Rheinland-Pfälzische
Technische Universität
Kaiserslautern
Landau

Electrical and Computer Engineering

Flexibility and Intrusion Resilience for Time-Triggered Real-Time Systems in Open Safety-Critical Environments

Mohammad Ibrahim Alkoudsi

Dissertation 2026

Flexibility and Intrusion Resilience for Time-Triggered Real-Time Systems in Open Safety-Critical Environments

vom
Fachbereich Elektrotechnik und Informationstechnik
der Rheinland Pfälzischen Technischen Universität Kaiserslautern-Landau
zur Verleihung des akademischen Grades eines

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von
Mohammad Ibrahim Alkoudsi
geboren in Damaskus, Syrien

D 386

Eingereicht am:	12.07.2026
Tag der mündlichen Prüfung:	21.09.2026
Dekan des Fachbereichs:	Prof. Dr.-Ing. Daniel Görges

Promotionskomission

Vorsitzender:	Prof. Dr.-Ing. Wolfgang Kunz
Berichterstattende:	Prof. Dipl.-Ing. Dr. Gerhard Fohler
	Prof. Sanjoy Baruah

Erklärung gem. § 6 Abs. 3 Promotionsordnung

Ich versichere hiermit, dass ich diese Dissertation selbst und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt und die aus den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Dissertation wurde weder als Ganzes noch in Teilen als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung eingereicht. Es wurde weder diese noch eine andere Abhandlung bei einem anderen Fachbereich oder einer anderen Universität als Dissertation eingereicht.

Mohammad Ibrahim Alkoudsi
Kaiserslautern, den 10.07.2026

Abstract

The *Time-Triggered Architecture* (TTA) has long served as a foundational model for safety-critical distributed real-time systems in domains such as avionics and automotive control, offering strong guarantees regarding extra-functional properties such as determinism, temporal predictability, fault containment, temporal composability and the ability to satisfy complex timing constraints. Historically, these systems relied on a closed-world assumption where they functioned in fully controlled environments and interfaced exclusively with design-time-known and trusted components. However, the evolving requirements of modern safety-critical real-time systems are driving a shift away from closed architectures toward open, integrated platforms characterized by high connectivity, increased complexity, adaptive resource demands, interaction with non-real-time components, and extended autonomous mission durations. This shift exposes two structural limitations of TTA. First, its static design restricts flexibility when systems must handle dynamic workloads. Second, its original fault model, which focuses solely on accidental faults, fails to address intelligent adversaries who exploit predictable temporal structures and extended mission times for time-domain and resource-exhaustion attacks.

In this thesis, we examine the combined challenges of *flexibility* and *intrusion resilience* in open TTA-based systems and propose solutions that preserve TTA's extra-functional properties. As a first step, we investigate and classify the dependencies between TTA's core dependability services and its extra-functional properties, thereby establishing a foundation that enables architects to identify which aspects of TTA are essential and which can be modified or replaced while preserving the desired system properties. Building on this foundation, the thesis advances two central objectives.

In the first objective, we advance flexibility within time-triggered systems. We introduce dynamic time-triggered schedulers that allocate computation slots at runtime while preserving the timing guarantees established through offline scheduling. The Dynamic Task Set Scheduler (DTSS) represents our solution within this class and supports operation with static and dynamic granularity.

We enable DTSS to admit, remove and modify aperiodic tasks at runtime under time-triggered operation by integrating the Remaining Processor Capacity Analysis (RPCA) into its mechanisms. We design RPCA as part of this work to provide the analysis required for these runtime adaptations. We further examine the integration of periodic and sporadic tasks into time-triggered systems with dynamic schedulers and develop exact feasibility tests for this purpose. For sporadic tasks, we also develop runtime exact tests, the *(Quick) Processor Demand Analysis* ((Q)PDA*), which use periodic-demand preprocessing and quick convergence to determine feasibility within practical computational limits (achieving two to three orders of magnitude better runtime performance

compared to state-of-the-art methods and our exact tests without these techniques). Our scheduler and exact tests improve flexibility under time-triggered operation while preserving TTA's extra-functional properties.

In the second objective, we focus on analyzing the vulnerability of TTA-based systems to time-domain attacks, designing resource- and synchrony-exhaustion attacks targeting TTA nodes and their local bus guardians, and developing corresponding countermeasures. We establish an impossibility result for synchronizing network-agnostic guardians in the presence of malicious nodes and design our countermeasures to circumvent this limitation, thereby strengthening the resilience of time-triggered systems against intruders. Our countermeasures rely on a *replica-aware rejuvenation service* integrated into a trusted hardware component named *SyncGuard*. This component operates as a local *network-agnostic guardian* that coordinates network access with proactive and reactive *node rejuvenation*. It maintains synchrony, defends the system against resource-exhaustion attacks, and enables collision-free communication across heterogeneous network topologies without incurring the substantial overhead associated with classical *network-aware variants*.

The contributions in this thesis establish a basis for the next generation of intrusion-tolerant and flexible distributed real-time systems, preserving the stringent dependability guarantees of the time-triggered architecture while meeting the flexibility requirements of modern open architectures.

“To exist is to change, to change is to mature, to mature is to go on creating oneself endlessly” — Henri Bergson

Acknowledgments

Completing a Ph.D. is *never* a solitary journey. As I reflect on the path that led to this dissertation, I am deeply grateful to the many individuals who have supported, guided, and enabled me to reach this point. Because so many people have contributed to this journey, I fear I may inadvertently omit someone in these lines. Please know that I sincerely appreciate every gesture of support I have received along the way.

My journey began with a life-changing opportunity, for which I owe a profound debt of gratitude to **Prof. Dr.-Ing Wolfgang Kunz**. Long before this Ph.D. commenced, he selected me for a scholarship to attend the *European Master in Embedded Systems* (EMECS). Given the severe economic hardships I faced back in Syria, arriving where I am today would have been an impossibility without this support. It enabled me to experience Germany and Norway firsthand and fundamentally altered my life's trajectory. I also wish to extend my sincere thanks to **Prof. Kunz** and **Prof. Sanjoy Baruah** for honoring me by serving on my Ph.D. defense committee.

My deepest professional appreciation belongs to my supervisor, **Prof. Dipl.-Ing Dr. Gerhard Fohler**. He did much more than offer me the opportunity to pursue a Ph.D. He was a profoundly patient and dedicated mentor. The many skills I have cultivated throughout this journey, from researching, reviewing papers, and presenting findings, to supervising and teaching students, are a direct result of his guidance. **Prof. Fohler** was also pivotal in opening doors that allowed me to engage with esteemed researchers from the real-time systems community. I am incredibly privileged to have collaborated and discussed my work with **Prof. Sanjoy Baruah**, **Prof. Marcus Völp**, **Prof. Paulo Verissimo**, **Prof. Pontus Ekberg**, **Prof. Joël Goossens**, **Prof. Damir Isovic**, **Dr. Björn Brandenburg**, and **Silviu Craciunas**.

To the members of the Real-Time Systems Chair at RPTU Kaiserslautern: you have been a second family to me. Your readiness to help without hesitation has been invaluable. A specific thank you to **Markus Müller** for his unfailing IT support and for actively conversing with me in German, which truly broke the ice and helped me find my voice. I am equally grateful to **Stephanie "Steffi" Jung** for her indispensable support in navigating the labyrinth of bureaucracy.

Furthermore, I owe a great deal to my current and former colleagues. The endless yet immensely fruitful discussions we shared continually elevated the quality of my research and its presentation. I am especially grateful to **Luiz Maia**, **Shreya Kulhalli**, and **Gautam Gala** for their excellent collaboration on our joint research, and to **Isser Kadusale**, **Alan Paz**, **Florian Heilmann**, and **Kristin Krüger** for the valuable and insightful discussions that shaped my thinking.

Finally, no words can adequately capture my gratitude for my family. To my **parents**: thank you for your unwavering belief in me and your consistent, grounding support throughout my life. To my dear **wife**: you stood by me through some of the most challenging times of this journey, including an event that nearly prevented me from finishing this thesis. I am incredibly thankful for your constant patience, love, and encouragement along the way.

Mohammad Ibrahim Alkoudsi
Kaiserslautern, Germany

The work presented in this dissertation was supported in part by the *Deutsche Forschungsgemeinschaft* (DFG) through the European projects ByzRT (project no. 4328784) and ReSAC (project no. 490872987).

Publications

I have authored or co-authored the following publications:

Journal papers and technical notes

- Mohammad Ibrahim Alkoudsi, Damir Isovic, and Gerhard Fohler. "Revisiting Slot-Shifting's Offline Acceptance Test for Sporadic Tasks: A Technical Note". *Leibniz Transactions on Embedded Systems (LITES)*, Volume 10, Issue 1, pp. 4:1-4:6, 2025. doi: 10.4230/LITES.10.1.4.
- Mohammad Ibrahim Alkoudsi, Marcus Völp, and Gerhard Fohler. "Dependency Analysis of the Extra-Functional Properties of the Time-Triggered Architecture". *IEEE Access*, accepted for publication, 2026.

Conference and refereed workshop papers

- Mohammad Ibrahim Alkoudsi, Marcus Völp, and Gerhard Fohler. "Tolerating resource exhaustion attacks in the time-triggered architecture." In *Proceedings of the XII Brazilian Symposium on Computing Systems Engineering (SBESC)*, IEEE, 2022. doi: 10.1109/SBESC56799.2022.9964703.
- Mohammad Ibrahim Alkoudsi, Marcus Völp, and Gerhard Fohler. "A network-agnostic approach to enforcing collision-free time-triggered communication." In *Proceedings of the 28th Pacific Rim International Symposium on Dependable Computing (PRDC)*, IEEE, 2023. doi: 10.1109/PRDC59308.2023.00018.
- Gautam Gala, Tilmann Unte, Luiz Maia, Johannes Kühbacher, Isser Kadusale, Mohammad Ibrahim Alkoudsi, Gerhard Fohler, and Sebastian Altmeyer. "Safety-Critical Edge Robotics Architecture with Bounded End-to-End Latency". *Real-Time Cloud (RT-Cloud) Workshop, co-hosted with the 36th Euromicro Conference on Real-Time Systems (ECRTS)*, 2024. doi: 10.48550/arXiv.2406.14391.
- Mohammad Ibrahim Alkoudsi and Gerhard Fohler. "Scheduling Dynamic Task-Sets in Time-Triggered Real-Time Systems." In *Proceedings of the 32nd International Conference on Real-Time Networks and Systems (RTNS)*, 2024. doi: 10.1145/3696355.3699699. **[Outstanding Paper Award]**.
- Marcus Völp, Mohammad Ibrahim Alkoudsi, Azin Bayrami Asl, Kristin Krüger, Julio Rodrigues Mendonca da Neto, and Gerhard Fohler. "Defending Event-Triggered Systems against Out-of-Envelope Environments." In *Proceedings of the*

2nd Workshop on Real-Time Autonomous Systems Security (RT-AutoSec), 2025. doi: 10.48550/arXiv.2512.06331

- Mohammad Ibrahim Alkoudsi, Sanjoy Baruah, Pontus Ekberg, Gerhard Fohler, Joël Goossens, and Maryam Moslehi. "Extending Periodic Task Sets with Sporadic Tasks: Computational Complexity and Exact Feasibility Tests." In *Proceedings of the 33rd International Conference on Real-Time Networks and Systems (RTNS)*, 2025. **[Best Student Paper Award]**.

- Luiz Maia, Mohammad Ibrahim Alkoudsi, and Gerhard Fohler. "Ensuring End-to-End Latencies of Multi-Rate Cause-Effect Chains in Multi-Mode Systems Through Migration." In *Proceedings of the 13th European Congress of Embedded Real Time Systems (ERTS)*, 2026. doi:10.82331/ERTS.2026.22.

- Shreya Vithal Kulhalli, Mohammad Ibrahim Alkoudsi, and Gerhard Fohler. "Exploring and Exploiting Synchrony Limitations of Time-Triggered Network-Agnostic Guardians." In *Proceedings of the 29th International Symposium on Real-Time Distributed Computing (ISORC)*, 2026. doi:10.1109/ISORC70347.2026.11605755.

Contents

Abstract V

Acknowledgements IX

Publications XI

List of Figures XVIII

List of Tables XIX

List of Acronyms XXI

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Trends . . . 6
  - 1.2.1 From Federated to Integrated Architectures . . . 6
  - 1.2.2 From Predictable to Complex Applications and Hardware . . . 8
  - 1.2.3 From Closed to Open World Assumptions . . . 10
- 1.3 Problem Statement . . . 11
- 1.4 Contributions . . . 13
- 1.5 Organization . . . 15

2 Background: Fundamental Concepts and Theory 17
- 2.1 Real-Time Systems Theory Overview . . . 17
  - 2.1.1 Real-Time System Model . . . 17
  - 2.1.2 Origins of Temporal Constraints . . . 19
  - 2.1.3 Task Models . . . 21
  - 2.1.4 Real-Time System Classes and Utility-Based View . . . 27
  - 2.1.5 Time and Clocks . . . 31
  - 2.1.6 Scheduling in Real-Time Systems . . . 33
- 2.2 Distributed Systems Theory . . . 62
  - 2.2.1 Foundational Concepts of Dependability and Security . . . 62
  - 2.2.2 Degree of Synchrony and Operational Envelopes . . . 68
  - 2.2.3 Failure Modes and Adversary Models . . . 72

2.2.4 Theoretical Limitations of Distributed Computing 74
2.2.5 Fault Tolerance 81
2.2.6 Distributed Fault-Tolerance Problems 83
2.2.7 Wormholes Model and Architectural Hybridization 87
2.2.8 Intrusion Tolerance, Resilience, and Rejuvenation 89

3 The Time-Triggered Architecture: Foundations and Dependencies 97
3.1 Overview of the TTA 97
3.1.1 System Model and Real-Time Computer Design 97
3.1.2 TTA Cluster Topologies 99
3.1.3 Cluster State 101
3.1.4 Time-Triggered Communication 101
3.1.5 Fault Hypothesis 106
3.1.6 TTA Node 107
3.1.7 TTA's Real-Time and Dependability Services 110
3.2 TTA's Extra-Functional Properties 129
3.2.1 Hierarchical Structure of TTA Properties 129
3.2.2 Dependency Analysis Between TTA Services and Properties 132

4 Flexibility in Time-Triggered Scheduling 149
4.1 Related Work 150
4.1.1 Global Task Set Replacement 150
4.1.2 Task Set Extension with Event-Triggered Tasks 151
4.2 Dynamic Time-Triggered (DTT) Scheduling 153
4.2.1 Rationale 153
4.2.2 Dynamic Time-Triggered Schedulers 154
4.2.3 System Description and Task Model 156
4.2.4 Slot-Shifting's Preservation of TTA Properties 158
4.2.5 Dynamic Task Set Scheduler (DTSS) 161
4.3 Task Set Extension Feasibility Analyses 165
4.3.1 Extending Time-Triggered Task Sets with Aperiodic Tasks 166
4.3.2 Extending Time-Triggered Task Sets with Sporadic Tasks 181
4.3.3 Extending Time-Triggered Task Sets with Periodic Tasks 190
4.4 Feasibility Tests Compatibility with DTT Schedulers 195
4.4.1 DTT Scheduler Operation Granularity 196
4.4.2 Event-Detection and Timestamping Capabilities 196
4.5 Experimental Evaluation 199
4.5.1 DTSS Overheads 200
4.5.2 Evaluating Feasibility Tests for Sporadic Task Integration 203

4.5.3 Evaluating Feasibility Tests for Periodic Task Integration 210

5 Intrusion Resilience in Time-Triggered Systems 215
5.1 Related Work 216
5.1.1 Security and Dependability in the TTA 216
5.1.2 Real-Time Systems Attacks and Countermeasures 216
5.1.3 Rejuvenation 217
5.1.4 Guardians 218
5.2 System and Fault Models 219
5.3 TTA's Vulnerability to Resource Exhaustion Attacks 220
5.3.1 Synchrony Exhaustion 220
5.3.2 Fault Masking Exhaustion 222
5.4 Replica-Aware Rejuvenation Service 222
5.4.1 Rationale 222
5.4.2 Proactive-Reactive Rejuvenation of TTA Nodes 223
5.4.3 Rejuvenation Enforcement 230
5.5 Network-Agnostic Guardians 232
5.5.1 Rationale 232
5.5.2 System Model 233
5.5.3 Node-SyncGuard Clock Precision 234
5.5.4 Timing Attacks Against Network-Agnostic Guardians 237
5.5.5 Fault Model 240
5.5.6 SyncGuard Design 241
5.5.7 SyncGuard Overhead 248
5.5.8 SyncGuardLite 250

6 Conclusion and Future Directions 253
6.1 Summary of Contributions 253
6.2 Future Directions 256

Bibliography 280

Summary 281

Zusammenfassung 287

Curriculum Vitae 293

List of Figures

2.1.1 Model of a real-time system. . . . 18
2.1.2 Model of a real-time task τ_x. . . . 22
2.1.3 Example of two precedence DAG instances, $G_{0,i}$ and $G_{1,k}$. . . . 26
2.1.4 Illustration of priority inversion. . . . 27
2.1.5 Utility profiles for different real-time task classes. . . . 30
2.1.6 Illustration of the global time base and reasonableness condition. . . . 33
2.1.7 States of a real-time job. . . . 34
2.1.8 Comparison of preemptive, non-preemptive, and limited-preemptive execution models. . . . 36
2.1.9 Comparison of RM and EDF schedulability regions in utilization space. . 42
2.1.10 Illustrative EDF and RM schedules for a periodic task set with utilization of 1.0. . . . 42
2.1.11 Illustration of Dhall's effect on multiprocessor systems. . . . 49
2.1.12 Illustration of the processor demand bound $B(\Gamma)$. . . . 52
2.1.13 Illustration of the schedule in Table 2.1.2. . . . 57
2.1.14 Structure of a time slot for computation. . . . 58
2.1.15 DAG instances from Figure 2.1.3 after merging dependent tasks. . . . 60
2.2.1 Failure mode hierarchy for distributed systems. . . . 72
2.2.2 Illustration of Fundamental Limit of Time Measurement (I). . . . 75
2.2.3 Illustration of Fundamental Limit of Time Measurement (II). . . . 76
2.2.4 Illustration of Fundamental Limit of Time Measurement (III). . . . 77
2.2.5 Illustrative example of proactive node rejuvenation. . . . 95

3.1.1 TTA Ultimate implementation. . . . 100
3.1.2 TTA Prototype implementation. . . . 100
3.1.3 TTA Star implementation. . . . 100
3.1.4 Illustrative example of communication structure in the TTP/C protocol. . 103
3.1.5 Transmission window alignment in TDMA-based communication. . . . 104
3.1.6 Acceptance window for message reception in TT communication. . . . 105
3.1.7 TTP/C slot structure. . . . 106
3.1.8 TTA node structure 108
3.1.9 Dependencies between TTA services. . . . 112

3.1.10 $1\,MT/4\,MT$ sparse time base. 114
3.1.11 TTA's temporal firewall interface. 119
3.1.12 Structure of a TTA node with an FTU layer. 126
3.2.1 Hierarchical dependencies between TTA's extra-functional properties. . 130

4.3.1 Illustrative example of EDL schedule change upon job set extension. . . 177
4.3.2 Illustrative example of EDL schedule change when jobs execute in non-EDL order. 179
4.3.3 Illustrative example of the pessimism in Slot-Shifting's offline test for sporadic tasks. 184
4.5.1 Relative reduction in DTSS invocations (dynamic vs. static mode). 201
4.5.2 Relative reduction in aperiodic task admission ratio (dynamic vs. static mode). 202
4.5.3 Reduction of created intervals by DTSS compared to Slot-Shifting. 204
4.5.4 Mean runtime of exact feasibility tests vs. task set size. 206
4.5.5 Mean runtime of exact feasibility tests vs. hyper-period length. 207
4.5.6 Mean runtime of feasibility tests vs. task set utilization. 209
4.5.7 Schedulability success percentage vs. task set utilization. 209
4.5.8 Schedulability success percentage of the feasibility tests as a function of Γ_{PS} utilization across different baseline Γ_{P} utilizations. 212
4.5.9 Mean runtime of feasibility tests as a function of Γ_{PS} utilization. 213

5.4.1 Example proactive rejuvenation scheme in the TTA 226
5.4.2 Example proactive-reactive rejuvenation in the TTA 227
5.5.1 Network-agnostic guardians in a time-triggered distributed system. . . . 234
5.5.2 SyncGuard structure 242
5.5.3 SyncGuard's abstract state machine. 242

List of Tables

1.1.1 Correlation of DO-178C DAL and ISO 26262 ASIL levels. 3

2.1.1 Timing parameters for tasks in Figure 2.1.3 57
2.1.2 Global scheduling table for the tasks in Table 2.1.1 57
2.1.3 Resulting job set derived from the jobs in Table 2.1.1 and their offline schedule in Table 2.1.2 after applying the merging step and recalculating their WCETs, release times, and deadlines. 61

3.1.1 TTA fault hypothesis for its alternative implementations. 107
3.2.1 Dependency classification of extra-functional properties on TTA services. 133

4.3.1 Slot-Shifting's spare capacity intervals for example job set. 168
4.3.2 Processor demand analysis for the extended example job set. 170
4.3.3 Example task set showing parameters after Slot-Shifting's offline phase. 183
4.3.4 Properties of the original offline acceptance test compared against our proposed modifications to account for the ASAP scheduling of TT tasks. 186

5.1.1 Overview of guardian implementation in different TT protocols 219
5.5.1 Extreme cases of clock drift among the node, SyncGuard, and global time, and corresponding values for c_{drift}, c_{ncorr} and c_{gcorr}. 236

List of Acronyms

ADAS Advanced Driver-Assistance Systems
AFR Await-First Resynchronization
APEX Application EXecutive
API Application Programming Interface
ASIC Application-Specific Integrated Circuit
ASIL Automotive Safety Integrity Level
ASR Await Subsequent Resynchronization
AVI Attack-Vulnerability-Intrusion
BCET Best-Case Execution Time
BE Best-Effort
BER Backward Error Recovery
BFT Byzantine Fault Tolerance
CAN Controller Area Network
CAP Consistency, Availability, and Partition Tolerance
CBS Constant Bandwidth Server
CC Communication Controller
CDMA Code-Division Multiple Access
CNI Communication Network Interface
COI Controlled Object Interface
COTS Commercial Off-The-Shelf
CPM Core Processor Module
CPU Central Processing Unit
CRC Cyclic Redundancy Code
CSI Channel State Interface
DAL Design Assurance Level
DM Deadline Monotonic
DP Dynamic-Priority
DS Deferrable Server
DSS Dynamic Sporadic Server
DTSS Dynamic Task Set Scheduler
DTT Dynamic Time-Triggered
ECC Error Correcting Codes

ECR Error-Containment Region
ECU Electronic Control Unit
EDF Earliest Deadline First
EDL Earliest Deadline as Late as possible
EESS Exact EDF Slack Stealing
EMI Electromagnetic Interference
ET Event-Triggered
FCR Fault Containment Region
FER Forward Error Recovery
FLTM Fundamental Limits of Time Measurement
FP Fixed-Priority
FPGA Field Programmable Gate Array
FTT Flexible Time-Triggered
FTU Fault-Tolerant Unit
G-EDF Global Earliest Deadline First
GPGPU General-Purpose Graphics Processing Unit
G-RM Global Rate Monotonic
GST Global Stabilization Time
g-state Ground State
H-CBS Hard Constant Bandwidth Server
HMAC Hash-based Message Authentication Code
HP Hyperperiod
h-state History State
IDLE Idle Phase
IFG Inter-Frame Gap
ILP Integer Linear Programming
IMA Integrated Modular Avionics
I/O Input/Output
IoT Internet of Things
i-state Initialization State
IVA Input-Validation
JIT Just-In-Time
LCM Least Common Multiple
LIF Linking Interface
LLF Least Laxity First
LRU Line-Replaceable Unit
MAC Message Authentication Code
MEDL Message Descriptor List
MIC Minimum Integration Count

MILS Multiple Independent Levels of Security
MPSoC Multi-Processor System-on-Chip
MT Macrotick
mt Microtick
MTTF Mean Time To Failure
MTTR Mean Time To Repair
NBW Non-Blocking Write
NWA Network Access
NGI Node Guardian Interface
NGU Never-Give-Up
NMR N-Modular Redundancy
NP Nondeterministic Polynomial time
NRT Node Rejuvenation Trigger
OS Operating System
OTA Over-The-Air
PCP Priority Ceiling Protocol
PDA Processor Demand Analysis
PE Priority Exchange
PFSM Periodic Finite State Machine
PIP Priority Inheritance Protocol
PRM Proactive Resilience Model
PRP Post Receive Phase
PRS Proactive Rejuvenation Slot
PS Polling Server
PSP Pre-Send Phase
QM Quality Management
QPDA Quick Processor Demand Analysis
RBNBW Rate-Bounded Non-Blocking Write
RC Rate-Constrained
RM Rate Monotonic
ROM Read-Only Memory
RPCA Remaining Processor Capacity Analysis
RRS Reactive Rejuvenation Slot
RT Real-Time
RTA Response-Time Analysis
RTOS Real-Time Operating System
SAE Society of Automotive Engineers
SES Schedule Entry Selection
SMR State Machine Replication

SoS Slightly-off-Specification
SPoF Single Point of Failure
SRP Stack Resource Policy
SRU Smallest Replaceable Unit
SS Sporadic Server
SWaP Size, Weight, and Power
TADL Task Descriptor List
TAS Time-Aware Shaper
TBS Total Bandwidth Server
TC Transmission Controller
TCB Trusted Computing Base / Timely Computing Base
TDMA Time Division Multiple Access
TEW Target Execution Window
TLB Translation Lookaside Buffer
TMR Triple Modular Redundancy
TP Transmission Phase
TSN Time-Sensitive Networking
TT Time-Triggered
TTA Time-Triggered Architecture
TTCB Trusted Timely Computing Base
TTNoC Time-Triggered Network-on-Chip
TTP Time-Triggered Protocol
TTP/C Time-Triggered Protocol Class C
UAV Unmanned Aerial Vehicle
USIG Unique Sequential Identifier Generator
UTC Coordinated Universal Time
V2X Vehicle-to-Everything
WCET Worst-Case Execution Time
WCRT Worst-Case Response Time

"The art of progress is to preserve **order** amid **change** and to preserve **change** amid **order**."

(Alfred North Whitehead)

Chapter 1

Introduction

1.1 Context

Real-time systems encompass a broad class of computing systems in which the timing of computations carries semantic significance. Unlike general-purpose systems where overall performance is measured primarily by average computational throughput, real-time systems are required to produce logically correct results within defined temporal bounds. These temporal bounds vary widely across different application domains. Many real-time systems, such as multimedia streaming, interactive gaming, or virtual reality, operate under soft or non-critical timing constraints, where occasional violations (e.g., deadline misses) are benign; they lead to temporary, observable quality of service degradation but do not compromise the safety or structural integrity of the system or its users.

In stark contrast, systems with hard real-time requirements, including certain safety-critical functions, must provide strict and verifiable adherence to timing guarantees. In such environments, untimely behavior can directly jeopardize system correctness and lead to catastrophic consequences, such as loss of life, severe environmental destruction, and massive damage to property. As the functional scope of these systems grows increasingly complex, and the architectural demand for modularity, topological scalability, and strict fault isolation increases, real-time architectures are increasingly implemented as distributed systems. Distribution allows computational load to be shared across multiple nodes, but simultaneously, it introduces a set of architectural challenges, which include maintaining a system-wide notion of time, coordinating activities across spatially separated nodes, and ensuring predictable communication despite network delays, faults, or interference. Consequently, the correct behavior of the system emerges from the coordinated and timely interaction of system nodes [1].

Meeting these challenges becomes particularly important in safety-critical domains, where failures are not merely undesirable but potentially catastrophic. Safety-critical real-time systems operate at the intersection of computational strictness and physical

dynamics, making their correctness dependent not only on the logical validity of computations but also on the precise time instants at which results are produced. In such systems, timing is inseparable from functionality; a logically correct value delivered too late may be indistinguishable from an incorrect one, as the controlled physical environment has already evolved past the point where the specific intervention by the real-time computer system was appropriate or safe.

These stringent timing requirements are evident in avionics and automotive applications. For instance, a flight-control subsystem must recurrently process multi-axis pilot inputs and adjust control actuators within a tightly bounded time interval to maintain aerodynamic stability. Similarly, in an automotive steer-by-wire or braking function, excessive sensing-to-actuation latency can directly affect vehicle controllability and stopping distance. In these strict settings, the real-time system extends beyond the digital computer to encompass the environment it monitors and controls. This tight cyber-physical coupling forms continuous feedback loops in which environmental dynamics influence computational demands, and computational outputs directly alter the behavior of the environment. The presence of inherently safety-critical functionality in such systems imposes uncompromising, legally mandated demands for safety, predictability, and reliability.

Providing high assurance of these properties requires the system architecture to comply with rigorous regulatory frameworks and assurance standards legally mandated by international certification authorities. Prominent examples include the *DO-178C* standard for airborne systems, covering commercial and civil airplanes as well as *unmanned aerial vehicles* (UAVs) [2]. The *ISO 26262* standard is the equivalent framework for functional safety in the automotive domain [3]. Within these stringent standards, each isolated software application within the system must be assigned a specific assurance level, which dictates the rigorousness of the design, verification, and validation objectives that must be satisfied to achieve certification. The *DO-178C* standard defines five *Design Assurance Level*s (DALs), whereas *ISO 26262* defines four *Automotive Safety Integrity Levels* (ASILs) alongside a *Quality Management* (QM) baseline. Both classifications have a profound impact on system design and validation efforts. Table 1.1.1 provides an illustrative correlation between these assurance levels. In practice, each level is determined by domain-specific hazard analyses, and there is no official one-to-one mapping between the standards.

Facilitating the certification of systems with highly critical functionalities (e.g., DAL A/B and ASIL C/D) requires deploying them upon safety-critical architectures that guarantee strong dependability and timing properties. In this thesis, we focus on the following extra-functional properties because they are central to dependable distributed real-time systems:

Table 1.1.1: Illustrative correlation of DO-178C DAL and ISO 26262 ASIL levels with failure conditions and practical examples

DAL	ASIL	Failure condition	Typical system impact	Example functionality
A	D	Catastrophic	Multiple fatalities and possible loss of the platform	Flight-control, emergency braking, steer-by-wire
B	C	Hazardous / Severe-major	Large reduction in safety margins; severe injuries possible	Autopilot functions, ADAS functions (e.g., lane keeping, adaptive cruise) tightly coupled to vehicle control
C	B	Major	Reduced safety margins and increased operator workload	Monitoring/navigation support, instrument cluster functions
D	A	Minor	Slight reduction in safety margins or passenger inconvenience	Cabin environmental control, exterior lighting
E	QM	No safety effect	No relevant safety impact	In-flight entertainment, non-safety infotainment

- **Determinism and temporal predictability**: Certification authorities require evidence that the timing requirements of safety-critical functionalities are consistently satisfied, even under worst-case operational conditions (further detailed in Section 2.1). A practical example is a flight-control loop that must periodically sample sensors, execute control laws, and actuate surfaces within a known latency bound. Any deviation beyond this bound can compromise stability and potentially lead to an unrecoverable aerodynamic stall.

  Determinism ensures that, for any given sequence of inputs and initial conditions, the system's externally observable behavior is uniquely determined and repeatable. Temporal predictability complements this notion by requiring that timing bounds on this behavior can be analytically established prior to deployment, thereby guaranteeing that all timing constraints will be met in operation [1, 4].

- **Replica determinism**: High reliability in airborne and automotive systems is commonly achieved through active hardware redundancy, e.g., by replicating the *Flight Control Computer* in a *triple modular redundancy (TMR)* configuration. Replica determinism ensures that replicated components, when initialized identically and provided with identical inputs, produce outputs that are consistent in both their value and timing. With deterministic replicas, simple voting-based mechanisms can be employed. The failure of a minority of replicas can thus be outvoted and compensated for by the healthy majority, assuming fault independence and a low likelihood of correlated failures. If non-deterministic actions cause the internal

states of the replicas to diverge over time, the voting mechanism will fail, rendering the costly hardware redundancy useless [5].

- **Diagnosability**: Safety-critical systems require adequate diagnosability to reliably determine the operational status of their components [6]. During runtime, diagnostic information enable fault-tolerance mechanisms to detect failures and trigger predefined recovery actions, such as masking a transient failure, activating a spare node, or degrading the system's quality of service. Diagnosability is also essential for offline activities, where diagnostic information supports testing, fault analysis, and maintenance, particularly in the presence of permanent failures.

- **Fault and error containment**: Safety-critical applications require that faults remain confined to the specific subsystem or *Fault Containment Region* (FCR) in which they initially occur (further detailed in Section 2.2.1.4). They must not propagate to other FCRs, especially through shared communication networks. A classical example of fault propagation is the *babbling idiot* failure mode frequently observed in *Controller Area Network* (CAN) buses. In this scenario, a faulty automotive ECU monopolizes the shared network by continuously transmitting spurious, high-priority data frames. Because CAN relies on non-destructive bitwise arbitration based on message IDs, this faulty node will always win arbitration, completely saturating the bus bandwidth and preventing other nodes from successfully transmitting their data.

- **Timely online agreement**: Distributed agreement requires that all correct nodes converge on a unanimous, irrevocable decision despite the presence of failures [7] (see Section 2.2.6.1). In distributed real-time systems, agreement on aspects such as node status, modes of operation, or sensor values must be reached in a timely manner. For example, consider a distributed braking system in a vehicle. ECUs must agree quickly on the failure of a braking actuator to redistribute control responsibilities and maintain safe inter-vehicle distances.

- **Constructive testability**: The enormous state space created by the many possible interleavings of tasks and messages in distributed real-time systems makes exhaustive testing infeasible, complicating certification. Constructive testability is the property that system behavior can be verified in a systematic and reproducible way. A system exhibits this property only if its design constrains nondeterminism sufficiently to make its behavior analyzable. In real-time systems, constructive testability extends beyond functional correctness to also include timing correctness across all possible execution scenarios.

  Constructive testability relies on the assumption of *repeatability*, meaning that the

system produces identical externally observable behavior under identical conditions. Repeatability eliminates non-reproducible software faults (*Heisenbugs*) [8] and ensures that execution traces and interleavings remain constrained and analyzable, making practical verification of system behavior feasible.

- **Temporal composability**: Ensuring that the timing properties of a component established in isolation remain valid when integrated into a larger system defines temporal composability [9]. This enables a node-based design paradigm, where independently developed, verified, and certified nodes can be reused without unforeseen temporal interference. By permitting modular certification, composability significantly reduces the overall certification costs of the system.

  For example, when integrating a new node into a shared communication network, the integration must ensure that the timing behavior of existing nodes, particularly those transmitting safety-critical messages, remains unaffected. Otherwise, certification must be redone on all affected nodes.

- **Meeting complex temporal constraints**: Distributed real-time applications often involve complex temporal requirements that span multiple computational nodes (see Section 2.1.3). These include precedence constraints between distributed tasks, mutual exclusion for shared resources, and stringent communication jitter bounds [10]. For example, in a sensor fusion pipeline of an autonomous vehicle, sensed data must be pre-processed, transmitted across the network, fused with other sensory data using specialized algorithms and then fed into actuators.

  Enforcing such complex, distributed temporal constraints dynamically at runtime is computationally prohibitive. The underlying scheduling problem is NP-hard, causing the worst-case computational complexity to grow exponentially with system size. As a result, attempting to satisfy these constraints at runtime is impractical for systems that must provide strict real-time guarantees. Therefore, hard real-time systems must rely on offline mechanisms that guarantee these complex constraints by design before deployment, rather than attempting to resolve them dynamically during operation.

By providing a simplified and formally verified solution to the extra-functional properties outlined above, the *time-triggered* (TT) paradigm has proven highly suitable for safety-critical distributed real-time systems. Its success has been demonstrated both in academic research and industry (notably by the commercial success of the company TTTech). TT principles are now widely adopted across domains such as avionics, automotive, railway, and space.

In contrast to the *event-triggered* (ET) paradigm, where computational activities and network transmissions are initiated by asynchronous events (e.g., interrupts), the TT

paradigm restricts all such activities to predefined, mutually exclusive *time slots* governed by a system-wide notion of *global time*. These slots are allocated according to the worst-case execution and communication demands of each activity and are encoded in static schedules deployed across system nodes.

This distinction does not imply that ET designs are fundamentally inadequate. Compared to TT systems, ET systems offer superior flexibility, better responsiveness and average-case resource efficiency, particularly for bursty and irregular workloads. However, under overload or contention, the ET approach makes it harder to guarantee bounded latency, low jitter, and fault containment for the most critical functions. These challenges complicate certification. Consequently, in hard safety-critical domains, the tradeoff between flexibility and predictability necessarily favors predictability.

The TT paradigm is concretely instantiated in the *Time-Triggered Architecture* (TTA), which is built around a sparse, fault-tolerant global time base and predefined offline schedules. The TTA delivers a suite of architectural services, including clock synchronization, membership agreements, atomic multicast, bus guardians, and deterministic mode changes, that collectively provide a solution for the extra-functional properties discussed above.

At its inception, the TTA was engineered for physically isolated, closed-world environments free from intelligent adversaries and unpredictable, dynamic workloads. In these highly controlled settings, prioritizing predictability and resource over-provisioning at the expense of flexibility represented a reasonable and effective design trade-off. Today, however, this strict design philosophy is being fundamentally challenged. As modern distributed real-time systems increasingly operate in open-world, highly interconnected, and actively adversarial environments, immense pressure is placed on TTA's original design assumptions. This architectural tension motivates an investigation of how time-triggered systems can evolve to meet the demands of contemporary open-world and adversarial settings without compromising their foundational safety guarantees.

1.2 Trends

1.2.1 From Federated to Integrated Architectures

The historical limitations of early distributed real-time systems, particularly their reliance on physical resource over-provisioning at the cost of operational flexibility, became a critical bottleneck as the functional complexity of modern cyber-physical systems escalated. During the 1960s and stretching into the late 1990s, safety-critical industries, including commercial and military aerospace, relied on *federated architectures* to provide inherent fault containment without the vulnerability of a single point of

failure [11].

Within a federated system, each specific avionics or automotive function utilizes a dedicated, physically segregated computing unit, commonly referred to as a *Line-Replaceable Unit* (LRU) in aerospace or an *Electronic Control Unit* (ECU) in the automotive domain. These self-contained modules encapsulate all hardware, software, power supplies, and input/output (I/O) interfaces required for a single functionality. System engineers typically grouped these units by domain, connecting them via standardized, low-bandwidth digital data buses such as the unidirectional ARINC 429 protocol [12] or the military MIL-STD-1553 bus [13].

The primary advantage of the federated approach lies in its certification simplicity. Physical segregation guarantees fault containment; a failure in one node cannot corrupt the memory or disrupt the execution cycles of another node. However, this paradigm scales poorly. Because computing resources cannot be shared across different subsystems, designers are forced to accept substantial hardware redundancy. This redundancy significantly increases *Size, Weight, and Power* (SWaP) metrics, complicates wiring, and constrains future system scalability and functional upgrades [14].

Addressing these escalating SWaP constraints and enabling advanced cross-domain functional coordination drove the industry toward *Integrated Architectures* in the late 1990s. These architectures allow multiple application subsystems to share common hardware platforms and communication networks, thereby drastically reducing physical redundancies and associated costs. Nevertheless, this transition introduced complex certification challenges. When multiple applications share resources, they become susceptible to unintended temporal and spatial interference, effectively voiding traditional physical isolation guarantees [15].

Consequently, robust partitioning mechanisms became a strict regulatory requirement. In the avionics domain, *Integrated Modular Avionics* (IMA) emerged to enforce strict isolation between software applications residing on the same *Core Processor Module* (CPM) through hardware-assisted time and space partitioning[1]. The ARINC 653 standard formalized this avionics partitioning model and defined the *APplication EXecutive* (APEX) interface [16]. Concurrently, the automotive domain shifted towards domain-centralized architectures, with the AUTOSAR adaptive platform providing analogous partitioning and isolation mechanisms for automotive computing platforms [17]. These frameworks enable applications with different criticalities to coexist safely on shared hardware platforms while preserving memory protection and deterministic scheduling.

Parallel to the structural shift toward integrated platforms, communication paradigms had to evolve to support shared, deterministic networking without introducing fault propagation vulnerabilities. Time-triggered concepts, initially conceptualized in the late

[1] Leading aircraft that successfully exemplify the IMA approach include the Boeing 777, Airbus A380, and Boeing 787 [15].

1970s and 1980s, provided a strong foundation for distributed, fault-tolerant systems requiring strict temporal isolation [18, 19]. These foundational ideas matured into the TTA, bridging the gap between predictable federated isolation and resource-efficient integrated sharing.

Designed in the early 1990s, the *Time-Triggered Protocol* (TTP/C) was specifically engineered to satisfy *Society of Automotive Engineers* (SAE) Class C requirements (which can be correlated with ASIL D or C in Table 1.1.1), providing predictable message transmission, fault-tolerant clock synchronization, membership services, rapid mode changes, and seamless redundancy management [20][2]. TTA-based networks were instrumental in advancing major aerospace programs; for example, the Airbus A380 and Boeing 787 utilized TTP in specific flight control and distributed systems, while the Sikorsky S-92 and NASA's Orion Crew Exploration Vehicle adopted it for critical subsystems.

Although TTP/C provided strong predictability on dedicated buses, the ultimate realization of integrated architectures required unified networks capable of handling both critical control data and high-volume payload traffic simultaneously. Therefore, engineers extended time-triggered principles to standard Ethernet technologies. The resulting protocol, *TTEthernet* (standardized as SAE AS6802), aligns well with the integrated paradigm and now supports spacecraft and advanced aircraft networks, currently utilized in high-profile platforms such as NASA's Orion and ESA's Ariane 6 [22].

Modern avionics and space programs demand the strong temporal determinism and redundancy management of classical time-triggered networks, yet they also require the high bandwidth, switched topologies, and extensive commercial off-the-shelf (COTS) ecosystem that standard IEEE 802.3 Ethernet provides. TTEthernet bridges this gap by combining a robust global time base and scheduled traffic with advanced mechanisms for fault containment. Within a TTEthernet network, strictly scheduled (TT), rate-constrained (RC), and best-effort (BE) traffic classes coexist concurrently on a single physical infrastructure [23, 24]. More recently, *IEEE Time-Sensitive Networking (TSN)* standards and most notably *IEEE 802.1AS* (for time synchronization) [25] and *IEEE 802.1Qbv* (the *Time-Aware Shaper* (TAS) for network traffic) [26] adopt time-triggered principles to guarantee bounded latency and deterministic delivery in converged industrial, automotive, and aerospace networks.

1.2.2 From Predictable to Complex Applications and Hardware

The evolution of computing hardware in safety-critical real-time domains reflects a continuous struggle between the need for temporal predictability and the demand for com-

[2] Industry milestones underscore TTP's success, including the first brake-by-wire prototype with Daimler Chrysler in 1997, the founding of TTTech in 1998 to commercialize TTP/C, the formal publication of the mature TTA model in 2003 [19], and the standardization of TTP/C as SAE AS6003 in 2011 [21].

putational performance. Historically, avionics and automotive control applications in the 1980s and early 1990s predominantly implemented control and monitoring functions such as engine control, flight control, and basic cockpit displays [11]. In that era, the size and complexity of embedded software were still modest, and the number of interacting functions per computing node were limited. For these systems, a pure time-triggered design with periodic activation and globally synchronized schedules found a natural fit to provide strict timeliness guarantees and facilitate certification. Since the mid-1990s, however, the functional scope of safety-critical systems has expanded dramatically. In the automotive domain, the amount of software in vehicles has increased exponentially, with modern premium carosting hundreds of functions and tens to over one hundred ECUs, driven by *advanced driver-assistance systems* (ADAS), complex infotainment hubs, and *Vehicle-to-Everything* V2X connectivity modules [27,28]. In modern avionics, integrated flight management, advanced guidance and navigation, enhanced vision systems, complex monitoring, passenger entertainment, and diverse aircraft subsystems significantly increased both local data processing and global communication bandwidth demands [11, 15].

The hardware evolution trajectory mirrored this rising application complexity. During the 1980s and 1990s, embedded systems predominantly relied on simple, single-core processors with in-order pipelines and predictable memory access delays, making them highly amenable to static timing analysis and the time-triggered paradigm [1, 19]. In the 2000s and 2010s, however, microarchitectural innovations, including speculative and out-of-order execution, deeper pipelines, multi-level caches, and advanced branch prediction, boosted average performance but increased timing variability, complicating the derivation of tight *Worst-Case Execution Time* (WCET) bounds [29, 30]. The subsequent rise of *Multi-Processor System-on-Chip* (MPSoC) platforms in the 2010s introduced shared memory hierarchies, complex interconnects, and dynamic power management, further challenging predictability [31].

In order to reconcile high integration with time-determinism, time-triggered concepts were adapted for on-chip interconnects and realized as the *Time-Triggered Network-on-Chip* (TTNoC), providing globally scheduled, sparse time bases for MPSoCs and enabling time-synchronized message delivery consistent with TTA principles [32, 33]. Nevertheless, modern heterogeneous MPSoCs increasingly incorporate *General-Purpose Graphics Processing Units* (GPGPUs), *Field Programmable Gate Arrays* (FPGAs), and dedicated *Machine Learning* (ML) accelerators, explicitly designed to support computation-heavy perception, sensor-fusion, and decision-making workloads, whose dynamic, data-dependent timing defies traditional static worst-case predictability [34].

This hardware shift is particularly evident across multiple domains. The automotive industry's transition toward higher levels of autonomous driving requires unprecedented processing power for AI-driven perception, prediction, and planning modules, while

mandating strict ISO 26262 ASIL D certification for safety-relevant functions [3, 35]. Similarly, aerospace systems and particularly unmanned aerial vehicles (UAVs) must accommodate compute-intensive functions such as advanced guidance, vision-based navigation, and onboard data analytics under DO-178C DAL A requirements [2, 36].

1.2.3 From Closed to Open World Assumptions

Historically, domains such as avionics, automotive, and industrial control operated under a "closed-world" assumption, meaning that all interacting components were known at design time and the operational environment contained no intelligent adversaries attempting to violate safety or integrity guarantees. Physical isolation served as the primary, and often the only, security perimeter, since federated, function-specific control systems shared no computational resources and external interfaces were minimal and tightly controlled [37, 38].

Within these environments, the prevailing fault and threat models focused exclusively on rare accidental hardware faults, physical component degradation, or environmental disturbances such as electromagnetic interference or cosmic radiation. Traditional safety-critical systems also operated only for short, bounded mission durations under active human supervision.

Over the past two decades, safety-critical systems have been transitioning from closed into increasingly open, highly interconnected architectures[3]. This paradigm shift is driven by demands for autonomous operation and pervasive connectivity paradigms such as the *Internet of Things* (IoT) and V2X communication. Ubiquitous connectivity inevitably expands the external attack surface, exposing previously isolated control networks to remote access channels, *over-the-air* (OTA) updates, and cloud-based backends.

The rising complexity and heterogeneity of these systems further imply that vulnerabilities cannot be fully eliminated through testing alone. As a result, the dominant threat model has shifted from mitigating accidental faults to actively defending against intelligent and malicious adversaries who actively exploit software vulnerabilities, protocol weaknesses, and design flaws [38, 39]. Physical access is no longer required to endanger safety-critical systems; previous exploits have demonstrated that remote compromise of in-vehicle functions is feasible through diverse attack vectors, including Bluetooth, cellular, and multimedia interfaces [39, 40]. Beyond automotive systems, successful attacks on safety-critical real-time infrastructures such as power grids, water management systems, industrial automation, and autonomous platforms have been

[3] In the context of this thesis, an *open* safety-critical environment is defined as a system characterized by ubiquitous connectivity, adaptive and dynamic resource demands, interaction with non-real-time components, extended autonomous mission durations, and active exposure to intelligent adversaries.

documented [41–46].

Modern autonomous and highly automated systems are additionally required to operate continuously without direct human intervention over extended mission durations [47]. Continuous and unsupervised operation increases security risks because adversaries gain extended time windows to remain covertly lodged within the system, silently infer critical control information, and mount sophisticated multi-stage intrusions while masquerading as normal operational behavior.

Consequently, safety-critical real-time systems must increasingly be co-designed with security mechanisms such as intrusion detection, runtime monitoring, and secure communication protocols [38, 48]. This is reflected in modern regulatory standards such as the automotive ISO/SAE 21434 [49], which mandates structured cybersecurity risk management across the entire vehicle lifecycle. In the avionics domain, the DO-326A standard (Airworthiness Security Process Specification) [50] now represent the mandatory *Acceptable Means of Compliance* (AMC) for managing intentional cyber threats, operating in tandem with the DO-178C safety standard [2].

The TTA already provides dependability mechanisms, including global time, offline-defined schedules, membership services, temporal firewalls, and bus guardians, which inherently support certain security features against attacks on the time-triggered network. Prior academic work has analyzed and extended these mechanisms with secure start-up, time, and communication services to further protect against the malicious manipulation of software on host computers and prevent network address spoofing [51].

1.3 Problem Statement

The central hypothesis of this thesis is that the TTA, while sound and well-founded, was conceived under closed-world assumptions with comparatively simple, highly predictable hardware platforms in mind. Its deployment into contemporary open, highly networked, mixed-criticality, and actively adversarial environments therefore leads to a massive, growing mismatch between its original, restrictive design assumptions and the operational realities of modern cyber-physical systems.

In light of the technological trends outlined above, we have identified the following assumption gaps that challenge the applicability of the TTA in its classical form and which have motivated the research in this thesis:

- **Assumption gap 1 (The periodic workload model):** The TTA assumes that all workloads are periodic or can be reduced to periodic behavior to support offline schedule construction. While classical deterministic control loops and periodic tasks remain the backbone of safety-critical systems, these now must coexist with

complex, data-driven, and often non-periodic tasks (e.g., sporadic events, aperiodic service requests). A rigidly periodic interpretation of all activities can therefore naturally lead to wasteful over-provisioning and poor resource utilization.

- **Assumption gap 2 (The fixed task set):** The TTA implicitly assumes a fixed and fully known set of tasks whose parameters can be accounted for at design time and embedded into offline schedules. In modern systems, however, task sets evolve dynamically with operating modes, optional functions, degradation strategies, and maintenance updates. Under these scenarios, exhaustively precomputing offline schedules for all possible configurations quickly becomes intractable, and therefore scales poorly.
- **Assumption gap 3 (The known WCET):** The construction of safe time-triggered schedules critically depends on the availability of tight and trustworthy worst-case execution time (WCET) bounds for all tasks. For modern platforms that rely on multicore processors, GPUs, AI accelerators, and complex software stacks (e.g., full-fledged operating systems, hypervisors, managed runtimes, and JIT compilation), deriving safe WCET estimates is increasingly infeasible in practice. Even when it is theoretically possible, it typically comes at the cost of extreme pessimism to compensate for the complex timing behavior and interference patterns of such platforms. This, in turn, results in substantial over-dimensioning of computational resources and can render strict WCET provisioning economically unattractive for many industrial stakeholders.
- **Assumption gap 4 (Non-adversarial environments):** TTA's fault model focuses mainly on rarely occurring accidental failures. In open and connected systems, however, compromised nodes and sustained adversarial behavior must also be considered. While traditional security mechanisms can mitigate or deter certain classes of attacks, the inevitable presence of residual vulnerabilities, combined with long mission durations, implies that safety-critical systems must be designed not only to tolerate accidental failures of nodes but also to withstand and limit the impact of successful intrusions. Consequently, these systems require dedicated mechanisms to remain resilient in the presence of compromised nodes. This creates the need for intrusion-tolerant architectures (see Section 2.2.8) that preserve timing and safety guarantees even when some components become malicious.

In this thesis, we address the following research questions that arise from the assumption gaps outlined above.

RQ1: How can TT systems accommodate dynamically changing task sets without requiring exhaustive offline enumeration of all possible configurations in static scheduling tables?

RQ2: How can TT architectures be extended or modified to efficiently support mixed periodic, sporadic, or aperiodic workloads, while preserving TTA's extra-functional properties?

RQ3: How can TT architectures be equipped with intrusion-resilient mechanisms that preserve hard real-time guarantees and safety properties, even in the presence of compromised nodes and sustained adversarial behavior?

Collectively, this thesis investigates how to preserve the architectural strengths of the TTA while relaxing selected static assumptions where this can be done safely, and while strengthening the architecture against adversarial behavior. Although resolving the challenges of WCET uncertainty on modern heterogeneous platforms (Assumption gap 3) remains a broader open problem beyond the scope of this thesis, we offer insights into how the capabilities of our proposed solutions can accommodate this uncertainty. As detailed in Chapter 6, the dynamic scheduling mechanisms developed in this work provide a foundation for reactive overrun management, enabling the system to enforce budgets and isolate critical tasks even when bounds on execution times are uncertain.

1.4 Contributions

In this thesis, we make five main contributions. The first establishes a theoretical foundation, while the subsequent four advance the two overarching objectives of the thesis: enhancing flexibility in TT scheduling and strengthening the security and resilience of TT systems deployed in open and adversarial environments.

1. Dependency analysis of TTA properties and services. As an initial step, we develop a systematic understanding of the TTA by analyzing the relationships among its core dependability services and the extra-functional properties they collectively realize. Based on this analysis, we introduce a classification framework that categorizes the resulting service-to-property dependencies as *fundamental*, *implementation-based*, or *application-dependent*. This framework provides a principled basis for distinguishing architectural constraints from implementation choices, thereby enabling architects to identify which aspects of the TTA can be adapted, relaxed, or replaced while reasoning about the resulting trade-offs in its extra-functional properties [52].

Objective 1: Advancing Flexibility in Time-Triggered Scheduling

2. Dynamic time-triggered scheduling for flexible task set management. Addressing RQ1, we introduce a class of time-triggered schedulers that allocate computation slots at runtime while preserving timing guarantees established during offline scheduling. The *Dynamic Task Set Scheduler* (DTSS) represents our solution within this class. DTSS

supports operation with static and dynamic granularity. DTSS enables task admission, skipping, and removal, thereby demonstrating that controlled runtime flexibility can be reconciled with time-triggered operation while retaining its extra-functional properties. By utilizing this scheduler, systems can maintain the deterministic predictability required for industry certification while unlocking the flexibility to safely admit, skip, or remove tasks at runtime [53].

3. Feasibility analyses for extending time-triggered task sets. Addressing RQ2, we develop exact feasibility tests that enable the safe modification of task sets at runtime. First, we design and integrate the *Remaining Processor Capacity Analysis* (RPCA) into DTSS to support the safe integration of aperiodic tasks under time-triggered operation. We then investigate the integration of periodic and sporadic tasks into time-triggered systems with dynamic schedulers and derive exact feasibility tests for this purpose. For sporadic tasks, we examine closely related state-of-the-art approaches, specifically identifying and resolving sources of pessimism in their offline acceptance tests [54]. Building on these insights, we derive exact runtime feasibility tests: the *(Quick) Processor Demand Analysis* ((Q)PDA*). By pre-processing the periodic-demand and employing quick convergence, our tests relocate the majority of the computational effort to the design phase. This approach lowers the required scheduler's runtime complexity from necessarily exponential to pseudo-polynomial, yielding two to three orders of magnitude better runtime performance compared to state-of-the-art methods [55].

Objective 2: Enhancing Intrusion Resilience in TTA-based Systems

4. Strengthening TTA nodes against resource-exhaustion attacks. Focusing on the vulnerability of TTA-based systems to targeted resource exhaustion attacks, we analyze how stealthy adversaries can bypass standard fault-tolerance mechanisms, such as active replication and clock synchronization. As a countermeasure, we design a TTA-tailored proactive and reactive node rejuvenation service that restores nodes to a secure state, outpacing adversaries before they can compromise a critical threshold of replicas. In order to achieve exhaustion safety without introducing extensive architectural changes, our rejuvenation service leverages existing TTA dependability services. It utilizes global time abstractions for scheduling proactive recovery, the membership service for immediate reactive fault detection, and bus guardians to externally enforce node rejuvenation via secure startup certificates [56].

5. Synchronizing and securing network-agnostic guardians in adversarial environments. Addressing RQ3, we investigate time-domain attacks targeting the synchrony of network-agnostic guardians. These guardians enforce network access according to offline-defined schedules while remaining agnostic to the underlying network protocol and topology, substantially reducing design, verification, and integration complexity.

However, because they must rely on their controlled node's networking infrastructure to maintain clock synchrony, existing solutions assume nodes only fail-silently; an assumption that no longer holds in open and adversarial environments. We demonstrate that the safe re-synchronization of these guardians cannot be guaranteed when nodes may behave maliciously. Architected to circumvent this synchronization limitation, we introduce *SyncGuard*, the first network-agnostic and time-domain attack-resilient guardian. SyncGuard extends the guardian's responsibility from coordinating network access to coordinating the proactive and reactive rejuvenation of nodes. SyncGuards maintain system synchrony, ensure resource-exhaustion safe operation, and enforce collision-free TT communication across heterogeneous network topologies [57].

The contributions presented in this thesis establish an architectural basis for the next generation of intrusion-tolerant and flexible distributed time-triggered real-time computing systems. They preserve the stringent dependability guarantees of the TTA while meeting the flexibility requirements of modern open architectures.

1.5 Organization

The remainder of this thesis is organized as follows:

- **Chapter 2** introduces the theoretical foundations used throughout this thesis, covering real-time scheduling concepts, task and timing models, dependability and fault-tolerance, adversarial and intrusion-tolerance concepts, and the role of global time and synchronization in distributed real-time systems.
- **Chapter 3** presents the TTA and analyzes the dependencies between its core dependability services and the resulting extra-functional properties required in safety-critical system design. It provides the conceptual framework needed to justify which aspects of the architecture can be relaxed or substituted while preserving its extra-functional properties.
- **Chapter 4** addresses the flexibility objective of the thesis. It introduces DTSS, and presents the feasibility analyses used to extend time-triggered task sets with aperiodic, sporadic, or periodic tasks while preserving timing correctness.
- **Chapter 5** addresses the security and resilience objective. It analyzes resource- and synchrony-exhaustion attacks in TTA-based systems, develops a replica-aware rejuvenation service, and presents the SyncGuard approach for enforcing collision-free time-triggered communication under adversarial conditions.
- **Chapter 6** concludes the thesis, summarizes the main findings, and outlines future research directions.

"A distributed system is one in which the **failure** of a computer **you didn't even know existed** can render your own **computer unusable**."

(Leslie Lamport)

Chapter 2

Background: Fundamental Concepts and Theory

In this chapter, we provide an overview of the background concepts required for the remainder of the thesis. We begin with the foundations of real-time systems, including the real-time system model, timing requirements, task models, and scheduling paradigms. We then review the main concepts of dependability, security, and intrusion resilience, with emphasis on fault and intrusion tolerance and the assumptions and mechanisms required for their enforcement.

2.1 Real-Time Systems Theory Overview

Real-time systems require behavioral correctness that depends not only on the logical results of computation but also on the timing of those results. In many real-time systems, a late but logically correct result can be as harmful as an incorrect result; for example, if a braking command arrives too late, the controlled physical process, such as a vehicle, may have already evolved into a dangerous state. Therefore, system designers must ensure that real-time systems satisfy the temporal requirements dictated by the environment [58].

This section reviews the origins and modeling of temporal requirements and then presents scheduling paradigms for meeting those requirements, together with analyses that verify feasibility under each scheduling method.

2.1.1 Real-Time System Model

Following Kopetz [1], a real-time system decomposes, as shown in Figure 2.1.1, into three distinct, interacting subsystems, each assigned specific roles and interfaces:

1. **Real-time (RT) computer system**: The core computational subsystem responsible for implementing control algorithms and maintaining system behavior within spec-

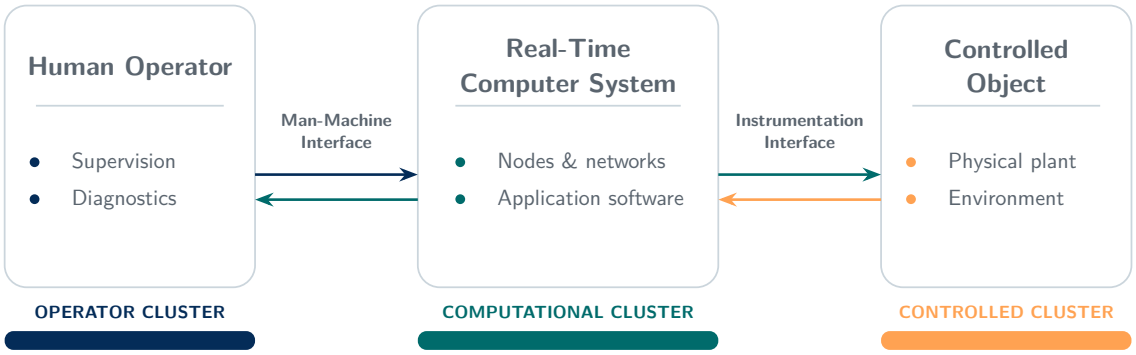


Figure 2.1.1: Model of a real-time system.

ified temporal bounds. It comprises the hardware infrastructure, including nodes and the network, and the software stack, including operating systems and application programs. The correctness of this subsystem depends on both functional accuracy, which ensures correct outputs, and temporal accuracy, which ensures outputs arrive within specified time constraints. When the RT computer system comprises several computational nodes that communicate over a shared network and employ a common set of distributed protocols to coordinate the execution of distributed activities, it qualifies as a distributed system [58].

