by the theoretical calculations. From Figure 5.8b, it is evident that the curves are relatively stable and the gap between faulty and fault-free conditions is generally not very large. The situation is different in Figure 5.8a, where the trend is less stable and, more importantly, there is a substantial difference in resource utilization between normal operation and scenarios with hardware failures. This indicates that the model performs better when the system handles a limited number of tasks and/or tasks with higher computational demand. By contrast, when the number of tasks increases, the system becomes less efficient in utilizing the available resources, as a significant portion must be reserved for fault management.

Average Task Response Time

The plots in Figure 5.9 illustrate how the average response time of tasks varies as the number of tasks in the system increases. It can be immediately observed that there is no significant difference between the curves with and without hardware failures, which once again confirms that the model effectively adapts to the redistribution of task load across nodes when a fault occurs. However, it is important to highlight that the simulations we performed do not represent the worst-case scenario for task response time. In fact, we assumed that all tasks are completed within each network cycle, thus neglecting the impact of preemption on task response time. To make this effect more explicit, we analyze the example shown in Figure 5.10, where two tasks $\{task_1, task_2\}$ are considered, both having compute time C_i and period T_i such that the task compute time is $S_i = 1\,ms$,



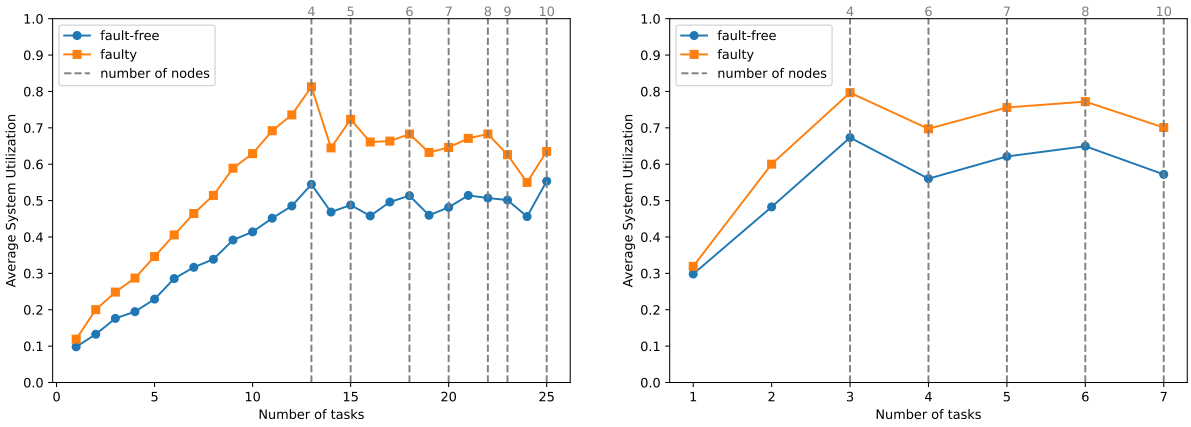
(a) Low load condition. **(b) High load condition.**

Figure 5.6: Comparison of the average network utilization, in the presence and absence of faults, for high and low load conditions.

while the network cycle is $T_c = 5\,ms$. We further examine three different cases:

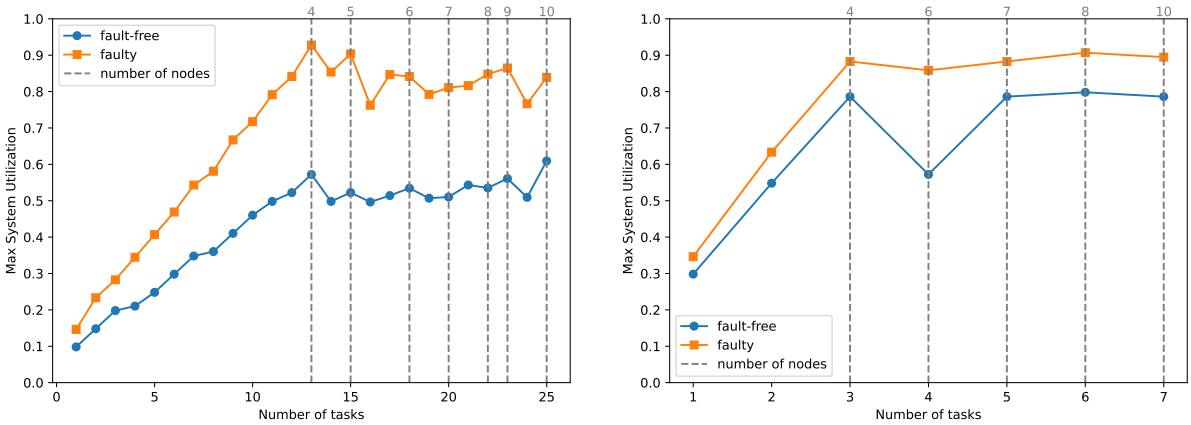
- **Case 1:** both tasks are completed within each network cycle.
- **Case 2:** $task_1$ is completed in every network cycle, while $task_2$ is completed every two network cycles.
- **Case 3:** $task_1$ is completed in every network cycle, while $task_2$ is completed every three network cycles.

Figure 5.10 shows how the average response time of tasks varies across the three cases. This highlights how preemption and the duration of the network cycle directly affect task response time. Between Case 1 and Case 2, the average response time nearly doubles. By contrast, the difference between Case 2 and Case 3 is less pronounced, which indicates that as a task requires more network cycles to complete, the average response time tends to stabilize, following a logarithmic trend.



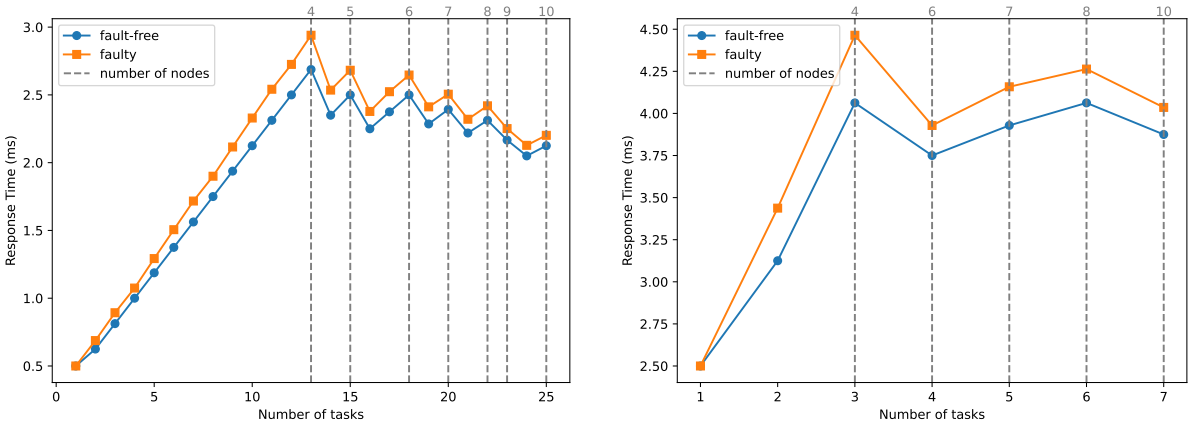
(a) Low load condition. (b) High load condition.

Figure 5.7: Comparison of the average system utilization, in the presence and absence of faults, for high and low load conditions.



(a) Low load condition. (b) High load condition.

Figure 5.8: Comparison of the maximum system utilization, in the presence and absence of faults, for high and low load conditions.



(a) Low load condition. (b) High load condition.

Figure 5.9: Comparison of the average task response time, in the presence and absence of faults, for high and low load conditions.