2. **Controlled object**: The physical system or process that the RT computer system controls or monitors constitutes the controlled object. This subsystem follows physical laws and exhibits dynamics that constrain the temporal requirements imposed on the RT computer system. The RT computer system implements real-time digital control over the controlled-object to achieve the desired output, while meeting its constraints to retain stability.

   The RT computer system interfaces with the controlled object through an instrumentation interface composed of sensors and actuators. Sensors (such as microphones, cameras, transducers, and so on) enable the computer to observe significant state variables (e.g., angle, temperature), called *RT entities*, and actuators (such as motors, pumps, engines, and so on) enable the computer to exert control over the controlled object. Depending on who has the authority to change the semantic content of an RT entity, some RT entities fall within the computer system's control sphere (e.g., the intended position of a control valve), while others lie within the environment's control sphere (e.g., the pressure in a valve).

3. **Human Operator**: A human responsible for system monitoring, diagnosis, and maintenance. The RT computer system interfaces with the human operator via the *Human-Computer Interface* (HCI), which comprises input devices, e.g., a keyboard, and output devices, e.g., a display to enable the human operator to interact with

the RT computer system.

It is the responsibility of the RT computer system to detect anomalous behavior by monitoring state variables, and to aid the human operator in diagnosing the cause of alarm events. For that, the system logs all related events in their temporal order to reveal causal relationships. When the human operator can establish what happened first and what followed, they can more easily trace the chain of events that led to the alarm and identify the underlying fault.

2.1.2 Origins of Temporal Constraints

Temporal constraints in real-time systems are derived from identifiable characteristics of the application domain and the implementation platform [59]. We distinguish between the following sources of temporal constraints:

- **Dynamics of RT entities.** A primary source of temporal constraints is the dynamics of RT entities in the controlled object. Since the value of an RT entity evolves as a function of physical time, an observation of it is only meaningful together with the instant at which the observation was made. An observation is termed as a *state observation* if it records the RT entity's absolute value along with the time of observation, and as an *event observation* if it records a change in the RT entity's value and the time at which that change was observed.

  Following Kopetz, the computer-internal representation of such an observation is an *RT image*, and its admissible use is limited by a *temporal accuracy interval*, whose length depends on the dynamics of the corresponding RT entity [1]. If the observed RT entity changes state rapidly, then the temporal accuracy interval is short; if it changes slowly, the interval is longer.

  This source of timing requirements can therefore be interpreted as a *freshness* problem: the delay between observation and use must remain sufficiently small such that the RT image remains valid (i.e., the RT image still represents the state of the RT entity with sufficient accuracy). For example, the angular position of a rapidly rotating shaft in a vehicle engine becomes stale far more quickly than a temperature reading. Consequently, the admissible observation-to-use delay for the shaft angle is much smaller than that for temperature.

- **Environmental interaction.** The physical interaction between the controlled object and the environment within which it operates is another source of timing constraints. The issue here is not only whether the system uses sufficiently fresh data, but whether the complete sensing-to-actuation chain reacts within a bounded time interval imposed by the environment.

This source of timing requirements is best viewed as an *end-to-end latency* problem. For example, in an emergency-braking function, obstacle detection, decision making, message transmission, and brake actuation together must complete before the vehicle reaches a hazardous state. The resulting requirement is therefore expressed naturally as an end-to-end deadline on a *cause-effect chain* (CEC) rather than merely as a local deadline on a real-time task.

- **Implementation characteristics of the RT computer system.** Temporal constraints can also arise from the implementation of the RT computer system itself, including scheduling, handling shared resources and communication protocols. Although these aspects do not originate in the physical process, they constrain when computations and communication take place. In particular, in the distributed setting, the time at which a message is received need not coincide with the earliest time at which it may be acted upon safely [1, 58].

  A central notion in this context is *message permanence*, where a message becomes permanent at a receiver once the receiver knows that all earlier relevant messages have either already arrived or will never arrive. The interval between message transmission and the instant at which the message becomes usable is captured by the *action delay*, which exclusively depends on the properties of the communication network. The usability of a message carrying an RT image is therefore constrained from both sides by the action delay and by the temporal accuracy interval. The former imposes a not-to-use-before constraint that depends on the RT computer system implementation, while the latter imposes a not-to-use-after constraint that depends on the dynamics of the RT entity [58].

  For example, consider a monitoring node that relies on temporally ordered messages from several distributed nodes. Suppose one node reports a sudden pressure change, while another node is expected to send a message indicating that the change was caused by a planned actuator command. If the pressure-change message arrives first and is acted upon immediately, the monitoring node may incorrectly trigger an alarm. Correct behavior, however, requires waiting until the communication protocol guarantees that no earlier in-transit message could still arrive and change the interpretation of the event sequence. Acting before the message becomes permanent can lead to irrevocable or unsafe responses, thereby compromising real-time services [1].

- **Human operator in the loop.**

  Integrating a human operator into a control loop introduces temporal constraints shaped by human perceptual, cognitive, and reaction times (typically on the order

of a few hundred milliseconds). In such settings, timing requirements arise from the need for human intervention, supervision, and acknowledgment.

For example, in manual or assisted driving, the driver continuously perceives the environment, interprets traffic conditions, and adjusts steering and speed accordingly. These human-in-the-loop dynamics impose inherent delays that must be explicitly considered when designing the RT computer system [1, 60].

Beyond control and safety-critical applications, temporal constraints may also originate from multimedia and interactive applications such as video processing and cloud gaming. In such systems, temporal constraints stem from media sampling and presentation rates, inter-stream synchronization requirements, and human perception sensitivity to latency and jitter.

In all of the above cases, temporal constraints are abstracted into the formal quantities used in real-time systems theory, such as deadlines, minimum inter-arrival times, end-to-end latency bounds, and jitter constraints on tasks and messages, which are then used as inputs to schedulability analyses or offline scheduling [60].

2.1.3 Task Models

Real-time systems theory models the computational demand imposed on the RT computer system by means of *task models*. A task model provides a formal abstraction of system activities together with the timing information required for analysis, such as periodicity, execution demand, deadlines, and precedence relations [60–62].

Formally, let

$$\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$$

denote a finite task set system assigned to a computation processor. Each task $\tau_x \in \Gamma$ represents a recurrent activity rather than a single execution instance; the entities actually scheduled at run time are the task instances (jobs) generated by these tasks.

A task τ_x is modeled, as shown in Figure 2.1.2, by the tuple

$$\tau_x = \big(\phi_x, C_x, D_x, T_x\big),$$

where

- ϕ_x is the *initial offset* of τ_x: the first job of τ_x arrives at time-instant ϕ_x;
- C_x is the *Worst-Case Execution Time* (WCET) of τ_x: each job of τ_x needs to execute for no more than C_x time units;

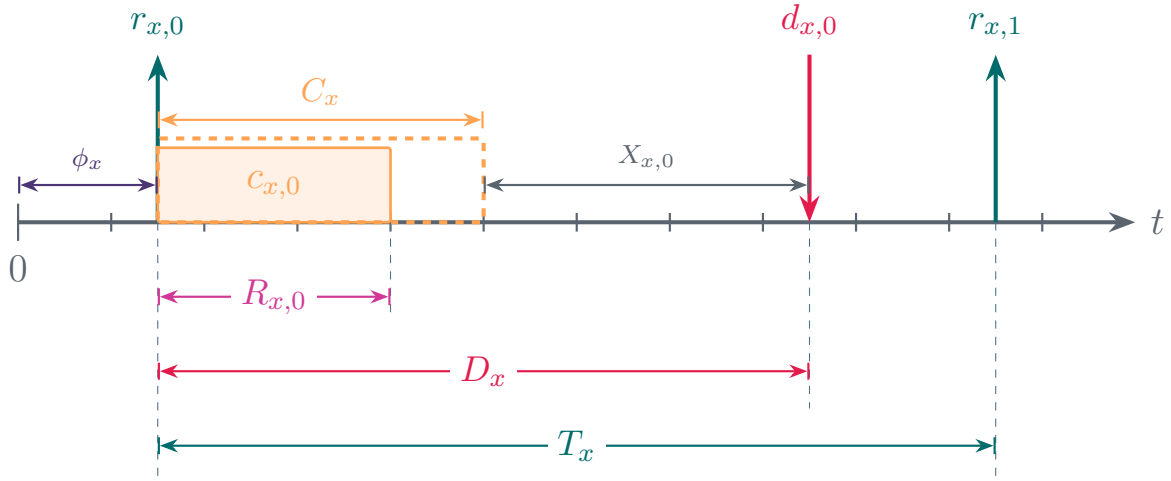


Figure 2.1.2: Model of a real-time task τ_x.

- D_x is the *relative deadline* of τ_x: each job of τ_x must complete execution within a duration D_x of its arrival; and
- T_x is the minimum inter-arrival time of τ_x: successive jobs of τ_x arrive at least T_x time units apart.

The yth job generated by τ_x, denoted by $j_{x,y}$, is characterized by the tuple

$$j_{x,y} = (r_{x,y}, c_{x,y}, d_{x,y}),$$

where

- $r_{x,y}$ is the release, arrival, or request time of $j_{x,y}$: the time at which $j_{x,y}$ becomes ready for execution.
- $c_{x,y}$ is the execution time of $j_{x,y}$: the time needed by the processor to fully execute $j_{x,y}$ without interruption, with $0 < c_{x,y} \leq C_x$.
- $d_{x,y}$ is the absolute deadline of $j_{x,y}$: the time before which $j_{x,y}$ must complete execution, with $d_{x,y} = r_{x,y} + D_x$

Hence, the task tuple captures the periodicity, deadline, and WCET specification, whereas the job tuple captures one occurrence of that specification in time.

The execution window of a job $j_{x,y}$ is the interval between its release time and its deadline. Within that window, $j_{x,y}$'s laxity or slack is defined as the maximum delay $j_{x,y}$ can tolerate after its release $r_{x,y}$ and still complete by its deadline $d_{x,y}$; at the job's release the slack is $X_{x,y} \stackrel{\text{def}}{=} d_{x,y} - r_{x,y} - C_x$. Note that delaying the execution of a job reduces its remaining laxity and increases its response time ($R_{x,y}$), that is, the time from job release until its completion or finish time.

2.1.3.1 Release Models

The most common task release or arrival models in real-time scheduling theory are periodic, sporadic, and aperiodic. For a periodic task τ_p, jobs are released strictly every period T_p, satisfying:

$$r_{p,k} = \phi_i + k \times T_p, \qquad k \in \mathbb{N}.$$

For a sporadic task τ_s, releases are irregular but satisfy the minimum separation constraint:

$$r_{s,k+1} - r_{s,k} \geq T_s.$$

For an aperiodic task τ_a, no positive lower bound exists on the inter-arrival time of successive jobs, i.e., τ_a's jobs may arrive at arbitrary points in time. Furthermore, an aperiodic task can either be deadline-constrained (i.e., firm aperiodic task) or deadline-free (i.e., soft aperiodic task).

2.1.3.2 Deadline Models

The deadline model for a task τ_x is determined by the relation between its relative deadline D_x and its minimum inter-arrival time T_x as follows:

- *implicit-deadline*: if $D_x = T_x$;
- *constrained-deadline*: if $D_x \leq T_x$;
- *arbitrary-deadline*: if no restriction is imposed on the relation between D_x and T_x, i.e., $D_x \leq T_x \vee D_x > T_x$

2.1.3.3 Utilization

A basic quantitative parameter in the task model is the utilization of a task. For a periodic or sporadic task τ_i, its utilization is defined as

$$U_x \stackrel{\text{def}}{=} \frac{C_x}{T_x},$$

that is, the fraction of processor time demanded by the task assuming worst-case recurrence, i.e., successive jobs of τ_x are separated by the minimum inter-arrival time T_x.

For a task set Γ, the total utilization or the processor utilization factor, $\mathsf{U}(\Gamma)$, is defined as the sum of each task's utilization:

$$\mathsf{U}(\Gamma) \stackrel{\text{def}}{=} \sum_{\tau_x \in \Gamma} \frac{C_x}{T_x}.$$

This quantity is fundamental in real-time scheduling theory; it appears in schedulability tests and analyses used to determine whether a given scheduling policy can meet the temporal constraints of the task set [60, 63].

2.1.3.4 Independent tasks

An *independent real-time task* is a sequence of jobs whose temporal behavior is fully characterized by task parameters (e.g., offset, WCET, relative deadline, and minimum inter-arrival time), as in the classical periodic and sporadic task models [1, 60, 61]. Independence means that:

- The release of the task's jobs does not depend on the initiation or completion of jobs of any other task, i.e., no precedence or release dependencies.
- The task's jobs interact with the jobs of other tasks only through competition for the processor: they do not hold or wait for mutually exclusive shared resources and do not synchronize on shared variables or other primitives that can cause blocking between tasks, i.e., no resource-sharing-based blocking (see Section 2.1.3.6).
- Once released, the task's job may be preempted by the scheduler but does not voluntarily suspend itself due to synchronization or waiting for external events.

For the purposes of this dissertation, we assume that independent tasks can have periodic, sporadic, or aperiodic arrival patterns and that their jobs may be either deadline-constrained or deadline-free, depending on the application semantics. In the former case, each released job must be completed by its absolute deadline; in the latter case, shorter response times improve the perceived quality of service.

2.1.3.5 Dependent tasks

A *dependent real-time task* is a sequence of jobs whose temporal behavior is not fully represented by a single independent task tuple (e.g., offset, minimum inter-arrival times, WCET, and deadline). More precisely, a task is said to be *dependent* if, in addition to its simple timing parameters, it has at least one of the following constraints [1, 60, 62, 64, 65]:

1. *Precedence or data-availability*, whereby a job becomes ready for execution only after the completion of one or more predecessor jobs of other tasks and the availability of their produced data.
2. *Mutual exclusion*, whereby jobs that are otherwise logically independent must execute in mutually exclusive fashion due to shared resources (see Section 2.1.3.6).

3. *Suspension*, whereby jobs intentionally self-suspend waiting for external events (e.g., completion of I/O or hardware accelerator task) so that their execution windows are shaped by suspension segments and associated synchronization constraints.

In such cases, the workload's semantics are more naturally described by a *structured* model (e.g., task graphs or self-suspending tasks) that explicitly encodes these internal dependencies. For the purposes of this dissertation, we assume that these internal dependencies are modeled by means of directed acyclic graphs (DAGs). Let $G_A = (V_A, E_A)$ denote the DAG associated with dependent tasks. Each vertex $v \in V_A$ represents a task, and each directed edge $(u, v) \in E_A$ represents a precedence relation requiring that u completes execution before v may start. The DAG therefore captures the partial order among these dependent tasks.

A task graph $G_{A,}$ is characterized by the tuple

$$\langle \phi_{G_{A,}}, T_{G_{A,}}, D_{G_{A,}} \rangle,$$

where $\phi_{G_{A,}}$ denotes the release offset of the first DAG instance, $T_{G_{A,}}$ the minimum separation between successive instances, and $D_{G_{A,}}$ the relative deadline of one DAG.

The k-th instance of $G_{A,}$, denoted by $G_{A,k}$, has a release time

$$r_{G_{A,k}} = \phi_{G_{A,}} + k \times T_{G_{A,}},$$

and an absolute deadline

$$d_{G_{A,k}} = r_{G_{A,k}} + D_{G_{A,}}.$$

Consider Figure 2.1.3, which shows instances i and k of two example task DAGs, $G_{0,i}$ and $G_{1,k}$. Within these DAG instances, vertices are classified based on their precedence relations: those with no predecessors are *entry jobs* ($j_{a,i}$ and $j_{c,k}$), those with no successors are *exit jobs* ($j_{b,i}$, $j_{e,k}$, and $j_{f,k}$), and those with both are *inside jobs* ($j_{d,k}$). This distinction is crucial because boundary jobs (entry and exit) typically inherit their timing constraints directly from the overall DAG instance.

For example, as illustrated in Figure 2.1.3, the entry jobs $j_{a,i}$ and $j_{c,k}$ implicitly share the release time of their respective DAG instance, $r_{G_{0,i}}$ and $r_{G_{1,k}}$. The exit jobs $j_{b,i}$, $j_{e,k}$ and $j_{f,k}$ share the absolute deadlines, $d_{G_{0,i}}$ and $d_{G_{1,k}}$ of their DAG instance. The boundary jobs of different instances can be subjected to strict periodic activation constraints, e.g., when driven by sensing and actuation requirements in a control loop. The execution windows for inside jobs (e.g., $j_{d,k}$ in $G_{1,k}$) do not have to be strictly periodic. They may not have a priori assigned release times and deadlines, as indicated by the '-' in Figure 2.1.3. Their feasible execution windows are determined after resolving the graph's precedence

relations.

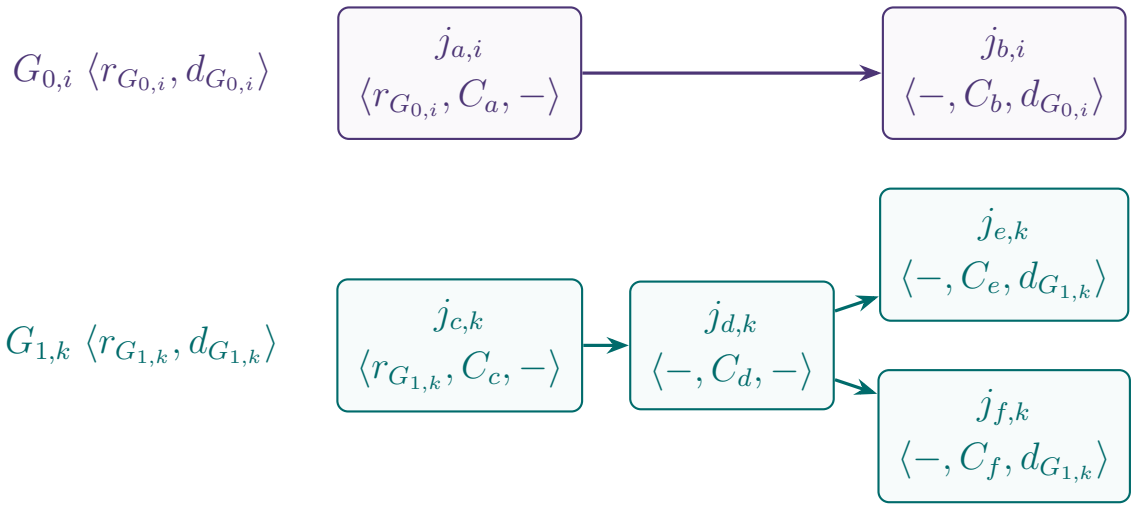


Figure 2.1.3: Example of two precedence DAG instances, $G_{0,i}$ and $G_{1,k}$.

In the distributed setting, task DAGs have vertices that include both computational tasks and inter-node messages, allowing temporal precedence and data-flow relations across multiple nodes to be captured. Messages are modeled similarly to tasks, with transmission times, periods, and deadlines. The sequence of alternating tasks and messages that transports data from sensors through processing stages to actuators is called a *cause–effect chain* (CEC). Each CEC is associated with an end-to-end deadline that constrains the total latency along its path. The end-to-end deadline is typically decomposed into local deadlines assigned to the individual vertices on the chain, such that satisfying these local constraints guarantees the end-to-end deadline of the CEC.

2.1.3.6 Resource Constraints

A *resource* is any software or hardware entity that a task may use to advance its execution. Typical examples include data structures, variables, regions of main memory, files, or registers of a peripheral device. A resource dedicated to a single task is said to be *private*, whereas a resource that may be used by multiple tasks is referred to as a *shared* resource. A shared resource that is protected against concurrent access is called an *exclusive* resource.

Shared resources must be accessed under *mutual exclusion* to maintain system consistency and prevent data corruption. Mutual exclusion is a concurrency-control property ensuring that access to a shared resource is strictly serialized: if one job is currently using the resource, any other job requesting access must wait until the resource is explicitly released.

The portion of code executed under mutual-exclusion constraints is referred to as the

critical section. Any job that wishes to enter its critical section must wait until no other job is holding the associated resource. A job waiting for an exclusive resource is said to be *blocked* on that resource; otherwise, it may proceed by acquiring the resource. When the job leaves its critical section, the associated resource becomes free and can be allocated to another waiting task, if any [60].

If a lower-priority job j_L is executing within a critical section and holds a resource needed by a higher-priority job j_H, then j_H is forced to wait, experiencing a *blocking delay*. Left uncontrolled, this situation can lead to *priority inversion*, as illustrated in Figure 2.1.4: a scenario in which a high-priority job j_H is indirectly delayed by an arbitrary chain of medium-priority jobs j_M that preempt the lower-priority resource holder j_L. This can extend the blocking duration, potentially causing a deadline miss despite the system not being overloaded.

Access to shared resources is regulated and serialized to bound resource-sharing-based blocking delays and improve timing predictability. The specific mechanisms and protocols used for handling shared resources depend on the underlying scheduling approach. For example, online scheduling requires explicit resource access protocols, whereas offline scheduling serializes all accesses to shared resources at design time. These mechanisms will be briefly discussed in their respective sections.

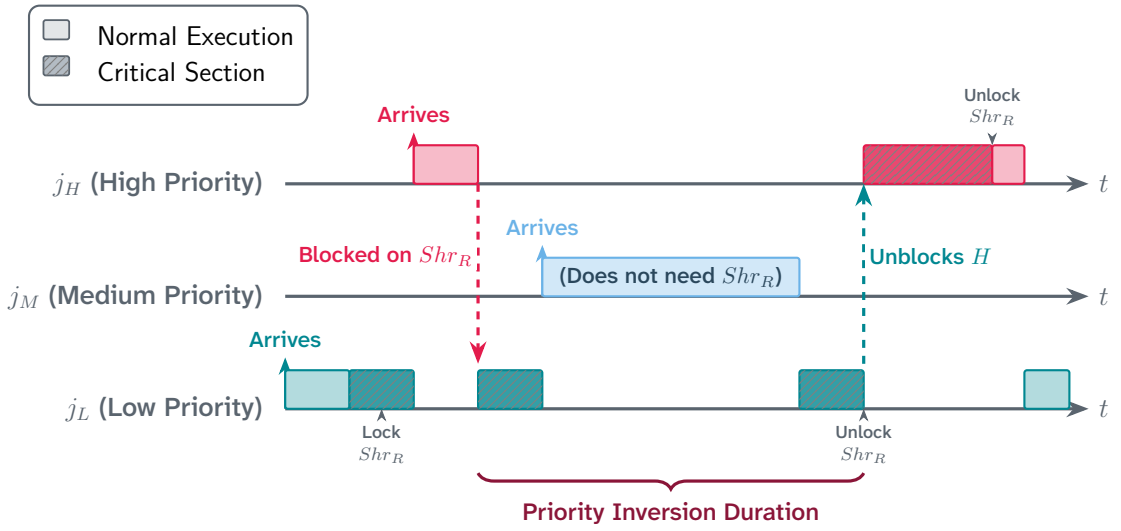


Figure 2.1.4: Illustration of priority inversion. A high-priority job j_H is blocked waiting for resource Shr_R held by a low-priority job j_L. A medium-priority job j_M preempts j_L, extending the blocking delay experienced by j_H.

2.1.4 Real-Time System Classes and Utility-Based View

A real-time system is a system that provides at least one real-time service, defined as a service that is required to be delivered within time intervals dictated by the environment.

In this context, service delivery may consist solely of sampling the state of RT entities at the required times [66].

Classical taxonomies distinguish between *hard* and *soft* real-time systems according to the admissibility and consequences of deadline misses in the services they provide [1, 60]. A real-time system is classified as follows:

- **Hard real-time system** if it provides at least one hard real-time task such that any deadline miss of that task constitutes an unacceptable failure due to the entailed catastrophic consequences.

- **Soft real-time system** if timing remains important, yet timing failures are benign. Missing the deadline of a soft real-time task does not entail catastrophic consequences. Instead, system behavior is characterized by quality-of-service metrics (e.g., average response time, deadline-miss ratio); lateness degrades performance but does not, by itself, constitute system failure. Within the soft real-time class, a task is called firm if it has a deadline after which computed results are discarded or treated as useless, but occasional misses within specified bounds may be acceptable from the application point of view. Consequently, the term 'soft' encompasses a range of distinct timing properties and utility profiles.

Utility-Based Unifying View

The hard and soft taxonomy delivers limited expressiveness when tasks may carry different importance values or when the value of a service provisioning varies with time. A more general abstraction associates with each real-time service within a system a *time-dependent utility function* $Utility(t)$ that maps the completion time t of a job (measured relative to a reference instant such as its release time) to an application-specific notion of usefulness or value [66, 67].

Three temporal markers structure the domain of $Utility(t)$: the *liveline* t_L denotes the earliest instant at which service delivery is admissible; the *targetline* t^* denotes the instant of maximum utility, which system designers aim to achieve; and the *deadline* $t_D \geq t^*$ denotes the latest admissible instant for service delivery. In the simplest case, t_L coincides with the job release time and utility is constant between t_L and t_D, yielding the classical deadline-only model.

Different real-time requirements correspond, as illustrated in Figure 2.1.5 to qualitatively different shapes of $Utility(t)$ [60, 66, 68]:

- **Hard real-time:** $Utility(t)$ is constant and maximal for all completion times up to a hard deadline t_D, and drops sharply, to very large negative values, for any $t > t_D$, modeling the fact that any deadline miss constitutes a system failure.

For certain safety-critical control functions (e.g., actuation that is dangerous if too early or too late), the $Utility(t)$ can be shaped by a *bounded admissible interval* $[t_L, t_D]$, with positive utility only within the interval. An example of hard real-time functionality is an airbag or emergency braking in a vehicle: actuation before or after the allowed window can compromise safety.

- **Firm real-time:** $Utility(t)$ is maximal up to a deadline and drops discontinuously to zero for $t > t_D$: late results are useless but not hazardous. A representative example is real-time audio-visual decoding, where an outdated video frame is discarded rather than displayed with delay. Similarly, a weapon-release computation that completes after the engagement window closes is discarded as operationally irrelevant, without itself endangering the platform.

- **On-time (jitter-sensitive):** $Utility(t)$ reaches its maximum at a target instant t^* and decreases monotonically with the absolute deviation $|t - t^*|$; both early and late completion are penalized. This jitter sensitive model arises in applications such as real-time musical note sequencing and audio rendering.

- **Soft real-time:** $Utility(t)$ is maximized for early completion and decays *smoothly* as completion deviates beyond a reference instant, capturing graceful quality degradation. A typical instance is adaptive multimedia streaming, where increased latency or jitter degrades user-perceived quality without constituting a correctness failure.

- **Non-real-time (best-effort):** $Utility(t)$ remains constant and positive for all t; there is no strict deadline, and later completion is still preferable to non-completion.

From an architectural standpoint, the utility-based view is valuable because it makes explicit which services require unconditional timeliness guarantees and which may degrade gracefully under overload or faults [66, 68].

Orthogonal Classifications

Beyond deadline criticality, real-time systems are classified along further orthogonal axes [1, 58]:

- **Criticality level.** Safety-critical, mission-critical, and non-critical systems differ in the severity of consequences under failure. In *safety-critical* systems, failure may cause harm to humans, damage to the environment, or severe material loss. In *mission-critical* systems, failure may not immediately endanger safety, but it

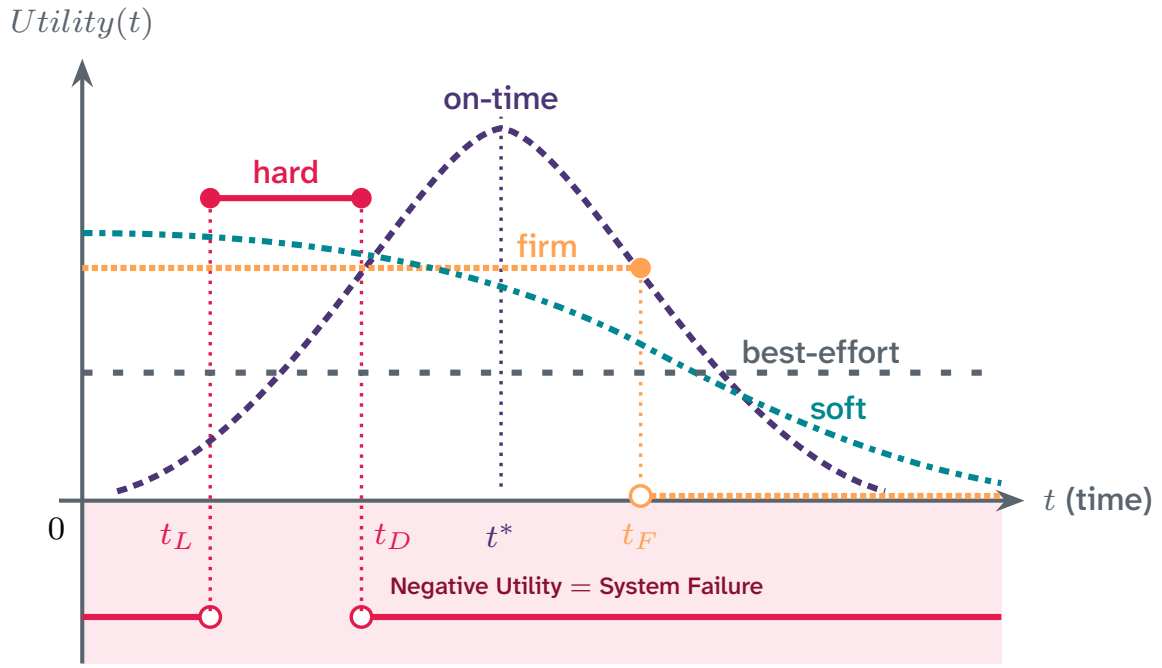


Figure 2.1.5: Illustrative $Utility(t)$ profiles for hard (bounded admissible window $[t_L, t_D]$ with negative utility outside), on-time (jitter-sensitive, peaking at targetline t^*), firm (drops discontinuously to zero at t_F); soft (smooth positive decay), and non-real-time (best-effort) tasks [60, 66, 67].

prevents the successful completion of an essential mission or service and can therefore lead to major operational, economic, or strategic losses. By contrast, failure in *non-critical* systems is generally tolerable and primarily results in degraded quality of service or user inconvenience.

In *mixed-criticality systems* (MCS), functions are assigned discrete criticality levels and multiple worst-case execution time (WCET) estimates reflecting different levels of assurance; under adverse conditions, the system guarantees timing only for the highest-criticality subset while potentially withdrawing guarantees from lower-criticality functions [69, 70].

- **Failure behavior:** *fail-safe* systems transition to, or remain in, a safe state upon detected failure, prioritizing safety over continued functionality. *Fail-operational* systems, by contrast, are designed to continue to provide a possibly degraded but functionally correct subset of critical services despite the failure of one or more components, typically through redundancy and active replication [1, 66].

  The distinction is architecturally significant since fail-operational systems require stronger fault-tolerance mechanisms compared to fail-safe ones to maintain service delivery.

2.1.5 Time and Clocks

Basic constants in physics are defined with respect to the physical second. The same unit of time governs both the evolution of the controlled physical plant and the time base of the real-time computer system that monitors and controls the physical plant. Thus, timing constraints in a real-time computer system must be interpreted with respect to a common physical time scale shared between the computer and its environment.

Real-time systems use a model of time grounded in Newtonian physics, in which time is treated as an independent variable unaffected by relativistic effects. The timeline is represented as a dense and ordered set of instants, providing the basis for reasoning about the temporal order of events, simultaneity, and durations.

A computing node does not observe reference time directly, but only through a local digital physical clock, which consists of an oscillator that generates periodic physical events and a counter that counts them. These oscillator events are called *microticks* (mt) and define the local granularity of the clock. Let $C_i(t)$ denote the reading of the local physical clock of node i at reference time t. Because physical clocks advance in discrete microticks, time is represented with finite granularity rather than as a continuum, which introduces an inherent digitization error [1].

A digital physical clock has two distinguishable aspects: its *phase*, captured by the counter reading $C_i(t)$, and its *frequency*, determined by the rate at which the oscillator generates microticks. Since no real oscillator is perfect, the rate of a digital physical clock drifts slightly from the nominal frequency. A standard bounded-drift (frequency error) assumption between two instants t_1 and t_2 is

$$(1-\rho)\,(t_2-t_1) \;\leq\; C_i(t_2)-C_i(t_1) \;\leq\; (1+\rho)\,(t_2-t_1),$$

where ρ denotes the maximum drift rate. Consequently, free-running clocks drift apart over time and cannot, by themselves, provide a common temporal reference across distributed nodes [1].

Distributed real-time systems, therefore, rely on clock synchronization algorithms to keep the digital physical clocks of nodes within a defined maximum offset from each other [6]. It is useful to distinguish two complementary notions:

- *Internal clock synchronization* maintains the clocks of all nodes in an ensemble approximately aligned within an error bound referred to as the *precision* π, defined as the maximum clock offset between any two correct local clocks in the ensemble at the same instant, measured with respect to reference time.

- *External clock synchronization* aligns a local clock or an ensemble of clocks with an external time reference such as *Coordinated Universal Time* (UTC). Its quality

is characterized by the *accuracy* α, defined as the maximum offset between a correct local clock and the external reference.

If an ensemble of clocks is externally synchronized within an accuracy α, then it is also internally synchronized within a precision $\pi = 2 \times \alpha$. The converse, however, does not hold [1].

Since clocks will continue to drift apart after synchronization, they must be re-synchronized periodically ensure that their accumulated deviation never exceeds the target precision π. If two clocks are synchronized to within π at a some instant, they will drift apart by at most $2\rho\, T_{\text{resync}}$ over a subsequent interval of duration T_{resync}. Keeping the deviation between clocks within π therefore requires that the re-synchronization interval satisfy

$$T_{\text{resync}} \;\leq\; \frac{\pi}{2\rho},$$

directly linking the achievable precision, the oscillator quality, and the required frequency of synchronization rounds [6].

The quality of clock synchronization is worsened by faults and communication jitter ϵ, which represents the uncertainty in message transmission delays. This uncertainty fundamentally limits the extent to which clocks can be synchronized in a distributed system. In a fully connected network where message delays lie within the interval $[\delta_{\text{min}}, \delta_{\text{max}}]$, i.e., where $\epsilon = \delta_{\text{max}} - \delta_{\text{min}}$, Lundelius and Lynch showed in [71] that the precision π between any two clocks cannot be made smaller than

$$\epsilon \left(1 - \frac{1}{N}\right)$$

even in the absence of faults. Thus, perfect clock synchronization is impossible: a residual uncertainty always remains and cannot be eliminated by any algorithm.

A *global time base* is an abstraction built on top of an ensemble of approximately synchronized clocks to provide a system-wide notion of time. The granularity of global time, referred to as *macroticks* (MTs), is defined by designating a sequence of subsequent microticks on each local clock to span one MT. The number of microticks composing a macrotick may differ across nodes, since local clocks may run at different speeds, but the duration of each MT must be identical system-wide.

Let $|MT|$ denote the duration of one macrotick and π the precision of the clock ensemble. The *reasonableness condition* for establishing a global time base is that $|MT| > \pi$, ensuring that the difference between any two correct clocks remains strictly smaller than one MT. Under this condition, an event observed concurrently by different nodes, as illustrated in Figure 2.1.6, will receive timestamps that differ by at most one MT [1]. The global time base thus provides a shared temporal scale that is anchored in physical time and yet implementable on top of imperfect local clocks.

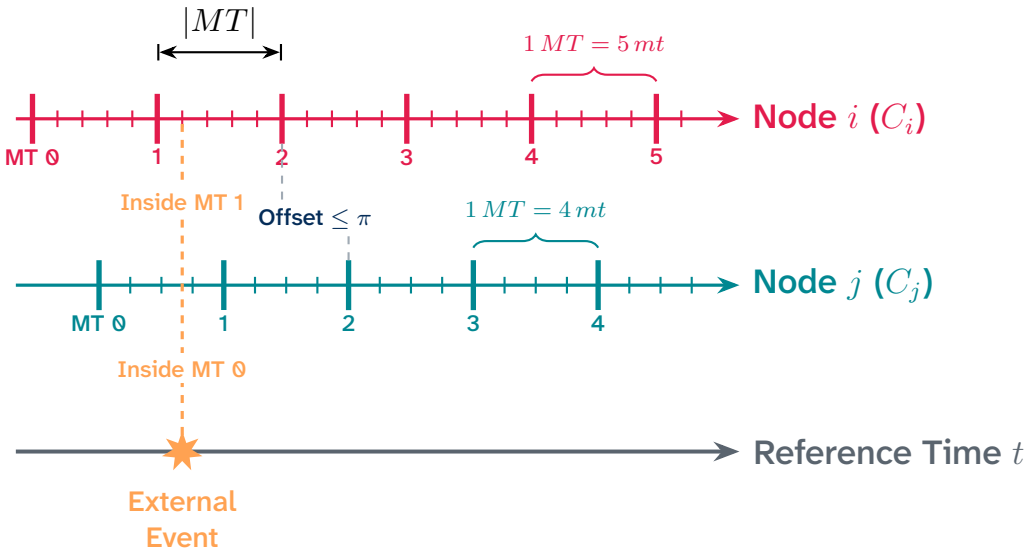


Figure 2.1.6: Illustration of the global time base and the reasonableness condition ($|MT| > \pi$). Despite a phase offset $\leq \pi$ and different microtick (mt) rates between the fast clock on Node i and the slow clock on Node j, an external event is timestamped with at most one MT of difference.

2.1.6 Scheduling in Real-Time Systems

Real-time scheduling concerns the allocation of jobs (and their associated messages) to computational resources such as processors or nodes, and to communication resources such as network channels, together with the determination of when these jobs and messages are executed and transmitted on those resources. The objective is to construct a temporal and resource allocation such that all timing constraints, e.g., deadlines, release times, precedence relations and WCETs are satisfied [60].

A schedule formalizes this allocation and ordering. For a processor P_k, a schedule can be represented by a function

$$\sigma^k : \mathbb{R}_{\geq 0} \rightarrow \mathcal{J}_k \cup \{\perp\},$$

where $\mathcal{J}_k$ denotes the set of jobs allocated to processor P_k, $\sigma^k(t) = j_{x,y}$ indicates that job $j_{x,y}$ is executing on P_k at time t, and $\sigma^k(t) = \perp$ denotes that processor P_k is idle at time t. If the execution of a job j_x may be interrupted, then it can span several disjoint intervals, whose total duration must be sufficient to accommodate j_x's WCET, C_x.

A schedule is called *feasible* if all temporal constraints of the considered activities (e.g., tasks and messages) are satisfied. A task set is called *schedulable* if there exists at least one algorithm that can produce a feasible schedule. An algorithm is said to be *optimal* when a cost function is defined over the task set and the algorithm always produces a schedule that minimizes this cost. When no cost function is defined and

the objective is solely to determine feasibility, an algorithm is considered *optimal* or, more precisely, *exact*, if it is able to construct a feasible schedule for every task set that admits one [60].

2.1.6.1 Job States

The task model introduced in Section 2.1.3 specifies the *temporal parameters* of recurrent activities. It does not, however, characterize the runtime *states* through which a job progresses from its creation until completion.Following [60], a job $j_{x,y}$ generated by task τ_x transitions through the states illustrated in Figure 2.1.7:

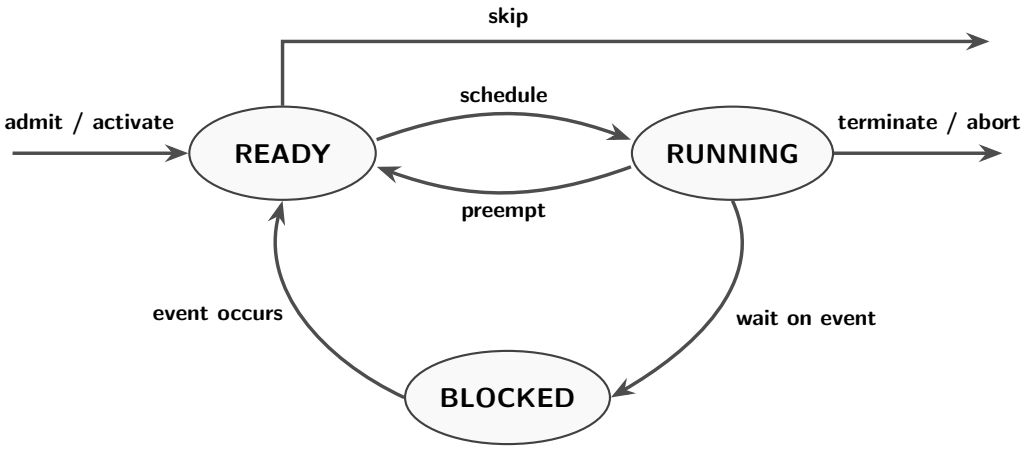


Figure 2.1.7: States of a real-time job.

Ready. Prior to the release time $r_{x,y}$, job $j_{x,y}$ has not yet been created by the system. Task τ_x itself is said to be *inactive* before its first job is generated. τ_x is *admitted* to the system task set Γ when the scheduler determines, possibly at runtime, that it can be accommodated without violating the timing constraints of the already-admitted tasks. A job $j_{x,y}$ is ready if it has all data required to begin execution and is waiting only for processor allocation. The ready state is the entry condition for all scheduling decisions. At any given time t, all ready jobs reside in the *ready queue* $\mathcal{Q}(t)$. Job $j_{x,y}$ enters $\mathcal{Q}(t)$ upon its release, upon admission into the system, or when it returns from the running or waiting states after the occurrence of an event that blocked its progress. Conversely, it is removed from $\mathcal{Q}(t)$ when it is dispatched for execution on the processor or skipped, i.e., the job will not execute at all.

Running. A job $j_{x,y}$ is in the running state while it is actively executing on the processor. Here, it is important to distinguish the *scheduler*, which implements the set of rules that decide *which* job executes next (e.g., by sorting $\mathcal{Q}$), from the *dispatcher*, which

is the low-level runtime mechanism that enacts this decision. At each scheduling point (e.g., a job release or completion), the dispatcher removes the head of $\mathcal{Q}$ (i.e., the job selected by the scheduler) and grants it processor access. If the scheduling policy is *preemptive* (see Section 2.1.6.2), the dispatcher may interrupt $j_{x,y}$ in favour of a newly arrived higher-priority job. Consequently, a job's execution may spread across several disjoint intervals, under preemptive scheduling policy.

Once job $j_{x,y}$ has completed execution after accumulating $c_{x,y}$ units of processor time, it terminates. It is the responsibility of the scheduler to enforce that $c_{x,y} \leq C_x$, ensuring that $j_{x,y}$ does not overrun its allocated WCET budget.

If $j_{x,y}$ has not completed by its absolute deadline $d_{x,y} = r_{x,y} + D_x$, the scheduler may abort $j_{x,y}$'s execution. The abortion process must be performed safely, since terminating a job while it is modifying shared data may leave the system in an inconsistent state and thus jeopardize correct system behavior [60].

Blocked. A running job transitions to the blocked state when it must wait for a condition it cannot proceed without, such as access to a mutually exclusive shared resource currently held by another job (Section 2.1.3.6), or the arrival of an external event like an I/O completion. A blocked job is temporarily withdrawn from the ready queue $\mathcal{Q}(t)$ until the blocking condition is resolved. During this time, the job is held in a queue associated with the specific event it is waiting for.

2.1.6.2 Preemption Models

The preemption model formalizes under which conditions an executing job can be interrupted. It is a key design parameter affecting schedulability, implementation overhead, and timing predictability. Three main classes are distinguished in literature, as illustrated in Figure 2.1.8:

Fully preemptive. The scheduler may interrupt the currently executing job at any instant when a higher-priority job becomes ready. While this simplifies schedulability analyses, it incurs platform-dependent overheads and delays. For instance, preemption requires a *context switch*, i.e., saving the execution context (registers, stack pointer, etc.) so that the job can return to the ready state and resume execution later. This process incurs non-zero time overhead. Furthermore, preemption affects cache behavior: interrupting a job disrupts program locality and increases the number of cache misses due to cache lines being evicted by the preempting job. These delays increase the WCET of a job and must be accounted for in the WCET analysis.

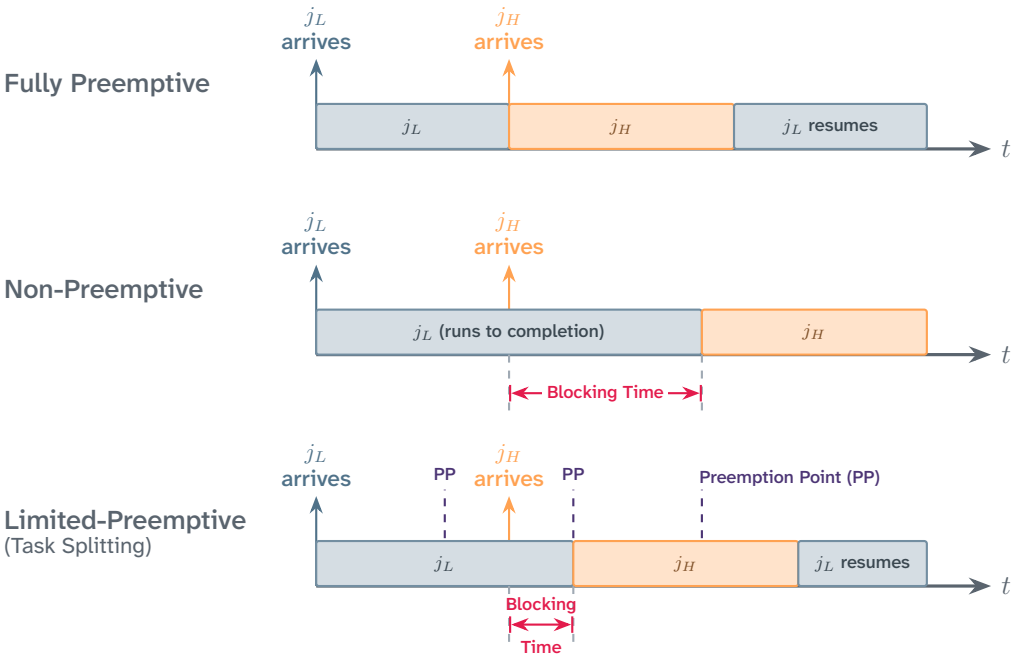


Figure 2.1.8: Illustration of fully preemptive, non-preemptive, and limited-preemptive execution models. A high-priority job (j_H) arrives while a low-priority job (j_L) is executing, demonstrating the differences in blocking time and preemption behavior.

Non-preemptive. Once a job starts executing, it runs to completion without interruption. This eliminates preemption overhead and simplifies mutual exclusion, but can reduce schedulability significantly because a higher-priority job may be blocked by the longest non-preemptive execution segment of any lower-priority job. A set of non-preemptive jobs can be viewed as a set of jobs sharing a single resource (the processor) for their entire WCET; that is, the whole job execution constitutes a critical section that, once started, prevents all other tasks from accessing the processor until it completes.

Limited-preemptive. Limited-preemptive approaches seek to balance the two extremes by constraining preemptions to a finite set of locations (e.g., preemption points introduced by explicit task splitting) or to specific conditions (e.g., deferred preemption or preemption thresholds). Under these approaches, an important objective is to identify, for each task, the longest non-preemptive interval that still preserves feasibility, assuming the task set is initially feasible under the fully preemptive model. We refer the reader to [60] for a more in-depth comparison of limited-preemptive approaches.

2.1.6.3 Computational Complexity of Scheduling

Real-time scheduling problems vary in computational complexity depending on the system model and constraints considered. In this context, it is helpful to distinguish between two fundamental problem types: *decision problems* and *optimization problems*.

A *decision problem* asks a yes or no question about feasibility, such as: "Does a valid schedule exist that satisfies all timing constraints?". In contrast, an *optimization problem* seeks the best possible schedule according to a specific metric, such as: "What is the minimum number of processors needed to satisfy all deadlines?".

This distinction is important from a computational complexity perspective. For decision problems, if a candidate schedule is provided as proof of feasibility, its correctness can be checked efficiently. That is, an algorithm can check in polynomial time whether every job respects its release time, deadline, and precedence and mutual exclusion constraints. Problems whose solutions can be checked in polynomial time belong to the complexity class *NP* (Nondeterministic Polynomial time).

However, finding a feasible schedule from scratch is typically much harder than verifying one[1]. No efficient (i.e., polynomial time) algorithm is known for any NP-complete problem and it is widely believed in computer science that no such algorithm exists [72,73]. Instead, exact scheduling algorithms for NP-complete problems generally have a computational complexity that grows *exponentially* with the number of tasks, which limits their applicability to large problem instances.

Optimization problems are at least as difficult as their decision counterparts. For example, finding the shortest possible schedule length (makespan) is an *NP-hard* problem because solving it automatically answers the decision problem "Is the makespan less than or equal to L?" for any value of L. Moreover, NP-hard problems are typically difficult not only to solve optimally but also to verify optimality, since confirming that no better solution exists is itself as hard as solving the optimization problem.

Sources of Computational Complexity

Computational intractability arises when any of the following are present:

1. **Asynchronous periodic tasks:** Real-time control systems consist of recurring tasks characterized by execution times, deadlines, periods, and distinct initial offsets (phases). Verifying the schedulability of such asynchronous and independent periodic tasks is strongly *coNP-complete* [63]. (The class coNP is used because it is often easier to check *unschedulability* by identifying a specific time window

[1] When a decision problem is classified as *NP-complete*, it means two things: first, checking schedulability is easy (it is in NP); second, the problem is among the hardest in the NP class, because every other problem in NP can be mathematically transformed into it in polynomial time.

where demand exceeds processor capacity). The intractability here arises because the offsets shift the relative release times of the tasks, forcing an exact analysis to check an exponentially growing number of task releases.

2. **Shared resources:** Real-time systems typically include mutually exclusive resources (e.g., shared memory region or variables). Introducing such resource constraints introduce NP-completeness even on uniprocessor systems [72]. The complexity arises because blocking behavior becomes state-dependent: a high-priority job may be delayed by a lower-priority job holding a lock on the resource, creating complex interference patterns.

3. **Precedence constraints:** In many systems, a job J_i must complete before J_j can begin. Ullman proved that on two or more identical processors, deciding whether a set of non-preemptive, precedence-constrained jobs can meet a common deadline is NP-complete [74]. The difficulty arises from the exponential number of valid topological orders and processor assignments that must be explored to determine whether any feasible schedule exists. Even on a uniprocessor system, deciding whether precedence-constrained jobs with release times and individual deadlines admit a feasible schedule is NP-complete [75].

4. **Multiprocessor allocation:** Even without precedence constraints, deciding whether a set of independent jobs can be feasibly scheduled on a fixed number of identical multiprocessors is NP-complete [76]. This hardness is closely related to the bin-packing problem, where the scheduler must distribute jobs across processors without exceeding the capacity of each processor.

Practical Implications

The real-time scheduling literature reflects the field's historical and ongoing emphasis on design trade-offs. Since optimal or exact exponential-time algorithms scale poorly, designers and researchers rely on:

- *Offline scheduling*, where the exponential cost of finding a schedule is paid once before deployment as in native time-triggered systems (see Section 2.1.6.6). For larger problem instances, offline scheduling can be paired with *heuristic algorithms* to quickly find feasible, though not necessarily optimal, schedules for complex multiprocessor and distributed systems.

- *Restricted models and platform assumptions*. For example, uniprocessor platforms with independent, bounded utilization, and synchronous task set systems

to reduce problem complexity and admit efficient exact or sufficient schedulability tests (see Section 2.1.6.5). Such restrictions reduce generality, but enable tractable analysis.

Understanding these underlying theoretical limits is essential for making principled algorithm and system architecture choices.

2.1.6.4 Online Scheduling

Online scheduling addresses the runtime decision problem of determining, at each scheduling instant, which ready job executes on a processor or which pending message accesses a communication resource. It is the dominant approach in event-triggered systems, where the scheduler is invoked by the occurrence of events such as job releases, job completions, or message arrivals, and where at each invocation, the scheduler selects the next job or message based on the current state of the ready queue and a predefined set of rules (e.g., a priority assignment) [1, 60].

Making scheduling decisions online provides a high degree of flexibility. It enables efficient handling of tasks whose temporal properties are not known a priori, such as aperiodic tasks that may arrive at arbitrary times [60], and supports mechanisms for reacting to unforeseen conditions at runtime. For example, the system can be equipped to dynamically guarantee higher execution rates for a real-time task when needed, even if such behavior was not anticipated at design time [77].

In this section, we classify online scheduling along three axes:

- the **processor model**: uniprocessor or multiprocessor,
- the **preemption model**: preemptive or non-preemptive (Section 2.1.6.2), and
- the **task model**: periodic, sporadic, or aperiodic tasks with implicit, constrained, or arbitrary deadlines (Section 2.1.3).

2.1.6.4.1 Fixed-Priority and Dynamic-Priority Scheduling

Online schedulers are typically *priority-based*, i.e., at each scheduling instant, the scheduler selects, among all ready jobs, the job with the highest priority (with ties broken according to some tie-breaking rule).

In *fixed-priority* (FP) scheduling, each task τ_i is assigned a static priority before runtime. Every job of τ_i inherits this priority, and the relative order of tasks does not change over time. Classical examples include *Rate Monotonic* (RM) and *Deadline Monotonic* (DM), where priorities are derived from periods or relative deadlines [61,78] FP scheduling is widely supported in real-time operating systems (e.g., FreeRTOS [79], Zephyr [80]

and AUTOSAR OS [81]) and is attractive due to its implementation simplicity and well-developed analysis techniques.

In *dynamic-priority* (DP) scheduling, a job's priority may change over time, typically as a function of its urgency (e.g., closeness of its deadline or remaining laxity within its execution window). The classical algorithm in the DP scheduling category is the *Earliest Deadline First* (EDF) algorithm, which assigns at each scheduling instant the highest priority to the ready job with the earliest absolute deadline. [60,61] DP policies generally achieve better processor utilization on uniprocessors at the cost of a more complex implementation.

2.1.6.4.2 Fully Preemptive Uniprocessor Scheduling

The classical baseline, originating with Liu and Layland [61], is a fully preemptive uniprocessor model with independent periodic tasks: each task generates an infinite sequence of jobs released periodically with a constant period, has a WCET not exceeding its period, implicit deadline and identical offset, i.e., the first job by all tasks is released at the same time instant. Additionally, no precedence constraints or mutual-exclusion constraints are assumed in this setting. Subsequent work typically adopts the same structural assumptions when generalizing from periodic to *sporadic* tasks by treating the period as a minimum inter-arrival time [63,82]. Within this model, classical optimality and utilization results are known for both *FP* and *DP* scheduling.

Fixed priorities: RM and DM.

Consider a set Γ of periodic or sporadic tasks scheduled on a single fully preemptive processor, under the assumptions above. For *implicit-deadline* periodic tasks ($D_x = T_x$), the *Rate Monotonic* (RM) policy assigns priorities inversely proportional to periods: tasks with shorter periods receive higher priorities. In this model, RM is optimal among all FP assignments: if a task set in this model cannot be scheduled by RM, then no other FP assignment can [61].

Furthermore, RM under this model admits simple utilization-based and sufficient tests that are useful for early design and quick admission control. Let $\mathsf{U}(\Gamma) = \sum_{\tau_x \in \Gamma} C_x/T_x$ be the total utilization. Liu and Layland derived the well-known sufficient schedulability bound

$$\mathsf{U}(\Gamma) \;\leq\; n\big(2^{1/n} - 1\big),$$

which converges to ≈ 0.693 as the number of tasks n increases [61]. This bound is *sufficient but not necessary*: whenever it holds, the task set is schedulable, but some schedulable task sets violate it [60].

A tighter sufficient bound, called the *hyperbolic bound*, was derived later by Bini and

Buttazzo in [83], which states that a task set is schedulable under RM if

$$\prod_{\tau_x \in \Gamma} (1 + U_x) \leq 2$$

This bound accepts many task sets that fail the Liu-Layland's test, while remaining inexpensive to evaluate.

For *constrained-deadline* periodic or sporadic tasks ($D_i \leq T_i$), the DM algorithm assigns priorities inversely proportional to relative deadlines and is optimal among all fixed-priority assignments under the same fully preemptive uniprocessor assumptions [60, 78].

For both implicit- and constrained-deadline fixed-priority systems, *Response-Time Analysis* (RTA) gives an *exact* feasibility test: it is necessary and sufficient, so a task set is schedulable if and only if every task's worst-case response time does not exceed its deadline [60]. This is illustrated in Figure 2.1.9, where the exact response-time-based region strictly contains the more conservative Liu and Layland and hyperbolic sufficient regions.

Dynamic priorities: EDF.

The EDF algorithm assigns, at each scheduling instant, the highest priority to the ready job with the earliest absolute deadline. Under the fully preemptive uniprocessor setting, EDF is *optimal*: if any online algorithm can schedule the task set without deadline misses, EDF can as well. Moreover, in the implicit-deadline model, EDF schedulability is exactly characterized by the utilization as follows:

$$\mathsf{U}(\Gamma) \leq 1$$

which is exact, i.e., both necessary and sufficient for feasibility on a fully preemptive uniprocessor system [61, 63] (see Figure 2.1.9).

In order to demonstrate EDF's ability to fully utilize processor capacity compared to RM, consider a task set where $\mathsf{U}(\Gamma) = 1.0$, defined by $\tau_1 = (0, 2, 4, 4)$ and $\tau_2 = (0, 5, 10, 10)$. Under RM, τ_1 receives the higher priority due to its shorter period. As shown in Figure 2.1.10, the job of τ_1 released at $t = 8$ preempts τ_2, causing τ_2 to miss its deadline at $t = 10$. Conversely, under EDF at $t = 8$, the newly released job of τ_1 has deadline $d_1 = 12$, while the currently running job of τ_2 has deadline $d_2 = 10$. Because its deadline is earlier, τ_2 retains the highest priority, completes at $t = 9$, and the task set remains feasible.