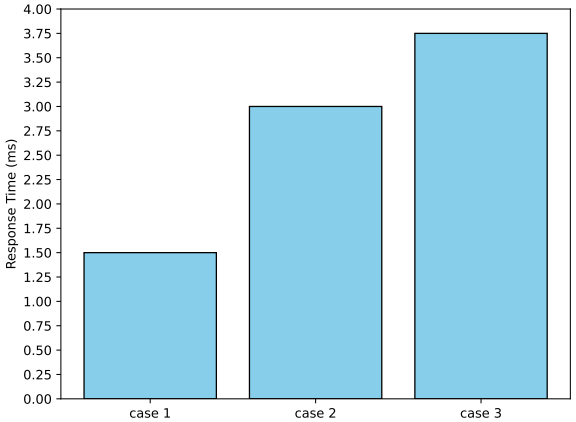


Figure 5.10: Illustration of the impact of preemption on the average task response time.

6 | Conclusions and Future Developments

This chapter brings the work of the thesis to a close by reflecting on the main outcomes and contributions of the proposed model. It begins by discussing the limitations identified during the evaluation phase, which provide useful insights into the current boundaries of the approach. Building on these observations, the chapter then outlines possible directions for future research aimed at enhancing efficiency, scalability, and fault tolerance. Finally, the overall conclusions are presented, emphasizing the relevance of the model within the broader context of hard real-time distributed systems.

6.1. Limitations

The following paragraphs describe the main limitations that emerged during the testing phase, which in turn result in a reduction of the model's overall efficiency and performance.

State Recovery Window

The presence of a state recovery window represents a first limitation to the number of tasks supported by the model, especially in the absence of failures, where this mechanism remains unused.

Network Calculus Limitations

Network Calculus proved to be an excellent tool for estimating the state recovery window. However, it is much less accurate when used to estimate the maximum latency of the quorum agreement window, leading to an overall underutilization of the available resources.

6.2. Future Work

Future research can mainly proceed in two directions: improving the efficiency of the model and extending its scope in order to enhance fault tolerance.

Multicast

The introduction of multicast mechanisms would significantly improve the performance of the model, as it would reduce the amount of traffic in the network and mitigate congestion. This, in turn, would shorten both the state recovery window and the delays during the quorum agreement phase, thereby increasing the number of supported tasks.

Latency Estimation

Another key factor negatively impacting efficiency is the latency overestimation introduced by Network Calculus. More refined techniques could be employed to directly simulate communication flows between nodes, thus obtaining a more accurate latency estimation. Given the high degree of determinism in the proposed model, integrating such methods should not be particularly complex.

Optimality of Network Scheduling

Although the current heuristic for network scheduling is effective, it does not ensure optimality. Evaluating its performance can reveal potential limitations and indicate areas for improvement. Should significant gaps be identified, alternative strategies could be explored to provide stronger guarantees and broaden applicability to more complex scenarios.

State Recovery Window Removal

Considering the generally low network utilization, it may also be possible to design a variation of the model that relies on backup replicas of tasks in a passive listening state. In this approach, quorum members would also send quorum agreement messages to the passive replicas, allowing them to remain updated and ready to activate without the need for explicit recovery coordination. This modification would increase network utilization but would significantly reduce the gap between normal and failure conditions.

Load Balancing Techniques

The static load balancing strategy for task allocation presented in this thesis could be modified or extended to handle multiple hardware failures simultaneously. The trade-off

would be a reduction in performance in exchange for stronger fault-tolerance guarantees. Dynamic load balancing techniques could also be explored to improve the model's flexibility, at the cost of partially sacrificing determinism and efficiency.

Moreover, these techniques could also be tailored to handle tasks with different priority levels, thereby enabling the model to support mixed-criticality systems. This extension would allow the proposed approach to address scenarios where tasks with varying degrees of importance and timing constraints must coexist, as is often the case in real-world safety-critical applications such as automotive, avionics, or industrial control systems. By incorporating mixed-criticality scheduling, the model could adapt its resource allocation strategy to guarantee that high-criticality tasks always meet their deadlines, even under fault conditions. Such an enhancement would significantly broaden the applicability of the model and increase its robustness in heterogeneous and mission-critical environments.

Tolerance to External Attacks

An aspect not fully addressed in this work, but of increasing importance today, is security against malicious third-party attacks. A quorum-based system can also be exploited to manage Byzantine failures arising from adversarial behaviors. By relying on majority consensus, the system can tolerate a subset of faulty or compromised participants while still guaranteeing correctness and consistency of the global state, thus extending its resilience beyond hardware failures to encompass security threats as well.

6.3. Conclusions

In this thesis, we have introduced and evaluated a quorum-based model for hard real-time distributed systems. Unlike traditional consensus-based approaches, the proposed model is able to detect and recover from Byzantine failures without requiring any task to be re-executed. We have also demonstrated how quorum mechanisms can be effectively leveraged both for failure detection and for the recovery of hardware faults, ensuring that task deadlines are consistently met. The evaluation revealed that the model achieves promising levels of determinism and fault tolerance. At the same time, several limitations were identified, particularly concerning latency estimation, and the impact of the state recovery window. These aspects, while not undermining the validity of the model, highlight opportunities for refinement and provide valuable directions for further research. Future work will therefore play a crucial role in addressing these limitations, with the aim of improving efficiency, scalability, and adaptability to realistic deployment scenarios. In particular, exploring more accurate latency estimation techniques, multicast support, dy-

namic load balancing, and enhanced tolerance to adversarial behavior could substantially strengthen the guarantees offered by the model. Overall, this thesis contributes a novel approach that enriches the design space of dependable distributed systems. By establishing a foundation for further advancements, the proposed model opens the way toward solutions that can deliver stronger fault-tolerance guarantees while remaining compatible with the stringent timing requirements of real-time and safety-critical applications.

Bibliography

[1] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software engineering journal*, 8(5):284–292, 1993.

[2] P. Chevochot and I. Puaut. An approach for fault-tolerance in hard real-time distributed systems. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 292–293, 1999. doi: 10.1109/RELDIS.1999.805106.

[3] J. Diemer, J. Rox, and R. Ernst. Modeling of ethernet avb networks for worst-case timing analysis. *IFAC Proceedings Volumes*, 45(2):848–853, 2012. ISSN 1474-6670. doi: https://doi.org/10.3182/20120215-3-AT-3016.00150. URL `https://www.sciencedirect.com/science/article/pii/S1474667016307832`. 7th Vienna International Conference on Mathematical Modelling.

[4] N. Gandhi, E. Roth, R. Gifford, L. T. X. Phan, and A. Haeberlen. Bounded-time recovery for distributed real-time systems. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 110–123, 2020. doi: 10.1109/RTAS48715.2020.00-13.

[5] A. Girault, C. Lavarenne, M. Sighireanu, and Y. Sorel. Generation of fault-tolerant static scheduling for real-time distributed embedded systems with multi-point links. In *IPDPS*, page 125, 2001.

[6] A. Gujarati, N. Yang, and B. B. Brandenburg. In-concretes: Interactive consistency meets distributed real-time systems, again! In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 211–224, 2022. doi: 10.1109/RTSS55097.2022.00027.

[7] J.-F. Hermant and G. Le Lann. Fast asynchronous uniform consensus in real-time distributed systems. *IEEE Transactions on Computers*, 51(8):931–944, 2002. doi: 10.1109/TC.2002.1024740.

[8] S. International. Time-triggered ethernet. Standard, SAE International, February 2023.

[9] D. Kozhaya, J. Decouchant, and P. Esteves-Verissimo. Rt-byzcast: Byzantine-resilient real-time reliable broadcast. *IEEE Transactions on Computers*, 68(3):440–454, 2019. doi: 10.1109/TC.2018.2871443.

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

[11] L. Lamport. *The part-time parliament*, page 277–317. Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450372701. URL `https://doi.org/10.1145/3335772.3335939`.

[12] J.-Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer, 2002.