For *constrained- and arbitrary-deadline* periodic or sporadic tasks, EDF feasibility cannot be decided based on the total utilization of the task set alone. Instead, it is determined by the *Processor Demand Analysis* (PDA), which compares the cumulative execution demand of all jobs that must execute within an interval to the available processor

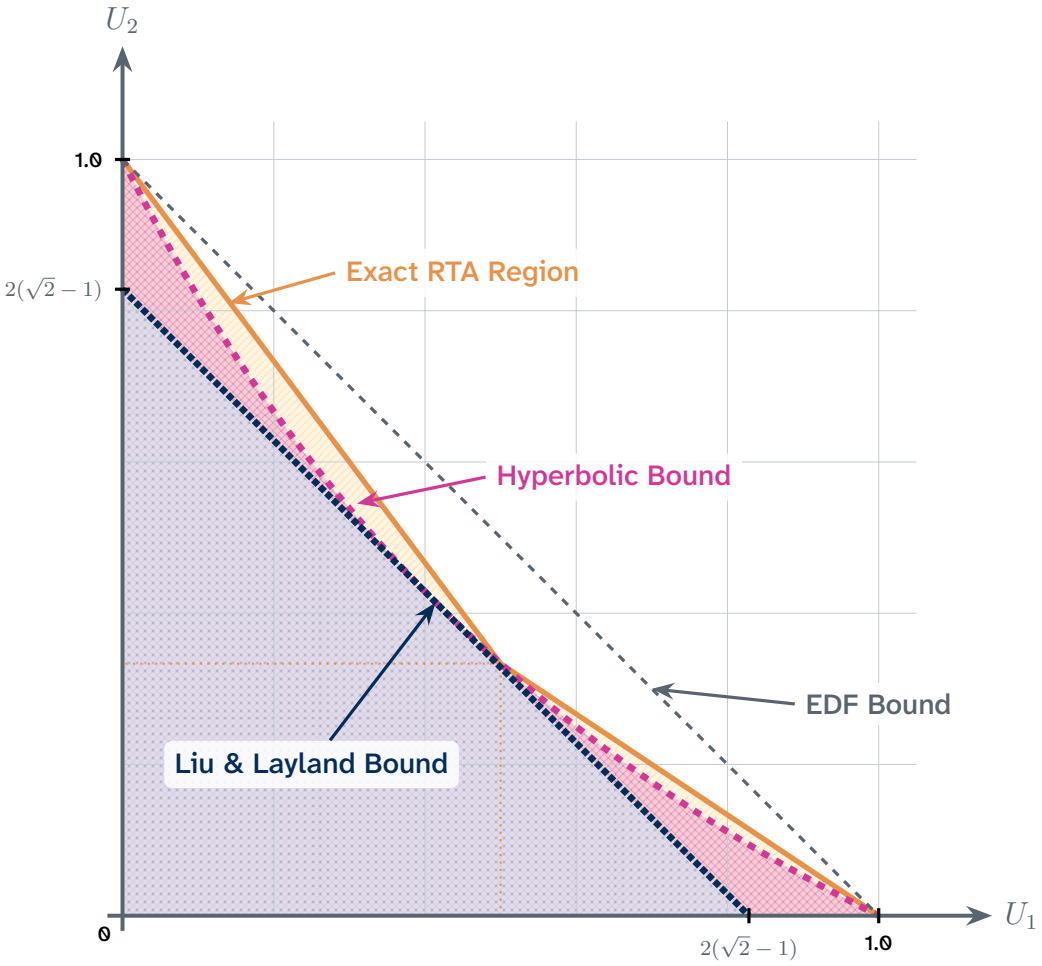
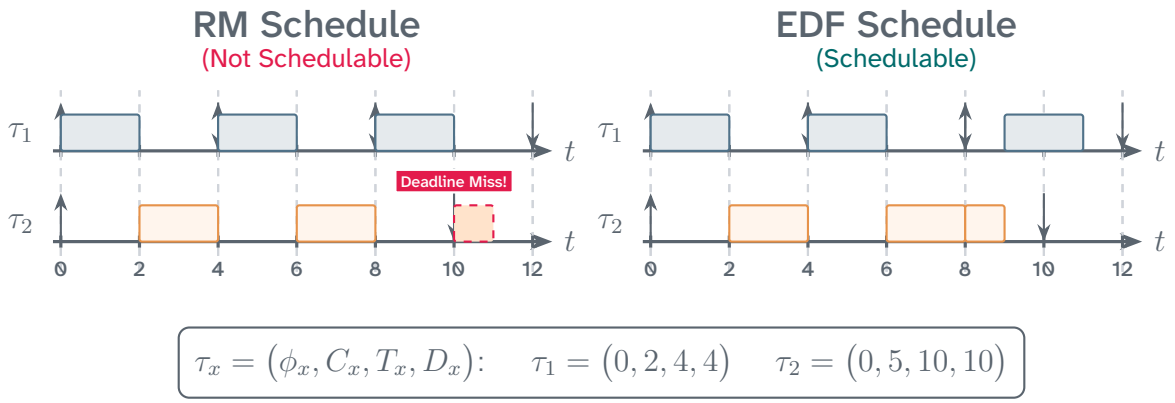


Figure 2.1.9: Comparison of schedulability regions in utilization space ($U_1 \times U_2$) for two implicit-deadline tasks under RM scheduling. The most conservative *Liu & Layland bound* limits total utilization to $2(\sqrt{2}-1) \approx 0.83$. The *Hyperbolic bound* encompasses a larger admissible region. The *Exact RTA region*, contains both sufficient bounds. The *EDF bound* covers the entire region, demonstrating EDF's ability to fully utilize uniprocessor capacity.



$$\tau_x = (\phi_x, C_x, T_x, D_x): \quad \tau_1 = (0, 2, 4, 4) \quad \tau_2 = (0, 5, 10, 10)$$

Figure 2.1.10: Illustrative EDF and RM schedules for a periodic task set with total utilization $U(\Gamma) = 1.0$ in a fully preemptive uniprocessor system.

capacity. The PDA and its associated computational complexity are further discussed in Section 2.1.6.5.

2.1.6.4.3 Non-Preemptive Uniprocessor Scheduling

In the *non-preemptive* model (Section 2.1.6.2), once a job starts executing it runs to completion without interruption. This eliminates preemption overhead and simplifies mutual exclusion, but it introduces additional *blocking*: a higher-priority job may be delayed by the longest non-preemptive segment of any lower-priority job.

Compared to the fully preemptive case, RM, DM, and EDF lose their optimality among all fixed-priority and dynamic-priority assignments respectively, even for independent implicit-deadline periodic or sporadic tasks on a uniprocessor. Feasibility now depends on both execution times and the placement of non-preemptive sections, and utilization-based bounds are no longer sufficient to characterize schedulability; feasibility tests must explicitly account for blocking [84, 85].

Nevertheless, for independent tasks on a uniprocessor system, many fully preemptive schedulability tests can be extended to the non-preemptive model by adding a bounded *blocking term* that upper-bounds the delay a job can experience due to lower-priority jobs that cannot be preempted immediately [84]. This blocking reduces schedulability compared to the fully preemptive model, with limited-preemptive models achieving schedulability results that lie between the fully preemptive and non-preemptive extremes [60, 84, 86, 87].

2.1.6.4.4 Scheduling Aperiodic Tasks

Real-time systems also execute *aperiodic* tasks, which have no specified arrival pattern. The scheduling approach depends heavily on the task model: *soft* aperiodic tasks are deadline-free. Hence, the main handling objective of these tasks is to improve their response times. In contrast, *firm* or *hard* aperiodic tasks are deadline-constrained and therefore require temporal guarantees even under worst-case execution times.

In the simplest scheme, aperiodic jobs execute in *background* service, running only when no periodic tasks are ready. While this preserves temporal guarantees for the periodic tasks, it cannot provide guarantees for deadline-constrained aperiodic jobs and yields highly variable response times for soft tasks [60].

In order to improve responsiveness and temporal guarantee, the literature introduces the concept of an *aperiodic server*. A server is not a physical task but a virtual *placeholder* characterized by a capacity (budget) C_s and a period T_s. It encapsulates aperiodic demand, reserving a fraction of processor utilization ($U_s = C_s/T_s$) so that schedulability tests can analyze it safely alongside standard periodic tasks [60].

Soft Aperiodic Task Scheduling: For soft aperiodic tasks, existing algorithms trade off implementation complexity against average response time minimization.

- **Fixed-Priority Servers:** Under preemptive RM scheduling, server algorithms include the *Polling Server* (PS), the *Deferrable Server* (DS), the *Priority Exchange* (PE) server, and the *Sporadic Server* (SS).

  These algorithms present direct trade-offs between implementation complexity, responsiveness, and schedulability. PS is the simplest to implement and safely preserves standard RM utilization bounds because it discards unused capacity immediately if no aperiodic requests are pending; however, this causes poor responsiveness for mid-period arrivals [88]. DS improves upon this by preserving its capacity throughout the period, yielding far superior average response times, but at the cost of a reduced schedulability bound for lower-priority tasks due to its deferred interference [89].

  In order to reclaim schedulability, PE preserves capacity by exchanging it with the execution time of lower-priority tasks. This yields a better periodic schedulability bound than DS, but it requires significantly higher implementation complexity to track capacities across different priority levels [88].

  Ultimately, SS achieves the best results compared to the other approaches: it replenishes its budget only after consumption, providing excellent average response times similar to DS and PE, but without affecting the schedulability bounds of lower-priority periodic tasks [90].

  Importantly, Tia, Liu, and Shankar proved that when periodic tasks are scheduled under a fixed-priority assignment, no algorithm can simultaneously minimize the response time of every aperiodic request and still guarantee the schedulability of the periodic task set, putting a theoretical limit on FP responsiveness [60].

- **Dynamic-Priority Servers:** under preemptive EDF scheduling, server algorithms include the *Dynamic Sporadic Server* (DSS), which adapts the efficient replenishment rules of its fixed-priority variant to safely achieve up to 100% processor utilization. Meanwhile, the *Total Bandwidth Server* (TBS) represents a paradigm shift: it is *not* a server in the traditional sense, but rather a dynamic *deadline assignment mechanism*. TBS assigns an absolute deadline to each arriving job, making it compatible with the EDF model [91]. However, TBS relies heavily on the assumption that exact worst-case execution times (WCET) are known and tasks can be trusted to not exceed these assumed bounds. In order to address this, the *Constant Bandwidth Server* (CBS) combines dynamic deadline assignment with strict budget enforcement, postponing deadlines during overruns to provide complete temporal isolation for unpredictable tasks [92].

Firm and Hard Aperiodic Task Scheduling: Servers alone cannot inherently guarantee firm or hard aperiodic tasks because these tasks have strict absolute deadlines. The system must execute an *online acceptance test* (admission control) upon the task's arrival. If the task can be accommodated without causing any existing guarantees to fail, it is accepted; otherwise, it is rejected [60]. In this context, an admission test provides *optimal acceptance* if it admits a firm/hard aperiodic task whenever there exists *any* feasible schedule that meets all deadlines.

- **Slack Stealing Tests:** Chetto and Chetto developed an exact acceptance test based on the *Earliest Deadline as Late as possible* (EDL) schedule, which exposes the exact worst-case idle intervals (slack) available in the schedule. If the total slack prior to the deadline of an arriving aperiodic job $j_a = (r_a, C_a, d_a)$ is at least its execution requirement (i.e., slack $\geq C_a$), then j_a can be safely admitted [93]. This test assumed *implicit deadlines* ($D_p = T_p$) for all periodic tasks. Later, the Exact EDF Slack Stealing (EESS) algorithm by Ripoll et al. proposed an exact test for hard aperiodic tasks under the assumption that periodic tasks have *constrained deadlines* ($D_p \leq T_p$). The test dynamically computes the shortest feasible deadline for an arriving aperiodic task [94]. Both exact methods rely on a fully preemptive execution model, synchronous releases of the periodic tasks, and complete a priori knowledge of the periodic task set.

- **Server-Based Admission Tests:** Instead of computing exact slack, bandwidth-preserving servers perform fast admission control by assigning virtual deadlines to aperiodic jobs and enforcing a reservation on the processor capacity. Given an arriving firm job $j_a = (r_a, C_a, d_a)$, a TBS or CBS server S_s with bandwidth $u_s = Q_s/T_s$ assigns a virtual deadline d_s to j_a and admits it if $d_s \leq d_a$. Under this condition, the reserved bandwidth of S_s is sufficient to complete j_a by its absolute deadline whenever the underlying EDF schedule remains feasible [60, 91].

  A misbehaving job serviced by TBS or CBS that overruns its assumed WCET introduces additional demand that may cause deadline misses. In a pure TBS setting, such an overrun can delay the execution of aperiodic jobs running on the same server as well as other ready jobs in the system, since the server budget is not enforced. When CBS is used instead, the overrun is contained within the server: the misbehaving task can interfere only with jobs assigned to the *same* server. Ensuring intra-server temporal isolation therefore requires additional mechanisms.

  In CBS, once the server budget Q_s is exhausted, the server may immediately receive new budget and postpone its deadline, allowing a job to draw execution from future periods and creating long intervals in which subsequent jobs receive little or no service. The *Hard Constant Bandwidth Server* (H-CBS) refines the CBS

budget-replenishment rules so that each server behaves as a hard reservation with a bounded worst-case service delay for its internal workload [95]. H-CBS prevents unbounded stealing from future budget by deactivating the server as soon as its budget is depleted: the server becomes ineligible for execution until a well-defined replenishment time, at which its budget and scheduling deadline are updated according to the reservation parameters. This behavior can bound the delay suffered by later jobs, but it also makes H-CBS non work-conserving at the task level, since jobs may be ready yet not executable while their server is deactivated, even if the processor is idle [95, 96].

These two methodologies present a stark contrast. Exact slack-stealing-based tests yield optimal admission ratios (accepting 100% of feasible tasks) but they impose pseudo-polynomial runtime complexity and high memory overhead to maintain slack tables over the schedule cycle. Conversely, applying admission tests on top of server-based mechanisms yields sub-optimal admission ratios, but incurs drastically smaller runtime overheads (the test has constant runtime complexity), making them more practical for resource-constrained environments.

2.1.6.4.5 Online Scheduling with Shared Resources

Section 2.1.3.6 introduced mutual exclusion and priority inversion. When tasks share *exclusive* resources protected by locks, a higher-priority job can be blocked by a lower-priority job holding the needed resource. If blocking is not controlled, this delay can be further extended by intervening medium-priority jobs that preempt the lock holder, leading to unbounded *priority inversion* (see Figure 2.1.4). A related hazard is a *deadlock*, which occurs when two or more tasks remain permanently blocked because each waits for a resource held by another task in the set.

Several resource-access protocols were proposed to bound blocking times and preserve analyzability under FP and EDF scheduling:

- The ***Priority Inheritance Protocol*** **(PIP)** is a synchronization protocol for *FP preemptive* scheduling. It implements a priority inheritance mechanism at each shared resource request: when a higher-priority task blocks on a resource held by a lower-priority task, the holder temporarily inherits the higher priority until it releases the resource. This limits unbounded priority inversion by preventing medium-priority interference. However, chained blocking may still occur, because a task can be blocked indirectly through nested or sequential resource dependencies, and deadlocks are not eliminated because PIP does not restrict the order in which resources are acquired [97].

- The ***Priority Ceiling Protocol*** **(PCP)** is a synchronization protocol for *FP preemptive* scheduling. It associates each shared resource with a fixed *priority ceiling*, defined offline as the highest priority of any task that may lock it. A task may enter a critical section only if its *current* priority is strictly higher than the ceilings of all resources currently locked by other tasks. Although task priorities are assigned statically under FP scheduling, PCP temporarily raises the priority of a lock holder through priority inheritance when it blocks a higher-priority task. Compared with PIP, PCP is more restrictive at lock-acquisition time, but in return it prevents deadlock and bounds each job to at most one blocking interval [97].

- The ***Stack Resource Policy*** **(SRP)** is a synchronization protocol for *EDF preemptive* scheduling. It associates each task with a static *preemption level*, where tasks with shorter relative deadlines are assigned higher preemption levels. Each shared resource is assigned a static ceiling equal to the highest preemption level of any task that may access it. A job may start executing (or preempt another job) only if its preemption level is strictly greater than the current *system ceiling*, which is the highest ceiling of all resources currently locked by any task. By ensuring a job cannot start until all the resources it might need are available, SRP eliminates deadlocks and guarantees a job is delayed by at most one lower-level critical section (one blocking interval by a smaller priority task). Furthermore, because jobs never block on a resource once they start running, SRP reduces context-switch overhead [98].

The primary benefit of these synchronization protocols is that they strictly bound the maximum blocking time suffered by each task. This improves temporal predictability by allowing shared-resource-induced blocking to be incorporated as an explicit term in schedulability analyses, such as RTA under FP or PDA under EDF. However, bounding the duration of blocking does not by itself guarantee *replica determinism*: replicated nodes may still diverge in their exact lock-acquisition and execution orders unless they are additionally forced to perform identical scheduling decisions.

2.1.6.4.6 Multiprocessor Scheduling

On multiprocessor platforms with m identical processors, online schedulers face a two-dimensional problem: they must decide not only *when* a job executes, but also *where* (on which processor) it runs. The literature lists three foundational approaches [99,100]:

- **Partitioned scheduling**: Tasks are statically assigned to specific processors offline. At runtime, each processor maintains its own local ready queue and is scheduled independently using a uniprocessor algorithm (e.g., RM or EDF).

- **Global scheduling**: The system maintains a single, shared ready queue. At any scheduling instant, the m highest-priority ready jobs are dispatched across the m available processors, while allowing jobs to migrate between processors.
- **Clustered scheduling**: Processors are grouped into disjoint clusters. Tasks are statically partitioned across the clusters but are scheduled globally and can migrate within their assigned cluster.

In *partitioned scheduling*, runtime execution decomposes into independent uniprocessor problems, allowing the direct application of the uniprocessor feasibility tests described in Section 2.1.6.5. However, from a computational complexity standpoint, the offline task-to-processor allocation is equivalent to the bin-packing problem. Consequently, partitioned scheduling is strongly NP-hard with respect to finding an optimal allocation [76] (see Section 2.1.6.3). Since optimal allocation algorithms scale exponentially, practical implementation rely on polynomial-time, sub-optimal heuristics (such as First-Fit or Best-Fit) [99,100].

In *global scheduling*, schedulability depends on both the dispatching policy and the interference caused by migrating jobs. While Global EDF (G-EDF) and Global RM (G-RM) generalize their uniprocessor counterparts, they lose their optimality on multiprocessors. A widely studied anomaly in multiprocessor scheduling is *Dhall's effect*, which demonstrates that G-EDF and G-RM can fail to schedule task sets with a total utilization slightly greater than 1, even on an arbitrary number of processors m [101].

For instance, as illustrated for $m = 2$ processors in Figure 2.1.11, consider a system running two light tasks (having higher priorities but negligible execution times) alongside one heavy task (having lower priority but a utilization approaching 1). Under global scheduling with G-RM (or with G-EDF), the two light tasks (τ_1 and τ_2) receive higher priority due to their shorter periods (or their earlier deadlines). They execute, at $t = 0$, across both processors, forcing the heavy task (τ_3) to wait. Because τ_3 is delayed until $t = 1$, it only receives 10 units of execution time before its deadline at $t = 11$. Consequently, τ_3 misses its deadline, despite the overall system being underutilized.

From a theoretical perspective, online global multiprocessor scheduling is fundamentally limited. Fisher, Goossens, and Baruah proved that on $m \geq 2$ identical processors, no online deterministic global scheduling algorithm can be optimal for the full class of constrained-deadline sporadic task systems: for every such algorithm, there exists a feasible constrained-deadline sporadic task system on which it misses at least one deadline [102]. The result naturally extends to more general models (e.g., arbitrary-deadline sporadic task systems). Avoiding deadline misses for all feasible systems in these models would require the scheduler to have complete prior knowledge of all future job arrivals (i.e., to be clairvoyant).

By contrast, for the more restrictive case of implicit-deadline periodic and sporadic

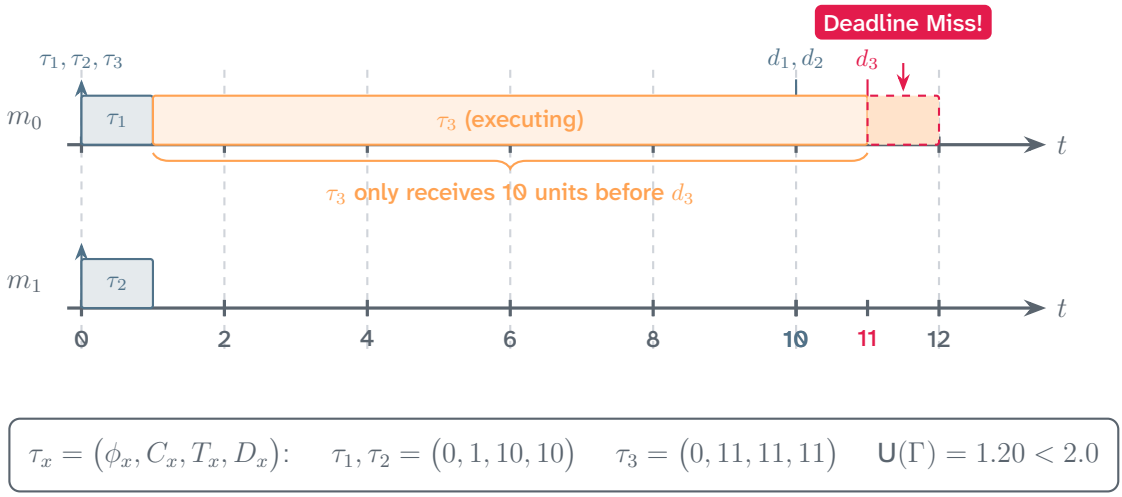


Figure 2.1.11: Illustration of Dhall's effect on $m = 2$ processors. Two high-priority light tasks (τ_1, τ_2) execute at release. This delays the single heavy task (τ_3), causing it to receive only 10 out of its required 11 execution units before its deadline, resulting in a deadline miss at $t = 11$. This failure occurs even though the total system utilization ($\mathsf{U} = 1.20$) is well below the available capacity of 2.0.

task systems, optimal online global schedulers are known to exist. In particular, the Pfair (*proportionate fairness*) algorithm, introduced by Baruah et al., divides time into equal-length quanta and enforces a proportionate-progress constraint that tightly bounds the lag between the service a task receives and its ideal fractional allocation [103]. Under this constraint, Pfair-based algorithms[2] are optimal for implicit-deadline periodic and sporadic task sets on identical multiprocessors: whenever such a task set is feasible on m unit-speed processors, a Pfair schedule exists and can be constructed by these algorithms [103–105]. However, the fine-grained time slicing and global dispatching that underpin Pfair scheduling cause very frequent preemptions and inter-processor migrations. These effects incur substantial runtime overheads and degrade processor affinity (and thus cache locality), compared to coarser-grained global EDF-like schedulers [106].

Semi-partitioned algorithms emerged as a practical compromise that combines the high utilization of global schemes with the low overhead of partitioned ones. In approaches such as the EKG algorithm [107], the vast majority of tasks are strictly partitioned to specific processors, while a carefully selected subset of tasks is split across neighbouring processors and scheduled according to local EDF-like rules. This hybrid strategy yields significantly higher utilization bounds than classical global EDF or FP (thus avoiding the low worst-case utilizations associated with Dhall's effect), yet retains a low number of preemptions and migrations compared to fully global fair-scheduling algorithms [99, 107].

[2] Prominent variants of the Pfair approach include the PF, PD, and PD^2 algorithms.

2.1.6.5 Feasibility and Schedulability Analysis

A task set is said to be *feasible* if every admissible job sequence that it may generate admits at least one schedule in which all deadlines are met. Feasibility analysis concerns the question of whether such a schedule exists under the assumed task model, processor architecture, preemption model, and resource-sharing protocol [60].

Feasibility tests are the analytical tools used to answer this question. In contrast, *schedulability* tests evaluate whether a *specific* scheduling algorithm (e.g., EDF, RM) will meet all deadlines for the task set. Hence, a task set may be feasible yet not schedulable under a particular algorithm, which makes it important to distinguish feasibility (an algorithm-independent property) from schedulability (an algorithm-specific property).

The analytical complexity of schedulability tests depends on the underlying task model (for example, whether tasks are synchronous or asynchronous, and whether they have implicit, constrained, or arbitrary deadlines) and on the processor architecture (uniprocessor or multiprocessor) [63, 72]. Moreover, schedulability tests may be either *sufficient*, guaranteeing correctness when their conditions hold but potentially rejecting some task sets that are in fact schedulable, or *necessary and sufficient*, providing an exact characterization of schedulability under the corresponding algorithm [60].

The reminder of this section provides a brief overview of EDF-schedulability on uniprocessor systems. For a detailed overview of RM-schedulability, the reader is referred to [60, 61, 83, 108].

2.1.6.5.1 EDF Schedulability on Uniprocessor Systems

Unless stated otherwise, the following analysis assumes a fully preemptive uniprocessor system executing a set of independent tasks. These tasks do not share resources (other than the CPU), do not suspend themselves, and incur negligible context-switching and scheduling overheads. Since EDF is optimal under these assumptions, evaluating exact EDF-schedulability is equivalent to evaluating task set feasibility [109]. Finally, the task set is initially assumed to be *synchronous* (meaning all tasks share a common initial release offset, e.g., $\phi_i = 0$ for all $\tau_x \in \Gamma$). The specific extension required for asynchronous periodic task sets is addressed in Section 2.1.6.5.2.

Implicit-deadline sporadic and periodic tasks. Under an implicit-deadline task model ($D_x = T_x$), EDF schedulability is characterized by the total utilization condition [60, 63]:

$$\mathsf{U}(\Gamma) \leq 1 \quad \Longleftrightarrow \quad \Gamma \text{ is schedulable under EDF.} \tag{2.1.1}$$

The test in Equation (2.1.1) is equally applicable to sporadic tasks, which in the worst-case scenario (the critical instant), all tasks arrive simultaneously and all subsequent jobs are released exactly at their minimum inter-arrival times, effectively mirroring the

synchronous periodic assumption. The computational complexity of this exact test is polynomial, executing in $\mathcal{O}(n)$ time where n is the number of tasks in Γ.

Constrained-deadline sporadic and periodic tasks. Under a *constrained-deadline* task model ($D_x \leq T_x$), the utilization condition in Equation (2.1.1) remains a necessary condition but is no longer sufficient: i.e., if the utilization condition does not hold, then the task set is for sure not EDF-schedulable. Exact EDF-schedulability is instead characterized by the *Processor Demand Analysis* (PDA). The core intuition behind PDA is to ensure that, for every possible time interval of length t, the total execution time demanded by jobs that must both arrive and complete within that interval never exceeds the available processing time t. This requirement is formalized using the *demand bound function*, $\mathsf{dbf}(t)$ [63]. For a given task τ_x, the function $\mathsf{dbf}(\tau_x, t)$ quantifies the maximum cumulative execution demand of all jobs of τ_x whose release times and absolute deadlines both lie within an interval of length t. Introducing the variable $\eta_x(t)$ to represent the maximum number of such job instances yields

$$\mathsf{dbf}(\tau_x, t) = \eta(\tau_x, t) \cdot C_x, \tag{2.1.2}$$

with

$$\eta(\tau_x, t) = \left\lfloor \frac{t - D_x}{T_x} \right\rfloor + 1 \tag{2.1.3}$$

The demand bound function for the entire task set aggregates the demand of all tasks as

$$\mathsf{dbf}(t) = \sum_{\tau_x \in \Gamma} \mathsf{dbf}(\tau_x, t) \tag{2.1.4}$$

A necessary and sufficient EDF schedulability test is obtained by checking $\mathsf{dbf}(t) \leq t$ for all relevant values of t. It was shown in [63] that t values can be restricted to absolute deadlines occurring within the critical analysis interval, denoted here as L_{max}.

For synchronous constrained-deadline periodic or sporadic task systems, the length of this analysis interval is bounded as follows [63, 108, 110]:

$$L_{\mathsf{max}} = \begin{cases} B(\Gamma) & \text{if } \mathsf{U}(\Gamma) < 1 \\ \mathsf{HP}(\Gamma) & \text{if } \mathsf{U}(\Gamma) = 1 \end{cases} \tag{2.1.5}$$

where the hyperperiod $\mathsf{HP}(\Gamma)$ and the utilization-based bound $B(\Gamma)$ are defined as

$$\mathsf{HP}(\Gamma) \;=\; \mathsf{LCM}\big(T_x \,\big|\, \forall \tau_x \in \Gamma\big) \tag{2.1.6}$$

$$B(\Gamma) \;=\; \frac{\sum_{\tau_i \in \Gamma} (T_i - D_i)\, U_i}{1 - \mathsf{U}(\Gamma)} \tag{2.1.7}$$

Accordingly, the exact test only needs to inspect absolute deadlines up to $L_{\max}$. When the total utilization is bounded by a constant strictly less than one ($\mathsf{U}(\Gamma) \leq c < 1$), the analysis interval is tightly bounded by $B(\Gamma)$, which yields a pseudo-polynomial exact test. Figure 2.1.12 illustrates how $B(\Gamma)$ arises as the intersection point of the continuous processor demand bound curve $\mathrm{dbf}^*(t)$ and the capacity line t. $B(\Gamma)$ guarantees that if the task set is unschedulable, i.e., if $\mathrm{dbf}(t) > t$ for some t, then we must have $t < B(\Gamma)$.

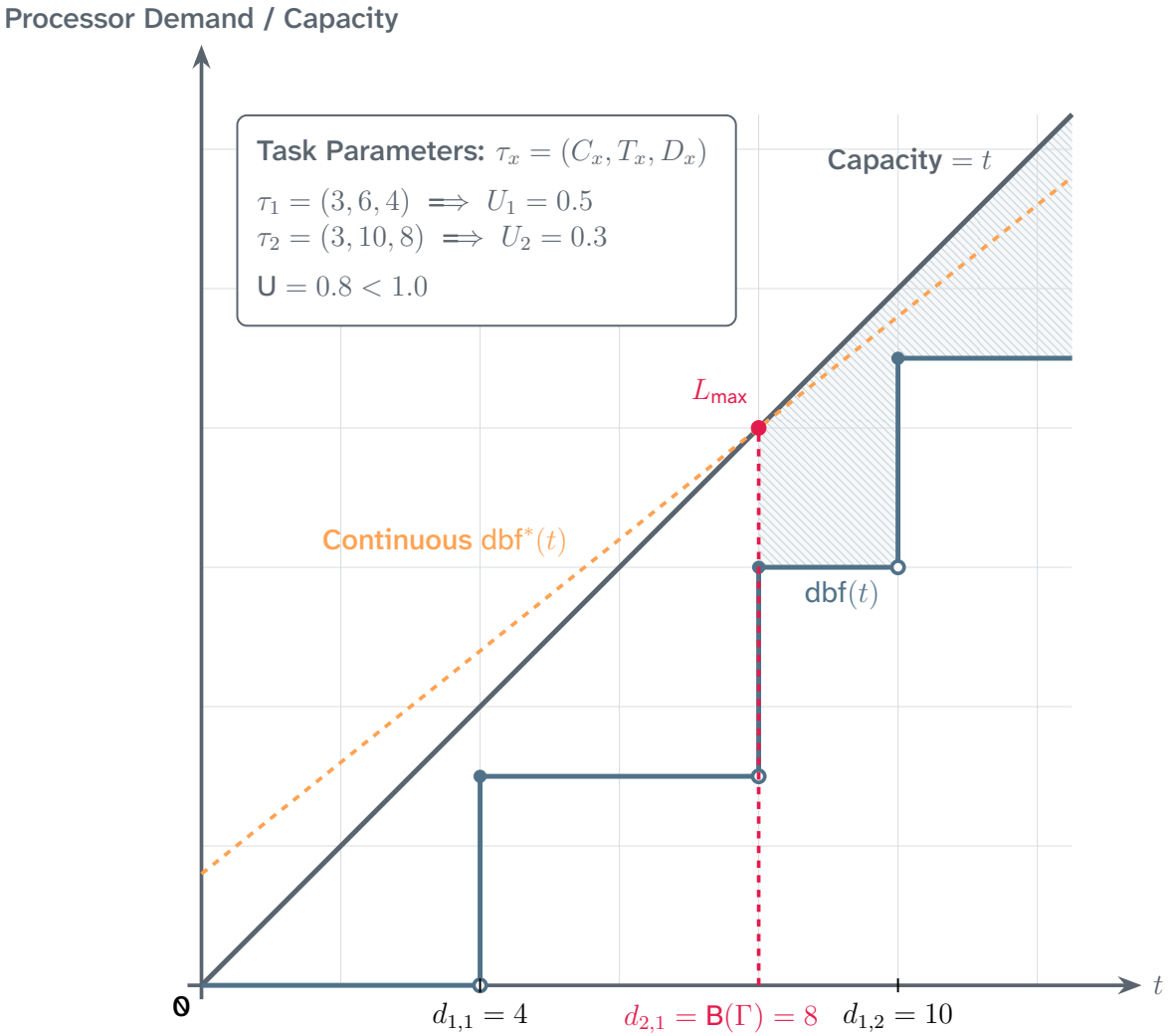


Figure 2.1.12: Illustration of the bound $B(\Gamma)$. The continuous upper bound curve $\mathrm{dbf}^*(t)$ intersects the capacity line at $B(\Gamma)$. Since $\mathrm{dbf}(t)$ is bounded by $\mathrm{dbf}^*(t)$, and both share a slope of $\mathsf{U}(\Gamma) < 1$, $\mathrm{dbf}(t)$ lies permanently below the capacity for all interval lengths $t > B(\Gamma)$ (indicated by the shaded region).

However, in the general setting where utilization reaches one, the critical analysis interval is bounded only by the hyperperiod $\mathsf{HP}(\Gamma)$. In this case, checking EDF-schedulability is known to be strongly coNP-complete [111], meaning that verifying a counterexample to schedulability is NP-complete and thus the problem is unlikely to admit polynomial time exact tests. The length of the critical analysis interval can con-

sequently grow exponentially with respect to the numerical size of the task parameters.

Arbitrary-deadline sporadic and periodic tasks. For an *arbitrary-deadline* sporadic or periodic task model, the individual task demand $\mathsf{dbf}(\tau_x, t)$ and the total system demand $\mathsf{dbf}(t)$ remain exactly as in equations (2.1.2) and (2.1.4), respectively. However, the number of job instances $\eta_x(t)$ contributing to the demand must be adjusted because deadlines may exceed periods. The number of instances, in this case, is defined as

$$\eta(\tau_x, t) = \mathsf{max}\left(0,\ \left\lfloor \frac{t - D_x}{T_x} \right\rfloor + 1\right). \tag{2.1.8}$$

This formulation ensures that only the demand from jobs whose release times and absolute deadlines fall entirely within the interval $[0, t]$ is considered. The exact schedulability test again requires checking $\mathsf{dbf}(t) \leq t$ for all relevant absolute deadlines up to L_{max}. However, in the arbitrary-deadline case, L_{max} is no longer bounded solely by the hyperperiod or by $B(\Gamma)$. The largest relative deadline must also be taken into account, as established in [112, 113]:

$$L_{\mathsf{max}} \ = \ \begin{cases} \mathsf{max}\left(\underset{\tau_x \in \Gamma}{\mathsf{max}}\, D_x,\ B(\Gamma)\right) & \text{if } \mathsf{U}(\Gamma) < 1, \\ \mathsf{HP}(\Gamma) & \text{if } \mathsf{U}(\Gamma) = 1. \end{cases} \tag{2.1.9}$$

2.1.6.5.2 PDA Extensions and Optimizations

Asynchronous periodic tasks. When periodic tasks are not assumed to share a common initial release time, they are modeled as *asynchronous* by introducing an offset parameter ϕ (yielding a four-parameter task $\tau_x = (\phi_x, C_x, D_x, T_x)$). Under this asynchronous model, the critical instant no longer corresponds to a simultaneous release at $t = 0$, because the fixed offsets constrain when jobs can actually be released relative to each other. The PDA accounts for this by checking the maximum execution demand over any possible time interval $[t_1, t_2)$, where t_1 corresponds to a job release time, and t_2 corresponds to a job absolute deadline.

The demand bound function $\mathsf{dbf}(\tau_x, t_1, t_2)$ is extended to quantify the execution demand of all jobs of τ_x whose release times are at or after t_1 and whose absolute deadlines are at or before t_2. This requires calculating the number of such job instances within that specific window, which is given by [63]:

$$\eta(\tau_x, t_1, t_2) = \mathsf{max}\left(0,\ \left\lfloor \frac{t_2 - \phi_x - D_x}{T_x} \right\rfloor - \mathsf{max}\left(0,\ \left\lceil \frac{t_1 - \phi_x}{T_x} \right\rceil\right) + 1\right) \tag{2.1.10}$$

The extended demand bound function is then $\mathsf{dbf}(\tau_x, t_1, t_2) = \eta_x(t_1, t_2) \cdot C_x$, and the task

set demand is:

$$\mathsf{dbf}(t_1, t_2) = \sum_{\tau_x \in \Gamma} \mathsf{dbf}(\tau_x, t_1, t_2) \tag{2.1.11}$$

For asynchronous periodic tasks, the task set is deemed EDF-schedulable if and only if $\mathsf{dbf}(t_1, t_2) \leq t_2 - t_1$ for all relevant release times t_1 and absolute deadlines t_2. It was also shown in [63] that the ranges on the values of t_1 and t_2 to consider for the PDA can be constrained as

$$0 \leq t_1 < t_2 \leq \max_{\tau_x \in \Gamma}(\phi_x) + 2\,\mathsf{HP}(\Gamma), \tag{2.1.12}$$

The extra dimension added by task offsets exacerbates the problem complexity: checking exact EDF-schedulability for asynchronous task sets is shown to be strongly coNP-complete [63], and the length of the critical analysis window can grow exponentially with respect to the numerical size of the task parameters, even under strictly bounded utilization. A more computationally efficient, albeit only sufficient, test was proposed in [109] to trade exactness for reduced complexity.

Quick Processor-Demand Analysis (QPA). Since the classical PDA requires exhaustively checking all candidate deadline points up to L_{max}, it incurs significant computational cost, particularly in high-utilization systems where L_{max} is very large. In order to mitigate this, Zhang and Burns introduced the *Quick Processor-demand Analysis* (QPA), an optimized algorithm that substantially reduces the number of absolute deadlines to check by exploiting the monotonic step structure of the demand bound function [113].

Rather than testing absolute deadlines in forward chronological order, QPA begins at the largest absolute deadline that is smaller than or equals to L_{max}, and then iterates backward toward zero. The core intuition is that because $\mathsf{dbf}(t)$ is a non-decreasing step function, if $\mathsf{dbf}(t) \leq t$ holds at some tested interval length t, the demand cannot possibly exceed the available processor capacity for any interval length between $\mathsf{dbf}(t)$ and t. Therefore, the next interval to test can safely jump directly down to the computed demand (by setting $t = \mathsf{dbf}(t)$). This significantly reduces the number of required $\mathsf{dbf}(t)$ checks while preserving exactness [113].

Scheduling with shared resources and non-preemptive execution. Both shared-resource synchronization and non-preemptive execution introduce bounded priority inversion under EDF, and in both cases the effect can be captured by augmenting the PDA with a blocking term $b(t)$, which represents the maximum delay that any job with deadline t may experience due to interference from jobs with deadlines greater than t. Detailed analyses of these scenarios are provided in [60, 98, 114].

Multiprocessor case. On $m \geq 2$ identical processors, EDF is typically implemented according to either a *partitioned* or a *global* scheduling approaches (see Section 2.1.6.4.6).

In the partitioned approach, each task is statically assigned to one processor before run time and thereafter executes exclusively on that processor; no migration is permitted. In the global approach, all ready jobs are kept in a single global queue ordered by absolute deadlines, and at each instant the m earliest-deadline jobs are dispatched on the m available processors, allowing job migration between processors [99].

Under partitioned EDF, once a feasible allocation of tasks to processors has been determined, each processor can be treated as an independent uniprocessor and analyzed in isolation using the uniprocessor EDF schedulability tests described above. The multiprocessor schedulability problem thus reduces to m independent uniprocessor analyses after fixing task allocation. The central difficulty therefore lies not in the per-processor test but in the allocation problem itself (See Section 2.1.6.3).

Under global EDF, jobs may migrate between processors, so each job can suffer interference from jobs executing on any of the other $m - 1$ processors, and the analysis does not reduce to independent uniprocessor tests [99]. Furthermore, global EDF suffers from the *Dhall effect*: task sets with total utilization arbitrarily close to 1 can be unschedulable by global EDF on any number of processors (see Section 2.1.6.4.6). More information regarding EDF-schedulability under global scheduling can be found in [99, 115, 116].

2.1.6.6 Offline Scheduling

Offline scheduling denotes the design-time construction of schedules for the computation and communication resources of a real-time system. Making all scheduling decisions offline requires complete *a priori* knowledge of the system and its operational environment. All system activities must be known in advance and must admit a temporal specification with respect to global time. Consequently, activities must follow a periodic activation model to remain compatible with the offline scheduling approach.

A major advantage of offline scheduling is that it can accommodate fairly complex constraints during schedule construction. These include precedence relations, distributed execution, communication over networks, task allocation, protocol parameters, and mode-dependent constraints [10, 117–121]. In this sense, resource conflicts are resolved by construction at design time: the offline scheduler assigns non-overlapping time slots to tasks sharing a resource, ensuring that mutual exclusion is guaranteed without requiring any runtime locking protocols. Precedence relations are handled in a similar manner by ordering the slot assignment of dependent activities on their respective resources [1]. However, since many such scheduling problems are computationally hard (see Section 2.1.6.3), offline scheduling methods often rely on heuristic algorithms to efficiently find a feasible schedule. If a scheduling attempt fails, the offline algorithm can be rerun with modified parameters, allocations, or search choices.

Generating Offline Scheduling Tables

In the first step of schedule generation, the offline scheduler takes the temporal constraints, e.g., DAGs representing tasks and their dependencies as input, and produces scheduling tables for each computation node and communication channel in the system. It assigns tasks to computation slots on nodes and their messages to communication slots on the respective network channels. Precedence constraints are resolved by ordering the slot allocations across these resources accordingly.

Since the activities considered in offline scheduling are periodic, schedule generation is performed over a finite time interval known as the *schedule cycle* T_Γ. When all tasks share the same initial offset, the system behavior becomes periodic immediately after tasks' first releases, and all relative phase alignments are covered within a single hyperperiod. In this case, T_Γ is simply set to the hyperperiod.

However, when tasks have distinct offsets, the system does not enter its periodic release pattern until after the largest offset has occurred. In order to check schedulability according to real-time theory, the schedule cycle is conservatively chosen to extend two full hyperperiods beyond the maximum offset [10, 109]. Given a set of periodic tasks Γ with distinct offsets, the schedulability interval is computed, in this case, following Equation (2.1.12).

Importantly, only one hyperperiod following the largest offset needs to be stored as the final static schedule, since this interval captures the periodic behavior executed at runtime. For a more comprehensive coverage of offline scheduling for periodic tasks, we refer the reader to Minaeva et al. [10].

As an illustrative example of scheduling tables, Table 2.1.1 presents the timing parameters of the jobs within the DAG instances depicted in Figure 2.1.3, where r_x denotes the release time, C_x the WCET, and d_x the absolute deadline. Table 2.1.2 provides a feasible global scheduling table for these DAG instances, specifying the assigned start and termination times for each job and message transmission.

As shown in Table 2.1.2, the schedule consists of a sequence of time instants, each associated with a time trigger that serves as a control signal responsible for initiating an activity on the corresponding resource. In practical implementations, each resource stores its own local schedule, which contains the activities planned for execution on that resource. The scheduling table for a computational node, commonly referred to as the Task Descriptor List (TADL), specifies the set of activities assigned to that node. In contrast, the scheduling table for a communication channel, known as the Message Descriptor List (MEDL), contains the messages that the communication controller must send and receive throughout the schedule cycle.

Table 2.1.1: Timing parameters for tasks in Figure 2.1.3

j_x	r_x	C_x	d_x
j_a	0	1	-
j_b	-	1	9
j_c	0	2	-
j_d	-	1	-
j_e	-	1	11
j_f	-	1	11

Table 2.1.2: Global scheduling table for the tasks in Table 2.1.1

Resource	Time	Actions
$Node\ 0$	0	Start j_c
$Node\ 0$	2	End j_c, Start j_d
$Node\ 0$	3	End j_d
$Channel\ 0$	4	Send j_d's message
$Channel\ 0$	5	Deliver j_d's message
$Node\ 0$	5	Start j_a
$Node\ 0$	6	End j_a, Start j_b
$Node\ 1$	6	Start j_e
$Node\ 0$	7	End j_b
$Node\ 1$	7	End j_e
$Node\ 1$	8	End Start j_f
$Node\ 1$	9	End j_f

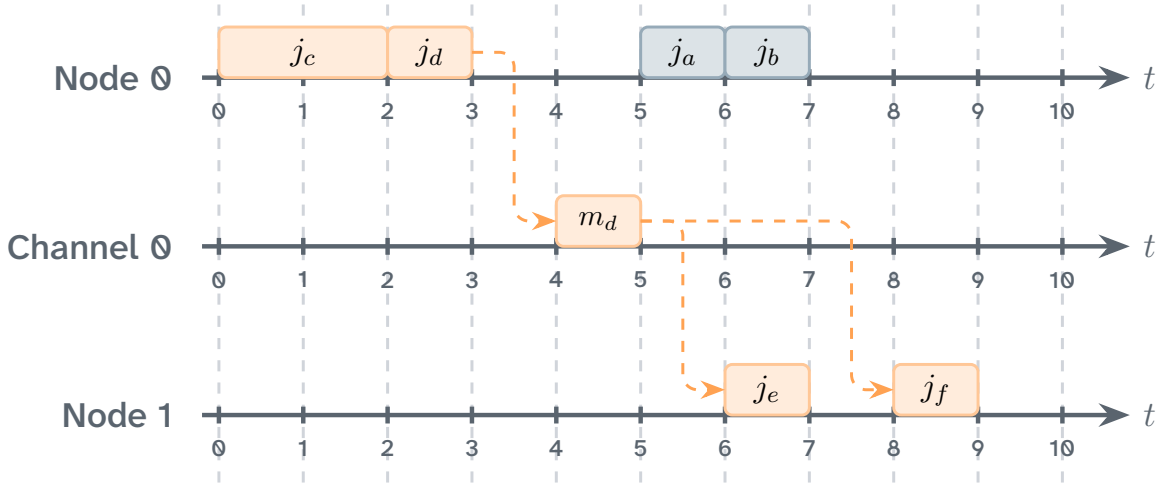


Figure 2.1.13: Illustration of the schedule in Table 2.1.2.

Table-Driven Runtime Execution

At runtime, scheduling tables are executed periodically and in synchrony with the progression of global time. Each offline-defined *time trigger* activates the local dispatcher, which initiates the execution of the activity specified in the scheduling table. The interval between two consecutive time triggers defines a *time slot* that is exclusively allocated to the corresponding job. These time slots may have a uniform duration, for example, in systems where the dispatcher is invoked by a strictly periodic clock interrupt [19, 122, 123], or they may vary in length if the scheduler is activated at non-equidistant instants explicitly listed in the scheduling table [119, 124].

The internal structure of a time slot is illustrated in Figure 2.1.14. It consists of two distinct phases: a *dispatching phase* and a *computation phase*. During the brief dispatching phase, the operating system's dispatcher is invoked by the time trigger to dispatch the job allocated to the current time slot according to the offline-defined schedule. The remainder of the slot forms the computation phase, where the assigned task executes exclusively and without interruption. To ensure that context-switching overhead remains negligible, the typical slot size is chosen to be significantly larger than the dispatching duration, often in the range of hundreds of microseconds to a few milliseconds, depending on the platform's processing speed and the control application's latency requirements [123].

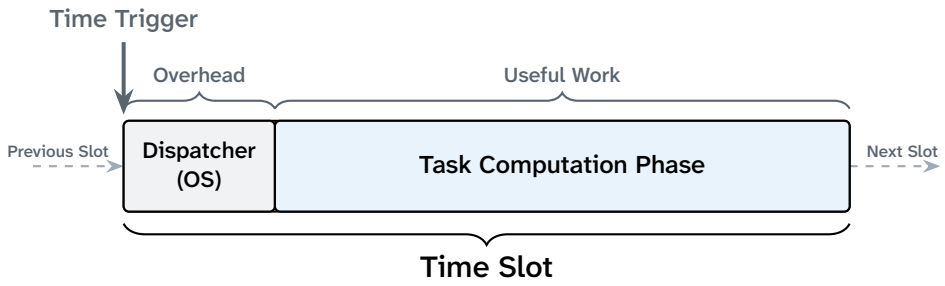


Figure 2.1.14: Structure of a time slot for computation.

Because all decisions are made offline, the runtime scheduler complexity and overhead is low. Runtime scheduling reduces to a simple table-lookup dispatching process without the need of accounting for the original complex temporal constraints. In this sense, offline scheduling shifts computational complexity from run time to design time, trading flexibility for analyzability and temporal determinism.

The offline-defined scheduling table provides a proof by construction that all timing constraints will be satisfied, provided that the temporal parameters supplied to the offline scheduling algorithm are valid. This proof is sufficient because the operation of the time-triggered system is rigidly tied to this offline-defined table. At the same time, obtaining the complete system knowledge required for offline scheduling can be very

costly. For this reason, offline scheduling is most commonly employed in safety-critical applications, where the benefits of strict determinism, predictability, and testability easily justify the additional design efforts and costs.

2.1.6.7 Combined Offline and Online Scheduling

A central limitation of conventional table-driven time-triggered scheduling is its inherent inflexibility; tasks whose parameters are not fully known at design time cannot be scheduled offline, and consequently they are not accommodated at runtime. In addition, processor capacity reserved for worst-case scenarios cannot be reclaimed at runtime. In contrast, pure event-triggered online scheduling lacks the ability to satisfy the strong predictability requirements of safety-critical activities, and it cannot resolve complex constraints such as precedence relations or task allocation without incurring substantial runtime overhead.

The combined offline and online scheduling approach presented by Fohler [117, 122] addresses this all-or-nothing tradeoff by shifting the bulk of the computational complexity to the design phase, where an offline scheduler resolves precedence relations, task allocation, inter-node message scheduling, generating a feasible scheduling table for each computational node and communication controller.

Although scheduling tables are sufficient to guarantee the timing requirements of dependent periodic tasks, they restrict flexibility, and are less attractive for implementing applications exhibiting a certain degree of runtime uncertainty (e.g., based on aperiodic and sporadic tasks).

In order to overcome this limitation, rigid scheduling tables can be supplanted by a set of independent periodic tasks, which can be directly handled by an online scheduler that manages both these tasks and additional runtime ones without compromising timing guarantees [117, 125]. This process relies on a detailed analysis of the generated offline scheduling tables and the parameters of the task DAGs to derive a *Target Execution Window* (TEW) for each task instance, that is, an earliest start time and a deadline. The precise execution times of tasks on their respective nodes are not required for this derivation.

The resulting TEWs preserve all original timing guarantees while exposing residual slack (spare processor capacity) to the online scheduler, which can exploit this slack to efficiently admit tasks at run time [122, 125]. The remainder of this section describes the derivation of target windows and the transformation of dependent periodic tasks into independent ones.

2.1.6.7.1 Merging dependent tasks

Merging dependent tasks is an optional preprocessing step intended to reduce the number of tasks that the online scheduler must manage. Dependent tasks assigned to the same processor, and not separated by precedence constraints involving other dependent tasks on different processors, can be combined into a single dependent task.

For instance, considering the allocation in Table 2.1.2, jobs j_c and j_d can be merged into a single job j_{cd}, with a worst-case execution time of $C_{cd} = C_c + C_d$. Figure 2.1.15 shows the resulting DAGs after applying the merging step on the dependent tasks in Figure 2.1.3. It is important to note that this merging step assumes partitioned scheduling, meaning that all jobs of a given dependent task are executed on the same processor.

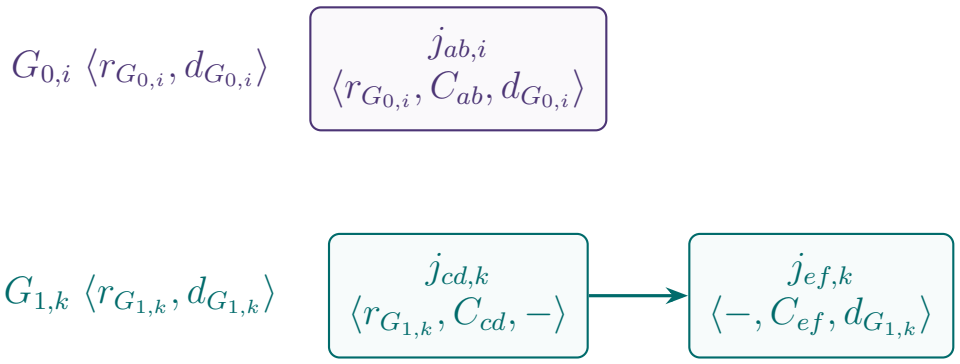


Figure 2.1.15: DAG instances from Figure 2.1.3 after merging dependent tasks.

2.1.6.7.2 Calculating target execution windows for dependent tasks

The timing constraints defined by task DAGs and their corresponding offline scheduling tables implicitly determine a time interval during which each dependent job must execute in order to satisfy both precedence and deadline requirements. For example, the release time of a dependent job may be constrained by the release time of its DAG instance, as in the case of an entry job, or by the delivery time of a message it consumes. Similarly, the deadline of a dependent job may correspond to the deadline of its DAG instance, as in the case of an exit job, or to the transmission time of a message it produces. The interval between a job's release time and its deadline defines its TEW.

Ensuring that all jobs of dependent tasks execute within their respective TEWs guarantees compliance with their complex timing constraints, thereby reducing the problem to the local scheduling of independent jobs with linear complexity, for instance using fixed- or dynamic-priority scheduling [126]. Algorithm 1 outlines the procedure for computing TEWs for jobs of dependent tasks. When applied to the example in Tables 2.1.1 and 2.1.2, this procedure yields the TEWs shown in Table 2.1.3.

It should be noted that the TEW-based transformation may come at the price of global optimality. The offline scheduling table on which the flexibility analysis is based repre-

Algorithm 1 Calculating TEWs for the jobs of dependent tasks

```
1: procedure calculate_tews(DAGs, Schedule)
2:     for i ← 0 to num_DAGs − 1 do
3:         num_instances_DAG_i ← T_Γ / T_{G_i}
4:         for k ← 0 to num_instances_DAG_i − 1 do
5:             for all job j_x ∈ G_{i,k} do
6:                 if j_x is an exit task then
7:                     d_x ← d_{G_{i,k}}
8:                 end if
9:                 if j_x sends a message at t_s then
10:                    d_x ← t_s
11:                end if
12:                if j_x is an entry task then
13:                    r_x ← r_{G_{i,k}}
14:                end if
15:                if j_x reads a message at t_r then
16:                    r_x ← t_r
17:                end if
18:            end for
19:        end for
20:    end for
21:    return
22: end procedure
```

Table 2.1.3: Resulting job set derived from the jobs in Table 2.1.1 and their offline schedule in Table 2.1.2 after applying the merging step and recalculating their WCETs, release times, and deadlines.

j_x	r_x	C_x	d_x
j_{ab}	0	2	9
j_{cd}	0	3	4
j_{ef}	5	2	11

sents only one feasible solution among potentially many, and the resulting target windows do not recover all alternative feasible schedules.

2.1.6.7.3 Deriving Independent Periodic Tasks

Each dependent task within a DAG G_A can be transformed into a 4-tuple τ_x represented as $\langle \phi_x, C_x, D_x, T_x \rangle$ according to the following rules:

1. The offset ϕ_x is set to the release time of the first job, that is, $\phi_x = r_{x,0}$.
2. The relative deadline D_x is defined as the length of the job's TEW, that is, $D_x = d_{x,0} - r_{x,0}$.
3. The period T_x and WCET C_x remain unchanged from the original dependent task.

Following this transformation, the local online scheduler on each processor is responsible only for the set of independent tasks, denoted by Γ, allocated to that processor. Any online scheduler capable of satisfying the timing constraints of all tasks in Γ may be employed (see Section 2.1.6.4).

2.2 Distributed Systems Theory

While the preceding section established the foundations of real-time scheduling, meeting temporal constraints addresses only one aspect of the design challenge. Safety-critical systems must preserve their strict timing guarantees even in the presence of accidental faults and intelligent adversarial attacks.

This section provides an overview of key concepts and principles in distributed systems theory. It begins with foundational concepts of dependability and security, followed by a discussion of synchrony assumptions that characterize distributed systems and their operational environments. The section then outlines fundamental theoretical limits in distributed systems and examines how dependability and intrusion tolerance can be achieved within these limits through fault tolerance, replication, architectural hybridization, and system rejuvenation.

2.2.1 Foundational Concepts of Dependability and Security

Before delving into the specific fault-tolerance and intrusion-resilience mechanisms required for distributed real-time systems, this section covers the foundational concepts and definitions that underpin these mechanisms. We overview the core attributes of dependability and security, outline the causal chain of system impairments (faults, errors, and failures), and introduce the foundational architectural principles of fault containment and secure startup.

2.2.1.1 Dependability and its Attributes

Dependability encompasses extra-functional attributes of a computer system that relates to the quality of service a system delivers. More precisely, it is a measure of the degree to which users, operators, and interacting systems can place justifiable trust in the services provided by the system [38, 58].

The attributes of dependability include:

- **Reliability**: A strict measure of the continuous delivery of correct service. Formally, reliability $R(t)$ is the probability that a system operates correctly and without interruption throughout a defined operational interval $[t_0, t]$, given that it was

fully operational at time t_0. Reliability is the paramount metric in mission-critical environments where in-mission repairs are physically impossible or prohibitively expensive (e.g., autonomous flight controllers). It is typically characterized by the *Mean Time To Failure* (MTTF). For components exhibiting a constant failure rate λ (failures per hour), reliability is modeled exponentially as $R(t) = e^{-\lambda(t-t_0)}$, with MTTF $= \frac{1}{\lambda}$.

- **Maintainability**: A measure of the time required to repair a system after the occurrence of a benign failure. It represents the probability that a failed system is restored to a fully operational state within a specified time interval d. Maintainability is characterized by the *Mean Time To Repair* (MTTR), which is the inverse of the constant repair rate μ (repairs per hour) of a reparable component. High maintainability requires diagnosability, modular architectures, and standardized interfaces.

- **Availability**: A measure of the delivery of correct service with respect to the alternation of correct and incorrect service delivery. It represents the probability that the system is operational and ready to deliver service at any given instant t. For components with defined MTTF and MTTR, availability is calculated as the ratio of uptime to total time as follows: $A = \frac{\text{MTTF}}{\text{MTTF}+\text{MTTR}}$. Hence, a highly available system can be achieved either by increasing the MTTF or by reducing the MTTR.

- **Safety**: The capability of the system to operate without causing catastrophic consequences for human users, the physical environment, or the system platform itself. Safety can be viewed as reliability with respect to critical failure modes, where the cost of failure is orders of magnitude higher than the utility of system services that are provided during normal operation. Consequently, safety focuses strictly on preventing high-cost and malign failures. In safety-critical systems, the design must first adhere to rigorous, domain-specific safety standards, and the resulting system must be demonstrated, through analysis, verification, and certification, to be safe, as discussed in Chapter 1.

A fundamental challenge in building distributed systems is that these dependability attributes are often orthogonal and therefore require explicit design trade-offs. For example, a system engineered to immediately stop operation and drive all physical actuators to a predefined safe state upon detecting a failure maximizes safety (a fail-safe strategy) [38]. However, such an instantaneous shutdown reduces availability and reliability. Conversely, incorporating component replication and fault-tolerant protocols enhances availability and reliability (a fail-operational strategy), but at the cost of increased architectural complexity and a larger attack surface [58].

2.2.1.2 Security

Historically, engineering disciplines treated dependability (and safety) as conceptually orthogonal to security. Dependability frameworks assumed a benign environment, addressing accidental faults arising from component degradation, electromagnetic interference, or unintended software defects. Security frameworks, in contrast, assumed a hostile environment and focused on intentional, deliberate faults introduced by intelligent adversaries seeking to subvert system goals.

Modern distributed architectures render this separation obsolete. The foundational taxonomy by Avizienis et al. explicitly integrates security under the broader umbrella of dependability, defining an attack as a malicious, intentional, and external fault [1, 38]. Under this unified perspective, security is treated as a composite dependability attribute comprising three properties:

- **Confidentiality**: The absence of unauthorized disclosure of information.
- **Integrity**: The absence of improper or unauthorized modification of system state.
- **Availability**: The absence of unauthorized withholding or denial of information or services.

However, when designing safety-critical real-time systems, the classical security triad is fundamentally insufficient. Data that is cryptographically intact and semantically correct can still be useless or dangerous if it is delivered and acted upon outside its specified temporal interval (see Section 2.1.4). Such timing deviations constitute a direct *safety* hazard because they can violate the physical invariants of the controlled plant. At the same time, modern adversaries can intentionally manipulate temporal behavior, effectively turning the temporal dimension itself into an additional attack surface. Under this perspective, *timing integrity*, the property that a system's temporal behavior, including communication latencies, execution times, and synchronization precision, remains bounded and predictable, represents a genuine *security* concern [51, 58].

Representative examples of timing-based attacks on real-time systems include [51, 127–130]:

- **Control destabilization via message manipulation**: A man-in-the-middle adversary may delay, drop, or replay network messages to artificially increase dead time and jitter beyond the bounds assumed in the control-loop design [51]. When the induced temporal distortion is sufficiently large, the loop stability margin is violated and the physical process may be driven into an unsafe state.
- **Synchronization subversion**: An attacker can inject or manipulate clock-synchronization messages to induce clock drift among distributed nodes, undermining the global time base required for coordinated actuation [51, 130]. A

concrete instantiation is the PCspooF attack [130], in which a best-effort device on a Time-Triggered Ethernet (TTE) network exploits electromagnetic coupling to cause a switch to forward crafted Protocol Control Frames, causing TTE devices to lose synchronization and drop time-critical messages.

- **Side-channel timing attacks**: An attacker may observe execution frequencies, task-activation patterns, or scheduling regularities to infer sensitive operational parameters and then selectively interfere with critical tasks or their messages. In the real-time setting, a particularly damaging variant is *cache-based interference*: by deliberately evicting cache lines used by a victim task, the attacker inflates that task's execution time and can cause it to exceed its assumed WCET, thereby inducing a deadline miss [128, 129].