[13] W. Luo, F. Yang, G. Tu, L. Pang, and X. Qin. Tercos: A novel technique for exploiting redundancies in fault-tolerant and real-time distributed systems. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, pages 275–282, 2007. doi: 10.1109/RTCSA.2007.70.

[14] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.

[15] S. D. Pasham. Fault-tolerant distributed computing for real-time applications in critical systems. *The Computertech*, pages 1–29, 2020.

[16] A. C. T. d. Santos, B. Schneider, and V. Nigam. Tsnsched: Automated schedule generation for time sensitive networking. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, pages 69–77, 2019. doi: 10.23919/FMCAD.2019.8894249.

[17] I. C. Society. Ieee standard for local and metropolitan area networks–frame replication and elimination for reliability. *IEEE Std 802.1CB-2017*, pages 1–102, 2017. doi: 10.1109/IEEESTD.2017.8091139.

[18] I. C. Society. Ieee standard for local and metropolitan area networks–timing and synchronization for time-sensitive applications. *IEEE Std 802.1AS-2020 (Revision of IEEE Std 802.1AS-2011)*, pages 1–421, 2020. doi: 10.1109/IEEESTD.2020.9121845.

[19] I. C. Society. Ieee standard for local and metropolitan area networks–bridges and bridged networks. *IEEE Std 802.1Q-2022 (Revision of IEEE Std 802.1Q-2018)*, pages 1–2163, 2022. doi: 10.1109/IEEESTD.2022.10004498.

[20] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[21] L. Zhao, F. He, E. Li, and J. Lu. Comparison of time sensitive networking (tsn) and ttethernet. In *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, pages 1–7, 2018. doi: 10.1109/DASC.2018.8569454.

[22] L. Zhao, P. Pop, and S. S. Craciunas. Worst-case latency analysis for ieee 802.1qbv time sensitive networks using network calculus. *IEEE Access*, 6:41803–41815, 2018. doi: 10.1109/ACCESS.2018.2858767.

[23] L. Zhao, P. Pop, Z. Zheng, and Q. Li. Timing analysis of avb traffic in tsn networks using network calculus. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 25–36, 2018. doi: 10.1109/RTAS.2018.00009.

List of Figures

List of Tables

Acknowledgements

I have now reached the conclusion of this journey, which has accompanied me over the past years and has given me valuable lessons not only from an academic perspective but also, and above all, on a personal level.

My deepest gratitude goes to my family: my mother, my brother, and Maurizio, who have always supported and encouraged me, showing understanding even in the most difficult times, and to Sid, who has been a constant presence by my side throughout these years.

A special thanks also goes to my friends, who, with their presence, have helped me find lightness and serenity, both in the moments we shared together and when I could not be there.

Finally, I wish to express my sincere gratitude to my supervisor, Federico Reghenzani, and my co-supervisor, Tomas A. Lopez, for their steady guidance, great availability, and patience in supporting me throughout the development and writing of this thesis.

Ringraziamenti

Sono giunto alla conclusione di questo percorso che mi ha accompagnato negli ultimi anni, offrendomi insegnamenti preziosi non solo dal punto di vista accademico, ma anche, e soprattutto, personale.

Il mio ringraziamento più grande va alla mia famiglia: a mia madre, a mio fratello e a Maurizio, che mi hanno sempre sostenuto e incoraggiato, dimostrando comprensione anche nei momenti più difficili, e a Sid, che in questi anni è stato una presenza costante al mio fianco.

Un grazie speciale va anche ai miei amici, che con la loro vicinanza mi hanno aiutato a ritrovare leggerezza e serenità, sia nei momenti di condivisione che quando non potevo esserci.

Infine, desidero esprimere la mia sincera gratitudine al mio relatore, Federico Reghenzani, e al mio correlatore, Tomas A. Lopez, per avermi seguito con costanza, guidato con disponibilità e accompagnato con pazienza nello sviluppo e nella stesura di questa tesi.