- **Temporal denial of service**: A faulty or compromised node may monopolize the communication medium by continuously transmitting garbage messages. This *babbling-idiot* failure, introduced as a fault-propagation example in Section 1, prevents correct nodes from delivering time-critical messages [127]. Sending messages at a high rate can similarly overwhelm a receiver with interrupt load, preventing it from meeting the timing requirements of its real-time tasks [51]. In both cases the attacker exhausts shared communication or processing resources to induce deadline misses without ever breaking a cryptographic primitive.

2.2.1.3 Faults, Errors, and Failures: The Pathology of Impairments

Avizienis et al. establish the following fundamental three-stage chain of causality that characterizes the pathology of system impairment [1, 38, 58]:

- A **fault** is the hypothesized root cause of a failure, representing the lowest level perturbation in the system. Faults may be classified by their origin (physical or logical), their intent (accidental or malicious), their persistence (transient, intermittent, or permanent), and their location (internal or external to the system boundary). Examples include a memory bit flip caused by *electromagnetic interference* (EMI) (accidental, physical, external, transient) or an adversary injecting forged messages into the network (malicious, logical, external, transient or intermittent).

- An **error** is the manifestation of a fault within the system's internal state, constituting a deviation between the intended correct state and the actual current state. For example, when electromagnetic interference induces a bit flip in memory, the corrupted variable value produced by that flip is the error. Errors may remain dormant for milliseconds or months, depending on when the erroneous state is next

accessed, propagated, or executed, and they represent the intermediate condition through which a fault may eventually lead to a failure.

Errors may also exhibit different observability characteristics. *Bohrbugs* correspond to deterministic, easily reproducible errors that manifest reliably under the same conditions, making them comparatively straightforward to diagnose. In contrast, *Heisenbugs* arise from faults whose manifestations are sensitive to timing or microarchitectural effects, making them hard to reproduce [8].

- A **failure** occurs when an error propagates to the system boundary and is consumed by an external entity, causing the delivered service to deviate from the system's specified behavior. A failure is therefore an externally observable deviation at the service interface [1].

  Failures can be further classified along several orthogonal dimensions. In terms of *domain*, a failure may be a *value failure*, in which the system outputs an incorrect or corrupted value, or a *temporal failure*, in which the system delivers the value outside its required time window. With respect to *severity*, failures may be *benign*, which have no harmful effect on the system or its environment, or *malign*, which causes hazardous or unsafe consequences. In terms of *frequency*, failures may occur *once*, *repeatedly*, or *permanently*, depending on whether the underlying error is transient, intermittent, or persistent. Finally, failures may be classified by their *view*: a *consistent failure* is observed identically by all external entities, whereas an *inconsistent failure* presents different erroneous behaviors to different observers, complicating detection, diagnosis, and recovery [58].

The causal relationship *Fault* $\rightarrow$ *Error* $\rightarrow$ *Failure* can be applied recursively when a system is decomposed into several interacting components. An error inside the system that arises from the failure of one of its components is interpreted, at the system level, as a fault. For example, a physical sensor that fails to sample the environment constitutes a failure at the device level, yet this same event appears as an omission fault to the centralized voting algorithm operating at a higher logical level [58].

In order to capture the pathology of security breaches in computer systems, the classical fault-error-failure taxonomy is extended into the composite *Attack–Vulnerability–Intrusion* (AVI) model [51, 58]. In the AVI model:

- An **attack** is the execution of a deliberate and malicious external fault. It is an intentional interaction initiated by an adversary in order to obtain unauthorized access rights or to perform actions that violate system's security.

- A **vulnerability** is an internal fault that creates the necessary preconditions for an attack to succeed. Vulnerabilities arise from design flaws, implementation defects,

or configuration weaknesses, such as a buffer overflow bug or an unprotected communication port.

- An **intrusion** is the resulting erroneous internal state produced when an attack successfully exploits a vulnerability, reflecting the adversary's ability to alter the system's data, control flow, or temporal configuration. If the intrusion remains undetected or unmitigated, it eventually manifests as a failure at the system boundary.

The central principle of fault and intrusion tolerance is that this causal chain must be actively interrupted. The architectural objective is to detect and neutralize the error or intrusion before it propagates outward and manifests as a system-level failure.

2.2.1.4 Fault and Error Containment

Since errors can propagate across interconnected components, ensuring dependability requires architects to partition the system both physically and logically. Dependable architectures are therefore organized around clearly defined *Fault-Containment Regions* (FCRs) [58, 131].

An FCR comprises one or more subsystems designed such that the immediate consequences of any fault that affects them, whether of accidental or malicious origin, remain localized to that region. In other words, errors originating within one FCR must not propagate into other FCRs.

In order to qualify as a fully independent FCR and avoid common-mode failures, a subsystem must avoid sharing critical operational resources with other regions. Kopetz specifies that a true FCR requires its own power supply, its own clock as a timing source, and independent computational resources, including dedicated processing units and memory that are physically isolated from adjacent FCRs [1].

It is important to distinguish an FCR from an *Error-Containment Region* (ECR). An FCR defines the boundary within which a fault is assumed to remain, whereas an ECR augments this boundary with explicit error-detection mechanisms that identify and report the manifestation of a fault before it propagates beyond the region [131]. The error detector of an ECR must reside in a different FCR from the one being monitored, since placing both in the same region would allow a single fault to corrupt the component under monitoring and the monitor itself [131]. Because distributed systems require FCRs to communicate, error-containment mechanisms are needed at the boundaries of these FCRs to mediate outbound interactions and prevent erroneous states within one FCR from propagating into others.

2.2.1.5 Secure Startup and the Trusted Computing Base

A secure startup ensures that the software and firmware executed within an FCR have not been tampered with before the system begins operating. Upon power-on, control is transferred to an immutable hardware root of trust implemented in *Read-Only Memory* (ROM), whose contents are fixed at fabrication time and therefore trusted by construction. The ROM stores both the verification routine and the reference cryptographic material (e.g., a hash or public key) required to authenticate the next boot stage. Before handing over execution, the verification routine computes the digest of the subsequent software component and compares it against the trusted value stored in ROM [132,133].

If verification fails at any point, indicating either a malicious modification or an accidental corruption, the boot process is aborted and the node is prevented from joining the distributed system, ensuring it cannot inject erroneous or adversarial messages [51]. A successful verification sequence establishes a well-defined *initial trusted interval*: the period during which the node's software stack has been authenticated and the node may safely begin participating in the distributed protocol before becoming exposed to external interaction.

The *Trusted Computing Base* (TCB) is the minimal set of hardware, firmware, and software whose correct and uncompromised operation is required to enforce the node's security and dependability mechanisms; any compromise within this set allows those mechanisms to be bypassed by definition [134]. Secure startup mechanisms verify the integrity of the software components that belong to this trusted set, but they do not define the TCB itself. In systems where the TCB is implemented entirely in simple, immutable hardware components, trust is established by physical construction rather than by a secure startup process.

Because the TCB forms the foundation upon which all higher-level security and dependability mechanisms depend, the primary engineering directive is to minimize it. A smaller TCB reduces the attack surface, simplifies assurance activities, and, in high-assurance architectures such as *Multiple Independent Levels of Security* (MILS) systems, enables rigorous formal verification at the highest evaluation assurance levels [48,58]. Designs that realize the TCB using minimal hardware state further facilitate formal verification and reduce the likelihood of design vulnerabilities.

2.2.2 Degree of Synchrony and Operational Envelopes

Before architects can define failure models or design fault-tolerance mechanisms, they must first establish the synchrony model for the system, an abstraction that specifies the timing behavior and bounds of processes, communication channels, and clocks, and that provides the foundation for reasoning about temporal correctness, fault propaga-

tion, and the feasibility of real-time guarantees [58].

In addition, architects must bound the system's operational envelope: all possible behaviors of the environment specified to provide design-time assurance that the system will meet its timeliness requirements. Outside this envelope, the system may still satisfy deadlines with high probability or only for selected high-priority functions [66].

2.2.2.1 Degrees of Synchrony

The synchrony model for a distributed system specifies the timing bounds on process execution, message delivery delays, and clock drift rates and offsets. The choice of synchrony model is not merely an implementation detail; it fundamentally determines which classes of distributed protocols and failure-detection mechanisms are theoretically achievable [58]. Three main models are distinguished in the literature by their degree of synchrony: the asynchronous, synchronous, and partially synchronous models.

2.2.2.1.1 The Asynchronous Model

The asynchronous model makes no timing assumptions whatsoever: there is no known upper bound on computation time, no upper bound on message delivery delay, and local clocks may drift at arbitrary and unbounded rates. Any algorithm that is correct in the asynchronous model runs correctly in all other models of synchrony [135].

However, the fully asynchronous model has limitations that restrict its usefulness in many practical systems. The absence of temporal bounds makes it impossible to distinguish a crashed node from one that is merely slow, rendering many fundamental problems in distributed systems without deterministic solutions [135, 136]. Moreover, the inability to provide timing guarantees under this model hinders its applicability for real-time systems.

2.2.2.1.2 The Synchronous Model

The synchronous model posits tight, known worst-case bounds on all temporal parameters: every computational task has a bounded WCET, every message is delivered within a known maximum latency, and the clocks of all non-faulty nodes have known and bounded offset, as well as a known and bounded maximum drift rate. These assumptions greatly simplify the design and implementation of dependable distributed applications because correct components can accurately predict the progress of their peers [1,58]. For example, if a component fails to respond by the time the maximum processing delay, maximum communication delay, and the clock-offset bound have elapsed from a common reference instant, it can be conclusively deemed faulty.

Time-triggered systems follow the synchronous model. The temporal parameters of tasks and their messages are fully specified, and an offline scheduling algorithm assigns exclusive time slots for these activities on a global time base maintained by a fault-tolerant clock-synchronization protocol (see Section 2.1.6.6).

2.2.2.1.3 The Partially Synchronous Model

Recognizing that purely synchronous assumptions are difficult to sustain across large-scale and open networks, and that purely asynchronous models cannot solve consensus deterministically, Dwork, Lynch, and Stockmeyer introduced the *partially synchronous* model [137]. This model assumes that temporal bounds on processing and communication exist, but are either unknown or hold only eventually, after some unknown *Global Stabilization Time* (GST). From the perspective of a time-asynchronous system, this corresponds to long intervals of unpredictable timing behavior punctuated by periods in which the system behaves synchronously.

Crucially, once the system has stabilized, the existence of bounded delays resolves the indistinguishability between a crashed node and a slow one, enabling deterministic solutions to problems such as consensus that are impossible in the purely asynchronous model [58].

This is the synchrony model typically implemented in *event-triggered* real-time systems, where bounds hold during normal operation but may be violated transiently under overload or attack. Hence, while partial synchrony is adequate for soft real-time applications, hard real-time systems (e.g., flight control) generally cannot tolerate the unpredictability inherent in this model and therefore requires strict synchrony instead.

2.2.2.2 Operational Envelopes and Demand Paradigms

Even under a strictly synchronous system, temporal guarantees depend on assumptions about the volume, rate, and timing of external stimuli the system must handle, i.e., the *operational envelope*. Requirement specifications do not describe the environment itself; rather, they impose assumptions on it. These assumptions must be both useful, enabling the establishment of timing bounds for design-time assurance, and realistic, so that the system can deliver its intended service [58].

In many engineering domains, however, real-world parameters are not sharply bounded but follow probability distributions with long, uncertain tails (e.g., temperature, wind speed, or sensor noise). Even when physical limits exist, they may be too large, impractical, or economically infeasible to design for. As a result, designers often impose *artificial* or *engineering* bounds based on empirical evidence or probabilistic reasoning. For example, a designer specifying a maximum supply voltage, or a command and

control system defining a minimum inter-arrival time for detected aircraft [66].

The region of environmental behaviors for which the system's timing guarantees remain valid is called the *Safe Operational Envelope* [66]. Because it is impossible to guarantee that all deadlines will be met when events occur outside this envelope, two paradigms on system design emerged:

2.2.2.2.1 The Bounded Demand Paradigm

In safety-critical engineering, environmental demands are assumed to be strictly bounded: sensors generate periodic or sporadic events with a known minimum inter-arrival time, and network traffic respects a known maximum message rate. The system is then dimensioned, often conservatively via resource overprovisioning, so that all deadlines are met for every admissible environmental trace within this envelope [66]. This approach requires identifying all relevant input events, classifying them as periodic or sporadic (so that they impose a bounded load over any finite interval), and determining bounds on RT entities that are both physically justified and commensurate with the control technology.

Time-triggered systems embody this paradigm: an offline schedule, combined with established WCET bounds, ensures that the cumulative resource demand of all tasks never exceeds the provisioned supply as long as the environment remains within the specified envelope [1, 19, 138]. If the environment violates these assumptions, for example, due to a sensor fault producing an interrupt storm or an attacker flooding the network, the events are either ignored or the system's behavior becomes undefined.

2.2.2.2.2 The Unbounded Demand Paradigm

Real-time systems deployed in highly connected environments (e.g., autonomous vehicles interacting with smart-city infrastructure or industrial control systems reachable from remote networks) cannot practically assume that external demands remain strictly bounded. In these settings, adequate system dimensioning cannot be established with certainty at design time: environmental bounds may exist in principle but may be undiscoverable, too extreme to support economically, or invalidated by aperiodic inputs that impose no fixed upper bound on instantaneous load. Moreover, the consequences of output events may be too complex for complete analysis, and system-sizing calculations often depend on large numbers of cross-dependent assumptions that are probabilistic rather than guaranteed [66]. When these assumptions fail, the system inevitably operates outside its nominal envelope.

The *unbounded-demand* paradigm accepts this reality and designs for *graceful degradation*. Instead of requiring that all deadlines be met under all admissible conditions, the system is engineered to shed non-critical tasks under overload (e.g., infotainment

functions) to preserve computational resources for core safety-critical functions [58]. This approach is inherently defensive: significant effort is devoted to defining the system's behavior outside the envelope, rather than assuming such conditions can be excluded.

Event-triggered architectures align with this paradigm, where priority-based schedulers can dynamically adjust task execution in response to fluctuating load. However, this flexibility comes at the cost of weaker worst-case guarantees compared to the bounded-demand approach.

In practice, both approaches can coexist within a single architecture. A system may provide hard real-time services that are protected and guaranteed sufficient resources, following the bounded demand paradigm, while simultaneously offering soft real-time or best-effort services that follow the unbounded demand paradigm. The key requirement is strict isolation: the hard real-time subsystem must remain isolated from overloads or attacks that push the rest of the system outside its operational envelope [58].

2.2.3 Failure Modes and Adversary Models

System designers must define how components may deviate from their specification and what capabilities an adversary may possess. These two concerns are tightly coupled: every failure mode in the classical taxonomy can, in principle, be deliberately induced by an attacker. Hence, treating both clarifies the cost of tolerating each failure mode and the security mechanisms required to contain it.

The modes of failure that a node in a distributed system may exhibit form a two-dimensional hierarchy, illustrated in Figure 2.2.1, ordered from the most benign and least costly to tolerate to the most unconstrained and expensive [1, 58].

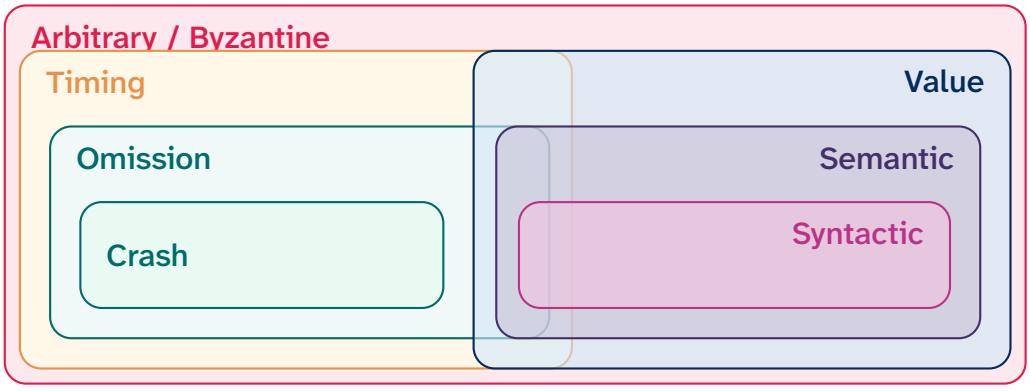


Figure 2.2.1: Failure mode hierarchy for distributed systems. Faults are categorized along a time and value domains axes. Each outer region subsumes the inner ones. The intersection of the two axes represents coupled failures, where a node fails in both the time and value domains simultaneously. All these classes are sub-classes of the Arbitrary/Byzantine behavior.

The Time Domain Axis:

- **Fail-silence and Crash Failures**: A fail-silent node operates correctly until it fails, at which point it permanently ceases to produce any output; this corresponds to the innermost crash failure.

- **Omission failures**: An omission failure occurs when a component sporadically fails to send or receive a message while otherwise continuing to execute correctly. A *send-omission* failure arises when an outbound message is not transmitted (e.g., dropped by the network interface), whereas a *receive-omission* failure occurs when an inbound message is not received (e.g., due to buffer overflow). Omission failures are typically transient, often caused by momentary EMI or burst congestion.

- **Timing failures**: A timing failure occurs when a syntactically and semantically correct value is delivered outside its prescribed temporal window. A *late* result, in real-time systems, is equivalent to an omission and is discarded. An *early* result may overwrite a valid state variable before the application has consumed it, leading to silent data corruption.

The Value Domain Axis:

- **Syntactic failures**: The innermost value domain failure where the delivered data is structurally corrupted or malformed (e.g., a truncated packet or a failed *Cyclic Redundancy Code* (CRC) checksum).

- **Semantic failures**: A value domain failure where the message is perfectly formatted and syntactically correct, but the actual information payload is logically wrong (e.g., a temperature sensor sending an incorrect reading).

- **Value failures**: A value failure occurs when a node delivers a result within its prescribed temporal window but with an incorrect value (whether syntactic or semantic). Under this failure mode, the faulty node is *symmetric* or *consistent*: it presents the same erroneous value to every receiver.

Coupled and Arbitrary Failures:

- **Coupled Failures**: The intersection of the time and value domain failure modes represents the interactions where a node fails in both time and value domains simultaneously. For instance, a node might deliver a semantically incorrect value well past its defined temporal window.

- **Byzantine (arbitrary) failures**: The Byzantine failure mode encompasses all previous classes, overlaps, and places no constraints on faulty behavior: a node may send contradictory values to different receivers, selectively omit messages, fabricate data, or deviate arbitrarily from the protocol. Its defining characteristic is *asymmetry*: a Byzantine node may report `TRUE` to one node, `FALSE` to another, and remain silent to a third, destroying the global consistency required for correct distributed operation. A Byzantine node may also behave *perfectly correctly* for extended periods, e.g., when compromised by a malicious adversary waiting for the most damaging moment to mount an attack. Because no restrictive assumptions are made, Byzantine failures are the most expensive class to tolerate.

  A physical class of faults that can manifest as Byzantine behavior is the *Slightly-off-Specification* (SoS) fault [1]. Examples of SoS faults include coding faults, in which a weak power supply may cause receivers to interpret divergent bit values from the same signal, and cut-off faults, in which a transmitter slightly overruns its allocated time slot, resulting in some nodes accepting the transmission while others reject it. Thus, a single, non-malicious analog degradation at the sender produces an asymmetric disagreement across the network.

The failure modes above arise both accidentally and under deliberate adversarial action [51]. *Node compromise* grants an attacker execution control over one or more nodes via software vulnerabilities or physical access, allowing the compromised component to exhibit arbitrary Byzantine behavior from within the system. The scope of damage is limited by the architectural error containment mechanisms at the boundaries of FCRs.

Network manipulation enables an adversary to interfere with communication without compromising any node. An attacker with access to the communication medium can eavesdrop, inject fabricated messages, or delay and reorder packets in transit. Cryptographic Message Authentication Codes (MACs) prevent external message injection by ensuring authenticity and integrity; however, no cryptographic mechanism can prevent an adversary from delaying, withholding, or selectively forwarding correctly authenticated messages.

Once an attacker gains access to system resources, whether through node compromise or through control over the communication medium, they may mount a wide range of attacks, including the timing attacks described in Section 2.2.1.2.

2.2.4 Theoretical Limitations of Distributed Computing

Designing real-time distributed systems requires first acknowledging the established theoretical results that constrain what is and what is not achievable under different assumptions about synchrony and failure modes. These results shape the design space:

they determine which guarantees can be provided in principle, which require additional architectural support, and which remain impossible.

2.2.4.1 Fundamental Limits of Time Measurement (FLTM)

Establishing the temporal order of events, particularly those related to observing the environment, is subject to limitations imposed by both the laws of physics and the imperfections of real-world implementations. Because of unavoidable uncertainties, such as variations in processing and message delivery delays, clock drift, and the finite accuracy of clocks, *perfect clock synchronization* (as discussed in Section 2.1.5) and, consequently, *perfect simultaneity* across spatially separated nodes is physically impossible [139].

Consequently, even under the reasonableness assumption of a global time base, determining the temporal order of events based on a global time base with granularity MT is constrained by the following *Fundamental Limits of Time Measurement* (FLTM) [1]:

- **FLTM (I):** A difference of one MT in the timestamps of two events is insufficient to reestablish their true temporal order, as illustrated in Figure 2.2.2. Even when an event e_1 occurs before an event e_2 in absolute reference time ($t_1 < t_2$), their observed clock readings can indicate $C_j(t_2) < C_i(t_1)$ due to the non-zero synchronization precision (π) between the nodes.

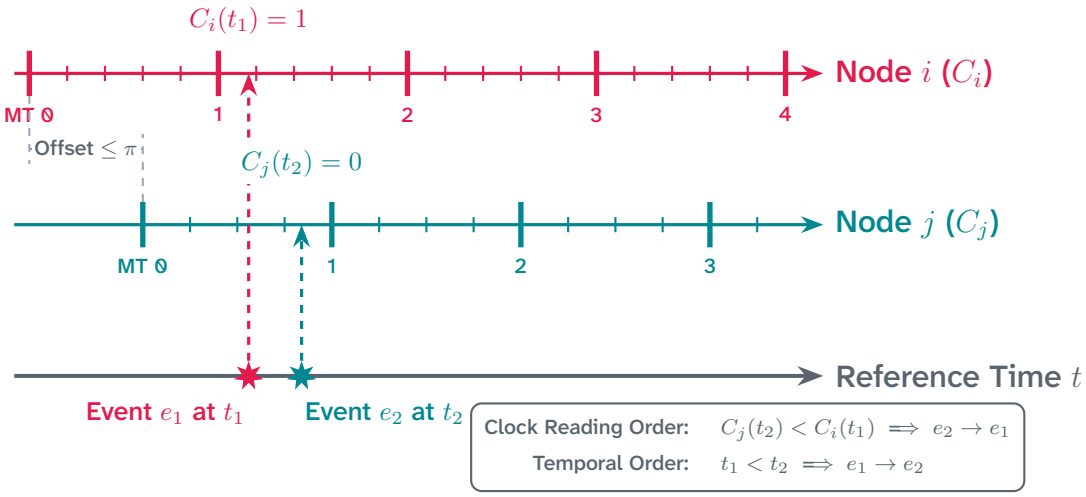


Figure 2.2.2: Illustration of FLTM (I). Two independent events e_1 and e_2 are generated at reference times t_1 and t_2, where $t_1 < t_2$. Event e_1 is observed by Node i (fast clock C_i), while event e_2 is observed by Node j (slow clock C_j). Because of non-zero π, their MT boundaries do not align. e_1 gets timestamp $C_i(t_1) = 1$, while e_2 gets $C_j(t_2) = 0$, falsely suggesting e_2 occurred first.

- **FLTM (II):** If the observed duration of an interval is d_{obs}, then the true duration d_{true} satisfies: $d_{obs} - 2MT < d_{true} < d_{obs} + 2MT$.

  This fundamental limit is illustrated in Figure 2.2.3, where the true duration between two events can be almost two full MTs longer (or shorter) than the timestamp difference suggests.

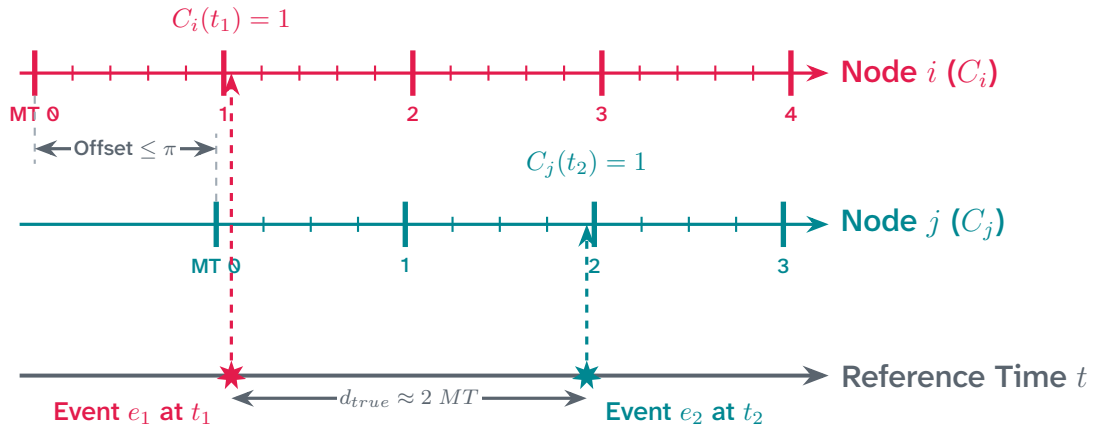


Figure 2.2.3: Illustration of FLTM (II). Event e_1 occurs immediately after Node i's MT 1 begins. Event e_2 occurs just before the end of Node j's MT 1. The observed duration is $d_{\text{obs}} = 1 - 1 = 0$ MT, but the true duration is $d_{\text{true}} \approx 2$ MT.

- **FLTM (III):** The temporal order of two events can be recovered from their timestamps only if the difference between the timestamps is at least two ticks.

  Hence, a true duration of $2\,MT$ between two events does not inherently guarantee that their temporal order can be recovered. Depending on the clock offsets, an interval of $2\,MT$ may yield an observed timestamp difference between 0 and 4 MTs. An example illustrating this limit is shown in Figure 2.2.4.

- **FLTM (IV):** The temporal order of events can always be recovered from their timestamps if the event set is $0/3MT$ precedent, i.e., concurrent events occur within an activity interval of length $0\ MT$, while successive events are separated by an interval of silence of length $3\ MT$.

While the first three limits describe what can be inferred from a given set of timestamped events, the fourth specifies how events must be organized so that their temporal order can always be reconstructed. However, since it is not physically possible to confine concurrent events to an interval of length $0\ MT$, this interval must be expanded to $1\ MT$, as done in the sparse time base concept described later in Section 3.1.7.1. Crucially, the system can enforce such precedence rules only for events that lie within the computer's sphere of control. Environmental events, by contrast, follow a dense time base and may occur at any instant on the timeline.

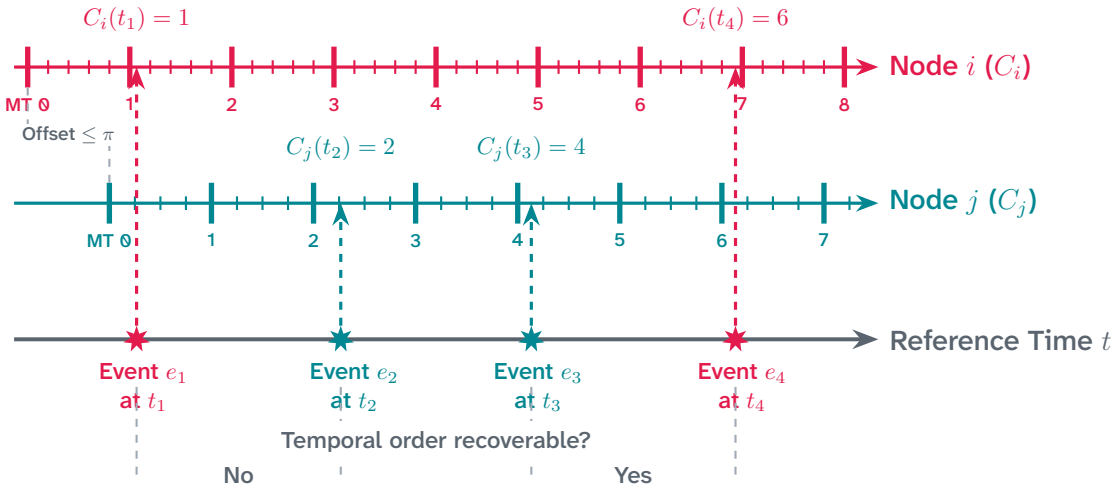


Figure 2.2.4: Illustration of FLTM (III). Two pairs of events (e_1, e_2 and e_3, e_4) are shown,both separated by an identical true duration of $2\,MT$. For the first pair, the clock offset causes the observed difference to be only $d_{obs} = 2 - 1 = 1\,MT$, meaning the temporal order cannot be reliably recovered. For the second pair, the timestamps yield $d_{obs} = 6 - 4 = 2\,MT$, crossing the threshold required for recovery.

2.2.4.2 The FLP Impossibility Result

In distributed systems, *consensus* is the fundamental problem of getting multiple nodes to agree on a single value or operational state. Consensus is a fundamental block of many higher-level distributed problems, including *State Machine Replication* (SMR), group membership, and coordinated actuation.

The FLP theorem formally proves that in a fully asynchronous system model (see Section 2.2.2.1), no deterministic algorithm can guarantee consensus if even a single node may fail by crashing [7].

The proof rests on the fact that an asynchronous system cannot distinguish between a node that has crashed and a node (or communication link) that is experiencing an arbitrarily long delay. Since no time bounds exist, a correct node can never safely conclude that a silent peer has failed. If the node waits indefinitely for the peer, the algorithm violates *liveness* since it may never terminate. If the node proceeds without the peer, and the peer's message arrives later, the algorithm violates *safety* since correct nodes may reach different decisions.

The FLP result has a direct architectural consequence: safety-critical real-time applications cannot rely on a purely asynchronous system. In order to ensure that safety-critical algorithms terminate deterministically in the presence of faults, the system *must* introduce synchrony assumptions, typically through architectural mechanisms that bound communication and processing delays [58].

It is worth highlighting that alternative strategies exist for bypassing the FLP result.

For example, nodes may employ randomized techniques such as coin-tossing or random back-off procedures to break symmetry, thereby ensuring agreement with probability 1 and substituting determinism with probabilistic termination [140]. However, since these approaches do not guarantee bounded termination times, they are unsuitable for the class of systems considered in this thesis.

2.2.4.3 Byzantine Consensus Bounds

Even if a system circumvents the FLP impossibility (see Section 2.2.4.2) by introducing synchrony assumptions, typically through the use of distributed failure detectors that eventually provide timely and accurate information about crashed nodes [141], achieving consensus in the presence of Byzantine faults remains fundamentally expensive.

Guaranteeing consensus in a system where nodes may exhibit the unconstrained and asymmetric behavior described in Section 2.2.3, requires satisfying the following constraints [142]:

1. **Spatial Redundancy Bound**: In systems with unauthenticated point-to-point channels, tolerating F Byzantine nodes requires the total number of nodes N to be strictly larger than three times the number of faulty nodes F, i.e., $N \geq 3F + 1$.

   This bound arises because a Byzantine node may *equivocate*, i.e., send mutually inconsistent messages to different recipients.

2. **Temporal Redundancy Bound**: Reaching consensus deterministically in the presence of F Byzantine faults requires at least $F + 1$ sequential rounds of message exchange among all participating nodes. This lower bound reflects the need to eliminate the influence of up to F potentially inconsistent or adversarial messages.

These spatial and temporal bounds can be economically or physically prohibitive (e.g., due to SWaP constraints), or even infeasible in real-time systems if the additional delay introduced by the required $F + 1$ communication rounds exceeds the system's end-to-end latency constraints. However, if the communication layer itself prevents equivocation, either by making it impossible for a sender to transmit different values to different receivers, as in hardware-enforced broadcast channels (e.g., TTP and FlexRay), or by making equivocation detectable (e.g., through cryptographic signatures) then Byzantine nodes lose the ability to influence different receivers with inconsistent information. In systems with such communication semantics, the spatial redundancy bound improves to $N \geq 2F + 1$.

Importantly, *no such reduction applies to the temporal redundancy bound*: even when equivocation is prevented or detectable, at least $F + 1$ communication rounds remain necessary to tolerate F Byzantine faults.

2.2.4.4 The CAP Theorem

The CAP theorem characterizes the fundamental tradeoffs that arise when a distributed system experiences network degradation. The CAP theorem states that in the presence of a network partition, a distributed system can guarantee at most two of the following three properties: *Consistency*, *Availability*, and *Partition Tolerance* [143]. Partition tolerance is not optional in any system deployed in failure-prone or hostile environments, because message loss, delay, and link failures are unavoidable. Consequently, in this setting, the architect must explicitly choose between maintaining consistency or maintaining availability [58][3].

In the domain of safety-critical real-time systems, these properties map onto classical dependability and security attributes:

- **Consistency (Safety)**: All correct nodes must maintain a consistent and valid view of the system state. Violations of consistency correspond to safety violations (e.g., conflicting actuator commands).
- **Availability**: The system must continue to accept inputs, compute outputs, and actuate the physical plant. Availability thus translates into fail-operational behavior.
- **Partition Tolerance**: The system must continue to operate despite arbitrary message loss, delay, or omission faults, whether caused by physical damage, interference, or malicious attacks.

2.2.4.5 Notions of Determinism and Common Knowledge

Determinism dictates that every action or outcome is inevitably caused by its preconditions and a fixed set of rules. A system model is deterministic if, given all inputs provided to the model, it defines exactly one possible correct behavior. However, judging whether a model is deterministic requires clearly defining what is interpreted as behavior from the perspective of an observer [4].

2.2.4.5.1 Deterministic Models

Two primary types of deterministic models are studied in the literature: *state-based models* and *data-flow-based models*. State-based deterministic models tie the notion of state to behavior, where system state encapsulates information about past inputs

[3] Recent work by Lee et al. revisits and generalizes the CAP theorem by quantifying the degree to which each property can be satisfied simultaneously [144]. Their GCAP framework replaces the binary (yes/no) formulation of CAP with a continuous tradeoff space, showing that systems may partially satisfy consistency and availability under bounded or probabilistic partition models.

such that knowing the state supplants the knowledge of the past [145]. However, relying on the notion of state requires defining the concepts of past, future, and present. Because the notion of present (perfect simultaneity) is not well-defined in the physical realization of parallel and distributed systems (due to imperfect clock synchronization, see Section 2.1.5), adopting this behavior definition is problematic [4].

In contrast, deterministic models based on data flow do not require observing an instantaneous global system state. Instead, the observer is restricted to observing output data, such as in *Kahn Process Networks (KPN)* [146], where the collection of message sequences on each channel is interpreted as the system behavior. While this bypasses the simultaneity problem, the absence of a global state makes coordinating complex decision-making and control-flow in pure data-flow models very difficult.

In order to resolve this coordination challenge, *synchronous-reactive languages* [147, 148] bridge the gap by reintroducing a discrete notion of state. They use a global discrete clock to model execution as *punctuated chaos*. In this paradigm, the system has a well-defined global state exactly at each clock tick, making decision-making straightforward. Between these ticks, computations are assumed to be carried out instantly and asynchronously, constrained only by data precedence. By defining behavior as the sequence of atomic state transitions that occur at these discrete ticks, the model is deterministic to an observer who evaluates the system solely at these boundaries.

2.2.4.5.2 Determinism and Common Knowledge

In distributed systems, achieving determinism in the temporal domain is fundamentally a problem of agreement: nodes must share *common knowledge* regarding a global clock and the order of events to process them identically. Common knowledge refers to the notion wherein each node knows a fact, knows that every other node knows the fact, and this recursive condition of mutual awareness extends indefinitely [149].

However, physical implementations are imperfect. Due to the inherent timing uncertainties and the FLTM discussed in Section 2.2.4.1, it is practically impossible to guarantee that spatially separated nodes initiate an activity simultaneously or produce outputs at precisely the same time. Consequently, true common knowledge and absolute temporal determinism are theoretically impossible in distributed systems without shared memory. Hence, establishing deterministic and temporally predictable behavior in distributed systems must rely on relaxed, practically achievable forms of common knowledge [149, 150]:

1. **Epsilon Common Knowledge**: Every node knows a fact within a bounded time interval of length ϵ. In real-time systems, this is often determined by the clock synchronization precision π. If global time at node i reads x, then after waiting for a duration π, node i can be certain that every other node's clock reads at least x.

Because consistent ordering of events solely based on imperfect timestamps cannot be guaranteed, systems relying on epsilon common knowledge must employ online agreement protocols to reach consensus on event ordering [151]. Notably, the agreed-upon order may not correspond to the actual temporal sequence of event occurrences [1].

2. **Simulated Common Knowledge**: Also known as internal knowledge consistency, this approach restricts the behavior of nodes such that they cannot detect any contradictions to the assumption that perfect common knowledge has been attained. It is applicable to internal specifications (events within the computer's sphere of control). By restricting event occurrence to sparse, discrete timeline, the system is able to simulate perfectly synchronized clocks, bypassing the simultaneity problem [5,152] (see Section 3.1.7.1).

2.2.5 Fault Tolerance

Ensuring system dependability begins with acknowledging the inevitability of faults. Traditional engineering practice emphasizes *fault prevention* through formal verification, static analysis, and physical measures (e.g., electromagnetic shielding). However, fault prevention alone is fundamentally insufficient for modern distributed real-time systems, where software and hardware complexity, environmental effects (e.g., cosmic radiation and thermal stress), and the presence of intelligent adversaries make it difficult to preclude the occurrence of faults.

Consequently, systems must incorporate *fault tolerance*: an active, runtime architectural capability that enables a system to continue delivering its specified services despite the presence of faults [38, 58]. Fault-tolerant architectures assume faults will occur according to a defined fault hypothesis and employ redundancy to detect, mask, or contain errors before they escalate into system-level failures.

The design of any fault-tolerant system begins with a fault model or hypothesis that precisely specifies the fault modes (see Section 2.2.3) that its FCRs may exhibit, their frequency, and their maximum number. The fault hypothesis further partitions the fault space into *normal faults* (level 1), which must be tolerated by the implemented fault-tolerance mechanisms, and *rare faults* (level 2), which lie outside the assumed model and are expected to occur only rarely [153].

Since fault-tolerance mechanisms do not cover rare level 2 faults (i.e., they are not designed to restore correct operation after the occurrence of such faults), safety-critical systems must also incorporate fallback or *never-give-up* (NGU) strategies that attempt to bring the system back into the domain of correct states [1].

2.2.5.1 Redundancy Dimensions

The main goal of fault-tolerance mechanisms is to prevent faults from propagating into failures. No single technique can address all fault scenarios; instead, the mechanisms to be deployed depend on the specific faults that must be tolerated and the service requirements. Whether the system is fail-safe, as in train control, where a safe state can be reached by stopping all trains, or fail-operational, as in an aircraft flight controller, which must continue operating until landing, fault tolerance requires using redundant resources, i.e., resources that would be unnecessary in a fault-free environment.

Redundancy is deployed across three primary dimensions:

1. **Space redundancy** involves replicating components such as sensors, processors, or communication buses. It is the primary mechanism for tolerating crash and Byzantine failures.

2. **Time redundancy** relies on repeated execution of computations or retransmission of messages. It is highly effective against transient faults (e.g., EMI bursts), but consumes the most critical resource in real-time systems: *time*. Its use must therefore be reflected in feasibility and schedulability analyses.

3. **Value redundancy** augments data with additional information to verify integrity or enable reconstruction. Examples include *Error Correcting Codes* (ECC) in memory, which mask transient bit-flips, and cryptographic signatures, which expose malicious manipulation.

2.2.5.2 Error Processing

Once an error-producing fault is identified within an FCR, error-processing mechanisms are activated to eliminate or neutralize it. This activity, known as *fault treatment*, proceeds through the following sequence of phases [58]:

1. **Error detection**: The prerequisite for any active recovery is recognizing that an error exists. Detection mechanisms must operate concurrently with normal system execution without significantly degrading WCETs. They span multiple abstraction levels: *temporal detection* relies on watchdog timers and scheduling monitors to identify crash, omission, and late-timing failures; *syntactic detection* uses parity bits, CRCs, and cryptographic signatures to expose value corruption or unauthorized fabrication; and *semantic detection* applies application-level reasonableness and plausibility checks to identify values that violate expected physical or logical constraints.

2. **Error confinement**: Once detected, the error must be confined so that it cannot propagate to other FCRs and cause failures. When the system lacks sufficient redundancy and mechanisms to recover from the error, the system or the faulty FCR may be forced into a controlled shutdown, restricting its behavior to crash failures, as is done in self-checking components.

3. **Error recovery**: Once the error is confined, the system must restore the corrupted state to a correct one. Two fundamentally different paradigms exist. *Backward Error Recovery (BER)* discards the erroneous state, rolls back to a previously saved and verified checkpoint, and re-executes the computation. Although BER is hardware-efficient, the time required for detection, rollback, and re-execution is highly variable, introducing substantial temporal jitter, and the action being rolled back may be irrevocable if its effects on the environment cannot be undone. This makes BER of limited utility in hard real-time systems, where missing a deadline constitutes a failure [1]. *Forward Error Recovery (FER)* identifies the erroneous state and applies corrective measures that cancel or mitigate the effects of the error, typically by using spatial or value redundancy to compute a new correct state and continue execution without rollback. The most stringent form of FER is *error masking*, in which sufficient redundancy outvotes or suppresses erroneous outputs so that correct service is delivered without any observable glitch; because masking is applied systematically, it does not require explicit detection. A common realization of error masking is replication through *Triple Modular Redundancy* (TMR), where three replicas execute in parallel and a majority voter masks a single faulty output; more generally, *N-Modular Redundancy* (NMR) extends this principle to N replicas, enabling tolerance of up to $\lfloor (N-1)/2 \rfloor$ faulty components [154].

2.2.6 Distributed Fault-Tolerance Problems

Replicating computation and state across a distributed system introduces a family of coordination problems that must be solved to maintain consistent behavior in the presence of faults. Nodes must agree on which values to apply, in what order to apply them, how to disseminate updates consistently, how to maintain a consistent view of which nodes are currently members of the system and how to synchronize replicated state so that faults can be detected and masks. These challenges give rise to fundamental problems such as consensus, atomic multicast, membership, and state-machine replication, which together form the algorithmic foundation of fault-tolerant distributed systems [58,142].

2.2.6.1 Agreement and Event Ordering

Replicating state across a distributed system fundamentally introduces an *agreement* problem. If two replicas observe the same external requests but in different orders due to communication delays or message reordering, their internal states will diverge. Preventing this requires replicas to agree both on the *values* to apply and on the *order* in which these values are applied [58].

2.2.6.1.1 Consensus and Interactive Consistency

The classical *consensus* problem (briefly described in Section 2.2.4.2) asks a set of nodes to propose values and eventually decide on a single value that satisfies:

- **Validity**: the decided value must satisfy an admissibility condition, typically that it was proposed by some node.
- **Agreement**: no two correct nodes decide differently.
- **Termination**: all correct nodes eventually decide.

Even under simple crash-failure assumptions, consensus is non-trivial: it cannot be solved deterministically in a purely asynchronous system (the FLP result; see Section 2.2.4.2). This motivates additional assumptions such as partial synchrony or the use of failure detectors to guarantee progress [58].

In more general failure models, consensus extends to *Byzantine agreement*, where faulty nodes may behave arbitrarily, including sending inconsistent messages to different peers. Closely related is the *interactive consistency* problem (the Byzantine generals formulation [155]), in which each process has an initial value and all correct processes must agree on a common vector of values that reflects the initial values of all correct processes, assigning a default value for faulty ones. Interactive consistency forms the basis for establishing consistent system-wide views in the presence of arbitrary faults.

2.2.6.1.2 Reliable and Atomic Broadcast

In practice, consensus and interactive consistency are often implemented using *group communication* abstractions:

- **Reliable multicast** provides baseline guarantees on message delivery, satisfying [156]:
  - **Validity**: if a correct sender sends a message, all correct receivers eventually deliver it.
  - **Integrity**: messages are delivered at most once and only if they were sent.

  - **Agreement**: if any correct receiver delivers a message, then all correct receivers eventually deliver the same message.

- **Atomic (total-order) multicast** strengthens reliable multicast with a *total-order* property [157]: if two correct receivers deliver messages m_A and m_B, they deliver them in the same relative order. Formally, all correct processes deliver the same set of messages and do so in an identical sequence.

Atomic broadcast is equivalent to consensus under standard system assumptions, and it is the fundamental building block used by most fault-tolerant replication protocols [58, 157].

2.2.6.2 Group Membership

Agreement on data must be complemented by agreement on *who* is part of the group (the group view). A *group membership service* maintains a consistent, system-wide view of which nodes are currently considered operational to participate in agreement and actuation. This information is represented as a sequence of *membership views*, each listing the set of nodes believed to be correct or reachable at a given time. Membership is inherently dynamic: the group may expand as new nodes join or shrink as nodes leave or fail [58, 157].

A group membership service can be characterized by the following properties:

- **Timely detection**: if a node fails in a detectable manner, all correct nodes must detect the failure and adopt a new membership view that excludes the failed node within a known, bounded time.

- **Primary partition**: in the presence of communication partitions, only the primary (majority) partition may continue to disseminate new membership views. Nodes in minority partitions must transition to a passive or fail-silent state and later rejoin the system, typically by catching up to the current view or rejoining as new members, once connectivity is restored.

The group membership service is highly dependent on failure detection. Unreliable failure detection can cause the membership to have erratic behavior. In the crash failure setting, membership can be implemented using timeouts, failure detectors, and majority quorum rules, typically integrated with the communication subsystem. Under more severe failure models (e.g., Byzantine faults), membership must also handle equivocation and malicious misreporting. In such cases, interactive consistency mechanisms become necessary to ensure that all correct processes adopt the same sequence of views [58].

2.2.6.3 Replication Management and State Machine Replication (SMR)

Replication management encompasses the mechanisms by which a distributed system ensures service continuity in the presence of faults. Among these mechanisms, SMR is the most widely used approach for strong consistency (i.e., ensuring that all replicas behave as a single logical copy by applying operations in the same total order). SMR models the service as a deterministic state machine replicated across multiple nodes. Clients submit requests; replicas receive these requests in exactly the same order (typically via an atomic broadcast primitive); and each replica processes them to evolve its local copy of the state.

If all replicas start from the same initial state and process the same totally ordered sequence of inputs, their states remain consistent, and faults can be masked by comparing replicas' outputs. In general, SMR protocols require $F + 1$ replicas to tolerate F crash failures, $2F + 1$ replicas to tolerate F value (consistent) faults, and $3F + 1$ replicas to tolerate F Byzantine (arbitrary) faults [58].

Within the broader space of replication management, three classical strategies are commonly distinguished:

1. **Passive replication (primary–backup).** A single primary replica executes all client requests and periodically sends state updates (or checkpoints) to one or more backup replicas. When the membership service detects that the primary has failed, a backup is promoted to primary and resumes request processing.

   This strategy assumes fail-silent behavior of the primary: backups trust the primary's state updates, making the approach unsuitable for tolerating Byzantine failures. Furthermore, the fail-over latency, comprising failure detection, primary election, and state synchronization, is typically too large and too variable for hard real-time control loops [1, 58, 66].

2. **Asymmetric Active replication (leader-based SMR).** In this strategy, all replicas execute the same commands, but a designated leader is responsible for making all scheduling decisions (e.g., the order in which client requests are processed) and disseminating this information to the follower replicas, which then execute the requests in the prescribed order. Asymmetric active replication thus relies on the leader, rather than an atomic multicast primitive, to impose a consistent execution order [58]. This approach underlies many practical SMR-style protocols for both crash-tolerant and *Byzantine Fault-Tolerant* (BFT) environments[4] in which a leader coordinates ordering while followers verify and execute. Although more

[4] Prominent examples include Paxos and Raft for crash tolerance [158–160], and PBFT, Prime, EBAWA for Byzantine environments [161–165].

robust than passive replication, leader-based replication still experiences temporary unavailability or degraded service during leader election and reconfiguration, which can significantly affect the timeliness of real-time tasks [1]. A further limitation, highlighted by the Prime protocol [163], is that performance can degrade significantly under faults or adversarial behavior: a malicious leader or even misbehaving clients can force the system into worst-case execution paths, reducing throughput and increasing latency despite maintaining safety.

3. **Symmetric Active replication (Decentralized SMR).** In this strategy, all replicas independently and concurrently execute the same sequence of commands in the same total order, producing identical outputs unless affected by faults. Replica consistency is ensured by an atomic multicast primitive that delivers client requests to all replicas in the same total order. Because every replica performs the full computation, symmetric active replication is resource-intensive: it requires full hardware and software redundancy at each replica and relies on the more expensive total-order message delivery property.

   A key advantage of symmetric active replication is true forward error masking: faulty outputs are outvoted at each step without rollback, reconfiguration, or fail-over delays. This makes the approach particularly attractive for mission- and safety-critical real-time systems (e.g., flight control), where avoiding fail-over gaps and ensuring tight bounds on fault-masking latency are essential [1, 58].

   A classical instance of this strategy is *Triple Modular Redundancy* (TMR), in which three replicas execute in lockstep and a majority voter masks a single crash or value (arbitrary but consistent) fault. TMR can tolerate such faults with only three replicas because the voter directly compares replica outputs at each step, eliminating the need for a distributed agreement protocol. However, this reduction in replica count does not extend to general Byzantine failures: tolerating arbitrary, inconsistent behavior requires distributed agreement on every step, which in turn necessitates at least $3F + 1$ replicas to mask F Byzantine faults.

2.2.7 Wormholes Model and Architectural Hybridization

The evolution of distributed computing has created a growing tension between what applications require and what infrastructures can reliably provide. Modern systems must reconcile uncertainty with predictability: applications demand responsiveness, determinism, robustness, and security, while underlying platforms are increasingly uncertain due to complexity, heterogeneous networks, and asymmetry [166, 167]. On the *time facet*, the asynchronous model cannot directly support the timed requirements of many

applications, yet real systems still fall short of a fully synchronous model. On the *security facet*, assuming a crash fault model for the trusted computing base (TCB, see Section 2.2.1.2) is unrealistic for secure system design, while the Byzantine model, although adequate for independent accidental faults, is ill-suited to reasoning about coordinated attacks and common-mode vulnerabilities [166].

Early work attempted to recover synchrony in the time dimension, using partial synchrony (Section 2.2.2.1), unreliable failure detectors [141], or timed-asynchronous models [168], under the assumption that systems would eventually behave synchronously long enough to make progress. Veríssimo observed that synchrony also varies in the *space* dimension: different components naturally exhibit different timing properties and failure modes [166]. By acting on this spatial dimension, synchrony can be *constructed* where needed, rather than merely assumed to appear, leading to more realistic and verifiable designs.

The *wormholes* model formalizes this idea by structurally separating a distributed system into subsystems with distinct synchrony and fault assumptions [166]. A *payload* subsystem contains the processes where applications normally execute, communicating over payload channels and following weak assumptions (e.g., asynchronous timing and possibly Byzantine failures). A *wormhole* subsystem contains a small number of simple, highly predictable processes, implemented in trustworthy components such as FPGAs, ASICs, or isolated RTOS-based controllers. Wormhole processes follow stronger assumptions (e.g., synchronous processing and crash-only failures). Communication between realms occurs exclusively through *wormhole gateways* with well-defined interfaces: payload processes observe only the abstract services provided at the gateway and need not know how the wormhole is built. Wormholes can be *local*, with each payload subsystem paired with its own wormhole, or *distributed*, with wormholes additionally interconnected through trusted channels. In both cases, the wormhole is a privileged mechanism, used only when necessary, to implement functionality that is hard or impossible to realize reliably in the weak payload environment.

Architectural hybridization generalizes this principle into a broader design paradigm [167]. Systems are intentionally constructed from subsystems with different extra-functional properties, timing guarantees, fault models, security guarantees, quality-of-service levels, and these properties arise from how each subsystem is engineered. Hybrid architectures thus provide a practical foundation for implementing abstractions that are infeasible in homogeneous fully asynchronous models, such as timely execution, synchronous channels, or perfect failure detectors [166].

Several leader-based BFT protocols exemplify the use of *local* wormholes. *MinBFT* [169] augments each replica with a small tamper-proof service, the *Unique Sequential Identifier Generator* (USIG), which assigns a monotonically increasing, uniquely signed identifier to each protocol message. This trusted local component enforces non-

equivocation and simplifies reasoning about message ordering, allowing MinBFT to reduce the classical SMR requirement from $3F+1$ Byzantine replicas to $2F+1$ while preserving safety and liveness under standard assumptions [169]. *CheapBFT* [170] builds on the same idea, implementing the USIG in FPGA hardware and exploiting the wormhole to run a fast path with only $F+1$ active replicas in the fault-free case, while still relying on a MinBFT-style protocol with $2F+1$ replicas to handle faults. In both designs, the trusted local component is a concrete instance of a wormhole: a small, strongly-assumed subsystem co-located with otherwise asynchronous, potentially Byzantine payload replicas.

At the other end of the spectrum, *BFT-TO* [171] avoids the performance issues of leader-based protocols while simultaneously reducing the number of required replicas to $2F+1$ by performing ordering in a distributed fashion. Instead of relying on a single leader, BFT-TO uses *distributed wormholes* to prevent equivocation and to coordinate ordering among replicas. The *Timely Computing Base* (TCB) and its trusted variant, the *Trusted Timely Computing Base* (TTCB), are experimental realizations of such distributed wormholes [68, 166, 172]. The TCB provides a set of synchronous, time-aware services, such as timely execution triggers, duration measurement, and timing failure detection, to an otherwise asynchronous payload [68, 166]. The TTCB extends this idea with security and intrusion-tolerance mechanisms, enabling efficient Byzantine-resilient protocols by offloading critical trusted operations to a small, well-engineered wormhole subsystem [172].

By combining weakly- and strongly-assumed subsystems in a principled way, the Wormholes model and architectural hybridization enable the construction of resilient, efficient, and realistic distributed systems that reconcile the need for timeliness and security with the practical limits of large-scale asynchronous infrastructures [166, 167].

2.2.8 Intrusion Tolerance, Resilience, and Rejuvenation

SMR provides the foundation for maintaining continuous service delivery through replication and fault masking. However, classical SMR is fundamentally predicated on a *static* fault model: correctness is guaranteed only as long as the number of simultaneously faulty replicas does not exceed the predefined threshold F [58]. This assumption is reasonable for short-lived faults or benign crash failures, but it becomes dangerously insufficient in open, long-lived systems that must operate in hostile environments.

Over extended operational periods, hardware components age and intelligent adversaries mount coordinated, persistent attacks, exploiting dormant vulnerabilities. Given enough time, an adversary will eventually compromise more than F replicas through stealthy intrusions. Once $F+1$ replicas are under adversarial control, the replication quorum is breached and the entire system can be subverted [136, 173]. This inevitability

transforms the static fault threshold from a safety guarantee into a time-limited assumption.

Ensuring long-term survivability therefore requires shifting from static fault masking to dynamic *proactive resilience*. The central mechanism enabling this shift is *rejuvenation*: the restoration of replicas to a known safe and secure state before accumulated faults or intrusions can exceed the tolerated threshold. By continuously refreshing replicas rejuvenation prevents adversaries from maintaining long-term footholds and protects the replication quorum from being compromised [174].

2.2.8.1 Intrusion Tolerance and Intrusion Resilience

Classical fault tolerance assumes that faults are accidental, independent, and probabilistically distributed. Under this benign model, once an error is masked through spatial redundancy, the system is considered safe because the probability of the same fault striking multiple replicas simultaneously is negligible.

Intrusion tolerance fundamentally alters this paradigm by explicitly acknowledging the presence of an intelligent, adaptive adversary capable of compromising system nodes [173]. Intrusion tolerance accepts the inevitability of partial compromise: the goal is not to prevent all intrusions, but to ensure that successful attacks remain structurally confined to a minority of replicas, never exceeding the replication threshold F at any given time, so that the system as a whole continues to operate correctly.

However, in adversarial settings, intrusions are neither independent nor random. If all replicas in an SMR ensemble run identical software stacks, a vulnerability exploited in one replica can be exploited immediately across all others. This correlation breaks the independence assumption and can cause the number of compromised replicas to exceed F, thereby invalidating the safety guarantees of classical SMR.

Intrusion resilience extends intrusion tolerance to long-lived systems. It ensures that the number of compromised replicas never exceeds F *over time*, even under continuous attack [136]. Achieving intrusion resilience requires preventing the adversary from accumulating permanent control over replicas. The key mechanism is rejuvenation: periodically restoring replicas to a known safe and secure state so that intrusions are removed faster than they can be acquired.

Both intrusion tolerance and intrusion resilience rely on architectural mechanisms that reduce exploit correlation, limit adversarial persistence, and protect the replication quorum. Three foundational principles support these goals:

- **Diversification**: Deploying heterogeneous implementations of the same logical function across replicas. Techniques such as N-version programming [175, 176], dissimilar operating systems, or varied hardware architectures break exploit cor-

relation. Diversification ensures that a single vulnerability cannot be weaponized uniformly across all replicas, enabling majority voting to reliably mask intrusions [136].

- **Obfuscation and randomization**: Dynamically altering the execution environment to invalidate adversarial knowledge. Mechanisms such as address space randomization and runtime obfuscation of code or scheduling decisions [129, 177, 178] hinder the reuse of collected information, making exploit formulation significantly more difficult.

- **Rejuvenation**: Ensuring that the adversary cannot compromise replicas faster than the system can recover them. In intrusion-resilient architectures, the critical resource is the number of correct replicas required to maintain the replication quorum. *Resource exhaustion* occurs when the number of simultaneously faulty replicas exceeds F. Preventing this, i.e., achieving *resource exhaustion safety*, requires proactive and reactive rejuvenation mechanisms that bound adversarial persistence and maintain the system within the tolerated fault threshold [174].

2.2.8.2 Rejuvenation Levels and Mechanisms

Executing rejuvenation within a distributed architecture requires determining the level at which rejuvenation is performed and specifying the conditions that trigger it.

2.2.8.2.1 Rejuvenation Granularity

The selected depth of rejuvenation is associated with an engineering trade-off between *fault coverage* (the depth of the security cleansing) and the *blackout interval* (the duration of the replica's unavailability during recovery) [179, 180]. This taxonomy of rejuvenation levels was originally developed in the software reliability literature to combat software aging and resource exhaustion. Castelli et al. introduced the necessity of selective rejuvenation [179], and subsequent empirical studies formalized a strict hierarchy of recovery depths [180, 181]. The intrusion-resilience community has since repurposed these varying recovery depths to eliminate malicious compromises [162, 182]. Ordered from the shortest to the longest blackout interval, the three main levels are:

- **Task-level rejuvenation.** This granularity targets exclusively the specific user-space task replica. The system terminates the task, flushes its allocated heap and stack memory, and restarts the task from a pristine binary image [181, 183]. Because it avoids reloading the operating system kernel, rejuvenation at this level incurs a minimal blackout interval, making it highly suitable for high-frequency

real-time control loops. However, its fault coverage remains bounded: any malicious compromises or aging effects embedded deep within the OS kernel, device drivers, or firmware will survive the recovery intact [180].

- **OS-level rejuvenation.** When an intrusion penetrates the OS kernel or breaches the task-OS boundary, a deeper cleansing is required. Reducing the extensive blackout interval of a full electrical power cycle involves executing a warm boot of the operating system that flushes all volatile software state (e.g., the kernel heap, page tables, network stack, and drivers state). Because a compromised OS cannot securely initiate its own replacement, this process must rely on an isolated, higher-privileged root of trust. Architects may implement this using a secure hardware watchdog that forces a jump to an immutable ROM [162], an architectural hardware wormhole [182], or a trusted hypervisor [184,185]. While this method successfully removes kernel-level intrusions faster than a cold boot, it requires specialized trusted components and leaves volatile hardware states (e.g., peripheral controller memory) uncleared [180].

- **Node-level rejuvenation.** The most exhaustive form of recovery is a complete hardware cold boot. Removing and restoring the supply voltage forces the node to reset main memory and all peripheral controllers to their power-on defaults, completely clearing all microarchitectural state (e.g., caches and TLBs) [180]. The node then re-executes the full secure-startup chain (see Section 2.2.1.5), re-authenticating every software layer from the hardware root of trust before fetching the updated service state from other correct replicas [182]. While a cold boot provides the ultimate recovery guarantee against both deep software compromises and volatile hardware faults, it incurs the largest blackout interval.

2.2.8.2.2 Rejuvenation Mechanisms

The frequency and the specific conditions under which rejuvenation is triggered define its role as an intrusion-resilience mechanism. Rejuvenation mechanisms are categorized into two paradigms with complementary goals: *reactive* rejuvenation, which is driven by diagnostics to recover a detected failed node, and *proactive* rejuvenation, which operates independently of the operational status of nodes to remove stealthy intrusions before they can propagate [182,186].

- **Reactive Rejuvenation.** Under this approach, recovery is initiated upon the detection of an error or an anomaly. The system utilizes error detection mechanisms to identify a compromised node and subsequently forces it into recovery. While highly efficient for restoring nodes that have visibly crashed or exhibited

overt faulty behavior, reactive recovery is fundamentally inadequate against sophisticated cyber-attacks. Advanced adversaries persistently execute *stealthy intrusions* that evade standard intrusion and error detection systems. Once compromised, a malicious replica can behave perfectly according to its specification during a dormant phase, waiting for a coordinated trigger signal or for the adversary to compromise the critical threshold of $F+1$ replicas required to subvert the voting quorum [186]. Because the intrusion produces no observable error during this dormant phase, reactive diagnostics are never triggered, granting the adversary infinite time to breach the rest of the system.

- **Proactive Rejuvenation.** Defeating stealthy, dormant intrusions mandates the implementation of proactive rejuvenation [162,174], which does not rely on diagnostics or error detection. Instead, it operates on the conservative assumption that any node in the network might currently harbor a silent compromise. The system continuously and periodically forces nodes to undergo a state recovery, regardless of their apparent health or recent operational behavior [162].

  Formalizing the execution of proactive recovery in distributed systems led Sousa et al. to develop the *Proactive Resilience Model (PRM)* [174, 186]. The PRM defines the mathematical constraints required to ensure that an adversary can never outpace the system's defensive rejuvenation period. The defining system parameter is the minimum time required for an adversary to successfully compromise $F+1$ replicas, denoted as $T^{min}_{Exhaust}$. This duration is inherently bounded by real-world constraints (e.g., the attacker requires time to gather information, develop exploits, and evade network detection), and is treated as a conservative lower-bound that architects must explicitly specify at system deployment time [174].

  The core architectural guarantee of the PRM is *exhaustion-safety*. A system is exhaustion-safe when the rate of proactive rejuvenation is strictly faster than the maximum assumed rate at which an adversary can discover vulnerabilities and compromise $F+1$ replicas. Let T_{Rejuv} denote the rejuvenation period, which defines the interval between consecutive triggers of the replica recovery procedure, and let T_{WCR} denote the worst-case rejuvenation time, which represents the WCET required to complete replica rejuvenation. The system maintains exhaustion safety as long as $T_{Rejuv} + T_{WCR} < T^{min}_{Exhaust}$ (see Figure 2.2.5), which ensures that the number of simultaneously compromised nodes, that is, the exhausted resources, remains perpetually below the maximum fault threshold F. Under this condition, continuous safe operation is guaranteed [186].

  A fundamental constraint of proactive rejuvenation is that a recovering node is temporarily unavailable. The system must therefore maintain a sufficient num-

ber of operational replicas during the blackout interval to ensure continuous fault masking and correctness of the provided real-time service. Proactively rejuvenating K replicas simultaneously (where $K \geq 1$) requires the total number of replicas N to be increased beyond the standard Byzantine threshold (see Section 2.2.4.3). The required replication degree depends heavily on the system synchrony model:

- **Asynchronous Model:** Under an asynchronous model, there are no reliable timing bounds. The system cannot distinguish between a replica that is taking a long time to rejuvenate and a replica that has suffered a Byzantine failure or crashed. Guaranteeing liveness (that the system can make progress without waiting for the recovering replicas) and safety (that the F faulty replicas cannot outvote the correct ones) under unbounded delays dictates that the system must add $2K$ additional replicas, resulting in a minimum replication degree of $N \geq 3F + 2K + 1$ [162].
- **Synchronous Model:** Under a synchronous model, or when using an isolated, synchronous wormhole to coordinate recovery [174,182], the rejuvenation execution time is strictly bounded and known. Therefore, the system can guarantee that a recovering replica is merely unavailable. Because the system must remain able to reach consensus among the remaining available replicas even if F faults occur while K replicas are offline for rejuvenation, the minimum required replication degree is reduced to $N \geq 2F + K + 1$ [182].

During runtime, all replicas periodically execute an offline generated proactive rejuvenation schedule, which comprises $G = \left\lceil \frac{N}{K} \right\rceil$ rejuvenation slots, taking the synchronous model as example, we have $N = 2F + K + 1$. During each such slot, up to K replicas of the group may rejuvenate in parallel. Figure 2.2.5 shows an example of a proactive rejuvenation scheme for a group with $N = 5$ and $K = 2$. A replica node n_x of the group is assigned to proactive rejuvenation slot PRS_w if $(x \mod G = w)$, where $x = \{0, \ldots, N-1\}$ and $w = \{0, \ldots, G-1\}$ indicate the unique index of the replica and rejuvenation slot respectively. The size of the rejuvenation slot T_{RS} needs to be set long enough to accommodate the worst-case node rejuvenation time T_{WCR}. As a result, the rejuvenation period is bounded from below by $T_{Rejuv} \geq G \times T_{RS}$ and from above by $T_{Rejuv} \leq T^{min}_{Exhaust} - T_{RS}$.

Hence, combining BFT SMR with proactive and reactive intrusion-resilience mechanisms yields an architecture capable of withstanding both accidental faults and malicious intrusions [136]. Maintaining this periodic cleansing process, together with diversification and obfuscation techniques, prevents adversaries from compromising the quorum required to subvert the system and thereby supports the uninterrupted delivery of real-time services in open and hostile environments.

Figure 2.2.5: Proactive rejuvenation of a group comprising $N = 5$ replicas, where up to $K = 2$ may rejuvenate in parallel.

"We may regard the **present state of the universe** as the **effect of its past** and the **cause of its future**."

(Pierre-Simon Laplace)

Chapter 3

The Time-Triggered Architecture: Foundations and Dependencies

In this chapter, we focus on the existing TTA introduced in [19], providing an overview of its system model, its node and network structure, and the real-time and dependability services it offers. We then introduce TTA's extra-functional properties from a hierarchical perspective that highlights their interrelationships. Building on this foundation, we present our dependency analysis, which classifies how these properties rely on specific TTA services and mechanisms, and discuss alternative solutions together with their trade-offs.

3.1 Overview of the TTA

The TTA is a concrete architectural realization of the TT paradigm. It combines a globally synchronized time base with offline-defined schedules (see Section 2.1.6.6), and provides a suite of architectural services, including clock synchronization, group membership, atomic multicast, guardians, and mode changes, that collectively enable the extra-functional properties required in safety-critical systems.

3.1.1 System Model and Real-Time Computer Design

The TTA adopts the three-subsystem real-time model introduced in Section 2.1.1: an RT computer system interacting with a controlled object and a human operator via instrumentation and human-computer interfaces. This model forms the basis for how the TTA designs its RT computer system. Accordingly, the TTA makes the following design decisions for its RT computer system:

- The RT computer system consists of one or more *clusters* of TTA nodes that communicate exclusively by exchanging state messages over TT networks. Communi-

cation is deterministic and periodic: nodes do not react to asynchronous external events but exchange state information at predefined points in global time.

- The system follows a state-based design methodology: the observation, transformation, and communication of RT entity states occur at periodic time instants. Each state message carries an RT image of one or more entities and is transmitted such that receivers can use it within its temporal-accuracy interval (see Section 2.1.2). The observation period for each entity must match the dynamics of the underlying physical process; short-lived but important state changes must be buffered until the next observation instant.

- The timing of all internal activities (e.g., task execution, state observation, and message transmission) is derived from the global time base (see Section 2.1.6.6). Temporal control is thus *internalized*: activities are triggered by the progression of global time rather than by asynchronous interrupts, ensuring globally coordinated behavior across all nodes.

Since all activities of the RT computer system are tied to predetermined global-time instants, the design process can cleanly separate the specification of the system architecture from the implementation of its constituent nodes.

In principle, TTA follows a two-step design methodology to separate architecture design from node design. In the first step, the architecture is designed, grouping TTA nodes into clusters and fully specifying their real-time service interfaces in the time and value domains. This involves specifying what RT images are communicated and their temporal constraints. A TT communication schedule is generated based on the specification (see Section 2.1.6.6).

In the node design phase, the application software that performs transformations on received RT images is implemented. In a top-down approach, the node implementation takes the fetch and delivery instants specified in the node's real-time interface as temporal constraints. Hence, the real-time operating system running on the node must implement a reasonable scheduling strategy to meet these temporal constraints. For a set of existing reusable nodes, a bottom-up approach can be followed instead, where the communication schedule is calculated with the available node's performance and application requirements as constraints. While this top-down approach is common, recent methods have opted to co-schedule networks and nodes to improve schedulability and resource efficiency [10, 187–189].

The design philosophy of the TTA highlights a fundamental trade-off: complexity is deliberately shifted into the design phase, where timing, interfaces, and communication patterns can be fully specified and verified. By internalizing temporal control and relying on globally coordinated behavior, the TTA achieves a high degree of predictability at the

cost of runtime flexibility. These choices reflect the technological context in which the TTA was conceived, an era characterized by simpler hardware and more stable operating conditions, yet the principles and design decisions of the TTA continue to offer benefits for systems that prioritize predictability over flexibility.

3.1.2 TTA Cluster Topologies

Following engineering insights and the real-time application structure, the application is mapped onto a set of TTA clusters, where each cluster is formed by grouping related TTA nodes to realize a particular application function (e.g., a cluster for vehicle dynamics control in an automotive system). The nodes of a TTA cluster share a communication subsystem – a TT network (such as TTP/C), which comprises the communication controllers of the nodes and the replicated communication channels.

These channels consist of the physical wiring used to transport messages and, depending on the network type and topology, may also include network switches.

Two different clusters can be interconnected by means of a gateway node that belongs to both clusters and provides a data-sharing channel between them. Such a gateway contains two communication controllers that may implement different TT network protocols. Furthermore, network access control and the prevention of untimely node accesses to the communication channel(s) are handled by additional hardware components called guardians. We discuss guardians in more detail in Section 3.1.7.4.

Figures 3.1.1, 3.1.2 and 3.1.3 show the three implementations of the TTA network interconnect specified in [190] for the TTP/C protocol, which differ primarily in their topology and in the number, capabilities, and placement of components.

- **TTA Ultimate (Fig. 3.1.1):** Provides the highest degree of fault isolation among the three implementations. It uses a bus topology in which each node is equipped with two independent, off-chip guardians, one for each replicated channel.

- **TTA Prototype (Fig. 3.1.2):** A cost-effective alternative that also employs a bus topology, but with a single on-chip guardian per node. This guardian is not fully independent of its host node: it has no direct access to the network and relies on the node to periodically synchronize its clock [127].

- **TTA Star (Fig. 3.1.3):** A compromise that mitigates the limitations of the Prototype design while remaining less costly than the Ultimate version. It uses a star topology with redundant central guardians located in the *star coupler*, which perform network access control for all nodes in the cluster [191].

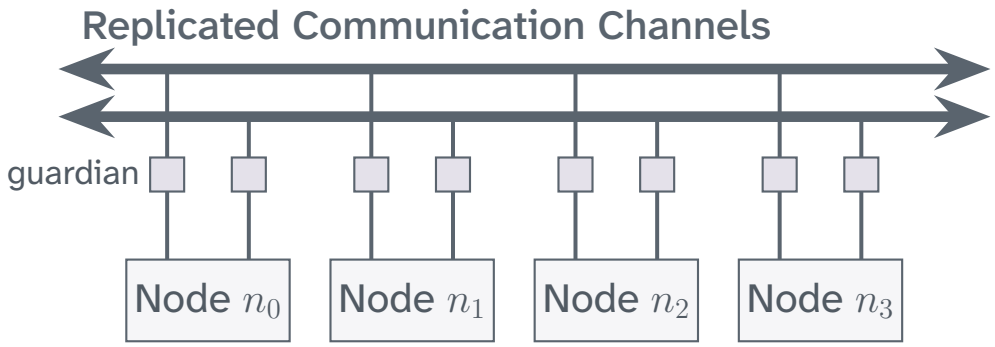


Figure 3.1.1: TTA Ultimate implementation.

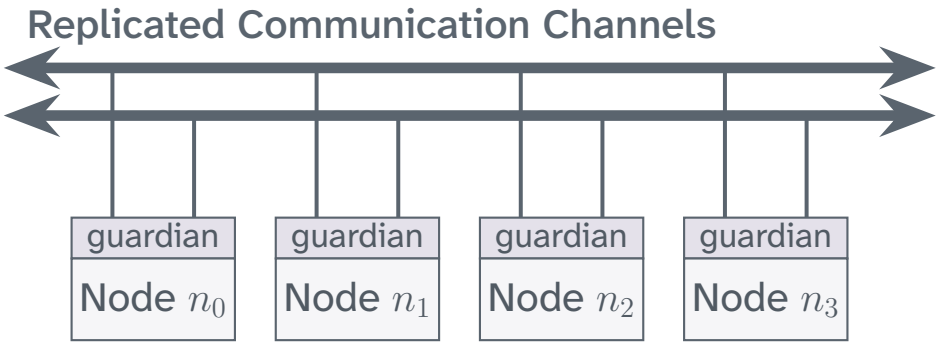


Figure 3.1.2: TTA Prototype implementation.

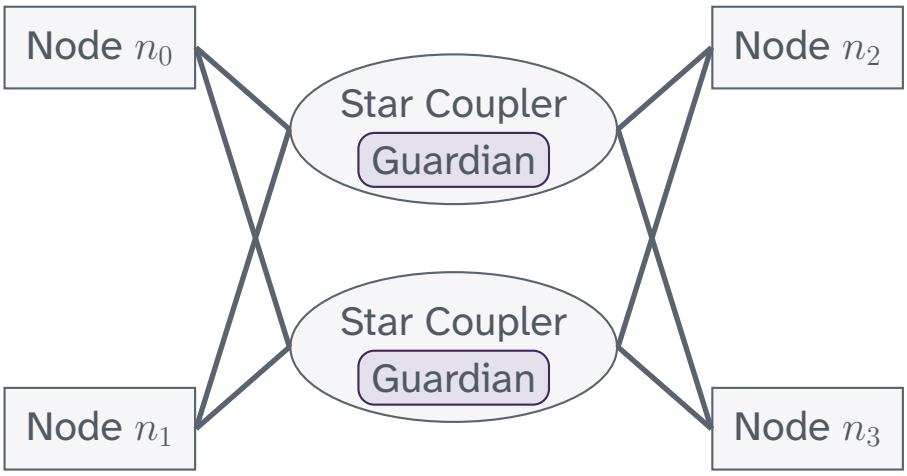


Figure 3.1.3: TTA Star implementation.

3.1.3 Cluster State

Regardless of the concrete realization, the cluster is global-time-driven, and the state of a TTA cluster is composed of the local states of its constituent nodes. According to [1], there are two types of local states:

1. **Initialization state (*i-state*)**: Represents a node's static data structures and code used during node initialization (e.g., host application and communication-controller schedules), which usually remain unchanged.

2. **History state (*h-state*)**: Represents a node's dynamic data structures (e.g., application-task state, local view of global time, operational state of other nodes) that evolve during system runtime.

The TTA identifies periodic instants where the $h\text{-}state$ is minimal (e.g., when no tasks are active and no messages are in transit) in order to facilitate node state resynchronization. The $h\text{-}state$ at this instant of maximum decoupling between past and future behavior is defined as the *ground state (g-state)* [1]. The interval between two successive global instants at which the state is the $g\text{-}state$ is called the *ground cycle*. In cyclic systems, the $g\text{-}state$ occurs naturally at the termination of one cycle and before the beginning of the next; thus, the ground cycle coincides with the cluster cycle (see Figure 3.1.4).

3.1.4 Time-Triggered Communication

A TTA cluster implements a *Time Division Multiple Access* (TDMA) strategy for message transmission, in which the capacity of the communication channels is shared among nodes by assigning their messages to unique, non-overlapping intervals of global time (*TDMA slots*). Assigning a message to a TDMA slot grants the corresponding node exclusive access to the communication channel during that slot. The communication schedule, which specifies the allocation of messages to TDMA slots, is stored in the communication controllers in a data structure known as the *Message Descriptor List* (MEDL). The MEDL represents a node's view of the temporal pattern of transmitted and received messages on the shared channel and is generated based on an offline scheduling algorithm during system design (see Section 2.1.6.6).

As part of the CNI specification, the MEDL contains information such as the global time of message transmission and reception, destination address within the CNI, message size, flow direction (send or receive), and other protocol-specific parameters. TDMA slots need not be of equal length, but their number is fixed within a *cluster communication cycle*. The cluster cycle consists of the ordered sequence of TDMA slots on

the channel, and consequently, the size of the MEDL depends on the number of slots in this cycle.

At the beginning of a TDMA slot, the sending communication controller fetches the corresponding message from its CNI and broadcasts it over the replicated channels. The receiving communication controller accepts only the message scheduled for the current slot according to its MEDL. As a result, only messages explicitly listed in the MEDL can be sent or received. The receiver infers both the sender identity and the message type from the TDMA slot itself, without requiring this information to be encoded in the message payload.

3.1.4.1 Communication Cycle Structure in TTP/C

In TT protocols such as TTP/C [20, 21] and the static segment of FlexRay [192], TDMA slots are typically assigned to nodes in a round-robin fashion, with each node receiving a fixed-size slot. For example, the communication cycle in TTP/C is organized into a hierarchy of temporal abstractions, as illustrated in Figure 3.1.4. Each node n_x is assigned a fixed interval of global time called TDMA slot $slot_x$, which gives n_x exclusive access to the communication channel to transmit one frame. The length of n_x's TDMA slot T_{slot_x} is chosen large enough to accommodate the largest frame broadcast by n_x. Here, $x \in \{0, \ldots, N-1\}$ is the unique ID of the node/slot in the cluster. A frame is a transmission of known length that carries both application and protocol data.

A sequence of N TDMA slots (one per node) forms a TDMA round $round$, the length of which is $T_{round} = \sum_{x=0}^{N-1} T_{slot_x}$. The phase of each node's TDMA slot within a round is fixed across all rounds, and the sequence of nodes within a round is determined at design time. A node that does not send frames in any slot is called *passive*. A node can be passive by design (no assigned slot) or becomes passive after diagnosing it as faulty. Slot multiplexing across rounds, assigning a slot to different nodes in different rounds, is possible by specifying such behavior in the MEDL.

The global communication schedule is formed by the sequence of R equally sized TDMA rounds containing all cluster messages, which is also referred to as the *cluster cycle* $ccycle$. The TDMA slot index (ranging from 0 to $N-1$) identifies the position of a slot within a given round, whereas the round slot index (ranging from 0 to $RN-1$) uniquely identifies the specific message instance transmitted within the $ccycle$. This distinction is necessary because a node is allowed to transmit different messages in its assigned TDMA slot across different rounds. Hence, the $ccycle$ determines the periodicity of communication between nodes (communication cycle) and has a length of $T_{ccycle} = R \times T_{round} = R \times \sum_{x=0}^{N-1} T_{slot_x}$

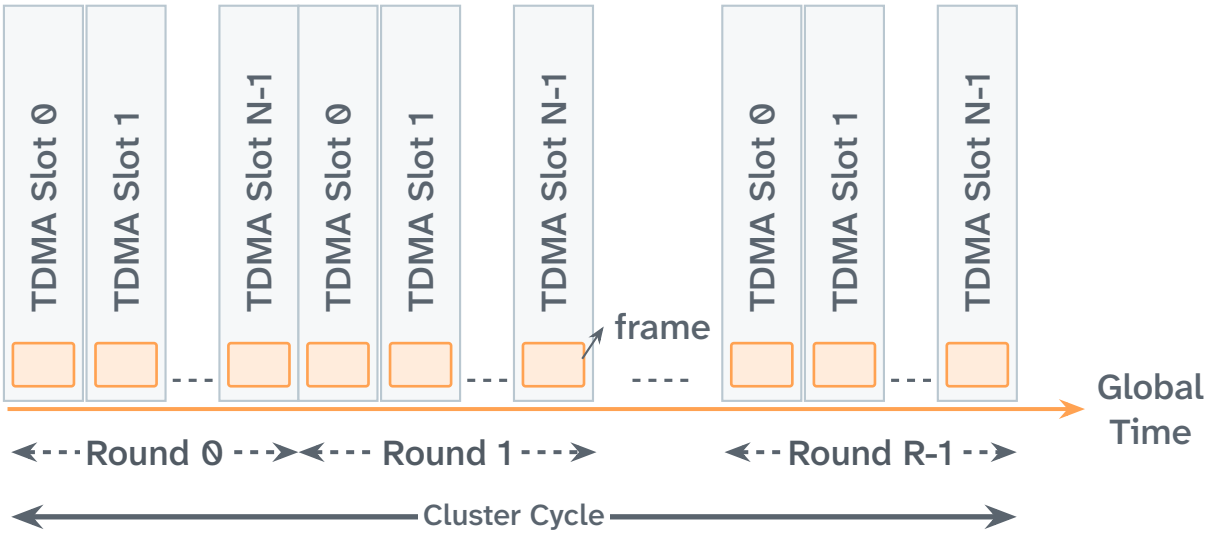


Figure 3.1.4: Example - Communication structure of TTP/C. The cluster cycle consists of R rounds, each containing N slots.

3.1.4.2 Transmission Window Alignment

A fundamental constraint in TDMA-based communication is that nodes are synchronized with a non-zero clock synchronization error, or precision π (see Section 2.1.5). As a result, different nodes have slightly different perceptions of when a TDMA slot begins and ends. Ensuring that all correct nodes interpret a transmission as belonging to the same slot requires incorporating this precision bound into the communication schedule.

As illustrated in Figure 3.1.5, two worst-case timing relationships must be considered: *Case 1*, where the sender's clock is the fastest and the receiver's is the slowest, and *Case 2*, where the roles are reversed. Although only one of these cases can occur at any time in a synchronized cluster, the communication schedule must accommodate both.

In *Case 1*, the sender must delay its transmission by at least π after its local slot start to ensure that the slowest receiver has already entered the same slot. In *Case 2*, the fastest receiver must keep its reception window open for an additional π beyond its perceived slot end to accommodate the slowest sender. Together, these constraints introduce an inter-slot gap of 2π (or $2\,MT$ of global time) between successive slots. This guarantees that transmissions from different nodes never overlap, regardless of individual clock drifts, thereby maintaining temporal isolation. When the architecture includes guardians, however, the inter-slot gap must be increased to 4π (or $4\,MT$), since guardians maintain their own notion of time that must be accounted for in addition to the sending and receiving nodes [191,193].

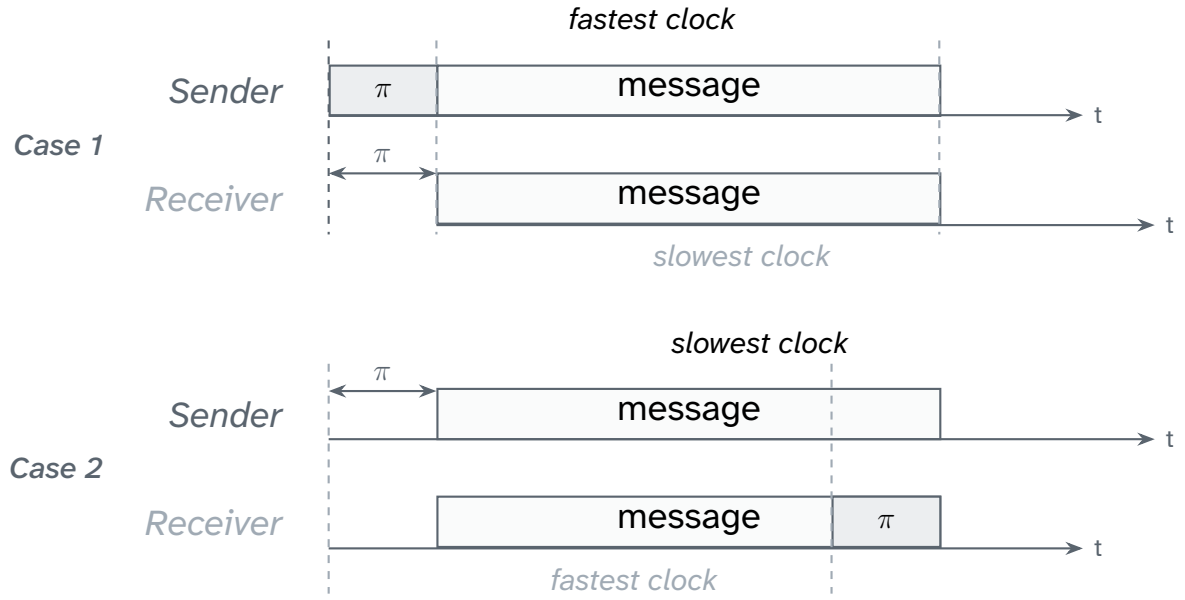


Figure 3.1.5: Transmission window alignment in TDMA-based communication. In order to ensure that a frame is consistently perceived within the correct slot boundaries, the system must account for the clock synchronization precision π. *Case 1* illustrates that transmission must be delayed by π to account for the a slowest receiver, whereas *Case 2* shows that reception must be extended by π to accommodate a slowest sender.

3.1.4.3 Message Acceptance Window

The process for accepting a received TT message and using its arrival time for clock correction is shown in Figure 3.1.6. Here, node n_x is the sender of message msg_x and node n_w is one of the receiving nodes. The planned message transmission or reception time at a given TDMA slot is called the action time (t_{at}), which is an agreed-upon global point in time stored in the MEDL. However, since clocks of nodes are approximately synchronized within precision π, n_x calculates a delayed action point ($t^x_{dat} = t^x_{at} + \pi$) for when it shall start transmitting msg_x such that all correct receivers are guaranteed to perceive the start of its TDMA slot (see Section 3.1.4.2). Similarly, n_w calculates a delayed action point ($t^w_{dat} = t^w_{at} + \pi + T^{est}_{COMM}$) for when it expects msg_x to arrive based on the clock precision and communication channel delay estimate (T^{est}_{COMM}). Node n_w additionally uses the calculated t^w_{dat} to define an acceptance window (T_{AW}) during which reception of msg_x should be considered valid from n_w's point of view. Since clocks drift in either direction, the minimum size for the acceptance window should be 2π. If the message is received within the acceptance window and used for clock correction, the estimated time difference (Δ^{est}_{DIFF}) between the expected t^w_{dat} and actual reception time (t^w_{rcv}) of msg_x is used by n_w to calculate a correction term for its clock. The exact correction term depends on the convergence function and is algorithm dependent.

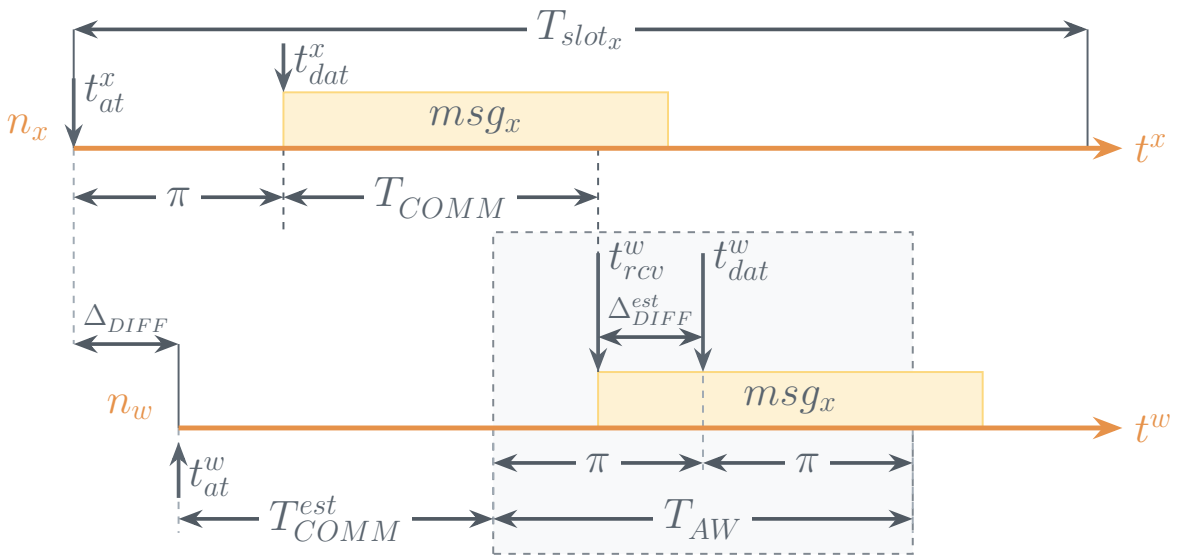


Figure 3.1.6: Acceptance window for message reception in TT communication.

Notice that the estimated time difference Δ^{est}_{DIFF} contains a reading error due to the uncertainty in network communication delays $\epsilon_{comm} = T^{max}_{COMM} - T^{min}_{COMM}$ and clock drift, which prevents clocks from converging closer than a certain bound (perfect clock synchronization is not attainable, see Section 2.1.5). Node n_w then applies the convergence function of the clock synchronization algorithm[1] to the time differences derived from message reception times on all communication channels. This is typically done using readings from multiple senders (e.g., from $3F+1$ nodes to tolerate readings from up to F arbitrary faulty nodes) in order to calculate the correction term for its local clock.

3.1.4.4 Slot Structure in TTP/C

Since it takes non-zero time to execute communication protocol services, frame transmissions are separated by a gap interval called the *Inter-Frame Gap* (IFG). The structure of a TDMA slot in TTP/C as an abstraction on top of global-time comprises, as shown in Figure 3.1.7, four phases:

1. Pre-Send Phase (PSP): where the slot owning node prepares its TTP/C frame for the transmission.

2. Transmission Phase (TP): where the actual transmission of the node's frame takes place.

3. Post Receive Phase (PRP): where the received frame gets evaluated, and the protocol services are executed.

[1] e.g., computing a fault-tolerant average by discarding the largest and smallest values.

4. Idle Phase ($IDLE$): used by the communication schedule designer to stretch the slot size and ultimately stretch the TDMA round size to match application requirements. During this phase, communication protocol execution is halted.

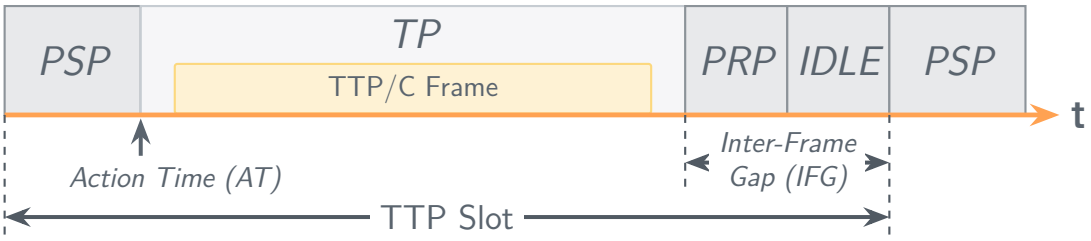


Figure 3.1.7: TTP/C slot structure.

3.1.5 Fault Hypothesis

The dependability of the TTA is based on a fault hypothesis that clearly defines FCRs to contain errors and prevent their propagation [131]. An FCR is a self-contained subsystem whose internal faults must remain contained; this assumption of independence relies on the absence of shared resources and on resilience against common-cause failures (see Section 2.2.1.4).

The fault hypothesis varies across the different TTA implementations for the TTP/C protocol. A comparative summary of these implementation-specific assumptions is provided in Table 3.1.1. As shown, all implementations follow a *single-fault hypothesis*, under which at most one FCR is assumed to fail at any given time. The distributed clock synchronization algorithm that establishes global time requires at least $3F + 1 = 4$ nodes and $F + 1 = 2$ broadcast channels to tolerate $F = 1$ arbitrary fault (see Section 2.2.4.3). It also requires sufficient temporal separation between successive faults, specifically at least two TDMA rounds, so that the cluster can reach agreement on nodes operational state before a subsequent fault occurs.

Since TTA is designed for closed systems, its fault hypothesis considers only accidental faults and does not account for intentional or malicious cyberattacks. Within this setting, the Star and Ultimate implementations assume that nodes may exhibit arbitrary (Byzantine) behavior.

In contrast, guardians and communication channels are restricted to more constrained failure modes. These components may exhibit inconsistent omission failures, such as delaying or discarding messages or delivering them only to a subset of nodes. However, they are assumed to be incapable of generating syntactically correct protocol messages due to the absence of CRC generation logic [191].

In a bus-based communication system, it is intuitive to expect that each message transmitted by a node is received consistently by all other nodes. However, perturbations

Table 3.1.1: TTA fault hypothesis for its alternative implementations.

Implementation	TTA Ultimate	TTA Star	TTA Prototype
FCRs	Each node Each guardian Each channel	Each node Each hub & channel	Each node & its guardian Each channel
Fault modes	Nodes: arbitrary Guardians & channels: inconsistent omission		Fail-silent
	Guardians and channels may not create syntactically correct protocol messages		
Number of faults	1		
	At least 4 nodes & both hubs/channels non-faulty to retain single fault hypothesis		
Arrival rate of faults	1 every 2 TDMA rounds		
Additional features	Resilience against spatial proximity faults		-
	Resilience against Slightly-Off-Specification (SoS) faults		

at the physical layer caused by SoS faults can lead to asymmetric reception, resulting in behavior that appears Byzantine (see Section 2.2.3).

The guardians in the Ultimate and Star implementations of TTA incorporate mechanisms that convert SoS faults into failures perceived symmetrically by all receivers. Coding faults are mitigated through active signal reshaping performed by the guardian, while cut-off faults are prevented by aborting transmissions that do not begin within a narrow window around the scheduled start time.

The Prototype implementation described in [190], which represents a commercialized product, adopts a more constrained failure mode for cost-reduction reasons, reducing both chip count and component complexity. In this design, the guardian is not fully independent of its associated node: it resides on the same chip, shares the power supply, and relies on the node's communication controller to maintain its view of global time and schedule progress [127]. As a consequence, Prototype guardians do not implement countermeasures against SoS faults, and TTA nodes in this implementation assume a fail-silent failure mode.

3.1.6 TTA Node

The node abstraction represents the basic building block of a TTA-based RT computer system. It is a self-contained computational unit that acts as an independent FCR (see Section 2.2.1.4) and provides a defined part of the real-time application service.

Each node contains its own software and hardware resources (CPU, memory, and I/O interfaces), including a local clock to track real-time progression. As shown in Figure 3.1.8, a TTA node comprises the following subsystems:

1. **Host computer** that executes the *Real-Time Operating System* (RTOS) and real-

time application tasks that deliver the intended service of the real-time node.

2. ***Communication Controller*** **(CC)** that implements the TT communication protocol and delivers a set of dependability services to host computers.

3. **Input/output subsystem** that manages the reception of data from sensors and its transmission to actuators.

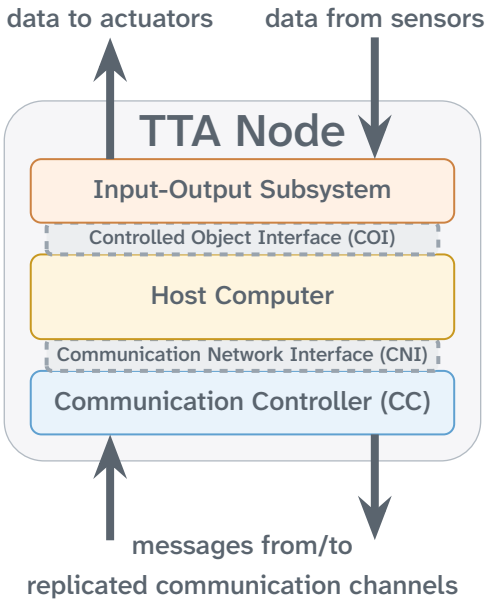


Figure 3.1.8: TTA node structure

3.1.6.1 Node Interfaces

The node structure permits system integration at a high level of abstraction, allowing each node to provide its services through well-defined message-based interfaces in the time and value domains, while encapsulating its internal implementation details behind these interfaces. Since no low-level signals such as clock, address, or interrupt lines are employed, the cognitive effort required to design and understand system behavior is significantly reduced.

TTA specifies the service provided by a node as a periodic sequence of timed state messages based on *global physical time*, an approach reminiscent of synchronous languages that model execution as streams of signals over *logical time* [148]. A TTA node comprises two types of interfaces:

1. **Operational interfaces** through which the node's services are provided:
   - *Communication Network Interface* (CNI): Also referred to as the *Linking Interface* (LIF), this interface connects the host computer to the communication controller through a shared memory region that is fully specified in both

the time and value domains. The CNI is *technology-agnostic*, permitting changes to the host computer implementation (e.g., CPUs, FPGAs, ASICs) without affecting interacting components. By mapping periodic state messages into this memory region, the TTA provides a logical abstraction of a globally shared memory with time-deterministic updates. In this model, the TT network protocol acts as a deterministic transport subsystem that synchronizes the distributed memory views and effectively decouples individual node computation from global communication.

The operational specification defines the syntactic and temporal properties of messages (names, structure, sequence), while the meta-level specification defines their semantic content.

- *Controlled Object-Interface* (COI): This interface connects the host computer to other TTA clusters or controlled objects via sensors or actuators. Nodes with COI are designated gateway nodes; those without are closed nodes.

2. **Control interfaces** for node configuration, debugging, and maintenance:
   - *Configuration and planning interfaces*: These interfaces support host-computer resource management (port naming, task assignment, start/restart) and communication controller schedule reconfiguration. Host computers cannot reconfigure communication controllers to prevent error propagation.
   - *Maintenance and debugging interface*: This interface provides platform-specific access to node internals for implementers with detailed implementation knowledge, and is distinct from the operational interfaces used by service consumers.

3.1.6.2 Host Computer Implementation

In a purely TT implementation of the host computer, the RTOS on each node is equipped with a data structure representing the host computer's scheduling table, also referred to as the task descriptor list (TADL). The TADL for every system node is calculated during design time using an offline scheduling algorithm as described in Section 2.1.6.6.

At runtime, the RTOS runtime scheduler on the node is triggered periodically with fixed granularity; that is, the time-trigger events that invoke the scheduler occur at equidistant points in time. This granule has been given different names in the literature, such as basic cycle time [194], macrotick [123], and time-slot [122]. Since the term *macrotick* is also used to denote the granularity of global time [1], and the term *time-slot* is too generic, we take the liberty of referring to the invocation granularity of

the runtime scheduler as the *invocation gap*. Additionally, we refer to the invocation time of the TT scheduler in response to a time-trigger as *scheduler point*.

The TADL holds a sequence of time-triggers, each responsible for initiating or terminating the execution of a task. The distance between two successive time-triggers in the TADL consists of one or more contiguous invocation gaps, which we refer to as a *computation slot*. The runtime scheduler executes the TADL periodically; at the beginning of each invocation gap it runs the task associated with that position in the TADL.

Task execution may be preempted depending on the size of the invocation gap and the placement of time-triggers in the TADL. For example, if the invocation gap is smaller than the WCET of a task, that task will be preempted by the scheduler. Furthermore, a task can be scheduled across a sequence of discontinuous computation slots.

Choosing the invocation gap to be equal to the largest WCET of all tasks allocated to the node ensures that no task is preempted. However, this approach underutilizes computational resources and may adversely affect schedulability. The size of the invocation gap therefore represents a lower bound on the responsiveness of the TT system. It is more efficient to size the invocation gap according to application needs and the capabilities of the execution platform. The system designer must ensure that the invocation gap is not so small that context-switching overhead dominates useful work, and not so large that system responsiveness is unnecessarily degraded. Experiments on the TMS570 platform in [123] show that the scheduling overhead becomes negligible with invocation gaps larger than $100\ \mu$s.

Note that TTA does not restrict the host computer to follow the offline scheduling approach. Instead, online scheduling can be used. However, replica determinism and temporal constraints must be respected in order to retain TTA's extra-functional properties. For example, a loss of replica determinism can occur if replicas interface with the CNI in an asynchronous manner, since different replicas may use different input messages and progress through different trajectories [195]. Replica determinism will be discussed in more detail in Section 3.2.

3.1.7 TTA's Real-Time and Dependability Services

The real-time and dependability-oriented services provided by the TTA include:

1. A **sparse global time base** to address the simultaneity problem and ensure a consistent temporal ordering of events across nodes [152].

2. **temporal firewall interfaces** that shield application tasks from interrupt showers [196], guardians that prevent untimely node accesses to the communication channels [127,191].

3. A **membership service** that maintains a consistent group membership view and reports node failures within a bounded time interval [197,198].

4. A **transparent implementation of fault tolerance** in which replica synchronization and fault masking are hidden from host computers [199].

5. An **atomic multicast service** that ensures reliable message delivery with a consistent total order across all receivers [19,191].

6. A **node integration service** that enables nodes to join an already operating and synchronized cluster [21].

7. A **mode change service** that supports timely switching among a set of offline-defined schedules to adapt to changing environmental conditions [200].

These architectural services form an intricate web of interdependencies that collectively realize the system's extra-functional properties. Figure 3.1.9 illustrates these relationships, with each arrow indicating the direction of dependency. By establishing how these services operate and rely on one another, we can subsequently map them to the extra-functional properties defined in Section 1.1. In Section 3.2.2, we will present a dependency classification framework that isolates which of these TTA services are fundamental for achieving the properties, and which represent implementation-specific design choices that can be adapted for greater flexibility.

In the remainder of this section, we examine each TTA service in detail, considering its functionality and dependencies.

3.1.7.1 Global Time

As established in Section 2.1.5, real-time systems adopt a Newtonian model of time in which physical time progresses as a dense sequence of instants measured in physical seconds. TTA-based systems follow this same model. Each node maintains a local digital physical clock that timestamps events and provides a basis for reasoning about temporal order. These clocks advance in discrete *microticks* (mt) and, due to manufacturing variability and environmental influences, inevitably drift apart unless periodically synchronized. Even after synchronization, a residual offset remains, since perfect synchronization of distributed clocks is impossible.

The TTA employs a fault-tolerant clock-synchronization algorithm [201] that periodically adjusts local clocks so that their readings remain within a bounded precision π. On top of this ensemble of approximately synchronized clocks, the TTA constructs a global time base that advances in units called *macroticks* (MT). As required by the reasonableness condition introduced in Section 2.1.5, the macrotick duration must satisfy

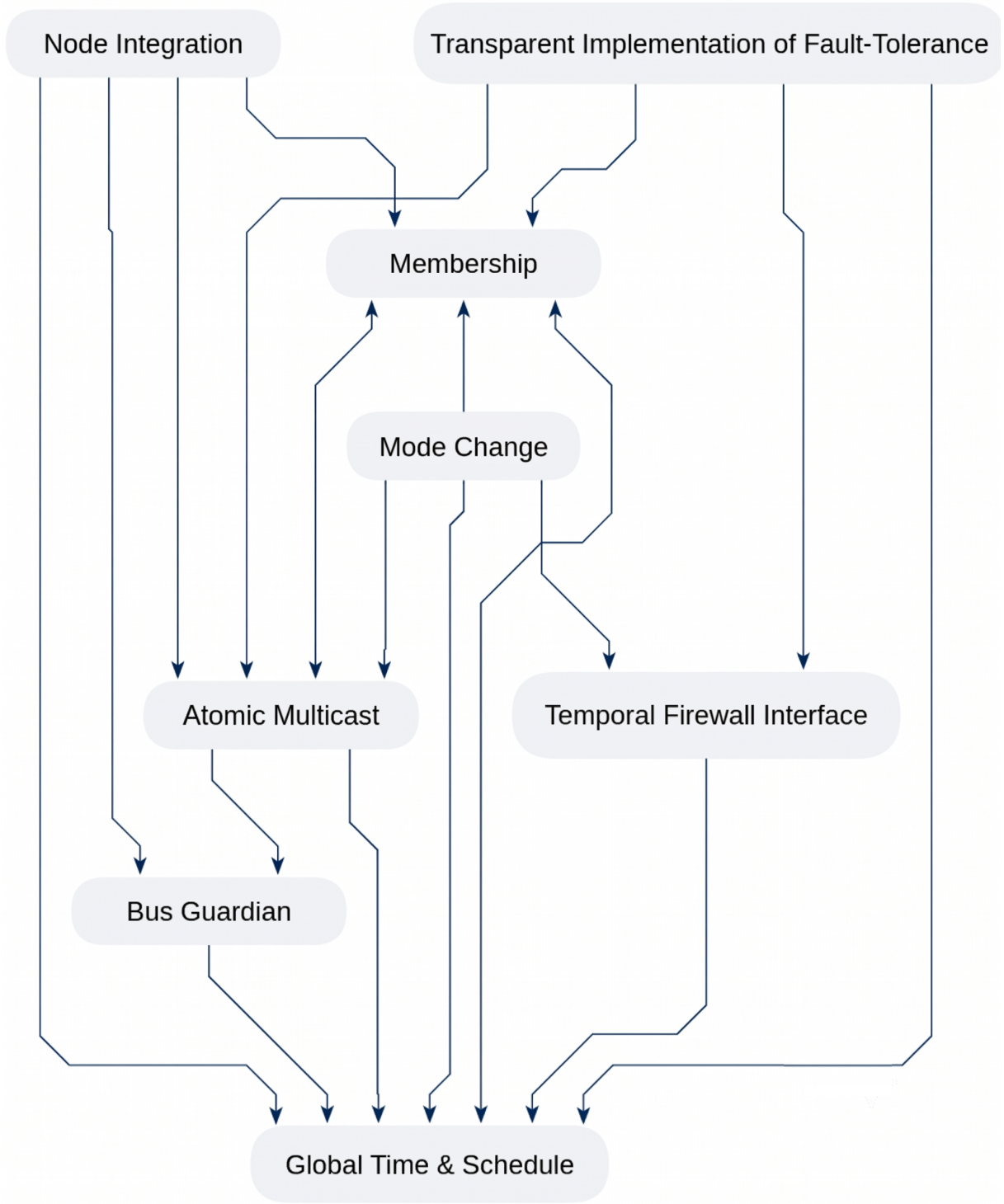


Figure 3.1.9: Dependencies between TTA services, where arrow direction represents use relationship.

$|MT| > \pi$. This ensures that two correct nodes observing the same event will assign timestamps that differ by at most one macrotick. The global time base thus provides a system-wide temporal reference that is anchored in physical time yet implementable on imperfect hardware clocks.

3.1.7.1.1 Sparse Global Time

Imperfect clock synchronization and the digitization error of digital physical clocks give rise to the four FLTM described in Section 2.2.4.1. These limits prevent a distributed system from reliably establishing the temporal order of events that occur within the same macrotick of the global time base or from timestamping events with perfect fidelity to their actual temporal order.

In order to mitigate these limitations, the TTA adopts the concept of a *sparse time base* to simulate perfectly synchronized clocks [152]. Sparse time divides global time into an infinite sequence of *activity intervals* and *silence intervals*. Events occurring within the same activity interval are treated as concurrent (not temporally ordered), whereas events separated by a silence interval are considered successive (temporally ordered). Instead of allowing events to occur at arbitrary microticks on the dense physical timeline, the sparse-time concept restricts their occurrence to activity intervals. The intervening silence interval absorbs the residual timing uncertainty imposed by the FLTM (Section 2.2.4.1) and ensures that nodes observe a consistent temporal order of events without executing an online agreement protocol, which would otherwise incur significant processing, communication, and latency overhead.

However, only events within the sphere of control of the computer system can be restricted to activity intervals. External events from the environment occur on a dense timeline and must therefore be *internalized* using an online agreement protocol that assigns each such event to an activity interval of sparse time. The resulting order is consistent across nodes but may not correspond to the actual temporal order of occurrence.

Recalling FLTM (IV) from Section 2.2.4.1, temporal order can theoretically *always* be recovered if events are $0/3MT$ precedent. However, because it is physically impossible to confine concurrent events to a true $0\,MT$ interval, the TTA expands the activity interval to $1\,MT$ to absorb the non-zero synchronization error (π). For events originating within the same synchronized cluster, a $3\,MT$ silence interval suffices. However, the TTA can even establish the temporal order of events across *non-synchronized* TTA clusters, without executing a costly online agreement protocol, by extending the silence interval to at least $4\,MT$ [1].

For example, if two events are generated at the same cluster-wide macrotick in one cluster, an observing cluster may timestamp them with $2\,MT$ difference due to syn-

chronization error. Similarly, if the two events are generated $3\,MT$ apart, they may also appear $2\,MT$ apart to the observer, making the two cases indistinguishable. By enforcing a $4\,MT$ silence interval, the system guarantees that successive events exhibit a timestamp difference of at least $3\,MT$, whereas concurrent events remain within $2\,MT$ or less. This separation enables the observer cluster to recover the intended temporal order without any runtime communication [1].

Figure 3.1.10 illustrates the relationship between local time, global time, and a $1\,MT/4\,MT$ sparse global time base. In practical implementations, the silence interval is extended so that each activity–silence pair forms a complete computation or TDMA slot. The activity interval contains the time-trigger that initiates the scheduled action, while the silence interval is dimensioned to accommodate the execution or transmission demands of the associated task or message, while inherently satisfying the minimum $4\,MT$ gap requirement.

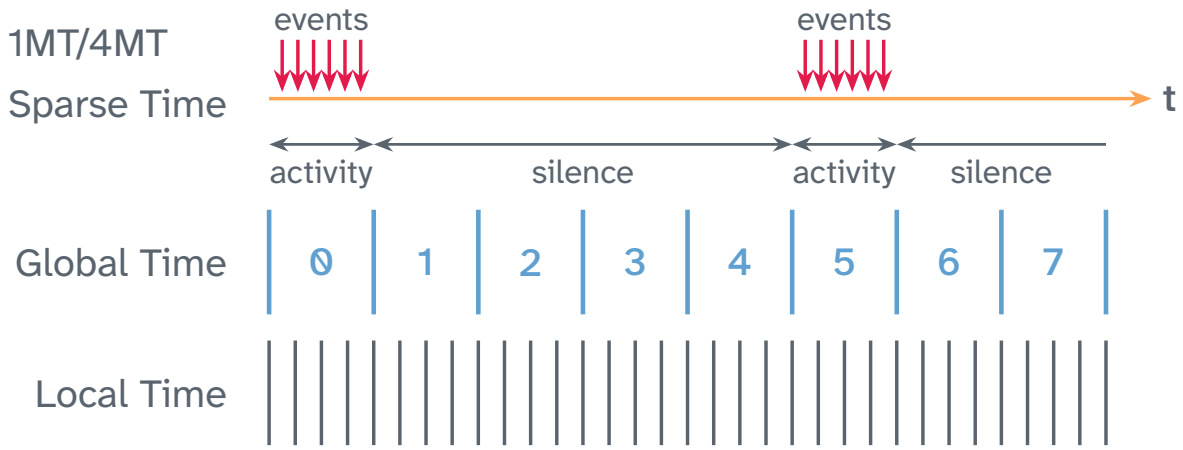


Figure 3.1.10: $1\,MT/4\,MT$ sparse time base.

3.1.7.1.2 Cyclic Representation of Global Time

The global time service forms the cornerstone of TTA's temporal architecture, providing a system-wide temporal reference that enables coordinated behavior across distributed nodes. In TTA-based systems, time triggers initiate the same activities periodically across the system. Consequently, these systems adopt a cyclic representation of global time by partitioning it into uniformly sized cycles (e.g., the cluster cycle in Section 3.1.4), where each instant is characterized by its *phase*, i.e., its angular deviation from the beginning of the cycle.

3.1.7.2 Membership Service

As outlined in Section 2.2.6.2, a group membership service maintains a consistent, system-wide view of which nodes are currently considered operational. In the TTA, this service provides timely and consistent information about the status of nodes in a TTA cluster [198, 202], which is essential for safety-critical applications (e.g., compensating for the loss of a wheel brake in an automotive system).

The TTA membership service operates through two recurrent and concurrent steps. The first step is *failure detection*, performed at every TDMA slot. A node's operational status is inferred from the reception of valid messages during its assigned TDMA slot. At each receiving node, the communication controller evaluates the messages received over all replicated channels and classifies the sender as correct only if the message satisfies the following conditions:

1. **Valid syntax**: The message conforms to the protocol's physical-layer format and encoding.

2. **Valid timing**: The message is received in a timely manner within the sender's allocated TDMA slot.

3. **Integrity**: The message passes a CRC check, ensuring that the embedded *communication controller state* ($CC\text{-}state$), which encodes the sender's view of global time, operational mode, and membership, matches the receiver's view. The $CC\text{-}state$ is either sent explicitly within the message to enable node integration or implicitly included in the CRC to save network bandwidth.

In the second step, the membership service achieves consistency by reporting failure information across the cluster. After evaluating the sender's message, each node updates its local $CC\text{-}state$ to reflect the observed status of the sender during its TDMA slot. This updated state is then included in the node's own subsequent transmission. Hence, the reception of a frame indicates agreement on the $CC\text{-}state$. Because every node broadcasts its view of the membership once per TDMA round, the status of a failing node is disseminated and agreed upon by all operational nodes within a single round. This cyclic exchange ensures that the membership view converges consistently across all correct nodes. The time required for this convergence defines the latency of the membership service, and consequently dictates how quickly the system can react to node failures, which is essential for safety-critical applications.

There exists a cyclic dependency between global time and the membership service (see Figure 3.1.9). The membership service depends on global time and schedule because the receiving node must determine whether the sender's message arrived within the expected TDMA slot, which requires the clock difference between nodes to remain

within the assumed precision. Conversely, the global time service depends on membership, since the clock synchronization algorithm only incorporates timing information from nodes deemed operational in the current membership view. This mutual dependency highlights the tight integration of temporal coordination and fault management in the TTA.

3.1.7.2.1 Membership View Structure and Communication

The membership view is implemented as a bit vector, where each node in the cluster is assigned one bit to indicate its operational state. The sending time of a message is called the *membership point* of its sending node and is defined in the MEDL, which allows receivers to interpret the arrival of the message as a life sign signal from the sending node. The receivers update their membership vectors during the PRP phase of every TDMA slot (see Section 3.1.4.4). If no message is received within the scheduled time, receiving nodes reset the membership bit of the sending node in their local membership vector. Hence, receiving messages at two consecutive membership points of the sender (e.g., in two consecutive TDMA rounds) suffice to establish the membership of the sender and conclude that the sender was alive in between these membership points, assuming that a failing node does not recover within this interval.

There are two ways for a node to communicate its membership view during its TDMA slot:

1. **Implicitly**: by encoding the membership vector within the extended CRC of communication protocol messages. The associated frames in TTP/C are called n-frames.

2. **Explicitly**: by sending the membership vector as part of the message to enable node integration. The associated frames in TTP/C are called i-frames (only contain the $CC\text{-}state$) or x-frames (contain $CC\text{-}state$ and application data).

While the implicit approach conserves data bandwidth, it does not support node rejoining within the TTA cluster. For nodes to reintegrate within the cluster they need to synchronize their $CC\text{-}state$ to the current view by member nodes in the TTA cluster. This clarifies the dependency of the node integration service on the membership in Figure 3.1.9.

3.1.7.2.2 Membership Fault Assumptions and Safety Properties

TTA's membership service is designed under benign fault assumptions, where faults are consistently detected by correct nodes. The service relies on the self-confidence principle, where each node assumes its own correctness unless it receives contradictory evidence at two consecutive membership points. Specifically, a node evaluates its

status based on the acknowledgments of its successors (the nodes scheduled to transmit immediately after it in the TDMA round). If the first successor fails to acknowledge the node's participation within its transmitted $CC\text{-}state$, the node interprets this as a potential local reception fault. If the second successor also fails to acknowledge the node, it concludes that it has experienced a sending fault. In such an event, the node defects from the current group membership and adopts the membership view broadcast by its successors to ensure cluster-wide agreement.

The membership service has the following safety properties:

- **Agreement**: the membership vector is identical at all correct nodes.
- **Validity**: there is at most one faulty node in the membership vector at any point in time which is classified as correct (because it takes some time until a node is diagnosed as faulty).

Hence, a maximum rate of failure occurrence is assumed to be one failure every two communication rounds.

3.1.7.2.3 Membership-Triggered Mechanisms

Clique avoidance and resolution. Although guardians mitigate inconsistent message receptions arising from accidental SoS faults at the physical layer (see Section 3.1.5), they do not provide full coverage, and the remaining inconsistencies fall outside the primary fault hypothesis. An inconsistent reception can lead to the formation of disjoint cliques that do not communicate with each other and thus inter-clique clock synchronization is not possible. This implies that the cliques will have different view about the global time and may corrupt the transmissions of each other due to the overlapping of TDMA slots. As a countermeasure, clique avoidance and resolution algorithms can be built on top of the membership service [153,198]. These algorithms detect the formation of disjoint cliques by identifying nodes with conflicting membership vectors and initiate a resolution strategy based on the CAP theorem's fundamental trade-offs (see Section 2.2.4.4). In the face of a network partition (P), the TTA cluster explicitly prioritizes Consistency (C) over Availability (A): these algorithms employ a majority rule that forces nodes in the minority clique to cease transmission within two TDMA rounds and transition to a passive, fail-silent state. This works by having each node maintain two counters called *accept* and *reject*, which starting from the node's TDMA slot, respectively count the number of frames accepted and rejected within a TDMA round. During the next PSP of a node slot (see Section 3.1.4.4), a node freezes if the *accept* counter is smaller than the *reject* counter. By sacrificing the availability of the minority nodes, the system guarantees a globally consistent membership view, thereby ensuring that

the remaining majority can continue safe and predictable operation, while the ones in the minority clique can rejoin using the integration service (see Section 3.1.7.6).

Communication blackout. The membership service is used to trigger a communication blackout mechanism at all nodes. This happens if no valid messages are received within the last TDMA round, e.g., due to a massive transient fault. The communication blackout mechanism restarts all nodes in the cluster in an attempt to re-establish the synchronous operation with at least the minimum required service [21].

3.1.7.3 Temporal Firewall Interface

The CNI is a memory abstraction that connects the host computer to the communication controller and separates local information processing from global communication within the TTA cluster [1,196] (see Section 3.1.6.1). The CNI is the most important interface in the TTA, providing a well-defined boundary that encapsulates the implementation details of the host computer. It supports the autonomous operation of the communication system and prevents nodes from exerting unpredictable temporal influence over one another. For this reason, the CNI is referred to as the *temporal firewall interface* within the architecture.

3.1.7.3.1 Communication Model

Nodes exchange information in two domains, data and control. Depending on the direction of control flow relative to data flow, communication models can be classified as either *information push* or *information pull* [1,6]. In the information-push model, data and control flow are in the same direction: the sender initiates transmission requests and thus determines when receivers must process incoming information. This gives the sender temporal influence over the receiver. Moreover, if the communication system is busy, outgoing requests must be buffered; without bounds on the sender's rate, buffer dimensioning becomes impossible and receiver overload is unavoidable.

In the information-pull model, data and control flow are in opposite directions: the receiver initiates transfer requests to the sender, as in the classical client–server model [203]. Incoming messages are buffered in an intermediate storage and do not interrupt the receiver, allowing it to decide when to fetch data. This protects the receiver from interrupt storms and preserves its temporal autonomy. Buffer dimensioning in this model requires bounds on message arrival rates and service times at the receiver to avoid buffer overflow.

As illustrated in Figure 3.1.11, the CNI implements a *hybrid push–pull communication model*. The host computer follows a push model when providing its services: it deposits state messages into the CNI. Conversely, it follows a pull model when consuming

information: it retrieves state messages from the CNI at times of its choosing. The communication controller bridges these two domains autonomously by executing the MEDL according to the progression of global time. At the sending node, it fetches and transmits state messages from the CNI; at the receiving node, it accepts valid incoming messages and writes them into the CNI. The global-time points at which these actions occur are predefined in the MEDL, which explains the dependency of the temporal firewall interface on global time and the schedule in Figure 3.1.9.

This hybrid push–pull model achieves a temporal decoupling that neither model provides in isolation. It offers *host-computer-to-network shielding* by preventing the host computer from influencing the actual transmission times and rates of messages, thereby avoiding network overload. Simultaneously, it provides *network-to-host-computer shielding* by preventing incoming messages from interrupting the host, eliminating the risk of interrupt showers and allowing the host to retain full control over its temporal behavior.

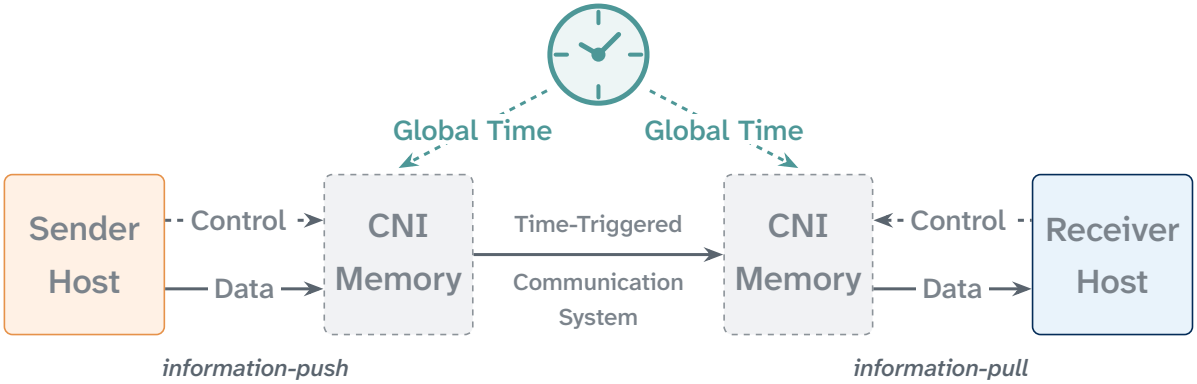


Figure 3.1.11: TTA's temporal firewall interface.

3.1.7.3.2 Flow Control Semantics

An interface is called *composite* when the receiver can control the production rate of the sender through explicit flow control mechanisms (e.g., acknowledgments or a direct physical link). Such back-pressure allows the receiver to prevent overload by exerting temporal control over the sender. However, this bidirectional dependency also enables error propagation from the receiver back to the sender.

In contrast, the CNI employs an *elementary interface* that relies on implicit flow control based on the a priori knowledge of message sending times and rates. This eliminates temporal dependencies between senders and receivers, allowing the correctness of a sender to be reasoned about independently of receiver behavior. Moreover, because no control signals cross the CNI boundary, the CNI acts as a temporal firewall that prevents control-flow errors from propagating from the network to host computers.

3.1.7.3.3 Message Read and Write Semantics

Messages in the CNI follow an update-in-place semantic: new state messages simply overwrite older ones and must be refreshed periodically to ensure that their temporal accuracy interval is not exceeded (see Section 2.1.2). Reading from the CNI is non-consuming, i.e., state messages remain stored in the CNI memory until overwritten.

State messages reinforce sender–receiver independence by preventing the receiver from exerting back-pressure on the sender to synchronize the flow of information. Because they carry absolute rather than relative values of RT entities, state messages are idempotent: receiving multiple copies has the same effect as receiving one. This greatly simplifies fault-tolerance mechanisms such as message replication.

3.1.7.3.4 Access Synchronization

Access by the host computer to the shared CNI may be either synchronous or asynchronous with respect to the communication controller's access times as defined in the MEDL. In *synchronous interfacing*, the host computer accesses the CNI only at times that avoid conflicts with the communication controller, which typically requires synchronizing the host computer to global time and co-designing the TADL and MEDL accordingly. This approach restricts the temporal autonomy of the host but enables tight control over message lifespan and facilitates extra-functional properties such as temporal predictability and replica determinism [204].

In *asynchronous interfacing*, the host computer operates independently of the communication controller's access schedule and retains full control over its own execution pace. Ensuring correct and exclusive access to the shared CNI requires implementing a shared-memory access mechanism such as *Non-Blocking Write* (NBW) or *Rate-Bounded Non-Blocking Write* (RBNBW). These mechanisms maintain separate reader and writer copies so that one buffer can be read while the other is being updated. This approach increases message lifespan, since both transmission and reception may be delayed by up to one message period, and it complicates replica determinism (see Section 2.2.6.3), because ensuring consistent inputs across replicas becomes more challenging [195,204].

3.1.7.4 Guardians

Conflict-free access to the communication channels can be ensured by following the TDMA-based communication schedule defined in the MEDL and driven by global time progression. However, if nodes cannot be guaranteed to fail silently, for example, if they may transmit outside their allocated TDMA slots and thereby cause collisions or corrupt messages, additional mechanisms are required to convert timing failures into fail-silent behavior (see Section 2.2.3).

In the TTA, this mechanism is implemented in a hardware component called the *guardian*. The guardian enforces compliant network access according to the MEDL and prevents nodes from transmitting outside their assigned TDMA slots. Achieving fault independence from the node it supervises requires the guardian to maintain its own notion and understanding of the TT network protocol, including the MEDL, global time, channel coding, and protocol message structure. This explains the dependency on global time and schedule, as shown in Figure 3.1.9.

According to the fault hypothesis of the TTA (see Section 3.1.5), guardians follow an inconsistent omission failure mode, where they are assumed incapable of generating syntactically correct protocol messages. The assumption is enforced architecturally by not providing guardians with the means to construct valid protocol messages. Since at most one guardian may be faulty at any given time, a faulty guardian cannot compromise the consistency of transmitted data or the communication system, as the guardian supervising the other communication channel remains correct and masks such faulty behavior [191].

3.1.7.4.1 Central and Distributed Guardians

Different guardian solutions exist depending on the network topology. In star-topology networks (e.g., TTA Star implementation), a *central guardian* located at the star coupler controls the access of all connected nodes [23, 191, 192]. In contrast, bus-based networks require a *distributed guardian* solution, where each node is equipped with a local guardian for every communication channel (e.g., TTA Ultimate implementation) [190]. This approach is more costly but provides strong fault isolation.

Cost-oriented guardian designs reduce complexity at the expense of fault-tolerance guarantees. An example is the local guardian used in the TTA prototype implementation [127, 190]. To reduce hardware cost, the guardian resides on the same chip as the node, supervises access to both communication channels, and relies on the node to synchronize its clock. Chapter 5 analyzes this vulnerability and presents enhanced guardian designs that maintain correct operation even in the presence of Byzantine faults.

3.1.7.5 Atomic Multicast Service

The existence of an atomic multicast primitive significantly simplifies the design of distributed applications by ensuring that all correct nodes operate on the same data (see Section 2.2.6.1). The difficulty of realizing such a primitive hinges on the types of faults that must be tolerated by the system. In general, $3F + 1$ nodes, $2F + 1$ communication links, or $F + 1$ broadcast channels and $F + 1$ rounds of message exchange are required

to tolerate F arbitrary faulty nodes that may equivocate and thereby cause asymmetric reception at different receivers [205]. Leaving this problem to the application designer makes application development more cumbersome and undermines the purpose of a dependable architecture. Conversely, solving the interactive consistency problem at the architectural level increases complexity and cost. An alternative is to eliminate the failure mode with reasonably high coverage. Following this third approach, TTA ensures interactive-consistent messaging based on two methods:

1. **Fault tolerance features in central guardians.** In the star implementation of TTA, the central guardians implement additional fault-containment mechanisms to isolate a node's arbitrary failure and ensure that all correct nodes receive consistent data from a faulty node [191]. Fault containment is performed at both the physical interface and the semantic interface with the supervised nodes.

   At the physical interface, in addition to restricting write access to the communication channels according to the MEDL, the guardian is responsible for masking SoS faults (see Section 3.1.5). It performs active signal reshaping using its own power supply by decoding the electrical signal of a message and re-encoding it, thereby masking coding faults and ensuring consistent reception. The guardian also masks cut-off faults by aborting a sender's transmission if it does not begin within a start window of length $(\pm\pi)$ around the scheduled transmission time (from the guardian's perspective), where π denotes the clock-synchronization precision.

   At the semantic interface, the guardian performs semantic checks on integration messages received from the node and aborts their transmission if they are not consistent with its view of the communication system state (i.e., the $CC\text{-}state$), resulting in fail-silent node behavior. This mechanism, however, comes at the cost of channel bandwidth and is reflected in the guardian's performance.

   The constrained failure mode of the guardian limits its faulty behavior to either corrupting a correct transmission or creating an invalid transmission on the channel. Since communication channels and guardians are replicated and the system operates under a single-fault hypothesis, other nodes can still receive a valid message (if any) through the non-faulty guardian. Hence, system-wide consistency is preserved, reflecting the dependency edge from atomic multicast to guardians in Figure 3.1.9.

2. **Clique avoidance and cyclic redundancy checks.** A weaker form of interactive consistency is provided by the combination of CRC checks and the *clique avoidance* mechanism of the membership service (see Section 3.1.7.2). Nodes drop messages that fail the CRC check (e.g., messages corrupted by bit flips or have disagreeing $CC\text{-}state$). An asymmetric reception partitions the cluster into two

cliques: in one clique, nodes receive a valid message, whereas in the other the message is corrupted and fails the CRC check. Acknowledgement messages cannot resolve this inconsistency, as they may also be lost or received asymmetrically. Instead, TTA uses unidirectional message passing with implicit flow control, i.e., no explicit acknowledgements are sent. The receiver knows the sender's message production rate and expected arrival times and is therefore responsible for detecting and reporting noncompliance using the membership service. This behavior is equivalent to consistent acknowledgement.

The cyclic dependency between membership and atomic multicast in Figure 3.1.9 arises because membership relies on atomic multicast to disseminate messages, while atomic multicast depends on membership to consistently report failures. If the fault hypothesis is violated, the clique avoidance algorithm ensures the survivability of the majority clique while causing minority cliques to freeze and shut down. As mentioned earlier, the clique avoidance algorithm sacrifices validity for agreement under such violations; consequently, a non-faulty node may be excluded from the membership but can rejoin in the next TDMA round.

The order property of atomic multicast is achieved trivially on a per-channel basis through the TDMA schedule defined in the MEDL and the use of global time, reflecting the dependency relation in Figure 3.1.9. All correct nodes receive messages on a communication channel in the same relative order, as dictated by the sequence of TDMA slots assigned to that channel.

3.1.7.6 Node Integration

Transient faults are the most common faults that systems experience. They occur sporadically and temporarily, causing state corruptions without inflicting hardware damage. Correcting the state of faulty nodes and reintegrating them into the system within a bounded amount of time is crucial for re-establishing the provided service.

3.1.7.6.1 Identifying Suitable Integration Time

A suitable integration instant must be defined at which it is simplest to synchronize the state of a rejoining node with its environment, i.e., the other TTA cluster nodes and the controlled object. Rolling back to a past checkpoint is generally discouraged in real-time systems, as the checkpoint may have been invalidated by the progression of real time (see Section 2.2.5.2). Instead, dynamic reintegration requires identifying periodic instants in system behavior at which the node's history state is small and contains only a small set of application variables, i.e., corresponds to the $g\text{-}state$ (see Section 3.1.3). Since the TTA specifies system behavior in global time and in a cyclic manner, the

periodic integration points are straightforward to identify (at the end of the cluster cycle), which explains the dependency of this service on global time and the schedule (see Figure 3.1.9).

3.1.7.6.2 Node State Resynchronization

A node that has been excluded from the membership service must resynchronize its state with the other nodes in the cluster. For this purpose, a set of predetermined nodes are assigned dedicated TDMA slots in the MEDL to explicitly transmit their $CC\text{-}state$ view of the communication system using the atomic multicast service (see Figure 3.1.9). The integrating node enters listen mode and observes the communication channels for a specified interval that is guaranteed to include at least one integration message. In the TTP protocol [20, 21], this interval spans two TDMA rounds plus the node start-up timeout (the number of slots preceding its own TDMA slot in the MEDL). Therefore, it must be ensured that enough frames that explicitly carry $CC\text{-}state$ (i.e., $i\text{-}frames$ or $x\text{-}frames$) are transmitted during the cluster cycle.

Once an integration message is received, the integrating node adjusts its $CC\text{-}state$ accordingly and executes the following mechanism to avoid becoming an accomplice of a faulty sender:

1. The node stores a preconfigured constant called the *Minimum Integration Count* (MIC) and maintains an integration counter.
2. The integration counter is initialized to zero upon reception of the integration message.
3. The counter is incremented for each succeeding correct message that agrees with the $CC\text{-}state$ of the integration message.
4. If the counter reaches or exceeds the MIC, the node may switch to the active state and participate in synchronous operation by sending messages in its allocated TDMA slots.

This mechanism depends on the membership service (see Figure 3.1.9) to track agreement on the $CC\text{-}state$. An alternative mechanism in networks with central guardians is, as mentioned in Section 3.1.7.5, to delegate the semantic analysis of integration messages to the guardian, which relays only those messages that are consistent with its $CC\text{-}state$ view. Integration messages that disagree with the guardian's $CC\text{-}state$ are aborted. The drawback of this approach is that the guardian's processing speed limits the available medium bandwidth [191].

If the node has also lost synchrony with the environment, for example, due to re-initialization, it must additionally resynchronize with the controlled object. This involves

scanning the state of RT entities via sensors, analyzing the acquired information, and potentially selecting the appropriate reset vector to enforce on the RT entities within its sphere of control. Once the node has obtained all $g\text{-}state$ information relevant at the next reintegration instant, it resumes its tasks in synchrony with the rest of the cluster and the controlled object [1].

3.1.7.7 Transparent Implementation of Fault-Tolerance

In TTA, fault tolerance is achieved through redundancy by replicating both nodes and communication channels. Since timing faults are contained by guardians, value domain faults are masked by grouping replicated deterministic nodes following SMR (see Section 2.2.6.3).

In this setting, the TTA refers to a node as the *Smallest Replaceable Unit* (SRU), and multiple SRUs are combined to form a *Fault-Tolerant Unit* (FTU). In the TMR case, each FTU consists of three deterministic SRUs. When all SRUs belonging to a TMR-based FTU consistently receive the same messages, the failure of one SRU can be masked by majority voting over the outputs of the two remaining operational SRUs. Consequently, the FTU layer relies directly on the atomic multicast service to ensure that all replicas receive identical inputs (see Figure 3.1.9).

Hence, a TTA cluster comprises $U \in \{2, \ldots, N-1\}$ FTUs. Assuming that $j \in \{0, \ldots, U-1\}$ is the unique ID of an FTU in the TTA cluster, an FTU_j comprises $N_j = 2F_j + 1$ replicated nodes, where F_j is the tolerable number of nodes that can be simultaneously faulty in FTU_j. Note that if SRUs are assumed to be fail-silent in the value domain, only two SRUs are required per FTU to mask the failure of one SRU.

3.1.7.7.1 FTU Management Layer

Managing FTU replicas is achieved by inserting an FTU layer with its own dedicated processor between the communication controller and the host computer of the node [199]. As shown in Figure 3.1.12, the original CNI (SRU CNI) becomes the interface between the communication controller and the FTU processor, while a new FTU CNI interfaces the host computer with the FTU processor. From the perspective of the host computer, both CNIs are identical in structure, keeping the redundancy configuration transparent to the application running on the host computer. However, the time at which the host fetches a message from the CNI may change, since the FTU processor must first receive messages from all SRUs of an FTU, combine them into a single message, and place the result in the FTU CNI. This effect is particularly noticeable when the TDMA slots of the SRUs are spread across the TDMA round to tolerate transient faults that may affect duplicated communication channels simultaneously [206].

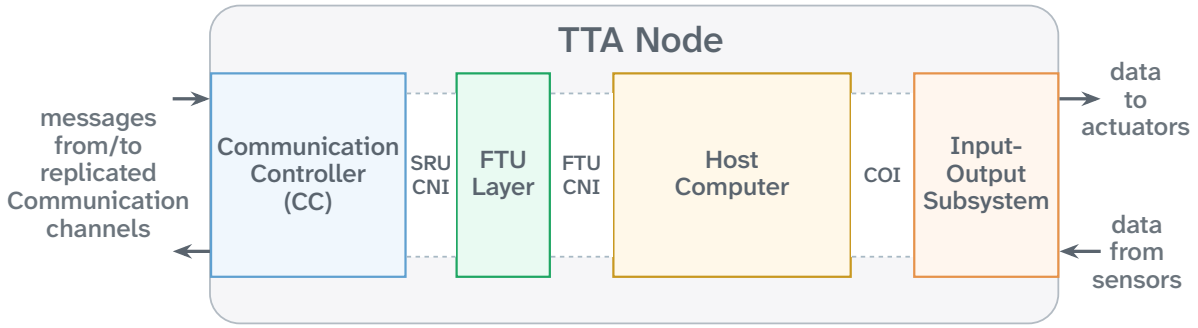


Figure 3.1.12: Structure of a TTA node with an FTU layer.

The FTU layer is responsible for the following functions [199]:

- **Replicated message handling:** The FTU layer combines the messages received from all SRUs of an FTU into a single correct message (e.g., via exact voting) and writes it to the FTU CNI. It also forwards all messages written by the host into the FTU CNI to the SRU CNI. The global time points at which these events occur are determined at design time and are identical across all SRUs of an FTU, demonstrating the FTU layer's dependency on global time and the schedule shown in Figure 3.1.9.
- **Membership fusion:** The FTU layer merges the membership bits of the SRUs into a single bit representing the operational state of the FTU. The fusion function depends on the FTU configuration: in TMR, two operational SRUs are required to set the FTU membership bit, whereas in double modular redundancy one operational SRU suffices. This reflects the FTU layer's reliance on the membership service.
- **Internal state synchronization:** The FTU layer synchronizes the state of all host computers belonging to the same FTU. It provides the same $g\text{-}state$ to all hosts upon initialization and during operation. The $g\text{-}state$ includes the $h\text{-}state$ of all stateful tasks when no tasks are active. The $h\text{-}state$ is externalized by having each node communicate it during its TDMA slots. The FTU layer merges the output $g\text{-}state$ of all hosts into a single message and writes it back to each FTU CNI to ensure that all hosts operate on the same $g\text{-}state$.
- **External clock synchronization:** A gateway node synchronizes the TTA cluster to an external time reference by periodically computing the difference between cluster time and the external reference and writing this value to the FTU CNI. The FTU layer ensures that this value is written to the control registers of the SRU CNI.
- **Diagnosis:** The FTU layer collects and logs information about detected faults from the communication controller and relays this information to the host, either in a special TDMA slot during normal operation or in a dedicated maintenance mode.

This information is useful when replacing SRUs (e.g., permanently failed ones) to restore the intended level of redundancy.

3.1.7.8 Rapid Mode Change Service

The TTA supports mutually exclusive modes of operation in real-time applications (e.g., an aircraft executing different task sets during take-off and cruise) by defining a dedicated TADL for each mode on every host computer and switching between them at runtime [121, 200]. The mode change service provides the architectural mechanism for transitioning between these pre-defined TADLs, which correspond to distinct operational envelopes (see Section 2.2.2.2), thereby allowing the system to adapt to environmental conditions without violating the bounded-demand paradigm. Assigning a per-mode TADL improves the schedulability of real-time applications and alleviates the inflexibility inherent in maintaining a single TADL across all operating conditions.

The rapid mode change service enables TTA nodes to achieve agreement on the current and requested modes of operation by having communication controllers include both modes in messages sent as part of the $CC\text{-}state$. It also governs which host computers may initiate a mode change by specifying, in the MEDL, the TDMA slots at which such requests are permitted.

Prior to transmitting in its TDMA slot, the communication controller of the requesting node checks whether its host computer holds the right to request a mode change. Confirming this right, the controller includes the mode change request in the message by indicating the position of the successor mode in a dedicated mode vector. Receiving nodes similarly consult the MEDL to determine whether the sending node is permitted to request a mode change in that TDMA slot, accepting or rejecting the message accordingly. Consequently, the membership service includes only nodes that agree on both the current and the requested mode.

Building on the previous description and as shown in Figure 3.1.9, the rapid mode change service relies on the following TTA services:

1. The temporal firewall interface, allowing the host computer to initiate a mode change by writing the mode change request to its CNI.

2. The atomic multicast primitive, disseminating the current and requested modes of operation to all other nodes in the cluster.

3. The global time and communication schedule, governing the timing and privilege of sending mode change requests.

4. The membership service, guaranteeing that all operational nodes share the same operating mode and agree on the same requested mode change.

3.1.7.8.1 Mode Change Types

A mode change is characterized by the duration from the moment of requesting the mode change until the system transitions into the requested mode. Timely behavior of the mode change service is therefore essential, especially in situations where the requested mode is an emergency mode. Two types of mode changes are specified in [200]:

1. **Immediate mode change:** the requesting node aborts all ongoing activities before switching modes. Immediate mode changes prioritize speed; however, aborting task execution must not leave tasks in an undefined state.

2. **Deferred mode change:** all cluster nodes begin executing the requested mode at the start of the next cluster communication cycle, i.e., when the system is in the ground state (since no tasks are executing and no messages are in transit). Deferred mode change enforces consistency, as all nodes agree on the current and requested modes.

The TTP/C protocol implementation supports up to three modes when using direct addressing, but can in principle accommodate a theoretically unbounded number of modes through relative addressing, with up to three successor modes reachable from any given mode [21].

3.1.7.8.2 Mode Change for Communication Schedules

On the network side, although the set of messages exchanged over the network changes from one operating mode to the next, the MEDL preserves the same TDMA slot assignments across all modes. Mode change approaches, therefore, consolidate the communication requirements of multiple modes into a single communication schedule, avoiding the need to update guardians or work around limitations in network switches.

For example, TTP [200] keeps the guardian independent of mode changes by requiring all operational modes to share the same TDMA slot pattern for all nodes. Allocated TDMA slots must therefore accommodate the largest message a node might send in any of the modes. A similar approach has been proposed in [120] to bring bandwidth-efficient mode changes to TTEthernet, given that switches lack the functionality needed for performing mode changes autonomously.

Other approaches adopt either a holistic schedule approach, assigning messages from different modes to distinct TDMA slots so that a single schedule covers all modes [187], or perform a comparatively lengthy online reconfiguration process [207].

3.2 TTA's Extra-Functional Properties

The collective interplay between the TTA services and mechanisms described in the previous section gives rise to the key extra-functional properties required in safety-critical systems. While Chapter 1 introduced these properties as a flat list, a deeper examination shows that they are not independent. Instead, they form a logical hierarchy: foundational properties enable analysis and coordination, which in turn support architectural properties, ultimately giving rise to system-wide emergent behaviors.

Understanding these logical dependencies is essential before attempting to relax or substitute any individual TTA property to achieve greater flexibility. In particular, the hierarchy clarifies which guarantees must be provided through alternative means if properties lower in the dependency chain are modified.

3.2.1 Hierarchical Structure of TTA Properties

Figure 3.2.1 illustrates the chain of hierarchical dependencies between TTA's extra-functional properties, which we categorize into four levels:

- **Foundational Properties:**
  1. ***Determinism*** forms the foundation upon which all other dependability properties are built. It ensures that for a given set of inputs and a given initial state, the system exhibits exactly one correct behavior. This causal uniqueness makes complex systems analyzable and provides the logical basis for higher-level guarantees.
- **Analysis & Coordination Properties:**
  1. ***Temporal predictability*** presupposes deterministic behavior, which allows future events or consequences to be derived analytically given sufficient knowledge of all preconditions and rules. Consequently, a nondeterministic system model is inherently unpredictable. For real-time systems, determinism enables the derivation of timing bounds on system activities.

     However, determinism alone does not guarantee predictability. High complexity or incomplete knowledge of all causal factors can limit the ability to forecast the entailed consequences. As well known from *chaos theory*, some deterministic models exhibit highly unpredictable behavior. For example, nonlinear deterministic models of atmospheric thermodynamics are notoriously difficult to predict [208]. Thus, predictability requires not only determinism but also a system whose complexity remains sufficiently manageable.

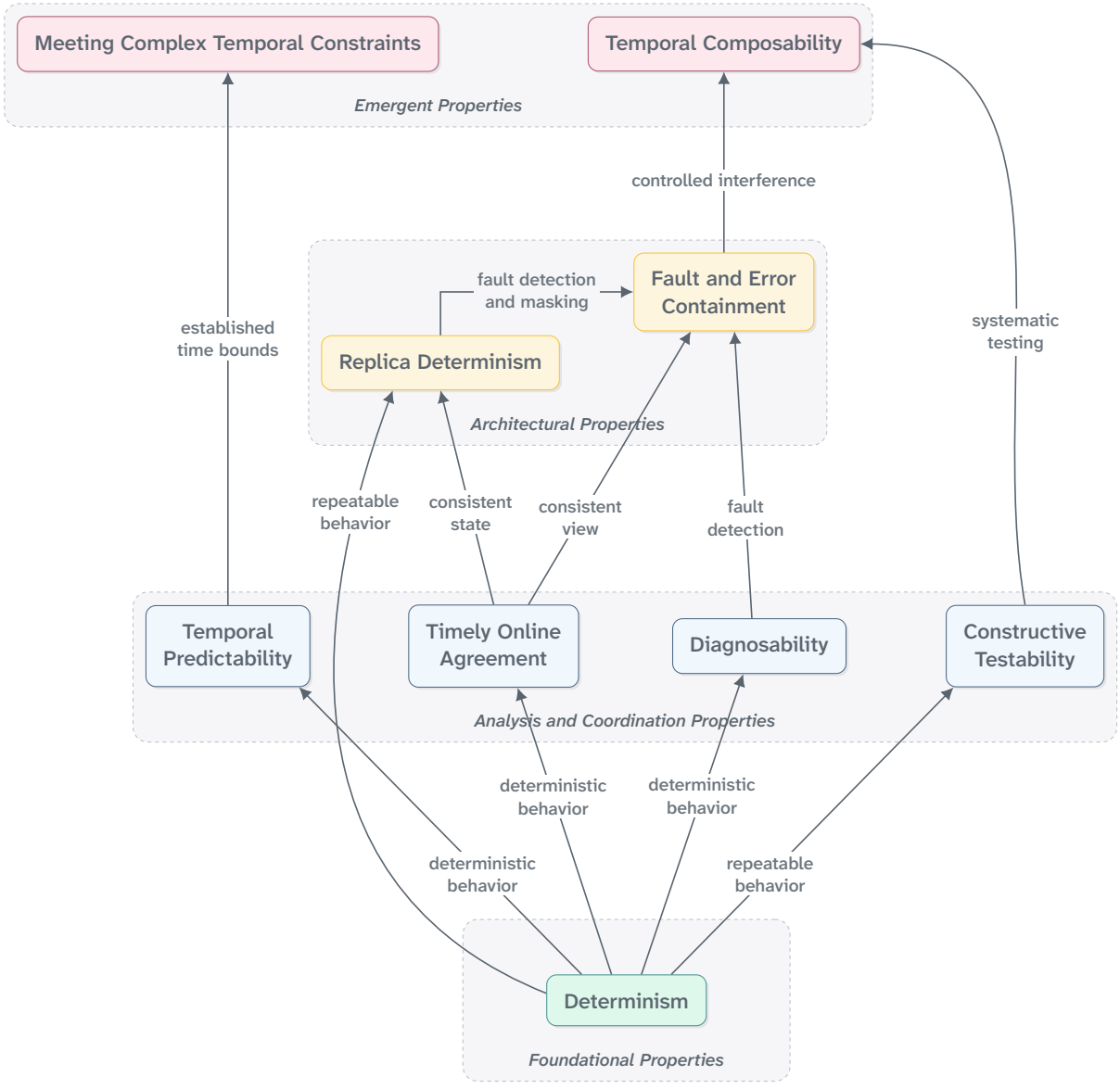


Figure 3.2.1: Hierarchical dependencies between TTA's extra-functional properties.

2. ***Timely online agreement*** builds on deterministic rules (e.g., leader election, log replication, or decision-making) to guarantee safety properties such as agreement (all non-faulty nodes reach the same decision) and validity (the decision value originates from a legitimate proposal). These rules prevent divergence once all healthy nodes share the same information. Ensuring progress (liveness), however, requires synchrony: only with bounded message-transmission and execution times can consensus be achieved in the presence of faults, thereby circumventing the FLP impossibility result (see Section 2.2.4.2). While liveness is typically distinguished from safety in classical distributed systems theory, in safety-critical real-time systems the safety of the overall system crucially depends on liveness, and specifically on the timeliness of results, since missing a deadline may constitute a system failure rather than merely a delay.
3. ***Diagnosability*** depends on a clear and unambiguous definition of correct behavior. Determinism provides this baseline, ensuring that any deviation from the correct temporal or value behavior can be reliably identified as a fault.
4. ***Constructive testability*** requires repeatability. Because deterministic systems behave identically under the same conditions, systematic and reproducible testing becomes tractable in the time and value domains [4, 209].

- **Architectural Properties.**
  1. ***Replica determinism*** depends on both determinism and timely agreement. It ensures that active replicas execute identically and share a consistent view of their state, enabling fault detection and masking through simple majority voting (e.g., in a TMR setting, see Section 2.2.6.3).
  2. ***Fault and error containment*** depends on multiple properties. Diagnosability provides the ability to detect faults, timely agreement provides a consistent view of system membership so that all nodes can isolate the same faulty component, and replica determinism enables coordinated voting to mask faults. Together, these dependencies allow containment regions (see Section 2.2.1.4) to enforce isolation and prevent faults and errors from propagating beyond their boundaries.

- **Emergent Properties.**
  1. ***Temporal composability*** represents the ultimate objective of a node-based design paradigm, in which complex systems are constructed from prevalidated nodes. Constructive testability ensures that components can be

verified systematically and independently, while fault containment guarantees that integrating new nodes does not introduce unpredictable interference with existing temporal guarantees. Together, these properties support the predictable composition of independently developed nodes into larger real-time systems.

2. ***Meeting Complex Temporal Constraints*** is an emergent property enabled by temporal predictability. Once the system's timing is predictable and bounds are known, complex temporal constraints can be guaranteed through known scheduling algorithms (see Section 2.1.6).

In summary, the graph shows a clear logical chain: Determinism enables analysis and coordination properties, which in turn enable architectural properties, ultimately leading to system-wide emergent properties.

3.2.2 Dependency Analysis Between TTA Services and Properties

Now that the logical dependencies among the properties have been established, this section connects these logical requirements to their practical realization within the TTA. We categorize the relationships between the extra-functional properties and the TTA services that implement them into three categories:

1. **Fundamental**: The property cannot be achieved by any other means across all applications. This represents a true architectural requirement where alternatives do not exist or would fundamentally compromise the property.

2. **Implementation-based**: The property can be realized through alternative approaches, potentially with different complexity, performance, or cost characteristics. These reflect TTA's specific design choices rather than fundamental requirements.

3. **Application-dependent**: The property can be realized through alternative approaches, but the feasibility of these approaches depends on the timing requirements of the target application.

This classification framework is a core contribution of this thesis. It clarifies which aspects of the TTA are essential and which can be modified or replaced while still preserving the extra-functional properties. By isolating *implementation-based* dependencies, we improve the understanding of the architectural degrees of freedom in the design space. This enables designing more flexible and secure solutions (e.g., for addressing the research questions in Chapter 1) or comparing existing alternatives to the TTA, while

highlighting the trade-offs that such solutions entail with respect to the extra-functional properties.

Table 3.2.1 presents a consolidated overview of the classification results. As shown, the majority of properties exhibit an implementation-based dependency, indicating a substantial design space for alternative solutions. None of the properties directly depend on the rapid mode change service (see Sec. 3.1.7.8), which constitutes the only form of flexibility in the TTA. However, because this service supports only a limited number of switchable schedules constrained by available memory, it often fails to cover the full range of environmental conditions [210].

Table 3.2.1: Dependency classification of the extra-functional properties (columns) on TTA services (rows).

TTA Service / Mechanism	Extra-Functional Property								
	Timely Online Agreement	Determinism	Replica Determinism	Diagnosability	Fault Containment	Temporal Composability	Temporal Predictability	Constructive Testability	Meeting Complex Temporal Constraints
Sparse Global Time	Imp.								
Offline Schedules	Imp.	App.		Imp.			App.	Fund.	Imp.
Atomic Multicast	Imp.			–		Imp.		–	
Membership	Imp.				–	Imp.		–	
Guardians	Imp.			–	Imp.				–
Temporal Firewall Interface	–		Imp.	–		Imp.		–	
Transparent Implementation of Fault-Tolerance	–		Imp.	–	Imp.			–	
Node Integration	–		Imp.	–		Imp.		–	
Rapid Mode Change	–								

Imp.: Implementation-based; **App.**: Application-dependent; **Fund.**: Fundamental; **-**: No direct dependency.

In the following, we revisit each property, describe how the TTA realizes it, identify the dependencies on the corresponding TTA services, and discuss potential alternative approaches.

3.2.2.1 Timely Online Agreement

Reaching consensus on events and decisions in a timely manner is essential for coordinated behavior in distributed systems. Circumventing the FLP impossibility result requires introducing synchrony into the system, typically through distributed failure detectors (see Section 2.2.4.2). In real-time systems, such failure detectors must have established time bounds on the detection of failed nodes to guarantee timely reactions to failures.

3.2.2.1.1 TTA's Strategy

The TTA operates under a strictly synchronous system model, wherein upper bounds are defined for processing speeds, message transmission delays, clock drift rates and clock differences (see Section 2.2.2.1). Within the fault hypothesis outlined in Section 3.1.5, the membership service functions as a timed failure detector characterized by the following properties:

1. **Timed strong completeness**: every failed node that overtly exposes its failure (e.g., by omitting a message or sending an invalid protocol message) is reliably and consistently suspected by all correct nodes within one communication round following the node's last TDMA slot (or equivalently, its last membership point).
2. **Strong accuracy**: no correct node is ever falsely suspected of failure.

These properties rely on the underlying atomic multicast service and guardians to ensure consistent message reception (see Section 3.1.7.5). If the fault hypothesis is violated, leading to inconsistent reception or multiple simultaneous faults, the TTA invokes the membership's clique avoidance algorithm as a fallback mechanism to stabilize the cluster, prioritizing consistency over availability in line with the CAP theorem (see Section 2.2.4.4).

Hence, timely online agreement relies, as summarized in Table 3.2.1, on sparse global time and offline schedules, which together establish the timing bounds required for liveness. Atomic multicast (which relies on bus guardians) provides consistent message delivery, and the membership service supplies the timed failure detection needed to agree on node operational status.

3.2.2.1.2 Dependency Classification and Alternative Approaches

Efficient online agreement in the TTA hinges on maintaining consistent reception without incurring the overhead of complex online agreement protocols, as discussed in Section 3.1.7.5. Because this service relies on global time, the offline schedule, membership, and guardians (see Figure 3.1.9), timely online agreement in the TTA also depends

on these properties. However, this dependency is *implementation-based*, as alternative approaches demonstrate that efficient timely agreement can be achieved through different architectural paradigms. For example, *eager execution* strategies reduce latency by initiating speculative execution on multiple processing cores before agreement is finalized [211]. Likewise, hardware-centric approaches such as *Crosstalk* [212] leverage redundant switched networks and hardware timestamping to achieve timely agreement within the network fabric itself, thereby eliminating the need for inter-replica communication mechanisms.

3.2.2.2 Determinism

As established in Section 2.2.4.5, determinism in distributed systems fundamentally depends on establishing a form of common knowledge regarding system state and event ordering. Without such a shared understanding, the nodes cannot reliably coordinate or reason about causality.

3.2.2.2.1 TTA's Strategy

Determinism in the TTA is achieved by simulating common knowledge through two mechanisms: The sparse global time base (Section 3.1.7.1) and offline-defined schedules (i.e., the TADLs and the MEDLs). By adopting a *Periodic Finite State Machine* (PFSM) model [213], TTA punctuates time into alternating activity and silence intervals, a structure reminiscent of the punctuated execution semantics in synchronous languages [148].

During each activity interval, nodes initiate concurrent activities on their respective resources following the MEDL and the TADL. The subsequent silence interval provides sufficient slack for all activities to complete before the next activity interval begins. This disciplined alternation ensures that, when the system state is observed at the end of each silence interval, all activities have deterministically concluded. The system exhibits deterministic behavior, provided that the observer examines the state of the system only at the end of each silence interval.

This design masks timing uncertainties and effectively simulates perfectly synchronized clocks for internal events, tackling the simultaneity problem and eliminating the need for costly online agreement protocols for every scheduling decision. However, for external events outside the computer's *sphere of control*, an online agreement protocol is still required to assign them a consistent timestamp on the sparse time base.

Hence, determinism relies, as summarized in Table 3.2.1, on the sparse global time base to simulate common knowledge and absorb timing uncertainties, while offline-defined schedules dictate the unique, repeatable sequence of operations. The member-

ship service, guardians, and atomic multicast mechanism ensure that all nodes maintain a consistent view of system state, thereby preserving deterministic operation.

3.2.2.2.2 Dependency Classification and Alternative Approaches

The dependency of achieving determinism on sparse global time is *implementation-based*. While it represents one viable strategy for the temporal coordination of distributed events, its functional objectives may alternatively be fulfilled by a centralized network component that emits periodic signals. These signals enable individual nodes to initiate activities according to an offline schedule and thus similarly simulate common knowledge of scheduling events and their order. The master node architecture proposed in the *Flexible Time-Triggered* (FTT) paradigm [214] exemplifies such an implementation. However, this centralized approach inherently introduces the master node as a single point of failure (SPoF). In comparison, global time in the TTA is maintained by a distributed fault-tolerant clock synchronization algorithm. Mitigating this SPoF vulnerability requires replication of the master node and the deployment of master election protocols, which in turn increase system complexity.

Determinism can also be achieved by simulating epsilon common knowledge regarding scheduling decisions or event order, albeit with higher latency or overhead. Online protocols enable nodes to reach agreement on event ordering or scheduling decisions [5,151,215]. However, the strength of the attained notion of temporal determinism is dependent on the latency bounds of the agreement protocols and the permissible time interval (relaxation of simultaneity) to produce the same output across replicas.

Simulating common knowledge regarding scheduling decisions based on offline schedules provides strong temporal determinism and strict timing guarantees. Yet many applications can accommodate a degree of temporal flexibility, an aspect not exploited by TTA's rigid offline schedules. In response, flexibility-oriented approaches have emerged that explore determinism hierarchically, ensuring determinism at higher abstraction levels while permitting controlled flexibility at lower levels [117]. These methods enable flexible scheduling and improved resource utilization while preserving overall deterministic behavior.

We categorize existing alternative approaches as follows:

1. **Flexible task scheduling**: In order to optimally handle runtime aperiodic tasks in TT systems, while still benefiting from the temporal partitioning that TT slot-based operation provides, the *Slot Shifting* [122, 125] enabled dynamic scheduling on top of TT activation. Unlike mode-change mechanisms in the TTA [121,200,216], which can switch between offline-defined schedules to replace the entire TT task set but cannot incorporate new aperiodic tasks at runtime, these flexible scheduling methods simulate common knowledge of *task parameters* rather than common

knowledge of complete schedules. Task parameters, derived from timing requirements and the sending and receiving times of relevant TT network messages (as described in Section 2.1.6.7), are treated as *globally known points in sparse time* that serve as anchors for coordination. The resulting notion of temporal determinism is comparatively weaker than that offered by the TTA; they guarantee that tasks always execute within their execution windows but not necessarily within the same computation slots every schedule cycle. The complexity of the runtime scheduler must be accordingly increased to handle the added flexibility, to levels that are comparable to classical online schedulers (see Section 2.1.6.4).

It is worth noting that Slot-Shifting assumes that synchronization between tasks for resource access is modeled as precedence relations and resolved during offline scheduling. Otherwise, it must either presume that resources are not held across slots or implement more sophisticated synchronization protocols.

Another approach that simulates common knowledge of task parameters to enable a set of replicas to employ dynamic preemptive scheduling while preserving their deterministic behavior was introduced in [5]. Rather than forcing replicas to make identical scheduling decisions, the method ensures that they use the *same message version*. In this approach the temporal partitioning offered by the slot-based allocation is lost and requires additional mechanisms under ET activation to be provided, e.g., wrapping tasks with bandwidth-constrained servers [60].

2. **Flexible message scheduling**: In the absence of simulated common knowledge regarding message scheduling decisions, achieving online agreement on the ordering and timing of network accesses becomes significantly more challenging. This difficulty arises primarily from the need for inter-node communication to establish such agreement. A viable alternative is based on a centralized runtime scheduling approach with epsilon common knowledge regarding scheduling decisions. For instance, the use of a centralized master node that assumes responsibility for network arbitration and disseminates scheduling decisions to all nodes during runtime. The FTT paradigm [214,217,218] exemplifies this approach, wherein the master node orchestrates communication by transmitting signals that mark the start of communication rounds, and where each signal embeds the corresponding round schedule.

   Alternatively, a hybrid TT/ET strategy may be employed. In this configuration, certain messages adhere to an offline defined schedule to ensure strict timing guarantees, while others are dynamically scheduled based on priority rules and specialized arbitration mechanisms. This approach underpins many contemporary real-time networking protocols and standards, including *FlexRay* [192], *TTEther-*

net [6, 22] and *Time-Sensitive Networking (TSN)* [219, 220], all of which seek to balance determinism with operational flexibility in complex distributed systems. Recent developments further emphasize increased adaptivity within these standards, for example through reconfiguration capabilities and *Software-Defined Networking* (SDN)-based orchestration in TSN [221, 222].

From the above discussion, it follows that the nature of the dependency between determinism and offline-defined schedules is governed by the specific requirements of the application. Accordingly, this dependency can be classified as *application-dependent*.

3.2.2.3 Replica Determinism

Determinism plays a critical role in facilitating fault tolerance through the replication of deterministic nodes. In particular, replica-determinism, where replicated nodes start from the same initial state and process identical inputs to produce outputs that are consistent in order and timing, considerably simplifies fault masking and constitutes a prerequisite for SMR (see Section 2.2.6.3).

3.2.2.3.1 TTA's Strategy

The TTA supports replica determinism by ensuring that all replicas of an FTU are initialized consistently with an identical state. This state is derived from global time and supplied to failed or rejoining replicas through the node integration service. The atomic multicast service and guardians ensures that all FTU replicas receive identical inputs, the offline schedule guarantees that they execute tasks and produce messages in the same order, and the FTU layer performs exact majority voting on their outputs, adjudicating them into a single result that it then deposits into the CNI of the receiving host computer. This coordinated execution model guarantees that all non-faulty replicas remain state-consistent and produce identical results. The dependencies of replica determinism on TTA services are summarized in Table 3.2.1.

3.2.2.3.2 Dependency Classification and Alternative Approaches

Replica determinism has also been achieved through more dynamic and flexible SMR protocols compared to the TTA. These protocols rely on establishing epsilon common knowledge of event ordering and scheduling decisions, as discussed in Section 3.2.2.2. This approach is widely used in ET systems to coordinate replicas and ensure determinism [151, 215, 223, 224], relaxing the strict synchrony and scheduling assumptions of the TTA while still preserving replica consistency and timing guarantees. While the dependency on offline schedules remains *application-dependent* due to the associated

latency and resource overheads inherent to online agreement protocols, the dependency on TTA's global time, atomic multicast, FTU layer, and node integration service is *implementation-based*, as equivalent guarantees can be realized through alternative mechanisms.

3.2.2.4 Temporal Predictability

Temporal predictability refers to the ability to analytically forecast the temporal behavior of a system, ensuring that timing constraints are met based on known system characteristics [4]. As established in Section 3.2.1, temporal predictability presupposes determinism and manageable complexity.

3.2.2.4.1 TTA's Strategy

The TTA strengthens temporal predictability through the following design choices:

1. **Internalized temporal control.** Nodes initiate actions based on predefined global-time triggers (in the TADLs and MEDLs) rather than reacting to external interrupts, thereby maintaining temporal independence and avoiding timing unpredictability caused by asynchronous events. These predefined triggers also serve as anchor points that establish offline agreement on key events such as network access and node integration times.

2. **Temporal isolation.** The temporal firewall (i.e., the CNI) acts as a low-pass filter that shields the host computer from event showers originating in the environment (see Sections 3.1.1 and 3.1.7.3). Similarly, Guardians enforce compliance with the defined MEDL and avoid unpredictable interferences between nodes.

3. **Avoiding nondeterministic communication effects.** The TTA avoids mechanisms such as retransmissions and back-off procedures, which introduce jitter and temporal uncertainty. Instead, it relies on periodic state messages and hardware-supported clock synchronization, thereby bounding communication jitter to the precision of clock synchronization and ensuring strict timing requirements [6].

TTA's membership, node integration, and FTU layer are not primary contributors to temporal predictability. However, they indirectly support temporal predictability by preventing inconsistent system views, stabilizing the system under faults, and ensuring replicas have consistent states.

3.2.2.4.2 Dependency Classification and Alternative Approaches

The dependencies on TTA's global time, CNI, guardians, and atomic multicast service for predictability are *implementation-based*. Equivalent guarantees can be realized through alternative coordination mechanisms. For example, centralized coordinators can serve as a single source of temporal control and may also enforce network access policies while supporting state-based messaging [214]. Likewise, *Code-Division Multiple Access* (CDMA) enables conflict-free channel sharing without requiring explicit guardian components [225].

In contrast, the dependency on offline schedules is *application-dependent*. Whether simulated common knowledge regarding scheduling is required, or whether more relaxed notions such as epsilon common knowledge suffice, depends on the specific level of determinism and flexibility required by the application, as discussed in Section 3.2.2.2.

3.2.2.5 Diagnosability

Diagnosability concerns the ability to reliably identify the operational status of nodes, which is crucial for triggering fault tolerance mechanisms or reconfiguration.

3.2.2.5.1 TTA's Strategy

The TTA leverages determinism, a-priori knowledge, and redundancy in the architecture to implement fault detection as follows: Space redundancy is utilized through duplicated communication channels and CRC codes to detect transmission faults (see Section 2.2.5.2), while actively replicating deterministic nodes in FTUs allows for the detection of value domain errors based on voting (see Section 3.1.7.7). Temporal redundancy allows detection of transient resource failures. Crucially, the a-priori knowledge encoded in the MEDL enables receivers to strictly enforce timing correctness: any message arriving outside its pre-allocated TDMA slot, or earlier or later than its scheduled time by more than the synchronization precision (i.e., outside the acceptance window for the message), is immediately identified as faulty.

The membership service then aggregates these observations to report a consistent system-wide view of nodes' health within a bounded time.

3.2.2.5.2 Dependency Classification and Alternative Approaches

Anomaly detection based on a-priori knowledge and redundancy constitutes a well-established strategy in dependable system design and is not exclusive to the TTA. For example, fault detection and masking based on active redundancy can be implemented under priority-driven scheduling [224]. Similarly, predefined environmental assumptions and operational constraints enable the identification of faulty sensor readings

or control outputs. However, when environmental conditions change, these detection mechanisms must adapt accordingly to avoid false fault identifications.

Similarly, TTA's approach to identifying untimely messages requires reconsideration in modern systems, where schedules may change dynamically and where not all messages follow a strictly periodic model. For instance, the central master node approach described in [214], which generates a new schedule at each round, equips nodes with sufficient temporal context to identify untimely transmissions. Furthermore, messages governed by a maximum allowable transmission rate can be subjected to plausibility checks that detect and discard those exceeding the specified rate [6]. These examples illustrate that equivalent functionality can be achieved through alternative mechanisms. Consequently, the dependency on TTA's global time, offline schedule and transparent implementation of fault tolerance is *implementation-based*.

While alternative membership management protocols, offering more flexibility (e.g., by accommodating a dynamic number of members [226]) exist, classifying the dependency on TTA's membership as *implementation-based*, the mechanism for determining node health through the periodic transmission of heartbeat messages remains *fundamental*.

Furthermore, TTA's fault detection mechanisms are based on the fault assumptions defined in its hypothesis (see Section 3.1.5). When these assumptions are violated, such as in open system models exposed to sophisticated security threats, TTA's fault detection capabilities may be circumvented, potentially compromising system safety. This underscores the need for adaptive and resilient fault detection and fault tolerance strategies that extend beyond the constraints of TTA's original scope (examined in more detail in Chapter 5).

3.2.2.6 Fault and Error Containment

As described in Section 2.2.1.4, the fault hypothesis partitions the system into independent FCRs, where a fault occurring in one region does not corrupt the operation of others. If FCRs are not properly isolated, a single fault can manifest as errors and cause cascading node failures, undermining the reliability of the entire system. Here, error containment is essential for preventing fault propagation from one FCR to the rest of the system. Crucially, error containment mechanisms must themselves be implemented within an independent FCR, ensuring that they block erroneous outputs and remain unaffected by faults occurring within the regions they monitor.

3.2.2.6.1 TTA's Strategy

The TTA defines each node as a distinct FCR and employs strict partitioning mechanisms to contain errors at the network level (see Section 3.1.5). The sparse global time and offline schedules (i.e., MEDLs) establish the temporal boundaries within which each node must operate. These boundaries are enforced by guardians, which restrict a node's network access to its assigned TDMA slots as defined in the MEDL (see Section 3.1.7.4). By synchronizing to global time and inserting safety gaps between slots to account for clock-synchronization error, guardians prevent *babbling idiot failures* and ensure collision-free communication.

Value domain errors are contained through spatial partitioning, which isolates private memory, and through the active replication of deterministic nodes within FTUs, where majority voting masks errors before they reach the application (see Section 3.1.7.7). Further, TTA's strictly synchronous system model allows for a reduction in the number of nodes required within an FTU to tolerate F arbitrary faults, from $3F + 1$ to $2F + 1$, in accordance with NMR principles (see Section 2.2.6.3).

While the temporal firewall interface (i.e., the CNI) participates in containing errors by design by separating local computation from global communication and protecting the host computer from interrupt showers (see Section 3.1.7.3), we do not include a direct dependency from error containment to the CNI in Table 3.2.1, since TTA assumes that the entire node, including the host computer, the communication controller, and the CNI, constitutes a single FCR (see Section 3.1.5).

3.2.2.6.2 Dependency Classification and Alternative Approaches

Guardians. TTA's guardians provide strict temporal partitioning and error containment in the time domain. While implementing the guardian functionality in an independent FCR remains a fundamental requirement for containing a monitored node's untimely behavior, other implementations, such as those described in [6,192,227], achieve the same objective, making the dependency on TTA's guardians *implementation-based*. Furthermore, the reliance of error containment on global time and offline schedules arises from the guardian's need to track schedule execution. This dependency is likewise *implementation-based*, since equivalent coordination can be realized through alternative synchronization mechanisms, such as a master node issuing periodic synchronization signals and embedding the network access schedule.

Contemporary systems, however, demand more advanced guardians that support flexibility, security, and scalability. Protocols employing dynamic allocation of TDMA slots (e.g., [214, 217, 218]) can achieve comparable temporal isolation to TTA's solution by informing guardian components of schedule changes before schedule execution. The guardian functionality can also be extended to rate-constrained traffic, where mes-

sages that exceed a predefined transmission rate are subjected to plausibility checks and throttled to limit their transmission frequency [23].

In heterogeneous environments where multiple network protocols and topologies coexist, the limitations of network-aware guardians become increasingly apparent. This motivates the need for network-agnostic approaches, such as our SyncGuard solution (see Section 5.5.6), which enhance scalability and highlight the importance of coordinated node rejuvenation as a means to strengthen resilience against cyberattacks and intrusions.

Spatial partitioning. The spatial partitioning principle of the TTA can be applied independently of its associated services and mechanisms. However, modern real-time systems present containment challenges not anticipated in the original TTA scope. Multicore and manycore architectures introduce heterogeneous processing elements, shared caches, complex memory hierarchies, diverse I/O devices, hardware accelerators, and shared power sources. These platforms often execute multiple applications of differing criticalities on the same node, which introduces timing interference [228], reduced predictability, and increased exposure to security threats [229].

In order to address these challenges, more advanced containment mechanisms combine fine-grained temporal and spatial partitioning. Modern solutions typically introduce additional layers of abstraction to enforce this isolation, allowing complex workloads of differing criticalities to coexist predictably on shared hardware[2].

Active replication. Finally, active replication is not confined to strictly synchronous models with global time and offline schedules. It has also been implemented in partially synchronous systems [215, 224]. In such models, the number of replicas can be reduced from $3F + 1$ to $2F + 1$ by incorporating trusted components and adopting a leader-follower SMR organization (see Section 2.2.7). This approach, however, entails drawbacks [1]:

1. Reduced application-level bandwidth due to the computational and communication overhead of inter-replica coordination.

2. Degraded temporal accuracy of observations caused by the latency overhead introduced by inter-replica coordination.

[2]Prominent examples of such abstraction layers include virtualization and containerization. Virtualization technologies provide strong isolation while enabling flexible resource allocation [230–235]. Compared to TTA's static partitioning, virtualization achieves higher resource utilization but at the cost of additional overhead and complexity. Containerization has emerged as another layer of abstraction, complementing or replacing virtualization for certain workloads [236, 237]. Containers provide lightweight isolation with lower startup latency and reduced overhead compared to full virtual machines, while enabling controlled resource sharing. When combined with real-time aware orchestration frameworks [238–240], containerization supports predictable execution and enforces temporal and spatial isolation.

3. Increased inter-replica dependency, where the interval between the leader's decision and its reception by followers creates a vulnerability window that can weaken error-containment boundaries.

3.2.2.7 Constructive Testability

Testing is only meaningful under the assumption of repeatability. Constructive testability reflects the ability to systematically exploit repeatability to verify timing correctness across all execution scenarios [209].

3.2.2.7.1 TTA's Strategy

The TTA facilitates constructive testability by drastically reducing the effective test space. It achieves this reduction by employing the following design principles:

- **Single source of temporal control:** Employing global time as the exclusive mechanism for initiating task executions and message transmissions eliminates the complexity arising from interactions between multiple sources of triggers. This approach significantly reduces the number of scenarios generated by interleaving such signals.
- **Offline schedules:** All computational and communication activities are fully specified during the design phase and allocated to dedicated time slots in the offline schedules. This removes runtime scheduling variability and avoids the need to test alternative timing interleavings of system activities.
- **Strict temporal partitioning enforcement:** The temporal isolation provided by exclusive time slots is rigorously enforced by the real-time operating system at the host level and by guardians at the network level. This guarantees that testing can be reliably performed at the granularity of individual tasks or messages.

3.2.2.7.2 Dependency Classification and Alternative Approaches

The dependency on global time is *implementation-based*, as alternative timing coordination mechanisms to global time can serve as a single source of temporal control; the master node concept exemplifies this approach [214].

On the other hand, rigid offline schedules create a finite and predetermined state space that cannot be replicated by dynamic scheduling approaches while preserving testability at the task or message level. By contrast, architectures employing *dynamic scheduling*, where execution order and timing are determined at runtime, can exhibit a combinatorial explosion of possible scenarios. In such systems, constructive testability

is significantly impaired, as all feasible interleavings of tasks and messages must be considered, including rare corner cases that may never occur in practice but cannot be excluded analytically.

Given that no other known architectural paradigm achieves a comparable reduction in the test space, the dependency on offline schedules for attaining task- or message-level constructive testability is *fundamental*. While this holds true for uniprocessor nodes in a distributed system, when scaling to multicore and manycore platforms, offline schedules alone no longer guarantee task-level testability due to interference from shared resource contention between tasks executing in parallel on different cores, demanding more sophisticated test methods also for TT systems.

Hence, a controlled degree of scheduling flexibility may still be a viable alternative. For example, constraining the set of activities permitted to execute within a given interval, thereby limiting potential interference, can still limit the number of test scenarios to a manageable level.

3.2.2.8 Temporal Composability

Composability enables a node-based design paradigm in which large systems are assembled from prefabricated nodes with specified and pre-validated properties [241]. System designers choose nodes based on functional and temporal needs; if no match exists, a new node must be developed [1].

An architecture is *temporally composable* if timing guarantees established for a node remain valid after integration. This allows system-level timing analysis to be derived from node-level guarantees without global re-analysis. According to [1, 6], a temporally composable architecture must satisfy:

1. **Independent development of nodes.** Nodes must be designable in isolation, which requires complete interface specifications in both value and time domains, including WCETs, communication latencies, and jitter bounds. If understanding an interface requires insight into internal implementation, composability breaks.

2. **Stability of pre-validated services.** The services delivered through a node's interface must remain unaffected after the node is integrated into the system. This stability ensures that properties validated at the node level continue to hold at the system level without revalidation.

3. **Non-Interfering Interactions and Constructive Integration.** Communication groups sharing a network must not interfere in value or time domains. Adding a node must not degrade existing timing guarantees. Otherwise, integration effort grows non-linearly.

4. **Diagnosability and Replica Determinism.** Faulty nodes must be detectable and replaceable without exposing internal details. Fault detection mechanisms require redundancy within the architecture, which constrains implicit sharing of resources among nodes. If replication is used, the architecture must also ensure *replica determinism*.

3.2.2.8.1 TTA's Strategy

The TTA satisfies the key principles of temporal composability through the following architectural design choices:

1. The TTA specifies node interfaces in both the value and time domains using the sparse global time base, enabling *independent development of nodes* and supporting compositional reasoning.

2. The CNI (see Section 3.1.7.3) implements an *information-pull* model at the receiver that decouples host computer timing from network timing, ensuring that a node's pre-validated temporal properties remain stable after integration.

3. Guardians prevent interference between nodes on the communication channels in the time domain and enforce access according to the predefined MEDL.

4. The MEDL can include unallocated slots that allow new nodes to be added without disturbing the timing guarantees of existing nodes to support constructive integration.

5. Replica determinism and fault detection are provided as described in Sections 3.2.2.3 and 3.2.2.5, ensuring that faulty nodes can be detected and isolated and that replicated nodes produce consistent outputs within bounded time.

3.2.2.8.2 Dependency Classification and Alternative Approaches

Temporal composability is an emergent property that depends on multiple other extra-functional properties (see Figure 3.2.1). Considering our discussions on these related properties, we can conclude that a comparable level of temporal composability, potentially with greater flexibility, can be achieved by replacing the global time base and offline-defined schedules with the previously mentioned master-node synchronization approach, making these dependencies *implementation-based*. Similarly, the reliance on guardians, membership, CNI, and FTU layer stem from their role in achieving fault detection and error containment. Since these functions can also be provided through alternative architectural means, their dependency nature is similarly *implementation-based*.

However, the strict non-interference model of TTA is too rigid for modern systems with shared resources (e.g., multicore systems with shared caches and communication infrastructure). Contemporary architectures may instead adopt a *bounded interference* model, where interference is not eliminated but analytically limited, allowing for composability with higher resource utilization [242, 243]. This notion of bounded interference complies with the requirements of *freedom from interference* principle of the *ISO 26262* standard [3], as long as it can be proven to not adversely affect safety-relevant functionality.

3.2.2.9 Meeting Complex Temporal Constraints

Distributed real-time applications often involve complex timing requirements, such as precedence constraints, distributed deadlines, and jitter limits. The scheduling problems arising in these settings are at least *NP-Complete* in computational complexity (see Section 2.1.6.3). Managing these constraints at runtime adds significant overhead and complexity to the operating system's runtime scheduler. Hence, a predictable architecture must provide mechanisms that guarantee these constraints are met efficiently.

3.2.2.9.1 TTA's Strategy

As explained in Section 3.1.1, the TTA shifts the complexity of scheduling to the design phase, where timing, interfaces, and communication patterns can be fully specified and verified. An offline scheduling algorithm (see Section 2.1.6.6) computes feasible schedules for distributed nodes and communication controllers (i.e., the TADLs and the MEDLs) that satisfy all specified timing constraints. At runtime, the dispatcher simply executes the decisions encoded in these schedules, requiring no awareness of the global constraints.

3.2.2.9.2 Dependency Classification and Alternative Approaches

TTA's offline scheduling approach presupposes a periodic task model in order to assign tasks or their messages to exclusive time slots. If the timing parameters of the task set are not fully known or cannot be expressed within a periodic model at design time, no timing guarantees can be established offline. The offline scheduling approach also depends directly on global time for the temporal specification and coordination of activities. This dependency is, however, *implementation-based*, since these functions can be realized through alternative mechanisms (e.g., a master node, as discussed earlier).

The dependency on offline schedules is likewise *implementation-based*. Priority-driven online scheduling algorithms generally rely on simple attributes (e.g., priorities or deadlines) as scheduling directives, limiting their ability to resolve complex distributed

constraints at runtime (see Section 2.1.6.4). However, alternative approaches such as *Slot Shifting* [122] and our dynamic task set scheduler introduced in Section 4.2.5 combine the strengths of offline and online scheduling. They transform a task set with complex distributed constraints into one with simpler local constraints offline (as described in Section 2.1.6.7), enabling an online priority-driven scheduler to efficiently handle both constrained and aperiodic tasks at runtime while implicitly preserving the original complex constraints.

3.2.2.10 Summary and Discussion

The dependency analysis presented in this chapter clarifies how the TTA achieves its extra-functional properties through the interplay of global time, offline scheduling, and the set of dependability services. A key insight is that most of these dependencies are not fundamental requirements but the result of specific architectural design choices. Consequently, TTA's extra-functional properties can, in principle, be realized through alternative mechanisms, opening a broad design space for hybrid or reinterpreted solutions that balance predictability, flexibility, and resource efficiency.

While the TTA provides a solid framework for building dependable distributed real-time systems under its original design assumptions (i.e., closed-world, only accidental faults, and simple hardware), modern safety-critical applications and computing platforms introduce challenges not fully addressed by the architecture. The push toward dynamic workload management conflicts with static slot-based allocation, and increasing connectivity and platform heterogeneity undermine the closed-system assumption and introduce new security concerns.

Despite the pressures of modern computing environments, TTA's architectural concepts and design principles remain valuable, as they continue to provide a foundation for predictable, analyzable, and resilient behavior in complex cyber-physical systems. Nevertheless, these principles must be adapted to address the evolving requirements of safety-critical applications.

The dependency analysis developed in this chapter serves as the baseline for the subsequent chapters, where we address the research questions presented in Chapter 1 by examining how resilience and flexibility can be achieved in time-triggered real-time systems without compromising TTA's extra-functional properties.

"It is **not the strongest** of the species that survives, **nor the most intelligent**, but the **one most responsive to change**."

(Leon C. Megginson)

Chapter 4

Flexibility in Time-Triggered Scheduling

Chapter 3 established a dependency framework that clarifies how the TTA achieves its extra-functional properties. This analysis revealed that many of these properties stem from implementation-specific design choices rather than inherent architectural constraints. Consequently, it becomes possible to identify where flexibility can be introduced without compromising the guarantees that the TTA delivers.

In the context of real-time scheduling, flexibility refers to the scheduler's ability to adapt at runtime, admitting tasks with potentially a-priori unknown arrival times or execution costs, dynamically changing priorities or *Quality-of-Service* (QoS) levels, and reallocating resources (e.g., CPU, network bandwidth). Although such flexibility can be applied to both communication (message scheduling) and computation (task scheduling), this chapter focuses exclusively on flexibility for task scheduling on TT computation nodes, as addressing both dimensions simultaneously exceeds the scope of this thesis.

Addressing RQ1 and RQ2 of this thesis, this chapter investigates how TT systems can be prepared to accommodate dynamically changing task sets without relying on exhaustive offline enumeration of all possible configurations. In particular, we examine how to efficiently determine whether an already feasible TT task set can be safely modified at runtime by adding or removing aperiodic, sporadic, or periodic tasks.

We begin by reviewing related work on flexibility for TT systems. We then introduce *dynamic time-triggered schedulers* (DTT), a class of runtime schedulers that allocate computation slots dynamically while preserving the guarantees established offline. Within this class, we analyze existing approaches and present our *Dynamic Task Set Scheduler* (DTSS) as a novel solution to admit, skip, and remove tasks efficiently at runtime under TT operation. These capabilities rely on our provided suite of runtime-optimized feasibility tests. Finally, we evaluate the proposed solutions experimentally and compare them to related work based on synthetic task sets and real-world automotive benchmarks.

4.1 Related Work

Existing flexibility-oriented methods for task scheduling under the TT paradigm modify the set of tasks executed by the system. Broadly, these approaches fall into two categories: global task-set replacement and task set extension with ET tasks.

4.1.1 Global Task Set Replacement

Methods in this category dynamically switch between a set of offline-defined schedules to accommodate aperiodic events, such as changes in the operational mode [120, 121, 200, 216]. Operational modes represent mutually exclusive phases of system behavior, such as startup, normal operation, emergency states, or specific mission phases (e.g., an aircraft transitioning from flight to landing).

Early foundational work on TT mode change [121, 200] demonstrated that a mode switch must be carefully orchestrated to avoid causing deadline misses in the newly established mode or leaving executing tasks from the source mode in an inconsistent state. In the distributed setting, this further requires global consistency: all nodes must reach an agreement on both the target mode and the exact global time at which the switch must occur. To guarantee a safe transition without violating temporal constraints, distributed systems typically employ deferred mode changes, waiting for the current schedule cycle to complete before initializing the new mode. Consequently, traditional mode change methods can incur significant delays before the subsequent mode is fully established.

Furthermore, a task set might be perfectly schedulable within its individual modes but become unschedulable during the mode change itself. In order to mitigate the high mode-change latency and resolve the potential unschedulability during the switch, Fohler proposed the use of *transition modes* [118, 121]. A transition mode serves as a temporary, intermediate schedule executed exclusively during the switch from an old mode to a new one. By executing a safe, pre-calculated subset of tasks from both the originating and target modes, transition modes allow the system to realize the mode switch without waiting for a full cycle boundary to pass. However, this early method is restricted to a single computational node, omitting the complexities of distributed network communication. Furthermore, while this approach successfully lowers latency, it exacerbates the memory overhead required for storing TT schedules, as additional transition schedules must be generated and stored for permutations of mode switches.

Finally, to simplify the complex distributed agreement process required for global mode changes, a central master component within the network infrastructure can be employed. This master node broadcasts the selected offline-defined schedule (or a newly computed one) to all network participants and blocks nodes that fail to comply.

While this centralizes and accelerates the transition, the master component inherently becomes a single point of failure, demanding redundancy to meet safety-critical requirements [217, 218].

4.1.2 Task Set Extension with Event-Triggered Tasks

Rather than swapping entire schedules, these methods focus on extending the TT task set locally at a given node with additional ET tasks. These approaches can be further sub-categorized based on when the schedule modification occurs:

4.1.2.1 Offline Schedule Modification

Offline schedule modification methods extend the task set by shaping the TT schedule during offline synthesis. We distinguish the following two representative approaches:

- **Periodic Server Abstraction [244–246]:** In this approach, a periodic server abstraction, e.g., one or multiple strictly periodic time slots in the TT schedule, is reserved for handling ET tasks based on a second-level fixed-priority-based scheduler. The advantage of this method is that it decouples the schedulability of ET tasks from the TT schedule generation problem. The system designer, however, has to solve the non-trivial server design problem, i.e., the problem of deriving suitable server parameters under which the ET tasks are schedulable.

  In [246], The authors propose two methods for implementing this hierarchical approach to handle sporadic tasks: simple and advanced polling. Simple polling assigns each sporadic task to its own periodic server, which, while simple, it can cause over-utilization when deadlines and periods diverge, resulting in suboptimal schedulability performance. In contrast, advanced polling involves assigning multiple sporadic tasks to the same periodic server, where schedulability testing is based on the response time of sporadic tasks. Comparative analysis reveals that the advanced polling approach yields superior schedulability results compared to the simple polling method.

- **Constraining Offline Schedule Generation [247, 248]**: In this approach, the offline slot-assignment procedure is deliberately constrained to leave unassigned slots that can later be used to service ET tasks at runtime, effectively providing execution capacity similar to periodic server abstractions.

  The method in [247] employs a list-scheduling heuristic with limited backtracking to guide TT schedule generation while improving the schedulability of ET periodic tasks. However, because tasks are allocated to strictly periodic slots, schedulability is restricted.

The *Burst Limiting Least Laxity First* (B3LF) method [248] achieves the strongest schedulability results within this category. It considers a combined TT periodic and ET sporadic task set. Periodic tasks are scheduled offline with highest priority, while sporadic tasks are assigned unique fixed priorities and executed dynamically in the free slots of the offline-generated TT schedule. Since B3LF assumes that periodic tasks have zero offsets, the worst-case arrival time for sporadic tasks is unambiguous: it occurs at the start of the longest synchronous busy period of periodic tasks (i.e., at $t = 0$). However, because such long interference from periodic tasks can significantly reduce the schedulability of sporadic tasks, the method derives a *Burst Limiting Constraint* (BLC) to intelligently distribute idle slots between TT busy periods[1].

In both cases, the structure of the TT schedule is fixed offline, meaning that new ET tasks cannot be added dynamically at runtime; any modification requires re-running the offline analysis.

4.1.2.2 Online Schedule Modification

The methods under this category dynamically adjust the allocation of time slots to tasks at runtime based on dynamic priority rules, allowing the system to guarantee and handle ET tasks without violating the guarantees provided to TT tasks.

- **Slot-Shifting:** The Slot-Shifting algorithm [122] is a well-known method in this category and operates in two distinct phases. In the offline phase, Slot-Shifting replaces the rigid TT schedule (generated as described in Section 2.1.6.6) with more flexible target execution windows for each task on each node (following the method in Section 2.1.6.7). In doing so, it simulates common knowledge of task parameters rather than common knowledge of scheduling decisions (see Section 3.2.2.2). Based on an As-Late-As-Possible (ALAP) execution of TT tasks, the algorithm computes and tabulates the spare processor capacity available within these windows.

  At runtime, each node's local scheduler maintains these spare capacities and uses them to feasibly admit arriving aperiodic tasks on slot-by-slot basis, and then assigns the slot, choosing a job from the ready queue following EDF rules.

[1]The BLC, based on real-time calculus, constrains the size and placement of TT busy periods to reposition idle slots such that the sporadic task set can be guaranteed. However, relying solely on dynamic priority algorithms (like EDF or LLF) alongside the BLC yields suboptimal schedulability, as these algorithms lack foresight regarding future task releases. The exhaustion of available burst budget (the number of consecutive slots assignable to TT tasks) due to the ASAP execution of tasks with high laxity can cause deadline misses for future tasks that have smaller laxity. To alleviate this, B3LF introduces a virtual task with dynamic laxity to the LLF algorithm; when this virtual task is selected, an idle slot is inserted into the schedule, effectively delaying the execution of periodic tasks.

Although Slot-Shifting was later extended to support sporadic task sets [125], several limitations remain. It provides no mechanism to remove or adjust guaranteed tasks, and it is tightly bound to TTA-style uniform slot structures (i.e., equally sized invocation gaps). Moreover, as analyzed in Section 4.3.2.1, its offline guarantee test for sporadic tasks pessimistically assumes that TT tasks always execute ALAP, which can significantly degrade schedulability.

Our contributions follow Slot-Shifting's principle of simulating common knowledge of task parameters but improves flexibility and addresses the above limitations without compromising the extra-functional properties of the TTA.

4.2 Dynamic Time-Triggered (DTT) Scheduling

In this section, we formally define the class of *Dynamic Time-Triggered Schedulers* (DTT) and introduce the system and task models used throughout this chapter. We then present Slot-Shifting as a representative member of this class and analyze its ability to preserve the extra-functional properties of the TTA. Finally, we introduce the Dynamic Task Set Scheduler (DTSS), our contribution within this class, which supports the admission, removal, and controlled skipping of tasks while retaining the extra-functional properties of the TTA.

4.2.1 Rationale

The foundational premise of the TTA is to shift all scheduling complexity to the design phase (see Section 3.1.1). By enforcing rigid, offline-computed scheduling tables, the TTA provides strong guarantees for determinism, temporal composability, and fault containment. However, modern real-time systems supporting safety-critical functionality such as autonomous driving, UAVs, and real-time cloud infrastructures operate in highly open and unpredictable environments. In these domains, the set of tasks the system must execute evolves continuously at runtime due to changing environmental conditions, dynamic resource provisioning, and varying application demands. This creates a fundamental tension between temporal predictability and runtime flexibility: the system must preserve rigid offline guarantees for critical tasks while accommodating the dynamic workloads required by modern applications.

Traditional methods for introducing flexibility fall short in addressing this continuous, highly dynamic behavior. Mode change methods preserve offline guarantees but inherit the same periodic model assumption for generating the schedule for each mode, making them unsuitable for efficiently supporting ET tasks except when switching between predefined TT task sets. Even then, their practicality is limited by the number

of modes that can be stored and by the latency introduced by the global agreement protocol required for switching modes consistently.

Methods that extend TT schedules locally with ET tasks also exhibit structural limitations. Offline schedule modification approaches such as periodic server abstractions or B3LF can accommodate known ET workloads, but because they ultimately freeze the schedule structure before execution, they cannot optimally handle ET tasks nor adapt to dynamic changes in the task set without re-running their offline analysis. Conversely, online schedule modification methods such as Slot-Shifting provide a step toward runtime flexibility by admitting arriving aperiodic tasks. While it can admit new aperiodic tasks, it lacks the mechanisms needed to safely remove, skip or dynamically adjust tasks once guaranteed. Moreover, its operational assumptions is limited by TTA's rigid, uniform invocation gaps and its offline guarantee test for sporadic tasks is, as we shall demonstrate, artificially constrained by pessimistic offline assumptions, leading to suboptimal schedulability results. Moreover, while it can be applied at runtime, it lacks the optimizations needed for runtime efficiency.

At present, no method within the TT paradigm fully supports dynamically changing task sets while preserving TTA's extra-functional properties. Deploying time-triggered principles in next-generation environments such as real-time clouds or autonomous vehicles, therefore requires extending these principles to support evolving task sets without compromising these properties, including fault containment and replica determinism. This motivates the formalization of *Dynamic Time-Triggered* (DTT) schedulers and the development of our *Dynamic Task Set Scheduler* (DTSS), which resolves the tension between predictability and flexibility by dynamically adjusting time triggers and integrating exact runtime feasibility analyses. It enables the system to admit, execute, skip, and remove tasks on the fly while retaining TTA's extra functional properties.

4.2.2 Dynamic Time-Triggered Schedulers

A significant advantage of the TT paradigm is its ability to resolve complex temporal constraints such as precedence relations, distributed end-to-end deadlines, and mutual exclusion, entirely during the offline schedule generation phase (as detailed in Section 2.1.6.6). Dynamic TT schedulers rely on an offline pre-processing step to preserve this architectural advantage while introducing runtime flexibility. By applying the transformation technique described in Section 2.1.6.7, complex dependent task graphs are reduced into an equivalent set of independent periodic tasks. This decoupling allows the dynamic TT scheduler to operate efficiently at runtime using simple priority rules, while implicitly upholding the complex constraints that were resolved offline.

In order to formalize the operation of dynamic TT schedulers, we first introduce their underlying temporal concepts. A *trigger* is an event that initiates an activity. When this

event is generated at a predetermined global-time instant, it is referred to as a *time-trigger* [1]. We refer to the global-time instant of a time trigger as a *scheduler point*. At each scheduler point, the runtime scheduler of a given resource is invoked by the associated time-trigger. The time interval between two successive scheduler points is the *invocation gap* (see Section3.1.6.2).

To encompass a broader class of TT schedulers that admit online flexibility, we generalize the classical notion of a TT scheduler as follows:

Definition 1. *A* time-triggered *(TT) scheduler is a runtime scheduler that selects the next real-time task to execute according to a defined set of dispatching rules. Dispatching decisions and preemptions are restricted to scheduler points, which may be* static*, i.e., determined offline as globally known instants, or* dynamic*, i.e., computed at runtime at a given scheduler point based on system state. Accordingly, the resulting invocation gaps may be* uniform *when derived from a fixed periodic clock, or* non-uniform *when their lengths vary with non-equidistant scheduler points.*

In comparison to ET schedulers whose invocation is driven by the arrival of external or asynchronous events, TT schedulers are activated exclusively at predefined scheduler points, which are determined either statically (offline) or dynamically (at runtime based on system state). Consequently, scheduling decisions occur only at these planned temporal instants, independent of when external stimuli such as message receptions actually occur. The conventional static table-driven TT scheduler of the TTA is an instantiation of this definition, characterized by static scheduler points and dispatching decisions that are fully prespecified offline.

Definition 2. *A Static Time-Triggered (STT) scheduler is a TT scheduler whose dispatching decisions are prespecified in a scheduling table constructed offline. Its scheduler points are necessarily static, and the task to execute at each scheduler point is determined prior to system operation.*

Definition 3. *A Dynamic Time-Triggered (DTT) scheduler is a TT scheduler that maintains a ready queue and selects the next real-time task to execute from it at each scheduler point according to a priority-based dispatching policy (e.g., EDF). Its scheduler points may be static, dynamic, or a combination thereof.*

A DTT scheduler can execute tasks feasibly with respect to their specified release times and deadlines, provided that the task set is schedulable under the implemented priority policy and that sufficient processor capacity is available. However, the added flexibility of online dispatching introduces sources of unpredictability that are absent in systems with STT schedulers, and the DTT scheduler must therefore implement additional mechanisms to preserve the extra-functional properties of the TTA.

In particular, the following challenges arise. First, the DTT scheduler must handle task overrun explicitly, for example by enforcing execution-time budgets and defining a policy for budget exhaustion; such handling is unnecessary in an STT scheduler where slot assignments are fixed. Second, if resource-access synchronization between tasks is not fully resolved offline (e.g., by fixing the order of resource accesses by tasks based on redefining task parameters), the DTT scheduler must implement explicit synchronization mechanisms at runtime to prevent priority inversion. Third, when a computation node is replicated for fault tolerance, all replicas must produce identical results in the same order to maintain replica determinism. This can be achieved either via online agreement on scheduling decisions (as in ET SMR protocols), or more efficiently by forcing replicas to use the same message versions assuming nondeterminism only occur due to replicas reading differing input data (see Section 3.2.2.3).

4.2.3 System Description and Task Model

We consider a distributed system of nodes consistent in structure to the TTA (see Section 3.1.2), where each node comprises a computation processor (host computer) and a communication controller. Nodes communicate with one another exclusively through time-triggered networks.

4.2.3.1 Time Model

We assume that all system nodes maintain approximately synchronized clocks, establishing a consistent notion of global time. Specifically, the difference between the local views of global time at any two correct nodes does not exceed one *macrotick* (MT), where MT denotes the granularity of global time (see Section 3.1.7.1).

Time slots for both computation and communication are constructed according to the sparse time base concept, which partitions global time into an infinite alternating sequence of *activity intervals* and *silence intervals*. Each activity interval spans exactly one MT, and each silence interval spans at least four MTs. Consistent with the TT paradigm, time triggers for global events are constrained to occur at sparse instants, established either offline or via an online agreement protocol, so that a consistent and unambiguous temporal ordering of distributed events is maintained across the system.

4.2.3.2 Communication Model

Communication controllers are responsible for satisfying the temporal requirements of messages produced by real-time tasks. We assume that each communication controller implements an STT scheduler, under which the controller is invoked at statically defined

scheduler points to initiate the transmission or delivery of messages over the shared communication channel.

The channel is accessed based on a TDMA strategy, where each message is allocated a dedicated, non-overlapping TDMA slot of predetermined length, and successive instances of the same message are transmitted at strictly periodic instants (up to the precision of the clock synchronization). This ensures that messages from different senders do not interfere with each other, and that message instances arrive at receivers in the same order in which they were sent.

Because every communication controller sharing a channel has knowledge of the TDMA schedule, each controller can filter incoming traffic and accept only those messages that arrive during their allocated TDMA slots.

We assume that all communicated messages conform to *state message* semantics: messages carry state information, read operations are non-consuming, and write operations follow write-in-place semantics, with each new value overwriting its predecessor. Furthermore, the CNI interface between each host computer and its communication controller continues to function as a *temporal firewall*, ensuring that timing variability in task execution on the host computer cannot propagate into the network domain and violate the static TDMA schedule (see Section 3.1.7.3).

4.2.3.3 Computation Model

Computation nodes are responsible for guaranteeing the temporal requirements of the real-time tasks assigned to them. Hence, they run an RTOS that can provide such guarantees. We assume that host computers implement a DTT scheduler, where at each scheduler point the scheduler initiates or terminates the execution of a given real-time task. Between two consecutive scheduler points, the selected task runs uninterrupted, and any events that arrive during this interval are not observed until the next scheduler point. A *computation slot* is formed by one or more consecutive invocation gaps allocated exclusively to a given task, providing sufficient processor time for the task to complete within its WCET (see Section 3.1.6.2).

4.2.3.4 Task Model

We assume that the system comprises a set of tasks Γ_{sys}, where each task τ_x is modeled as a sequence of instances or jobs $j_{x,y}$. A job $j_{x,y}$ is characterized by its release or arrival time $r_{x,y}$, and its worst-case execution time C_x, which is the same for all jobs of τ_x. A job $j_{x,y}$ is *deadline-constrained* if it is associated with an absolute deadline $d_{x,y}$ by which it must complete, and *deadline-free* otherwise (see Section 2.1.3). All task parameters are expressed in units of the global time granule, MT.

Although computation processors may operate at different speeds, we assume that replicated instances of the same task on different processors share identical sparse-time-based release times derived from the global time base, providing simulated common knowledge of task execution windows.

We assume that tasks communicate exclusively by message passing. Specifically, a task reads the state messages relevant to its current job at its release time $r_{x,y}$, and produces its output by its deadline $d_{x,y}$ The RTOS on host computers ensures that received messages by communication controllers are read and made available to the corresponding task at its release time. By having the same task across replicated nodes read the same message version, nondeterminism is avoided (see Section 3.2.2.3).

We assume that Γ_{sys} may contain both *independent* tasks, which have no synchronization dependencies that could block their execution, and *dependent* tasks, whose timing constraints may involve multiple nodes and messages and be subject to precedence and synchronization requirements. We assume that the temporal constraints of dependent tasks are modeled exclusively as *directed acyclic graphs* (DAGs). These DAG-based constraints are then transformed into an equivalent set of independent task parameters by deriving modified release times and deadlines to implicitly enforce precedence (following the method described in Section 2.1.6.7), after which the unified independent task model applies throughout the remainder of this chapter.

4.2.4 Slot-Shifting's Preservation of TTA Properties

In this section, we examine Slot-Shifting as a member of the DTT schedulers class, revisit its introduced flexibility mechanisms and discuss whether they are sufficient to preserve the extra-functional properties of the TTA.

4.2.4.1 Runtime operation

Slot-Shifting is a DTT scheduler that admits firm aperiodic tasks and efficiently handles soft aperiodic tasks. Because it was originally designed for TTA-based systems, it supports only *static* scheduler points with uniform invocation gaps derived from a periodic clock interrupt; the invocation gap is therefore a design-time parameter.

At runtime, Slot-Shifting applies dynamic priority rules to select tasks from the ready queue on a slot-by-slot basis. It uses EDF to schedule TT tasks and guaranteed firm aperiodic tasks, but prioritizes soft aperiodic tasks when sufficient spare processor capacity (i.e., free slots) is available to ensure that no deadlines are missed.

4.2.4.2 Handling runtime jobs

Different task types impose different execution requirements. Deadline-constrained tasks, such as periodic TT tasks and hard aperiodic tasks, may tolerate bounded delays but must ultimately meet their deadlines. In contrast, deadline-free tasks (e.g., soft aperiodic tasks) benefit from earliest possible completion under best-effort semantics.

Slot-Shifting performs scheduling locally at each node and maintains two ready queues: one for deadline-constrained jobs, ordered by deadline, and one for deadline-free jobs, organized in FIFO order.

In order to efficiently handle runtime jobs that are not part of the offline job set, Slot-Shifting maintains a dynamic view of free slots within one schedule cycle T_Γ (see Section 2.1.6.6). It partitions T_Γ into disjoint *spare-capacity intervals*, defined by job release times and deadlines, and computes the spare processor capacity within each interval under the assumption that guaranteed TT and firm aperiodic tasks execute ALAP. This assumption positions free slots as early as possible in the schedule. Using this information, Slot-Shifting can admit deadline-constrained jobs with deadlines shorter than T_Γ, or execute deadline-free jobs in the available free slots. The admission test and spare-capacity maintenance mechanisms are described in [122, 125].

Algorithm 2 provides a simplified pseudo-code for the main routine of Slot-Shifting's runtime scheduler. At the beginning of each slot (i.e., at a scheduler point), the scheduler enqueues newly arrived deadline-free jobs into the FIFO queue. For each newly released deadline-constrained job j_{dc}, it evaluates feasibility based on the number of free slots available before j_{dc}'s deadline. If the job is feasible, Slot-Shifting recomputes the spare-capacity intervals between its release time and deadline, potentially across the entire T_Γ. If the test fails, the job is rejected, as Slot-Shifting provides no mechanisms (e.g., job skipping or abortion) to free processor capacity.

Next, the scheduler checks whether deadline-free jobs are ready and whether the current spare-capacity interval still has free slots. If so, it executes the first FIFO job. Otherwise, it selects a ready deadline-constrained job using EDF, or executes the idle task if both queues are empty. After selecting a job, the scheduler updates the spare capacity of the affected intervals to reflect the actual number of remaining free slots.

4.2.4.3 Effects on TT's extra-functional properties

Because Slot-Shifting introduces runtime flexibility into the otherwise rigid TT execution model, it is essential to understand how this flexibility affects the extra-functional properties of the TTA. In the following, we discuss the impact of Slot-Shifting's flexibility on these properties.

Algorithm 2 Pseudo-code for the primary routine of Slot-Shifting's runtime scheduler.

```
1:  procedure OnSchedulerInvocation
2:                                   ▷ Process all newly arrived deadline-free jobs
3:      for all newly arrived j_df do
4:          Push j_df to deadline-free queue
5:      end for
6:                                   ▷ Process all newly arrived deadline-constrained jobs
7:      for all newly arrived j_dc do
8:          if IsAdmissible(j_dc) then
9:              Add j_dc to deadline-constrained queue
10:             UpdateSpareCapacityIntervals(r_dc, d_dc)
11:         end if
12:     end for
13:                                                  ▷ Scheduling decision logic
14:     if deadline-free queue is not empty and sc_current > 0 then
15:         schedule(first job from deadline-free FIFO queue)
16:     else if deadline-constrained queue is not empty then
17:         scheduleEDF(ready deadline-constrained jobs)
18:     else
19:         schedule(idle task)
20:     end if
21:                        ▷ Runtime update of spare capacity based on scheduling decision
22:     UpdateSpareCapacityRuntime
23: end procedure
```

- **Temporal determinism and predictability.** Compared to the fixed slot assignment in offline schedules, where slot-level determinism is guaranteed, Slot-Shifting relaxes this property to *execution window-level* determinism. That is, jobs are guaranteed to execute within their designated target windows and to complete predictably by the end of them. This follows directly from relaxing common knowledge of scheduling decisions to common knowledge of task parameters.

- **Replica determinism.**

  Preserving deterministic behavior among actively replicated nodes requires that they be initialized with the same state, process identical inputs, and generate identical outputs in the same order within a specified time interval (see Section 3.2.2.3). Slot-Shifting does not modify the TT communication schedule and therefore does not alter message ordering or versioning. Its role is to ensure that tasks on replicas meet their deadlines and thus produce identical outputs, assuming the operating system consistently reads inputs and writes outputs at each task's release and deadline. Consequently, replica determinism remains primarily the responsibility of the communication system, which must deliver identical input messages and

enforce output timing according to the TT communication schedule.

- **Fault and error containment.** Slot-Shifting preserves temporal isolation through slot-based execution: once a job is assigned to a slot, it executes without interruption until the slot ends. However, because it lacks a budget-tracking mechanism, tasks that exceed their assumed WCET may consume more slots than expected, potentially causing deadline violations and propagating errors, particularly if the misbehaving task is a guaranteed TT or firm aperiodic task. The absence of tracking also complicates fault attribution, reducing diagnosability and traceability. Soft aperiodic tasks cannot compromise TT or firm aperiodic tasks, but they may monopolize spare capacity and starve other soft aperiodic tasks queued in FIFO order.

- **Meeting complex timing constraints.** Slot-Shifting assumes that synchronization and resource-access constraints are represented as precedence relations and fully resolved during offline scheduling. As long as the runtime scheduler executes jobs within their designated execution windows, these complex timing constraints remain satisfied without requiring explicit awareness of them at runtime.

4.2.5 Dynamic Task Set Scheduler (DTSS)

The *Dynamic Task Set Scheduler (DTSS)* is a DTT scheduler that leverages the EDF policy for the dynamic allocation of slots to ready tasks. Compared to Slot-Shifting, DTSS offers enhanced runtime flexibility and improved fault containment through the following mechanisms:

1. **Dynamic Task Set**: DTSS adapts to changing environments by supporting the runtime admission, removal, and skipping of tasks. These modifications rely on the feasibility tests developed in Section 4.3, which provide the admission control needed to safely integrate aperiodic, sporadic and periodic workloads without violating offline guarantees.

2. **Dynamic Slot Size**: DTSS permits the duration of the computation slot, and consequently the invocation gap, to be defined dynamically in accordance with the execution state of tasks. This capability reduces context-switching overhead and enables more efficient reclamation of freed resources.

3. **Budget-Constrained Execution**: DTSS ensures that all tasks are bounded by their specified WCET. In contrast, Slot-Shifting enforces such constraints only for soft aperiodic tasks, which may allow overruns by guaranteed tasks to propagate interference and potentially result in deadline violations.

4.2.5.1 Runtime operation

DTSS is a slot-driven dynamic-priority-based algorithm, where at the beginning of each slot (i.e., scheduler point) on a given node, the scheduler dynamically assigns a ready job to the current slot according to EDF. With respect to mechanism of invoking the scheduler, DTSS can be configured to operate in one of the following two runtime modes:

1. **Static mode**: DTSS is invoked, as in purely TT dispatchers [19] and Slot-Shifting [122, 125], according to periodic clock interrupt, i.e., invocation gaps are uniform and defined before runtime. In this case, freed processor capacity due to early job completion is wasted until the next scheduler point. Freed future slots intended for the job's WCET are reclaimed.

2. **Dynamic mode**: DTSS calculates the next scheduler point dynamically, assigning the job to a computation slot, whose duration depends on the timing parameters of jobs in the job set. Precisely, the scheduler invocation gap is chosen such that it accommodates the remaining execution requirement of the job, or allows it to execute until the release of a job with an earlier deadline, whichever is smaller. Hence, only in the second case, the job execution is preempted. Furthermore, in the dynamic mode, the scheduled job can yield to the scheduler. Hence, an early job completion ends the job's slot prematurely, reclaiming freed processor capacity.

Note that DTSS preserves temporal isolation in both of its operational modes: once a job is assigned to a slot, it executes without interruption until the slot's duration has elapsed.

4.2.5.2 Handling Aperiodic jobs

Similar to Slot-Shifting, DTSS employs two distinct queues to accommodate the differing execution requirements of deadline-constrained and deadline-free jobs. The former is prioritized according to job deadlines, whereas the latter follows a FIFO order.

DTSS maintains a look-ahead *Earliest Deadline as Late as possible* (EDL) schedule for the job set to efficiently handle runtime jobs. This schedule is represented as an ordered list of busy-period endpoints (*EDLBP* deadlines), denoted by $\mathfrak{D}_{bp}$, spanning the length of the scheduling cycle T_Γ. EDL, introduced as a variant of EDF in [93], differs from the classical policy by executing ready tasks as late as possible rather than immediately. The resulting list of EDLBP deadlines facilitates the localization of available slack and thus simplifies task set modification (see Sec. 4.3.1.3).

4.2.5.2.1 Handling Deadline-Constrained Aperiodic Jobs

The procedure for handling deadline-constrained jobs in DTSS closely parallels that of Slot-Shifting, but incorporates job removal and skipping as an additional mechanism to enhance admission feasibility. The pseudo-code for the primary routine of DTSS is shown in Algorithm 3.

First, DTSS verifies whether the most recently executed job j_r differs from the job associated with the earliest deadline in the maintained deadline list $\mathfrak{D}_\mathfrak{J}$, i.e., indicating that j_r was executed in non-EDL order. If this condition holds, DTSS invokes the procedure in Algorithm 6 on j_r. This step guarantees that $\mathfrak{D}_{bp}$ accurately reflects all EDLBP deadlines in the look-ahead EDL schedule for the current job set.

If a deadline-constrained job j_{dc} is awaiting admission, Algorithm 5 is applied. If the job fails the test, DTSS checks whether the amount of execution time that can be freed through skipping jobs within j_{dc}'s execution window allows it to be guaranteed. In other words, assuming j_{dc} fails the admission test because the remaining processor capacity within its execution window is equal to $-c$, DTSS checks whether removing the execution requirement by skippable jobs within $[r_{dc}, d_{dc}]$ frees up c amount of processor time. If so, the execution requirement by the skipped jobs is removed and the list $\mathfrak{D}_{bp}$ is updated by executing Algorithm 6. Afterwards, j_{dc} is added to the deadline-constrained queue and the next ready task is chosen following the EDF policy.

DTSS leaves unspecified the precise mechanisms for selecting which jobs must be skipped and in what order simultaneously arriving hard aperiodic jobs are handled, while highlighting the possibility of implementing optimizations to avoid redundant steps. For instance, when two jobs with identical deadlines arrive during the same slot, the condition in line 27 of Algorithm 5 can be modified to consider the worst-case execution requirement by the jobs simultaneously.

Furthermore, when the scheduler is required to handle jobs whose relative deadlines exceed the duration of the scheduling cycle T_Γ, the maintained list of look-ahead EDLBP deadlines must be correspondingly extended into the future.

4.2.5.2.2 Handling Deadline-Free Aperiodic Jobs

Deadline-free jobs (e.g., soft aperiodic or non-real-time jobs) are handled by a server task whose deadline d_{sas} is calculated such that these jobs finish as early as possible and receive optimal response times. The server deadline d_{sas} is postponed depending on the availability of remaining processor capacity in the TEWs of deadline-constrained jobs. It gets updated whenever one of three conditions is satisfied: 1) the last executing job is not a deadline-free job; 2) new deadline-constrained jobs are guaranteed; 3) d_{sas} is reached, i.e., the remaining processor capacity before the current EDLBP deadline is exhausted.

Algorithm 3 Pseudo-code for the primary routine of DTSS.

```
1: procedure OnSchedulerInvocation
2:                                   ▷ Process all newly arrived deadline-free jobs
3:     for all newly arrived j_df do
4:         Push j_df to deadline-free queue
5:     end for
6:                                                  ▷ Non-EDL execution update
7:     if last job j_r executed in non-EDL order then
8:         UpdateBPdeadlines(j_r)                                  ▷ Acc. to Alg.6
9:     end if
10:                          ▷ Process all newly arrived deadline-constrained jobs
11:    for all newly arrived j_dc do
12:        if IsGuaranteed(j_dc) then                              ▷ Acc. to Alg. 5
13:            Push j_dc to deadline-constrained queue
14:        else if IsGuaranteedWithSkipping(j_dc) then
15:            UpdateBPdeadlines(on skipped jobs)                  ▷ Acc. to Alg. 6
16:            Push j_dc to deadline-constrained queue
17:        end if
18:    end for
19:                          ▷ Update deadline of server task for deadline-free jobs
20:    if conditions to update d_sas met then   ▷ The three conditions are listed in
   Sec. 4.2.5.2.2
21:        UpdateServerDeadline(d_sas)          ▷ Update d_sas according to Eq. 4.2.1
22:    end if
23:                                              ▷ Scheduling Decision Logic
24:    if deadline-free queue is not empty or deadline-constrained queue is not empty
   then
25:        scheduleEDF(ready deadline-constrained jobs, server task for deadline-free
   jobs)
26:    else
27:        schedule(idle task)
28:    end if
29: end procedure
```

The first condition ensures that any freed or moved processor capacity is reclaimed (e.g., when jobs finish executing within less time compared to their WCET). The second and third postpone d_{sas} to the closest time with available remaining processor capacity. i.e., d_{sas} is calculated based on the earliest EDLBP with a remaining processor capacity value larger than zero. Let d_f^{bp} be the end of that busy period, t_{now} the current time, and $\mathfrak{D}_{bp}$ be the ordered list of EDLBP deadlines. Equation 4.2.1 shows how to calculate d_{sas}.

$$d_{sas} = \mathsf{max}(t_{now}, d_{f-1}^{bp}) + rpc(t_{now}, d_f^{bp}) \tag{4.2.1}$$

Interestingly, delaying deadline-constrained jobs to service deadline-free ones as soon as possible increases the likelihood that the delayed jobs execute in EDL order. Hence, the likelihood of executing Algorithm 6 is reduced.

4.2.5.3 Effects on TT's extra-functional properties

DTSS provides the same level of temporal determinism as Slot-Shifting. In addition, it follows the same approach and assumptions for meeting complex timing constraints and supporting replica determinism (see Section 4.2.4.3). At the same time, it offers improved fault containment by enforcing strict control over the allocation of processor capacity to individual jobs. This is achieved by continuously tracking each job's remaining execution requirement and ensuring that it never exceeds its worst-case execution time.

4.3 Task Set Extension Feasibility Analyses

The predefined set of TT tasks rarely exhausts the computational resources available on each node, leaving residual spare capacity that can be strategically exploited to service runtime tasks. Consequently, whenever the baseline TT task set must be extended with additional tasks at runtime, the scheduler must perform a feasibility analysis. This analysis determines whether the new computational demand can be safely admitted without jeopardizing the hard deadlines of the already guaranteed tasks.

In this section, we present our exact feasibility analyses for extending TT task sets with aperiodic, sporadic, and periodic tasks. Throughout these analyses, we assume that all dependent TT tasks have already been transformed into an equivalent set of independent, asynchronous periodic tasks on each node, utilizing the methodology detailed in Section 2.1.6.7. This transformed baseline set, combined with the newly arriving tasks to be admitted, constitutes the formal input task set evaluated by the feasibility analyses introduced herein. Furthermore, given the optimality of the EDF algorithm on

preemptive uniprocessors and its central role in our DTSS solution, we assume that EDF priority rules govern runtime execution on top of TT activation.

Note that the feasibility tests introduced in this section evaluate schedulability from the perspective of a single, local node. If deployed to admit replicated tasks, e.g., within an FTU, additional operational rules are necessary to preserve replica determinism. Because independent dynamic behaviors across replicas (e.g., such as varying processor speeds or slack reclaimed from early job completions) inevitably cause local variations in available processor capacity, relying on this locally reclaimed slack for joint admission decisions would cause state divergence. Furthermore, because asynchronous task admission requests originate outside the computer's sphere of control, they must first be subjected to a global online agreement protocol to ensure consistent observation across all FTU nodes. Therefore, the admission of replicated tasks must be evaluated on the basis of a globally established state.

4.3.1 Extending Time-Triggered Task Sets with Aperiodic Tasks

This section investigates the following real-time feasibility analysis problem:

> *Given a task system comprising a collection of asynchronous periodic tasks that is known to be feasible upon a preemptive uniprocessor platform, is feasibility preserved upon adding an aperiodic task to the task system?*

For ease of presentation and to establish a formal baseline, we assume that the periodic task set on each node is unrolled into a set of N_j independent jobs, denoted as $\mathfrak{J}_p = \{j_0, \ldots, j_{N_j-1}\}$, which covers the periodic pattern of task arrivals up to the end of the first hyperperiod. Because this periodic task set originates from TT tasks, the periods are typically engineered (e.g., as harmonic periods) to bound the hyperperiod (and consequently the length of the TT schedule), preventing a combinatorial explosion. Furthermore, the unrolling of these periodic tasks into $\mathfrak{J}_p$ can take place entirely offline as a pre-computation step.

Each job $j_x \in \mathfrak{J}_p$ is characterized by a 3-tuple (r_x, C_x, d_x), where r_x represents the release time of j_x, C_x denotes its WCET, and d_x specifies its absolute deadline (as established in Section 2.1.3). Consequently, we evaluate the schedulability of the combined job set $\mathfrak{J} = \mathfrak{J}_p \cup \{j_a = (r_a, C_a, d_a)\}$ upon a preemptive uniprocessor platform, where j_a is the dynamically arriving aperiodic job.

4.3.1.1 Revisiting Slot-Shifting's Feasibility Test

Because the admission test for aperiodic tasks operates online, keeping its computational complexity low is paramount to avoid interfering with the execution of application

tasks. In order to minimize this runtime overhead, the Slot-Shifting algorithm calculates and maintains an ASAP view of spare processor capacity by assuming an ALAP execution model for periodic tasks. To achieve this, the algorithm partitions the schedule cycle into a sequence of *disjoint* spare capacity intervals, indexed from 0. These intervals are derived directly from job parameters and are explicitly annotated with their available spare processor capacity.

Formally, let $\mathfrak{D} = \{d_0, \ldots, d_{N_{dl}-1}\}$ denote the ordered set of unique job deadlines within the job set $\mathfrak{J}$. A spare capacity interval I_x is generated for each deadline d_x as follows:

$$\begin{aligned} end(I_x) &\stackrel{\text{def}}{=} d_x, \\ start(I_x) &\stackrel{\text{def}}{=} \max\left(\min_{\forall j_i | d_i = d_x}(r_i), end(I_{x-1})\right) \end{aligned} \tag{4.3.1}$$

Thus, the end of an interval I_x is defined by the deadline d_x, while its start is determined by either the earliest release time among all jobs sharing that deadline or the end of the preceding interval I_{x-1}, whichever is larger.

Applying Equation (4.3.1) across all job deadlines generates *at least* N_{dl} spare capacity intervals; depending on specific job parameters, additional empty intervals (containing no assigned jobs) may also be created. Once established, each interval I_x is annotated with its available spare capacity, denoted $SC(I_x)$ and expressed in multiples of the invocation gap. This capacity is computed as the difference between the interval's length $|I_x| \stackrel{\text{def}}{=} end(I_x) - start(I_x)$ and the cumulative execution demand of all jobs sharing the deadline $d_x = end(I_x)$, yielding $SC(I_x) \stackrel{\text{def}}{=} |I_x| - \sum_{\forall j_i | d_i = d_x} C_i$.

However, the cumulative execution demand within an interval I_x may exceed its length $|I_x|$. When this occurs, the calculated spare capacity becomes negative, indicating that the jobs assigned to I_x must borrow processor capacity from preceding intervals. To illustrate this capacity-borrowing phenomenon, consider an example job set comprising $j_0(r_0 = 0, C_0 = 2, d_0 = 4)$ and $j_1(r_1 = 0, C_1 = 4, d_1 = 7)$. The resulting capacity intervals are detailed in Table 4.3.1.

As shown, the computational demand of j_1 cannot be fully accommodated within I_1, yielding a negative spare capacity of $SC(I_1) = -1$. Consequently, the excess demand is shifted backward into I_0, reducing its effective spare capacity from 2 to 1. This demonstrates that computing the true spare capacity for any interval I_x must incorporate the execution demand of jobs with later deadlines that are forced to execute within I_x.

Table 4.3.1: Spare capacity intervals for two jobs: $j_0(r_0 = 0, C_0 = 2, d_0 = 4)$ and $j_1(r_1 = 0, C_1 = 4, d_1 = 7)$.

Interval I_x	$start(I_x)$	$end(I_x)$	$\sum_{\forall j_i \mid d_i = d_x} C_i$	$SC(I_x)$
I_0	0	4	2	1
I_1	4	7	4	-1

By refining the initial formulation to explicitly account for capacity borrowing from succeeding intervals, the effective spare capacity of I_x is recursively defined as:

$$SC(I_x) \stackrel{\text{def}}{=} end(I_x) - start(I_x) - \sum_{\forall j_i \mid d_i = d_x} C_i + \min(0, SC(I_{x+1})) \tag{4.3.2}$$

This recursive dependency necessitates computing spare capacities in reverse chronological order, beginning with the final interval ending at $d_{N_{dl}-1}$.

During runtime, the *instantaneous spare capacity* of an interval I_x at time t, denoted $SC(I_x, t)$, provides a dynamic view of the available capacity, accounting for task executions that have already occurred prior to t. Depending on the actual execution order, $SC(I_x, t)$ may be smaller than, equal to, or larger than the statically computed offline value $SC(I_x)$. For a more detailed treatment of spare capacity maintenance rules in Slot-Shifting, we refer the reader to [122,125].

With the spare capacity intervals established, the online admission test for a newly arrived aperiodic job $j_a(r_a, C_a, d_a)$ verifies whether the aggregated positive spare capacity, spanning from the current interval I_c (where the job is released) to the last interval I_l (which contains its deadline), is sufficient to accommodate C_a. This is reflected in the following inequality:

$$SC(I_{c,t}) + \sum_{x=c+1}^{l-1} \max(0, SC(I_x)) + \min(SC(I_l), d_a - start(I_l)) \geq C_a \tag{4.3.3}$$

where $SC(I_{c,t})$ represents the instantaneous spare capacity of the current interval I_c at time $t = r_a$, the summation term $\sum_{x=c+1}^{l-1} \max(0, SC(I_x))$ accumulates the positive spare capacity of all fully encompassed intervals between t and d_a, and the final term $\min(SC(I_l), d_a - start(I_l))$ calculates the exact spare capacity accessible to j_a within the last interval I_l.

4.3.1.1.1 Discussion and Optimizations

While the admission test provided by Slot-Shifting is highly efficient and executes in linear time, it is structurally constrained to TTA's model of uniform invocation gaps. Consequently, this test cannot be directly applied to DTT schedulers with nonuniform invocation gaps, such as our DTSS algorithm. Furthermore, the reliance on negative

spare capacity annotations, capacity borrowing, and the resulting cascaded capacity updates post admission makes the approach somewhat involved. These insights motivate the design of a generalized and invocation-gap-independent feasibility test that can be integrated with any DTT scheduler, including our DTSS from Section 4.2.5.

Beyond the requirement for nonuniform invocation gap support, an analysis of the interval creation logic defined in Equation (4.3.1) reveals an opportunity for optimization. Determining the start of an interval based on job release times is unnecessary for the analysis and increases memory overhead by frequently creating more intervals than the number of unique job deadlines. Therefore, the definition of spare capacity intervals can be amended to resolve this inefficiency as follows:

$$\begin{aligned} end(I_x) &\stackrel{\text{def}}{=} d_x, \\ start(I_x) &\stackrel{\text{def}}{=} \begin{cases} end(I_{x-1}) & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases} \end{aligned} \tag{4.3.4}$$

By applying Equation (4.3.4), spare capacity intervals are created based on the set of unique job deadlines alone. This ensures that exactly N_{dl} intervals are generated, meaning the number of spare capacity intervals matches the number of job deadlines and no empty intervals with no assigned jobs are created.

4.3.1.2 Processor Demand Analysis for Job Set Extension

As established in Section 2.1.6.5, the Processor Demand Analysis (PDA) presents a necessary and sufficient schedulability test for real-time periodic or sporadic tasks scheduled under preemptive EDF on uniprocessor systems.

While Equation (2.1.4) describes PDA's demand bound function for recurrent tasks, it can be easily rewritten for the context of job sets as follows:

$$\text{dbf}\,(t_1, t_2) \stackrel{\text{def}}{=} \sum_{\forall j_k \in \mathfrak{J} | r_k \geq t_1 \wedge d_k \leq t_2} C_k \tag{4.3.5}$$

dbf (t_1, t_2) calculates for a given interval $[t_1, t_2]$ the accumulative amount of processing time of all jobs in $\mathfrak{J}$ that are released and have deadlines within that interval.

Let $\mathfrak{R} = \{r_0, \ldots, r_{N_r-1}\}$ and $\mathfrak{D} = \{d_0, \ldots, d_{N_{dl}-1}\}$ be ordered sets of release times and deadlines of all jobs in the schedule cycle T_Γ, where N_r and N_{dl} are the size of the sets $\mathfrak{R}$ and $\mathfrak{D}$, respectively. The feasibility of $\mathfrak{J}$ can be checked using the PDA as per Theorem 1.

Theorem 1. *A job set $\mathfrak{J}$ with release times $\in \mathfrak{R}$ and deadlines $\in \mathfrak{D}$ is feasible on a*

preemptive uniprocessor system if and only if:

$$\forall r_x \in \mathfrak{R} \wedge \forall d_w \in \mathfrak{D} |\ r_x < d_w,\ \text{dbf}\,(r_x, d_w) \leq d_w - r_x \tag{4.3.6}$$

If we want to extend the job set $\mathfrak{J}$ with a real-time job j_a during runtime, then applying Theorem 1 on the newly formed job set $\mathfrak{J}_{new} = \{\mathfrak{J} \cup j_a\}$ is a valid option. However, naively evaluating the set of constraints checked by Theorem 1 grows in a quadratic manner with respect to the total number of jobs in the newly formed set. The incurred overhead is especially worsened if the analysis must be repeated iteratively, e.g., after selectively skipping existing jobs to gracefully degrade the system and feasibly schedule j_a.

Methods were proposed to cut down the number of unimportant constraints [113,249]. However, these methods assume a restricted model for the tasks, e.g., periodic tasks released synchronously and have constrained deadlines. Hence, they are not directly applicable to the more general task models or specifically tailored for runtime usage.

As an example, consider the feasible job set in Table 4.3.2, which comprises the 6 jobs $j_0, \ldots, j_5$. An additional job $j_a < r_a = 14, d_a = 19, C_a = 5 >$ is to be feasibly added to the job set according to Theorem 1. The steps performed by the analysis are illustrated in Table 4.3.2, where the processor demand within each valid interval, formed by a release time and succeeding deadline combination, is checked for not being larger than the length of the interval. In the analysis, the check fails for the interval $[14, 20]$, i.e., it is not feasible to extend $\mathfrak{J}$ in Table 4.3.2 with j_a without causing deadline misses.

Table 4.3.2: Example job set $\mathfrak{J} = \{j_0, \ldots, j_5\}$ and the processor demand analysis for it when extended with $j_a < r_a = 14, d_a = 19, C_a = 5 >$, showing for each release time r_i and deadline d_i, dbf (r_i, d_i) (left) and $d_i - r_i$ (right).

j_i	j_0	j_1	j_2	j_3	j_4	j_5
r_i	1	0	4	6	12	16
d_i	6	8	11	12	18	20
C_i	2	2	4	1	2	2

r_i/d_i	$d_0 = 6$	$d_1 = 8$	$d_2 = 11$	$d_3 = 12$	$d_4 = 18$	$d_a = 19$	$d_5 = 20$
$r_1 = 0$	$2 \leq 6$	$4 \leq 8$	$8 \leq 11$	$9 \leq 12$	$11 \leq 18$	$16 \leq 19$	$18 \leq 20$
$r_0 = 1$	$2 \leq 5$	$2 \leq 7$	$6 \leq 10$	$7 \leq 11$	$9 \leq 17$	$14 \leq 18$	$16 \leq 19$
$r_2 = 4$	$0 \leq 2$	$0 \leq 4$	$4 \leq 7$	$5 \leq 8$	$7 \leq 14$	$12 \leq 15$	$14 \leq 16$
$r_3 = 6$	-	$0 \leq 2$	$0 \leq 5$	$1 \leq 6$	$3 \leq 12$	$8 \leq 13$	$10 \leq 14$
$r_4 = 12$	-	-	-	-	$2 \leq 6$	$7 \leq 7$	$9 \leq 8$
$r_a = 14$	-	-	-	-	-	$5 \leq 5$	$\boxed{7 > 6}$
$r_5 = 16$	-	-	-	-	-	-	$2 \leq 4$

If the system is able to tolerate that j_5 be skipped with a bounded rate (e.g., some

control plants tolerate such behavior [250,251]), then enough resources are freed to admit j_a. However, Theorem 1 must be reapplied on the job set without j_5 and considering all intervals.

4.3.1.3 The Remaining Processor Capacity Analysis (RPCA)

In this section, we detail the *Remaining Processor Capacity Analysis* (RPCA), our runtime-tailored feasibility test job set extension. RPCA eliminates the capacity-borrowing effect inherent in Slot-Shifting by instead partitioning the schedule cycle according to the endpoints of EDL busy periods. Furthermore, because it is based on the PDA, the test retains generality and remains independent of invocation gaps. Importantly, RPCA achieves linear computational complexity by evaluating only the minimal set of constraints necessary to verify feasibility.

Definition 4. *An earliest deadline as late as possible (EDL) schedule for a feasible job set $\mathfrak{J}$ corresponds to maximally delayed job executions such that all deadlines are met. An EDL schedule thus comprises a sequence of N_{bp} EDL busy periods (EDLBPs), $bp_0, bp_1, \ldots$, and $bp_{N_{bp}-1}$, each ending at $d_0^{bp}, d_1^{bp}, \ldots, d_{N_{bp}-1}^{bp}$, and where any two consecutive EDLBPs are separated by a non-zero interval of processor idle time.*

A job deadline that marks the end of an EDLBP is referred to as an *EDLBP deadline*, and the set of all such deadlines is denoted by $\mathfrak{D}_{bp}$. An execution order of jobs is said to be *in EDL order* if it matches the order of execution in the EDL schedule for the job set; otherwise, it is *in non-EDL order*.

In order to characterize the unused processor capacity relative to the current execution state, the following definitions are introduced:

Definition 5. *The remaining execution requirement $c_{j_x}(t)$ of a job j_x is the remaining computation time that j_x needs at time t to finish its computation, and where the following statement always holds: $\forall t \in \mathbb{R}^+,\ 0 \leq c_{j_x}(t) \leq C_x \wedge c_{j_x}(r_x) = C_x$*

Definition 6. *The remaining processor demand function $rpd(t_1, t_2)$ for a set of real-time jobs and an interval $[t_1, t_2]$, is the total amount of remaining execution requirement at t_1 of the jobs that have deadlines within $[t_1, t_2]$, i.e.,:*

$$rpd(t_1, t_2) \stackrel{def}{=} \sum_{\forall j_k | t_1 \leq d_k \leq t_2} c_{j_k}(t_1) \tag{4.3.7}$$

Note that relative to t_1, only those jobs with release time $r < t_1$ can have remaining execution requirement smaller than their worst-case execution time.

Definition 7. *The remaining processor capacity (RPC) function* $rpc(t_1, t_2)$ *for a set of real-time jobs and an interval* $[t_1, t_2]$*, is the difference between the length of the interval* $[t_1, t_2]$ *and* $rpd(t_1, t_2)$ *at that interval, i.e.,:*

$$rpc(t_1, t_2) \stackrel{def}{=} t_2 - t_1 - rpd(t_1, t_2) \tag{4.3.8}$$

The following lemmas establish key properties of RPC within an EDL schedule and demonstrate why EDLBP deadlines provide an ideal reference for accurately tracking the RPC during runtime.

Lemma 1. *Given any EDLBP* bp_x *starting at time* t *and holding a set of ordered deadlines* $\mathfrak{D}_{bp_x} = \{d_i, d_j, \ldots, d_w, d_x^{bp}\}$*, the following inequality always holds:*

$$rpc(t, d_i) \geq rpc(t, d_j) \geq \cdots \geq rpc(t, d_w) \geq rpc(t, d_x^{bp}) \tag{4.3.9}$$

Proof. Let Q_x be the remaining execution requirement of all jobs with deadline d_x^{bp}, which can be calculated as follows: $Q_x = \sum_{\forall j_k \mid d_k = d_x^{bp}} c_{j_k}(t) = rpd(t, d_x^{bp}) - rpd(t, d_w)$. Since the processor is continuously busy throughout bp_x, i.e., there is no idle time in the interval $[t, d_x^{bp}]$, we write:

$$\begin{aligned} Q_x = rpd(t, d_x^{bp}) - rpd(t, d_w) &\geq d_x^{bp} - d_w \\ rpd(t, d_x^{bp}) - rpd(t, d_w) - t &\geq d_x^{bp} - d_w - t \\ d_w - t - rpd(t, d_w) &\geq d_x^{bp} - t - rpd(t, d_x^{bp}) \\ rpc(t, d_w) &\geq rpc(t, d_x^{bp}) \quad \text{[By using Equation (4.3.8)]} \end{aligned}$$

The same result is derived for any two successive deadlines within bp_x, giving us Equation (4.3.9). □

Lemma 2. *For a feasible job set* $\mathfrak{J}$*, the RPC function in Definition 7 calculates the amount of time the processor spends idle in the interval* $[t_1, t_2]$ *if* $t_1, t_2 \in \{0, d_0^{bp}, d_1^{bp}, \ldots, d_{N_{bp}-1}^{bp}\}$*, i.e., the considered interval is delimited by the ends of busy periods in the EDL schedule for* $\mathfrak{J}$*.*

Proof. Consecutive EDL busy periods are separated by a non-zero-length interval, during which the processor idles. Otherwise, by definition, there is a single busy period. The amount of idle time equals the length of the interval, delimited by the considered busy period ends, minus the RPD of jobs with deadlines contained within that interval, which is exactly what the RPC function calculates in Eq (4.3.8).

Since the RPD and RPC functions are cumulative, the same argument is valid for any interval $[t_1, t_2]$, where $t_1, t_2 \in \{0, d_0^{bp}, d_1^{bp}, \ldots, d_{N_{bp}-1}^{bp}\}$. □

Lemma 3. *Given the ordered set of EDLBP deadlines* $\mathfrak{D}_{bp} = \{d_0^{bp}, d_1^{bp}, \ldots, d_{N_{bp}-1}^{bp}\}$*, for any feasible job set* $\mathfrak{J}$*, the following inequality always holds:*

$$rpc(0, d_0^{bp}) < rpc(0, d_1^{bp}) < \cdots < rpc(0, d_{N_{bp}-1}^{bp}) \tag{4.3.10}$$

Proof. Considering the first two EDLBPs, we can, according to Lemma 2, write:

$$\begin{aligned}
&rpc(d_0^{bp}, d_1^{bp}) = d_1^{bp} - d_0^{bp} - rpd(d_0^{bp}, d_1^{bp}) > 0 \\
&rpc(d_0^{bp}, d_1^{bp}) = d_1^{bp} - d_0^{bp} - rpd(0, d_1^{bp}) + rpd(0, d_0^{bp}) > 0 \\
&=> d_0^{bp} - rpd(0, d_0^{bp}) < d_1^{bp} - rpd(0, d_1^{bp}) \\
&=> rpc(0, d_0^{bp}) < rpc(0, d_1^{bp})
\end{aligned}$$

By transitivity, the same result holds for any two successive EDLBPs, giving us Eq (4.3.10). □

4.3.1.4 Job set extension test

The RPCA explicitly identifies which release times and deadlines are critical for determining the feasibility of extending a job set. In the following, we will present which releases and deadlines can be skipped progressively to aid clarity and build intuition.

Lemma 4 demonstrates that when extending a job set by a job j_x, it is not necessary to consider intervals for the analysis that end at deadlines smaller than j_x's deadline, d_x.

Lemma 4. *A schedulable job set* $\mathfrak{J}$ *under a preemptive EDF policy, is feasibly extensible by an additional job* $j_x < r_x, d_x, C_x >$ *if and only if:*

$$\forall r_i \in \mathfrak{R} \wedge \forall d_j \in \mathfrak{D} \,|\, r_i < d_j \wedge d_j \geq d_x, \;\; rpd(r_i, d_j) \leq d_j - r_i \tag{4.3.11}$$

where $\mathfrak{R}$ *and* $\mathfrak{D}$ *are the sets of release times and deadlines of all jobs in* $\mathfrak{J}$*, including* j_x*'s.*

Proof. $rpd()$ is cumulative at job deadlines. Hence, by adding j_x to $\mathfrak{J}$, we get: $\forall d_j \in \mathfrak{D} \wedge d_j \geq d_x, rpd(r, d_j) = rpd(r, d_j) + C_x$. Since the demand value at all intervals ending before d_x remains unchanged and since the $\mathfrak{J}$ is assumed to be initially schedulable, it is sufficient to perform the analysis only over the affected intervals as per Equation (4.3.11). □

Lemma 5 uses the $rpd()$ function to further reduce the number of considered release times in Lemma 4 to a single release time (the added job release time).

Lemma 5. *A schedulable job set $\mathfrak{J}$ under a preemptive EDF policy, is feasibly extensible by an additional job $j_x < r_x, d_x, C_x >$ if and only if:*

$$\forall d_j \in \mathfrak{D} \wedge d_j \geq d_x,\ rpd(r_x, d_j) \leq d_j - r_x \tag{4.3.12}$$

where $\mathfrak{D}$ is the set of deadlines of all jobs in $\mathfrak{J}$ and of the additional job j_x.

Proof. Consider the two intervals $I_w = [r_w, d_e]$ and $I_x = [r_x, d_e]$, where $r_w < r_x$ and $d_e \geq d_x$. Since the execution of j_x may only affect the execution of tasks after it has been released, a scenario where $rpd(r_w, d_e) > d_e - r_w$ but $rpd(r_x, d_e) \leq d_e - r_x$ can only happen if deadline misses occur before r_x. Furthermore, since $rpd(r_x, d_e)$ considers the remaining execution requirements (at r_x) by all jobs with deadlines in the interval I_x (even if jobs are released before r_x), any deadlines misses within I_x are detected. Consequently, the deadline misses happen in the interval $[r_w, r_x]$ and cannot be attributed to $\mathfrak{J}$'s extension, contradicting the assumption that $\mathfrak{J}$ is initially feasible.

Now, consider a third interval $I_z = [r_z, d_e]$, where $r_z > r_x$. Since the processor is either executing tasks or idling in the interval [r_x,r_z], the following inequality holds: $rpd(r_x, d_e) \geq rpd(r_z, d_e) + r_z - r_x$. Hence, if $rpd(r_z, d_e) > d_e - r_z$ then the following inequality also holds: $rpd(r_x, d_e) > d_e - r_x$. Therefore, it suffices to analyze intervals starting at r_x and ending at deadlines larger than or equal to d_x as per Eq (4.3.12).

□

In order to further reduce the number of intervals considered by RPCA to a single interval, the properties of EDLBPs can be used as follows:

Theorem 2. *A schedulable job set $\mathfrak{J}$ under a preemptive EDF policy, is feasibly extensible by an additional job $j_x \langle r_x, d_x, C_x \rangle$ if and only if:*

$$rpc(r_x, d_m^{bp}) \geq 0 \ \Big|\ d_m^{bp} \stackrel{def}{=} \max\left(\underset{d_j \in \mathfrak{D}\ |\ d_j \geq d_x}{\arg\min}\ rpc(r_x, d_j)\right) \tag{4.3.13}$$

where $\mathfrak{D}$ is an ordered set of all deadlines of all jobs in $\mathfrak{J}$ and of the job j_x, and d_m^{bp} is end of the EDLBP, bp_m, within which d_x lies, i.e., $d_{m-1}^{bp} < d_x \leq d_m^{bp}$.

Proof. According to Lemma 5, we can decide the feasibility of extending $\mathfrak{J}$ by j_x using the following check:

$$\forall\, d_j \in \big(\mathfrak{D} \cup \{d_x\}\big)\ |\ d_j \geq d_x :\ rpd(r_x, d_j) \geq d_j - r_x \iff rpc(r_x, d_j) \geq 0$$

Let $\mathfrak{D}_{bp} = \{d_0^{bp}, \ldots, d_m^{bp}, \ldots, d_{m+l}^{bp}\}$ be the set of EDLBP deadlines for the job set that are strictly larger than r_x. Since $d_{m-1}^{bp} < d_x \leq d_m^{bp}$, the following inequality holds according

to Lemma 1:

$$\forall d_j \mid d_{m-1}^{bp} < d_j < d_m^{bp},\ rpc(r_x, d_j) \geq rpc(r_x, d_m^{bp})$$

Hence, if $rpc(r_x, d_m^{bp}) > 0$ then so is $rpc(r_x, d_j)$. Similarly, we can write following Lemma 3:

$$\forall d_j^{bp} \in \mathfrak{D}_{bp} \wedge d_j^{bp} > d_m^{bp},\ rpc(r_x, d_j^{bp}) > rpc(r_x, d_m^{bp})$$

Hence, the single check $rpc(r_x, d_m^{bp}) > 0$ suffices to decide the feasibility of extending $\mathfrak{J}$ with j_x.

d_m^{bp} represents the deadline of the EDLBP where d_x lies. Starting from d_x, the RPC reaches its absolute minimum value at the end of this busy period. Hence, d_m^{bp} can be calculated as follows:

$$d_m^{bp} = \max \left(\underset{d_j \in \mathfrak{D} \mid d_j \geq d_x}{\arg\min} \ rpc(r_x, d_j) \right)$$

The arg min operator returns the deadline d_j that produces this minimum capacity. Because multiple intermediate deadlines within the same busy period might share this exact same minimum RPC value, arg min can evaluate to a *set* of all such deadlines. Hence, we must select the largest deadline from this set to get the deadline of the EDLBP. □

4.3.1.5 Runtime usage of RPCA

Theorem 2 has shown that deciding job set extension feasibility under RPCA boils down to checking a single interval. However, the end of that interval, i.e., d_m^{bp}, is not known a-priori, and finding it is equivalent to performing the test according to Lemma 5. Nevertheless, the derived results can still be used during runtime by having the scheduler maintain a look-ahead EDL schedule for the job set. That is, the scheduler implements mechanisms to maintain the localization of EDLBPs and the RPC in between them. This section details a possible implementation of such mechanisms, based on keeping an updated list of EDLBP deadlines, $\mathfrak{D}_{bp}$.

Calculating the list of EDLBP deadlines.

Following Theorem 2, finding the next EDLBP deadline relative to time t corresponds to finding the largest deadline d_j at which the function value $rpc(t, d_j)$ is minimum.

Algorithm 4 shows an alternative and more efficient approach to calculate the list of EDLBP deadlines, $\mathfrak{D}_{bp}$. Procedure $find_bpdl_list()$ requires two lists as input parameters, a list of deadlines dl_list and a list rpc_list, holding for each deadline d_k in dl_list

the value $rpc(t, d_x)$ (see Eq (4.3.8)). Using the results of Lemma 1 and Lemma 3, the procedure iterates through all deadline in dl_list in reverse order, adding a deadline d_k to $\mathfrak{D}_{bp}$ if $rpc(0, d_k) < rpc(0, d_{k+1})$ (lines 6-12). Hence, $find_bpdl_list()$ incurs a time complexity of $O(N_{dl})$, where N_{dl} is the number of deadlines in $\mathfrak{D}$.

Algorithm 4 Procedure for calculating the list of EDLBP deadlines $\mathfrak{D}_{bp}$

```
    // parameters and variables
    𝔇                                          ▷ ordered list of unique job deadlines
    𝔈 ← {∀d_x ∈ 𝔇, rpc(t, d_x)}                ▷ RPC value at each deadline list
 1: 𝔇_bp ← find_bpdl_list(𝔇, 𝔈)
 2: procedure find_bpdl_list(dl_list, rpc_list)
 3:     bpdl_list ← ∅
 4:     N_dl ← dl_list.size()                  ▷ Number of deadlines in dl_list
 5:     x ← N_dl − 1                           ▷ Index of current EDLBP deadline
 6:     bpdl_list.front_insert(dl_list[x])
 7:     for (i ← N_dl − 2 to 0) do
 8:         if (rpc_list[i] < rpc_list[x]) then
 9:             x ← i
10:             bpdl_list.front_insert(dl_list[x])
11:         end if
12:     end for
13:     return bpdl_list
14: end procedure
```

Maintenance of $\mathfrak{D}_{bp}$

For the runtime scheduler to use an existing knowledge of EDLBPs (as in Theorem 2), it is required that this knowledge be accurate, i.e., elements in $\mathfrak{D}_{bp}$ are in fact EDLBP deadlines. Hence, the list $\mathfrak{D}_{bp}$ must be updated during runtime, whenever one of the following conditions occurs:

1. The EDLBP deadline has passed.
2. Jobs are added the job set.
3. Jobs are removed from the job set or are executed in non-EDL order.

Condition 1. When an EDLBP deadline $d_x^{bp} = t$ is reached, the runtime scheduler removes this deadline from $\mathfrak{D}_{bp}$ as it is no longer needed. Moreover, if d_x^{bp} originates from a periodic TT task (even if currently $d_x^{bp} \notin \mathfrak{D}_{bp}$), its future occurrence in the next schedule cycle T_Γ (i.e., at $t + T_\Gamma$) must be added back to $\mathfrak{D}_{bp}$. Hence, the size of $\mathfrak{D}_{bp}$ depends on the number of EDLBP deadlines within a T_Γ. In the worst-case $\mathfrak{D}_{bp}$=$\mathfrak{D}_\mathfrak{J}$, where $\mathfrak{D}_\mathfrak{J}$ is the list of all unique deadlines in the job set $\mathfrak{J}$.

Condition 2. In order to illustrate how $\mathfrak{D}_{bp}$ must be updated when the job set is extended, consider the four example job sets, $\mathfrak{J}_w$, $\mathfrak{J}_x$, $\mathfrak{J}_y$ and $\mathfrak{J}_z$, created by extending the set of jobs $\mathfrak{J}$ from Table 2.1.3 with: $j_w < r_w = 1, d_w = 15, C_w = 2 >$, $j_x < r_x = 1, d_x = 15, C_x = 5 >$, $j_y < r_y = 1, d_y = 16, C_y = 3 >$, $j_z < r_z = 1, d_z = 16, C_z = 6 >$, respectively. The EDL schedules for the job sets are illustrated in Figure 4.3.1. The cases covering EDLBP change due to job set extension are:

- ***Case1*** - EDL Schedule for $\mathfrak{J}_w$: the execution of j_w forms a completely new EDLBP that does not change the structure of other existing EDLBPs. d_w is an EDLBP deadline and must be inserted into $\mathfrak{D}_{bp}$.
- ***Case2*** - EDL Schedule for $\mathfrak{J}_x$: the execution of j_x causes the formation of a new EDLBP ending at d_x. The new EDLBP subsumes one or multiple existing EDLBPs whose deadlines in $\mathfrak{D}_{bp}$ must be replaced by d_x.
- ***Case3*** - EDL Schedule for job set $\mathfrak{J}_y$: the execution of j_y extends the length of an existing EDLBP within which d_y is located without changing the structure of other existing EDLBPs. $\mathfrak{D}_{bp}$ is unchanged.
- ***Case4*** - EDL Schedule for $\mathfrak{J}_z$: the execution of j_z extends the length of an existing EDLBP within which d_z is located such that it subsumes one or multiple EDLBPs that precede it. The deadlines of the subsumed EDLBPs are removed from $\mathfrak{D}_{bp}$.

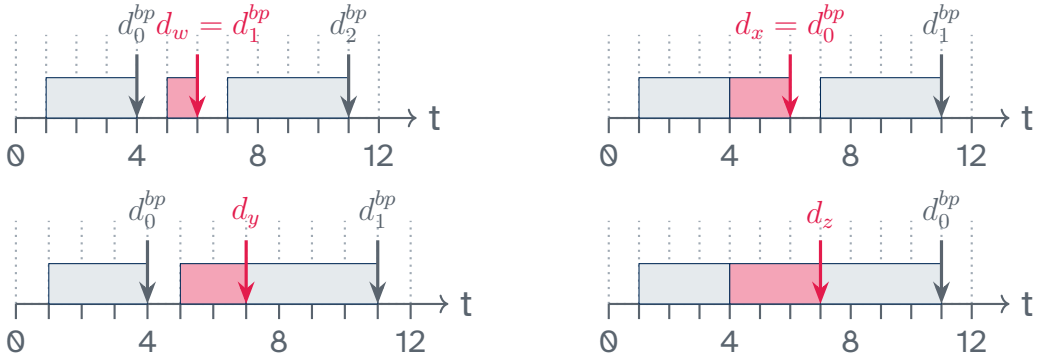


Figure 4.3.1: EDL schedules for job set $\mathfrak{J}_w$ (top-left), $\mathfrak{J}_x$ (top-right), $\mathfrak{J}_y$ (bottom-left), and $\mathfrak{J}_z$ (bottom-right)

Assuming that j_x's deadline, d_x, is smaller or equal to the largest deadline d_{N-1} within the schedule cycle, we propose Algorithm 5 to check the feasibility of the job set extension by j_x (following Theorem 2) and to update the list of EDLBP deadlines $\mathfrak{D}_{bp}$ according to the 4 cases above.

Algorithm 5 requires three lists, the list of deadlines $\mathfrak{D}$, the list of remaining execution requirement at each deadline $\mathfrak{Q}_{\mathfrak{D}}$, and the list of EDLBP deadlines. In lines (7-33), the

Algorithm 5 Checking feasibility of admitting $j_x < r_x, d_x, C_x >$ and updating $\mathfrak{D}_{bp}$ accordingly

// parameters and variables
$\mathfrak{D}$ ▷ ordered list of unique job deadlines
$\mathfrak{Q}_{\mathfrak{D}} \leftarrow \{\forall d_k \in \mathfrak{D}, \sum_{\forall j_w \mid d_w = d_k} c_{j_w}(r_x)\}$ ▷ list of remaining execution requirement at each deadline in $\mathfrak{D}$
$\mathfrak{D}_{bp}$ ▷ ordered list of EDLBP deadlines
1: $i \leftarrow 0$ ▷ index of EDLBP deadline d_i^{bp}
2: $rpci \leftarrow 0$ ▷ RPC at d_i^{bp}
3: $rpcx \leftarrow 0$ ▷ RPC at d_x
4: $\mathfrak{C}_{bp} \leftarrow \emptyset$ ▷ list of RPC values at EDLBP deadlines
5: $demand \leftarrow 0$ ▷ RPD at a job deadline d_j
6: $N \leftarrow \mathfrak{D}.size() - 1$
7: **if** $(d_x < \mathfrak{D}[0])$ **then**
8: $\quad demand+ = C_x$
9: $\quad rpcx \leftarrow d_x - r_x - demand$ ▷ $rpc(r_x, d_x)$
10: **end if**
11: **for** $(j \leftarrow 0$ to $N)$ **do**
12: $\quad demand+ = \mathfrak{Q}_{\mathfrak{D}}[j]$
13: $\quad$ **if** $(\mathfrak{D}[j] \leq d_x)$ **then**
14: $\quad\quad$ **if** $(j < N \wedge \mathfrak{D}[j+1] > d_x \vee j == N)$ **then**
15: $\quad\quad\quad demand+ = C_x$
16: $\quad\quad\quad rpcx \leftarrow d_x - r_x - demand$ ▷ $rpc(r_x, d_x)$
17: $\quad\quad$ **end if**
18: $\quad$ **end if**
19: $\quad$ **if** $(\mathfrak{D}[j] == \mathfrak{D}_{bp}[i])$ **then** ▷ $\mathfrak{D}[j] \in \mathfrak{D}_{bp}$
20: $\quad\quad rpci \leftarrow \mathfrak{D}_{bp}[i] - r_x - demand$ ▷ $rpc(r_x, d_i^{bp})$
21: $\quad\quad$ **if** $(\mathfrak{D}_{bp}[i] == d_x)$ **then**
22: $\quad\quad\quad break$
23: $\quad\quad$ **else if** $(\mathfrak{D}_{bp}[i] > d_x)$ **then**
24: $\quad\quad\quad$ **if** $(rpcx < rpci)$ **then** ▷ d_x is a d^{bp}
25: $\quad\quad\quad\quad i- = 1$
26: $\quad\quad\quad\quad rpci \leftarrow rpcx$
27: $\quad\quad\quad$ **end if**
28: $\quad\quad\quad break$
29: $\quad\quad$ **end if**
30: $\quad\quad \mathfrak{C}_{bp}.insert(rpci)$
31: $\quad\quad i+ = 1$
32: $\quad$ **end if**
33: **end for**
34: **if** $rpci \geq 0$ **then** ▷ Checking whether j_x can be admitted
35: $\quad$ **if** $rpci == rpcx \wedge \mathfrak{D}_{bp}[i] \neq d_x$ **then** ▷ d_x is a d^{bp}
36: $\quad\quad \mathfrak{D}_{bp}.insert_at(i, d_x)$
37: $\quad$ **end if**
38: $\quad N_{\mathfrak{C}_{bp}} \leftarrow \mathfrak{C}_{bp}.size()$ ▷ number of elements in $\mathfrak{C}_{bp}$
39: $\quad$ **for** $(z \leftarrow N_{\mathfrak{C}_{bp}} - 1$ to $0)$ **do** ▷ removing invalid EDLBP deadlines from $\mathfrak{D}_{bp}$
40: $\quad\quad$ **if** $\mathfrak{C}_{bp}[z] \geq rpci$ **then**
41: $\quad\quad\quad \mathfrak{D}_{bp}.remove_at(i + z - N_{\mathfrak{C}_{bp}})$
42: $\quad\quad$ **else**
43: $\quad\quad\quad break$
44: $\quad\quad$ **end if**
45: $\quad$ **end for**
46: **end if**

algorithm calculates $rpc(r_x, d_x)$, and builds a list of RPC values for all EDLBP deadlines in the interval $[r_x, d_m^{bp}]$, where d_m^{bp} is the EDLBP deadline within which d_x is located, i.e., $d_{m-1}^{bp} < d_x \leq d_m^{bp}$ (see Theorem 2). Afterwards, it is checked, whether $rpc(r_x, d_m^{bp}) > 0$ and if so, then j_x can be added to the job set (line 34). If $d_x = d_m^{bp} \wedge d_x \notin \mathfrak{D}_{bp}$, then d_x is inserted into $\mathfrak{D}_{bp}$ (lines 35-37). Based on the results of Lemma 3, the second loop takes care of removing all deadlines that are no longer EDLBP deadlines in the EDL schedule for the job set with j_x (lines 33-39). Since the scheduler maintains the remaining execution requirement at each deadline, i.e., the list $\mathfrak{Q}_{\mathfrak{D}}$, Algorithm 5 has a time complexity of $O(N_{dl}^m)$, where N_{dl}^m is the number of deadlines in the interval $[r_x, d_m^{bp}]$.

Note that by storing EDLBP's length alongside its deadline (i.e., the execution requirement by all jobs that form the EDLBP), the first loop in the algorithm can be avoided, leading to a faster decision on extension feasibility. However, this comes at the cost of maintaining EDLBP lengths at runtime.

Condition 3. The procedure for updating $\mathfrak{D}$ when a job is removed, skipped or executed in non-EDL order is the same. Consider a set of three jobs, $j_1 < r_1 = 2, d_1 = 3, C_1 = 1 >$, $j_2 < r_2 = 2, d_2 = 6, C_2 = 2 >$ and $j_3 < r_3 = 0, d_3 = 8, C_3 = 3 >$. Following Algorithm 4, the job set has a single EDL busy period ending at $d_3 = 8$. However, an algorithm that executes jobs as soon as possible (e.g., EDF) will allow j_3 to execute until at least the release of j_1 and j_2. At $t = 2$, j_3 has $c_3(2) = 1$, meaning that the original EDLBP from $t = 0$ in the EDL schedule is split, as shown in Figure 4.3.2, into three new EDLBPs, ending at each of the job deadlines. Removing or skipping j_2 at time $t = 1$, corresponds to resetting its remaining execution requirement, i.e., $c_2(1) = 0$, splitting the existing EDLBP into two EDLBPs, each ending at d_1 and d_3, respectively.

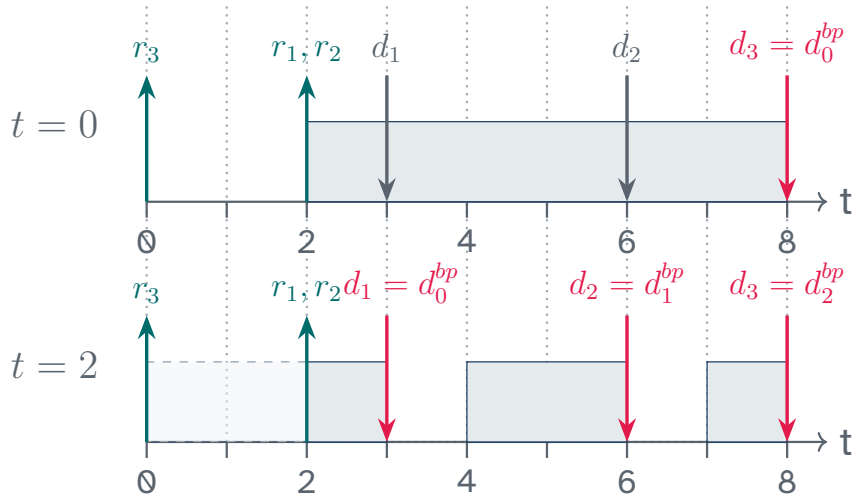


Figure 4.3.2: Example - changes to EDLBPs when jobs execute in non-EDL order

Catering for these cases, we propose Algorithm 6 to update $\mathfrak{D}_{bp}$ when a set of jobs $\mathfrak{J}_{rem} = \{j_{r_1}, j_{r_2}, \ldots, j_{r_K}\}$ is skipped or removed from $\mathfrak{J}$, or when a job executes in non-

EDL order (equivalent to removing a job with execution requirement that is equal to the completed execution). Firstly, the list of remaining execution requirements is updated by removing the amount of already executed demand or subtracting the demand of the to be skipped jobs. Next, the RPC function values considering all deadlines in the interval from smallest release time r_{min} in $\mathfrak{J}_{rem}$ until the EDLBP deadline d_{max} that is larger than or equal to the largest deadline in $\mathfrak{J}_{rem}$ are calculated. This can be done following similar steps as in the first loop of Algorithm 5. These deadlines and their RPC values are held in the lists $\mathfrak{D}_r$ and $\mathfrak{C}_r$, respectively. Any created EDLBP deadline in the interval [r_{min},d_{max}] is found using the $find_bpendlist()$ procedure from Algorithm 4, and added to $\mathfrak{D}_{bp}$. Assuming that N^r_{dl} is the number of job deadlines in the interval $[t, d_{max}]$, where t is the current point in time, Algorithm 6 has a time complexity of $O(N^r_{dl})$.

Algorithm 6 Updating $\mathfrak{D}_{bp}$ when a set of jobs J_{rem} is removed, or executed in non-EDL order.

// parameters and variables

$\mathfrak{D}$ ▷ ordered list of unique job deadlines

$\mathfrak{Q}_{\mathfrak{D}} \leftarrow \{\forall d_k \in \mathfrak{D}, \sum_{\forall j_w \mid d_w = d_k} c_{j_w}(t)\}$ ▷ list of remaining execution requirement at each deadline in $\mathfrak{D}$

$\mathfrak{D}_{bp}$ ▷ ordered list of EDLBP deadlines

Update $\mathfrak{Q}_{\mathfrak{D}}$ by subtracting the execution requirement of all $j_j \in J_{rem}$
$r_{min} \leftarrow \min_{\forall j_j \in J_{rem}}(r_j)$
$d_{max} \leftarrow \min(d^{bp} \in \mathfrak{D}_{bp} \geq \max_{\forall j_j \in J_{rem}}(d_j))$
▷ Identify range and RPC values
$\mathfrak{D}_r \leftarrow \{\forall d_k \in \mathfrak{D} \wedge r_{min} < d_k \leq d_{max}, d_k\}$
$\mathfrak{C}_r \leftarrow \{\forall d_k \in \mathfrak{D}_r,\ rpc(t, d_k)\}$
▷ Calculate EDLBP deadlines for the affected range
$\mathfrak{D}_{bpa} \leftarrow find_bpdl_list(\mathfrak{D}_r, \mathfrak{C}_r)$
▷ Replace the sub-list in $\mathfrak{D}_{bp}$ with the new list
$idx_{start} \leftarrow \mathfrak{D}_{bp}.find_first_index(\geq \min(\mathfrak{D}_r))$
$idx_{end} \leftarrow$ index of d_{max}
if $idx_{start} \neq null$ **then**
    $\mathfrak{D}_{bp}.replace_range(idx_{start}, idx_{end}, \mathfrak{D}_{bpa})$
else ▷ No overlap found, insert new EDLBP deadlines in order
    $\mathfrak{D}_{bp}.insert_sorted(\mathfrak{D}_{bpa})$
end if

4.3.2 Extending Time-Triggered Task Sets with Sporadic Tasks

This section investigates the following real-time feasibility analysis problem:

> *Given a task system comprising a collection of asynchronous periodic tasks that is known to be feasible upon a preemptive uniprocessor platform, is feasibility preserved upon adding a collection of sporadic tasks to the task system?*

Formally, we evaluate the feasibility of a combined task set $\Gamma = \Gamma_{\mathsf{P}} \cup \Gamma_{\mathsf{S}}$ upon a preemptive uniprocessor platform. The constituent task subsets are defined as follows:

- **Periodic task set (Γ_{P}):** The baseline task system comprises a set of *four-parameter periodic tasks*. Each task $\tau_x \in \Gamma_{\mathsf{P}}$ is modeled as a 4-tuple (ϕ_x, C_x, D_x, T_x). Here, ϕ_x denotes the initial offset (see Section 2.1.3), C_x is the WCET, D_x is the relative deadline, and T_x is the period separating successive job arrivals. We assume constrained deadlines, such that $D_x \leq T_x$.

- **Sporadic task set (Γ_{S}):** The extension consists of a set of *three-parameter sporadic tasks*. Each task $\tau_x \in \Gamma_{\mathsf{S}}$ is characterized by a 3-tuple (C_x, D_x, T_x). While C_x and D_x retain the same definitions as above (WCET and relative deadline, with $D_x \leq T_x$), the period parameter T_x here dictates the *minimum* inter-arrival time between successive jobs.

General feasibility analysis for both four-parameter periodic [63] and three-parameter sporadic [111] task systems is strongly coNP-complete. While bounding the total utilization ($\mathsf{U}(\Gamma) < 1$) reduces the sporadic problem to weakly coNP-complete (enabling pseudo-polynomial time algorithms [110, 111]), periodic systems with unequal offsets remain strongly coNP-complete due to their exponentially growing analysis intervals.

Consequently, our specific feasibility problem $\Gamma \stackrel{\text{def}}{=} \Gamma_{\mathsf{P}} \cup \Gamma_{\mathsf{S}}$ is inherently strongly coNP-complete in the general case. However, our scenario constitutes a *promise problem* [252]: we must determine the feasibility of the combined set *given the promise* that the baseline periodic set Γ_{P} is already feasible. Because sporadic systems under bounded utilization can be solved in pseudo-polynomial time, one might assume this promise would reduce the complexity of the extension problem.

This is not the case. As established by the authors in [55], determining the schedulability of $\Gamma_{\mathsf{P}} \cup \Gamma_{\mathsf{S}}$ remains strongly coNP-complete despite the promise that Γ_{P} is feasible. This strong coNP-completeness holds even when Γ_{S} consists of just a single implicit-deadline sporadic task ($D_x = T_x$) that increases the cumulative utilization by an arbitrarily small constant.

4.3.2.1 Revisiting Slot-Shifting's Offline Feasibility Test

In this section, we revisit Slot-Shifting's offline acceptance test for sporadic tasks, clarify the assumptions required for its correct application, identify sources of pessimism within it, and address its schedulability analysis interval.

4.3.2.1.1 Clarification of Assumptions

The offline acceptance test assumes that TT tasks experience maximum delay and are forced to complete their execution ALAP. Consequently, the identification of critical slots within each capacity interval (see Section 4.3.1.1) for the simultaneous arrival of sporadic tasks, and in turn the correctness of the acceptance test, is only established under this assumption[2].

Every slot in the schedule represents a global offset candidate for sporadic tasks. The critical slots defined in [125] serve to reduce the number of candidate slots to a single slot per capacity interval. However, in [125], there is no proof of the applicability of critical slots to ASAP scheduling.

4.3.2.1.2 Identification of the Pessimism

We now turn to identifying the pessimism in the offline acceptance test, which analyzes the feasibility of adding the worst-case sporadic demand alongside the ALAP-serviced TT demand based on the offline computed capacity intervals (see Section 4.3.1.1), and assuming that sporadic tasks arrive synchronously at the critical slot of each capacity interval in the TT schedule.

The runtime mechanisms defined in the Slot-Shifting extension prioritize ready ET tasks whenever spare capacity is available in the current interval. Consequently, ET tasks can impose forced delays on the execution of TT tasks, even when they do not have the earliest deadlines. In the worst-case scenario, TT tasks execute ALAP, while still meeting their deadlines. However, when no ET tasks are ready, TT tasks execute ASAP on a slot-by-slot basis and according to the EDF policy.

Since the considered acceptance test is performed offline and aperiodic tasks are only admitted online, analyzing the schedulability of the combined TT and sporadic task set offline does not need to take aperiodic tasks into account. This implies that, online, TT tasks execute ASAP prior to the synchronous arrival of sporadic tasks at a critical slot. Hence, the actual spare capacity values at a given slot deviate from those originally computed under the assumption of ALAP execution. Since we can precisely determine which TT tasks execute prior to the critical slot and for how many slots (based

[2]The reasoning in the proof of Theorem 1 in [125] is grounded in ALAP reservation of slots for TT tasks. This is evidenced by the absence of any change in spare capacity within the considered arrival interval when the arrival time of sporadic tasks is shifted forward.

on their release times, worst-case execution times (WCET), and EDF-based priorities), we can precisely calculate the change in capacity values. Each slot used by a TT task reduces its remaining demand by one. This does not consume a free slot but may shift it to a later point in time, meaning that there can be more available spare capacity online compared to the offline calculated values based on ALAP execution in the test. Consequently, assuming less available spare capacity than actually present based on ALAP execution of TT tasks introduces pessimism into the test. In particular, the test can produce false negatives by incorrectly classifying some schedulable task sets as unschedulable. However, this assumption does not lead to false positives, i.e., any task set deemed schedulable by the test is indeed schedulable.

Detailed Description and Example. In [125], Isovic and Fohler define, for each capacity interval I_i, a critical time slot $t_c^{I_i}$. They prove that if the combined TT and sporadic task set is feasible when sporadic tasks arrive synchronously at $t_c^{I_i}$, then it is also feasible when sporadic tasks arrive synchronously at every other time slot t_k located within the same interval I_i. Isovic and Fohler calculate the critical slot of interval I_i as follows:

$$t_c^{I_i} = start(I_i) + SC(I_i),$$

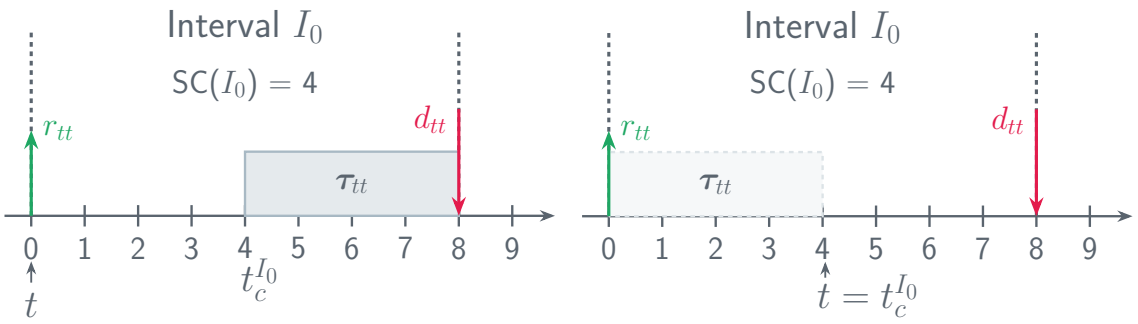
where $start(I_x)$ and $SC(I_i)$ denote the start of interval I_i and the spare capacity within it, respectively. In other words, since spare capacities in Slot-Shifting are calculated based on the ALAP execution of TT tasks in order to service aperiodic tasks ASAP, a critical slot of a capacity interval is the first time slot, where the spare capacity of the interval is completely used or wasted.

Let us consider the simple example task set system in Table 4.3.3, showing the parameters of a single TT task τ_{tt} after performing Slot-Shifting's offline phase (which resolves its complex constraints and transforms it into an independent task as described in Section 2.1.6.7) and a single sporadic task τ_{sp} with which we seek to extend the task set. Figure 4.3.3a shows the schedule for τ_{tt} with ALAP execution, annotated with its single spare capacity interval I_0 and its spare capacity $SC(I_0) = 4$. The critical slot for this interval is $t_c^{I_0} = 4$.

Table 4.3.3: Example task set, showing the parameters of a single TT task τ_{tt} after performing Slot-Shifting's offline phase and a single sporadic task τ_{sp}

Task	Type	Offset ϕ	WCET C	Deadline D	Period T
τ_{tt}	Periodic	0	4	8	8
τ_{sp}	Sporadic	-	1	4	4

Each sporadic task instance can be guaranteed depending on the availability of spare capacity within its execution window. If we let the sporadic task τ_{sp} arrive at the first critical slot, i.e., at $t = 4$, and attempt to guarantee it within the ALAP schedule in



(a) ALAP Schedule following execution until $t = 0$

(b) ALAP Schedule following execution until $t = 4$

Figure 4.3.3: Schedule for task τ_{tt} from Table 4.3.3, illustrating ALAP execution relative to time slot t, under the assumption that tasks are executed ASAP prior to t. For both cases, the capacity interval I_0 is indicated and annotated with its spare capacity $SC(I_0)$.

Figure 4.3.3a, we find that the first instance of τ_{sp} has no available free slots in its execution window interval $[4, 8]$. Therefore, it cannot be guaranteed according to the test in [125].

However, since Slot-Shifting's runtime scheduler executes tasks ASAP at each time slot, the corresponding ALAP schedule at $t_c^{I_0} = 4$ is shown in Figure 4.3.3b. By then, the task τ_{tt} has already finished executing, and all free slots have been relocated to the interval $[4, 8]$. Hence, the first instance of τ_{sp} can be guaranteed in the interval $[4, 8]$; there are 4 free-slots, and it needs only one. In fact, the task set in Table 4.3.3 follows the implicit deadline model, where the necessary and sufficient condition for EDF-schedulability is that the total utilization of the task set must be less than or equal to one (see Section 2.1.6.5). Since this condition is satisfied, the task set is schedulable.

Therefore, the test in [125] is pessimistic. The ASAP execution of TT tasks, shifts and does not consume free time slots within capacity intervals. Removing this pessimism requires recalculating the spare capacity values of all intervals based on the ASAP execution of TT tasks prior to the candidate slot for the global arrival of sporadic tasks. A possible approach is by simulating Slot-Shifting's online scheduling behavior on TT tasks up to the critical slot, thereby ensuring that spare capacities are updated accordingly across the entire schedule. In Table 4.3.4, we list simple modifications to eliminate this pessimism and make the test exact.

4.3.2.1.3 Addressing the Analysis Interval

A fundamental limitation in the original Slot-Shifting offline acceptance test for sporadic tasks [125] lies in its definition of the analysis interval. The original method evaluates feasibility relative to each critical slot strictly for the duration of the sporadic task

hyperperiod, HP(Γ_S). However, this specific bound is only formally proven for purely sporadic task sets operating in isolation [110]. When sporadic tasks co-execute with TT periodic tasks, applying this isolated bound lacks formal proof to justify correctness. Furthermore, the test leaves the underlying deadline model ambiguous.

We address the test analysis interval using formally proven bounds from scheduling theory. TT tasks are inherently periodic and characterized by distinct initial offsets and complex temporal constraints (e.g., precedence constraints and end-to-end deadlines). As detailed in Section 2.1.6.7, the offline scheduling phase resolves these complex dependencies prior to runtime, effectively reducing the TT tasks to a set of independent, asynchronous periodic tasks.

This transformation enables us to directly apply our proven schedulability bounds for asynchronous periodic tasks (see Theorem 3) to correctly dimension the analysis interval of the offline acceptance test, all while remaining compatible with the ASAP execution model. Because the bounds on t_1 and t_2 are established under the assumption of constrained deadlines, we adopt this identical deadline model. We then adapt the temporal bounds from Equations (4.3.17) and (4.3.18) to align seamlessly with Slot-Shifting's interval-based evaluation logic:

1. Set the critical slot t_c to t_1, marking the start for the current analysis iteration.

2. Omit t_2 as an explicit variable and instead force the test to evaluate all capacity intervals starting from t_c up to the strict theoretical maximum bound of $t_1 + \mathsf{B}(\Gamma)$.

By extending the analysis interval to capture the true maximum demand of the combined task set, this modification inherently increases the runtime of the test. However, the increased computational cost is necessary to ensure correctness.

4.3.2.1.4 Summary of Test Properties

Table 4.3.4 summarizes the properties of the original offline acceptance test alongside our proposed modifications, including the corrected analysis interval. Furthermore, we outline the specific adjustments required to align this evaluation with our exact formulation presented in the subsequent section. Particularly, these modifications explicitly incorporate the ASAP execution of TT tasks prior to the synchronous release of sporadic tasks at any candidate global offset slot. As demonstrated in our experimental evaluation (see Section 4.5), eliminating this analytical pessimism yields substantial improvements in schedulability.

Table 4.3.4: Properties of the original offline acceptance test compared against our proposed modifications to account for the ASAP scheduling of TT tasks.

Property	**Original Test [125]**	**Proposed Exact Modifications**
Candidate global offset slots for sporadic tasks	Only the critical slot of every interval I, calculated as $start(I) + sc(I)$.	Every slot that coincides with the release of a TT task in the schedule.
Spare capacity calculation	Evaluated once, assuming strictly ALAP execution of all TT tasks.	Evaluated iteratively for every candidate slot, assuming ASAP execution of TT tasks prior to that slot.
Analysis interval	Subject to the bounds in Equations (4.3.17), (4.3.18), and (2.1.7).	
Theoretical Exactness	Sufficient	Exact

4.3.2.2 Processor Demand Analysis for Mixed Periodic and Sporadic Task Sets

It follows from Equations (2.1.4) and (2.1.11) that we can decide the exact EDF-schedulability of the combined task set $\Gamma = \Gamma_{\mathsf{P}} \cup \Gamma_{\mathsf{S}}$ using the PDA by verifying that the inequality

$$\sum_{\tau_x \in \Gamma_{\mathsf{P}}} \mathsf{dbf}_{\mathsf{P}}(\tau_x, t_1, t_2) + \sum_{\tau_x \in \Gamma_{\mathsf{S}}} \mathsf{dbf}_{\mathsf{S}}(\tau_x, t_2 - t_1) \;\leq\; t_2 - t_1 \tag{4.3.14}$$

holds for all $t_1 < t_2$, where

$$\mathsf{dbf}_{\mathsf{P}}(\tau_x, t_1, t_2) \stackrel{\text{def}}{=} C_x \cdot \max\left(0, \left\lfloor \frac{t_2 - \phi_x - D_x}{T_x} \right\rfloor - \max\left(0, \left\lceil \frac{t_1 - \phi_x}{T_x} \right\rceil\right) + 1\right), \tag{4.3.15}$$

$$\mathsf{dbf}_{\mathsf{S}}(\tau_x, t) \stackrel{\text{def}}{=} C_x \cdot \max\left(0, \left\lfloor \frac{t - D_x}{T_x} \right\rfloor + 1\right) \tag{4.3.16}$$

are the standard demand bound functions for periodic and sporadic tasks, respectively.

As established in Section 2.1.6.5, the values of t_1 and t_2 can be confined to release times and absolute deadlines of tasks, respectively. However, since the synchronous arrival time of sporadic tasks is not known a priori, Equation(4.3.14) accumulates their maximum requested demand within the interval $[t_1, t_2]$ by assuming they arrive synchronously at t_1. Consequently, t_1 is restricted to the release times of periodic tasks

only, whereas t_2 can be the absolute deadline of any task in the combined task set, where for the sporadic tasks their absolute deadlines are those resulting from their synchronous arrival sequence starting at t_1.

For plain periodic tasks sets (i.e., not mixed with sporadic tasks), the ranges on the values of t_1 and t_2 to consider for the PDA can be constrained as per Equation (2.1.12):

$$0 \leq t_1 < t_2 \leq \max_{\tau_x \in \Gamma_\mathsf{P}}(\phi_x) + 2\,\mathsf{HP}(\Gamma_\mathsf{P})$$

where $\mathsf{HP}(\Gamma_\mathsf{P})$ denotes the hyper-period of the periodic taskset Γ_P. However, as the following theorem shows, it is in fact possible to reduce those ranges significantly, if total utilization $\mathsf{U}(\Gamma)$ is below 1, even for our more general problem with mixed task sets.

Theorem 3. *A combined constrained-deadline task set $\Gamma = \Gamma_\mathsf{P} \cup \Gamma_\mathsf{S}$ of N tasks, where Γ_P is a set of asynchronous periodic tasks, Γ_S is a set of sporadic tasks and $\mathsf{U}(\Gamma) < 1$, is EDF-schedulable on a single preemptive processor if and only if the inequality in Equation* (4.3.14) *holds for all $t_1 < t_2$, where*[3]

$$\max_{\tau_x \in \Gamma_\mathsf{P}}(\phi_x) \leq t_1 < \max_{\tau_x \in \Gamma_\mathsf{P}}(\phi_x) + \mathsf{HP}(\Gamma_\mathsf{P}) \tag{4.3.17}$$

$$t_2 < t_1 + \mathsf{B}(\Gamma) \tag{4.3.18}$$

$$\mathsf{B}(\Gamma) \stackrel{\text{def}}{=} \frac{\sum_{\tau_x \in \Gamma}(T_x - D_x)U_x}{1 - \mathsf{U}(\Gamma)} \qquad \text{from Equation (2.1.7)}$$

Proof. Γ is EDF-schedulable if and only if Equation (4.3.14) holds for all $t_1 < t_2$. We will show that the reduced ranges for t_1 and t_2 given in Equations (4.3.17) and (4.3.18) are sufficient to consider. First, we note that the range for t_1 in Equation (4.3.17) is the length of one hyper-period of the periodic tasks, $\mathsf{HP}(\Gamma_\mathsf{P})$, and that by starting at the largest offset $\max_{\tau_x \in \Gamma_\mathsf{P}}(\phi_x)$ the interval for t_1 will include all possible patterns of job releases from the periodic tasks, which are then repeated in the same way in the next interval of length $\mathsf{HP}(\Gamma_\mathsf{P})$ and so on. It follows that for any $t_1', \delta \in \mathbb{N}$, there exists $t_1 \in [\max_{\tau_x \in \Gamma_\mathsf{P}}(\phi_x), \max_{\tau_x \in \Gamma_\mathsf{P}}(\phi_x) + \mathsf{HP}(\Gamma_\mathsf{P}))$ such that $\sum_{\tau_x \in \Gamma_\mathsf{P}} \mathsf{dbf}_\mathsf{P}(\tau_x, t_1', t_1' + \delta) = \sum_{\tau_x \in \Gamma_\mathsf{P}} \mathsf{dbf}_\mathsf{P}(\tau_x, t_1, t_1 + \delta)$. Trivially, we also have $\sum_{\tau_x \in \Gamma_\mathsf{S}} \mathsf{dbf}_\mathsf{S}(\tau_x, t_1' + \delta - t_1') = \sum_{\tau_x \in \Gamma_\mathsf{S}} \mathsf{dbf}_\mathsf{S}(\tau_x, t_1 + \delta - t_1)$ for the sporadic tasks. Hence, if there are any values of t_1' and t_2', where t_1' does not fall within the constraints of Equation (4.3.17) and for which the inequality in Equation (4.3.14) does not hold, then the inequality in Equation (4.3.14) also does not hold for some other values of t_1 and t_2 where t_1 is constrained as in Equation (4.3.17).

[3] In the corner case where $\Gamma_\mathsf{P} = \emptyset$, and Equation (4.3.17) would be undefined, simply let $t_1 = 0$ instead. We note that in this corner case the proposed test simply reduces to the standard PDA test for sporadic tasks described in [110].

Second, it follows directly from Equations (4.3.15) and (4.3.16) that $\mathsf{dbf}_\mathsf{P}(\tau_x, t_1, t_2) \leq \mathsf{dbf}_\mathsf{S}(\tau_x, t_2 - t_1)$ for any periodic task τ_x and any $t_1 < t_2$[4]. From this it follows that for our combined task set $\Gamma = \Gamma_\mathsf{P} \cup \Gamma_\mathsf{S}$ and for any $t_1 < t_2$ we have

$$\sum_{\tau_x \in \Gamma_\mathsf{P}} \mathsf{dbf}_\mathsf{P}(\tau_x, t_1, t_2) + \sum_{\tau_x \in \Gamma_\mathsf{S}} \mathsf{dbf}_\mathsf{S}(\tau_x, t_2 - t_1) \leq \sum_{\tau_x \in \Gamma} \mathsf{dbf}_\mathsf{S}(\tau_x, t_2 - t_1) \tag{4.3.19}$$

It was shown in [108, 110] that for any Γ', where $\mathsf{U}(\Gamma') < 1$, if $\sum_{\tau_x \in \Gamma'} \mathsf{dbf}_\mathsf{S}(\tau_x, t) > t$, then we must also have $t < \mathsf{B}(\Gamma')$. Combining this with Equation (4.3.19), it follows that if $\sum_{\tau_x \in \Gamma_\mathsf{P}} \mathsf{dbf}_\mathsf{P}(\tau_x, t_1, t_2) + \sum_{\tau_x \in \Gamma_\mathsf{S}} \mathsf{dbf}_\mathsf{S}(\tau_x, t_2 - t_1) > t_2 - t_1$, then $t_2 - t_1 < \mathsf{B}(\Gamma)$. This matches the bound on t_2 in Equation (4.3.18) and completes the proof. □

The commonly-used ranges in Equation (2.1.12) give $\Theta(\mathsf{HP}(\Gamma)^2)$ combinations of t_1 and t_2, while the ranges in Equations (4.3.17) and (4.3.18) give only $\Theta(\mathsf{HP}(\Gamma_\mathsf{P})\,\mathsf{B}(\Gamma))$ combinations. The latter would typically be much smaller, which could improve the running time of checking schedulablity greatly.

Theorem 3 is also applicable when $\Gamma_\mathsf{S} = \emptyset$, and so as a *corollary* the reduced ranges of t_1 and t_2 can be used instead of the ranges in Equation (2.1.12) when checking schedulability of plain asynchronous periodic task sets (see Section 4.3.3).

4.3.2.2.1 Optimizations for Runtime Deployment

Despite reducing the values of t_1 and t_2 to consider, the runtime complexity for the test in Theorem 3 is still dominated by the periodic task set for which separate combinations of t_1 and t_2 matter, and not just their difference.

However, in the setting considered in this thesis, where Γ_P is fixed and we want to efficiently determine if adding a given Γ_S will make the system unschedulable, it is possible to shift much of this complexity to pre-runtime by pre-processing the sum $\sum_{\tau_x \in \Gamma_\mathsf{P}} \mathsf{dbf}_\mathsf{P}(\tau_x, t_1, t_2)$ in Equation (4.3.14) as follows: For each $t \in \{0, 1, \ldots, \mathsf{B}(\Gamma) - 1\}$, pre-compute values $\mathbf{dbf}_{\Gamma_\mathsf{P}}(t)$ as

$$\mathbf{dbf}_{\Gamma_\mathsf{P}}(t) \leftarrow \max_{t_1 \in R} \left(\sum_{\tau_x \in \Gamma_\mathsf{P}} \mathsf{dbf}_\mathsf{P}(\tau_x, t_1, t_1 + t) \right), \tag{4.3.20}$$

where $R = [\max_{\tau_x \in \Gamma_\mathsf{P}}(\phi_x), \max_{\tau_x \in \Gamma_\mathsf{P}}(\phi_x) + \mathsf{HP}(\Gamma_\mathsf{P}))$ is the range of values for t_1 from Equation (4.3.17).

Then, Theorem 4 describes our runtime-tailored feasibility test (referred to as *PDA** in Section 4.5).

[4] Note that we are here applying dbf_S to a periodic task. While dbf_S is *not* an exact demand bound function for a periodic task, it is still a well-defined function over three of the task's parameters.

Theorem 4. *A combined constrained-deadline task set $\Gamma = \Gamma_P \cup \Gamma_S$, where Γ_P is a set of asynchronous periodic tasks, Γ_S is a set of sporadic tasks and $U(\Gamma) < 1$, is EDF-schedulable on a preemptive uniprocessor system if and only if the inequality*

$$\mathbf{dbf}_{\Gamma_P}(t) + \sum_{\tau_x \in \Gamma_S} \mathrm{dbf}_S(\tau_x, t) \;\leq\; t \tag{4.3.21}$$

holds for all $0 < t < B(\Gamma)$, where $B(\Gamma)$ is calculated as per Equation (2.1.7)

Proof. Let t_1 be any release time of a periodic task, and consider any interval $[t_1, t_2]$ of length $t = t_2 - t_1$, with $0 < t < B(\Gamma)$. Since the demand of sporadic tasks is maximized in $[t_1, t_2]$ if they all arrive synchronously at t_1, their total demand depends only on t and equals $\sum_{\tau_x \in \Gamma_S} \mathrm{dbf}_S(\tau_x, t)$. In contrast, the demand of periodic tasks $\mathrm{dbf}_P(\tau_x, t_1, t_2)$ varies with the interval alignment, i.e., two intervals of the same length t but with different starting points t_1 can yield different demand. However, $\mathbf{dbf}_{\Gamma_P}(t)$ is exactly the maximum of periodic tasks' demand over all intervals $[t_1, t_2]$ of length t, and so if the inequality in Equation (4.3.14) fails for some values t_1 and t_2, then the inequality in Equation (4.3.21) will also fail considering $t = t_2 - t_1$ and vice versa. □

Theorem 4 allows us to check EDF-schedulability of $\Gamma_P \cup \Gamma_S$ for arbitrary Γ_S, having pre-computed the more costly analysis for Γ_P. The issue with this is that bound $B(\Gamma)$ depends on the particular Γ_S, which we by assumption do not know at the time of pre-computing $\mathbf{dbf}_{\Gamma_P}$. But if we reasonably assume upper bounds on allowed values of $U(\Gamma)$ and of $\max_{\tau_x \in \Gamma}(T_x - D_x)$, then we can directly calculate an upper bound on $B(\Gamma)$ to be used for the pre-processing step. When evaluating Equation (4.3.21) at runtime the exact $B(\Gamma)$ is used instead.

Indeed, with a predefined constant upper bound on allowed $U(\Gamma)$, say $U(\Gamma) \leq 0.99$, evaluating the schedulability with Equation (4.3.21) takes only pseudo-polynomial time (not counting pre-computing $\mathbf{dbf}_{\Gamma_P}$). Storing the values of $\mathbf{dbf}_{\Gamma_P}$ takes pseudo-polynomial space though. In contrast, evaluating the test in Theorem 3 is not in pseudo-polynomial time, but it requires only polynomial space.

Note that it suffices to check only those values of t at which the left-hand side of Equation (4.3.21) is discontinuous. The discontinuity points for $\mathbf{dbf}_{\Gamma_P}$ are computed during pre-processing[5]. For dbf_S, the discontinuities occur at the absolute deadlines of sporadic tasks from a synchronous arrival sequence in the interval $[0, B(\Gamma)]$.

In order to further optimize runtime performance, we apply the *quick convergence processor demand analysis* (QPA) technique [113], which enables the safe skipping of certain values of t using a simple backward iterative procedure. We adapt QPA to our

[5] In our evaluation in Section 4.5, we simply store the pre-processed values as pairs $(t, \mathbf{dbf}_{\Gamma_P}(t))$ for only the points t where $\mathbf{dbf}_{\Gamma_P}(t)$ is discontinuous.

problem by setting the $h(t)$ function from Theorem 5 in [113] to be equal to the left-hand side of Equation (4.3.21), and substituting the list of absolute deadlines with the list of discontinuity points. We refer to this adapted test, which combines QPA with preprocessed values of $\mathbf{dbf}_{\Gamma_\mathsf{P}}$, as *QPDA**. The corresponding pseudocode is provided in Algorithm 7.

Algorithm 7 Pseudocode for QPDA*

1: **Define** $h(t) \leftarrow \mathbf{dbf}_{\Gamma_\mathsf{P}}(t) + \sum_{\tau_x \in \Gamma_\mathsf{S}} \mathsf{dbf}_\mathsf{S}(\tau_x, t)$ `(left-hand side of Equation (4.3.21))`

2: **Define** $\mathcal{T}_{pts} \leftarrow$ `ordered list of unique discontinuity points of` $h(t)$

3: $t_{\mathsf{min}} \leftarrow \mathsf{min}\{t_i \mid t_i \in \mathcal{T}_{pts}\}$

4: $t \leftarrow \mathsf{max}\{t_i \mid t_i \in \mathcal{T}_{pts} \wedge t_i < \mathsf{B}(\Gamma)\}$

5: **while** $h(t) \leq t \wedge h(t) > t_{\mathsf{min}}$ **do**

6: $$t \leftarrow \begin{cases} h(t), & \text{if } h(t) < t \\ \mathsf{max}\{t_i \in \mathcal{T}_{pts} \mid t_i < t\}, & \text{otherwise} \end{cases}$$

7: **end while**

8: **Output:** Task set is **schedulable** if $h(t) \leq t_{\mathsf{min}}$; otherwise, **not schedulable**

4.3.3 Extending Time-Triggered Task Sets with Periodic Tasks

This section investigates the following real-time feasibility analysis problem:

> *Given a task system comprising a collection of asynchronous periodic tasks that is known to be feasible upon a preemptive uniprocessor platform, is feasibility preserved upon adding a collection of synchronous periodic tasks to the task system?*

Formally, we evaluate the feasibility of a combined task set $\Gamma = \Gamma_\mathsf{P} \cup \Gamma_\mathsf{PS}$ upon a preemptive uniprocessor platform. The constituent task subsets are defined as follows:

- **Asynchronous periodic task set (Γ_P):** The baseline task system comprises a set of *four-parameter periodic tasks*. Each task $\tau_x \in \Gamma_\mathsf{P}$ is modeled as a 4-tuple (ϕ_x, C_x, D_x, T_x). Here, ϕ_x denotes the initial offset (see Section 2.1.3), C_x is the WCET, D_x is the relative deadline, and T_x is the period separating successive job arrivals. We assume constrained deadlines, such that $D_x \leq T_x$.
- **Synchronous periodic task set (Γ_PS):** The extension consists of a set of *four-parameter periodic tasks*. Each task $\tau_x \in \Gamma_\mathsf{PS}$ is characterized by the same 4-tuple

formulation (ϕ_x, C_x, D_x, T_x), with the distinction that the initial offset is identical for all tasks within this set, i.e.„ $\forall \tau_x, \tau_y \in \Gamma_{\mathsf{PS}}$, $\phi_x = \phi_y = \phi_{PS}$.

There is an analytical distinction between this scenario and the previously discussed sporadic extension. While sporadic tasks may arrive at any instant, forcing the feasibility analysis to explicitly evaluate the worst-case critical instant relative to the arrival times of the baseline periodic tasks, a synchronous periodic task set is characterized by a fixed, shared offset. This shared offset dictates the arrival times of all its constituent tasks. Because these arrivals do not necessarily align with that worst-case critical instant, this scenario can potentially yield better feasibility results.

Nevertheless, despite this advantage, the computational complexity of this feasibility problem is identical to that of the extension with sporadic tasks. In the general case, analyzing the feasibility of the combined four-parameter task set $\Gamma = \Gamma_{\mathsf{P}} \cup \Gamma_{\mathsf{PS}}$ is strongly coNP-complete. Internally, the offset of the synchronous task set is only relevant relative to the asynchronous baseline. Consequently, if considered in isolation, the synchronous set could be reduced to a three-parameter task set (by omitting the shared offset) and would admit a pseudo-polynomial solution under bounded utilization.

However, the promise that the baseline asynchronous task set is already schedulable does not reduce the computational complexity of the extension problem. Checking the feasibility of the combined four-parameter task set Γ remains fundamentally as hard as verifying the feasibility of Γ_{P} from scratch. Consequently, the existence of an exact pseudo-polynomial time feasibility test for this problem is highly improbable.

Finally, this section will also consider the feasibility of runtime job migration on multi-core systems, demonstrating its equivalence to the periodic task set extension problem defined above.

4.3.3.1 Exact Processor Demand Analysis for Periodic Task Set Extension

Based on Equation (2.1.11) and our established bounds in Equations (4.3.17) and (4.3.18), the exact EDF-schedulability of the combined four-parameter task set $\Gamma = \Gamma_{\mathsf{P}} \cup \Gamma_{\mathsf{PS}}$ can be decided using the PDA. Specifically, the combined task set is feasible if and only if the following inequality holds for all $t_1 < t_2$:

$$\sum_{\tau_x \in \Gamma} \mathsf{dbf}(\tau_x, t_1, t_2) = \sum_{\tau_y \in \Gamma_{\mathsf{P}}} \mathsf{dbf}(\tau_y, t_1, t_2) + \sum_{\tau_z \in \Gamma_{\mathsf{PS}}} \mathsf{dbf}(\tau_z, t_1, t_2) \;\leq\; t_2 - t_1 \tag{4.3.22}$$

where the demand bound function and evaluation parameters are defined as:

$$\mathsf{dbf}(\tau_x, t_1, t_2) \stackrel{\text{def}}{=} C_x \cdot \max\left(0, \left\lfloor \frac{t_2 - \phi_x - D_x}{T_x} \right\rfloor - \max\left(0, \left\lceil \frac{t_1 - \phi_x}{T_x} \right\rceil\right) + 1\right),$$

$$\max_{\tau_x \in \Gamma}(\phi_x) \leq t_1 < \max_{\tau_x \in \Gamma}(\phi_x) + \mathsf{HP}(\Gamma),$$

$$t_2 < t_1 + \mathsf{B}(\Gamma),$$

$$\mathsf{B}(\Gamma) \stackrel{\text{def}}{=} \frac{\sum_{\tau_x \in \Gamma}(T_x - D_x)U_x}{1 - \mathsf{U}(\Gamma)}$$

While the exact evaluation points for t_1 and t_2 in Equation (4.3.22) can be restricted to actual task release times and absolute deadlines, evaluating the two-dimensional space of $t_1 < t_2$ remains computationally prohibitive for online admission control.

4.3.3.2 Related Work: Existing Sufficient Feasibility Tests

Avoiding the exponential complexity of the exact feasibility analysis in Equation 4.3.22 requires introducing analytical pessimism, yielding tests that are only sufficient but execute in pseudo-polynomial time. In the context of analyzing the feasibility of asynchronous periodic task sets under fully preemptive uniprocessor scheduling, several sufficient tests with varying degrees of complexity and pessimism have been proposed in the literature. The most conservative PDA-based approach evaluates the task set under synchronous arrival assumption by forcing all task offsets to zero. Although computationally efficient, this test yields highly pessimistic feasibility results [110].

Pellizzoni and Lipari [109] proposed a sufficient feasibility test that offers a controllable trade-off between pessimism and computational complexity by anchoring a subset of M tasks (hereafter referred to as MFT). It reduces the analysis complexity by evaluating a restricted set of critical arrival patterns derived from task offsets. Specifically, the algorithm selects a subset of M *fixed* tasks and evaluates all valid phase alignments among them. The remaining *non-fixed* tasks are subsequently pulled back to their minimum possible relative arrival times with respect to the fixed subset. This mechanism compresses the schedule, constructing an artificial worst-case busy period that maximizes task interference and thereby introduces analytical pessimism.

For each selection of an initial fixed task τ_i, the algorithm generates a distinct modified task set Γ'_{i} in which τ_i anchors the busy period, while all other tasks are pulled back according to the aforementioned rule. The processor demand is then evaluated over the resulting artificial busy period. This evaluation is iterated for every task in the system, resulting in N distinct feasibility checks, where N represents the total number of tasks. The asynchronous task set is declared feasible if and only if the processor demand test

succeeds across *all* N generated modified task sets.

By increasing the number of fixed tasks M, the method captures more of the true phase relationships between tasks, converging to an *exact* feasibility analysis when $M = N$. However, for any configuration where $M < N$, the test creates artificial busy periods that may never occur in the actual schedule. Consequently, the test remains *sufficient*.

Although the MFT test exhibits pseudo-polynomial complexity under bounded utilization when $M < N$, its runtime scales rapidly as $\mathcal{O}(N^{M+1})$. This growth renders the test computationally prohibitive for large task sets or elevated values of M, making it less attractive for extending task sets dynamically under such scenarios.

4.3.3.3 Proposed Optimizations for Runtime Deployment

In order to achieve the pseudo-polynomial efficiency required for online admission control, we utilize the offline pre-processing technique we used for sporadic task extension in Section 4.3.2.2.1. Because the baseline asynchronous periodic task set is fixed and known a priori, its worst-case demand for any interval length $t \in \{0, 1, \ldots, \mathsf{B}(\Gamma) - 1\}$ can be pre-computed offline using Equation (4.3.20) and stored in a one-dimensional lookup table. Let $\mathbf{dbf}_{\Gamma_\mathsf{P}}(t)$ be a function that returns the stored value for a given interval length t. By omitting the shared offset parameter from the set of synchronous periodic tasks (effectively treating them as three-parameter sporadic tasks), the online feasibility test simplifies to a one-dimensional evaluation of whether the following inequality holds for all $0 < t < \mathsf{B}(\Gamma)$:

$$\mathbf{dbf}_{\Gamma_\mathsf{P}}(t) + \sum_{\tau_x \in \Gamma_\mathsf{PS}} \mathsf{dbf}_\mathsf{S}(\tau_x, t) \;\leq\; t \tag{4.3.23}$$

where $\mathsf{dbf}_\mathsf{S}(\tau_x, t)$ is defined in Equation (4.3.16).

It suffices to only check the values at which the left-hand side of Equation (4.3.23) is discontinuous. Furthermore, not knowing the synchronous periodic task set a priori similarly requires assuming an upper bound on $\mathsf{B}(\Gamma)$ during pre-processing. At runtime, the exact bound is dynamically calculated and used instead. Finally, by setting the left-hand side of the inequality in Equation (4.3.23) as $h(t)$, we can use QPDA* to analyze the feasibility of the combined task set following Algorithm 7.

It is critical to note that for synchronous periodic tasks, Equation (4.3.23) constitutes a *sufficient but not exact* feasibility test. By extracting the maximum baseline demand as a function of the interval length t, the analysis loses the precise temporal phase relationship between Γ_P and the shared offset of Γ_PS. Consequently, the test pessimistically assumes that the worst-case critical instant of the synchronous task set coincides with the absolute busiest interval of the baseline task set.

However, this approach holds an analytical advantage over existing sufficient tests.

Unlike the MFT test, which shifts all non-fixed tasks and introduces pessimism across the *entire* task set, our approach incurs *zero pessimism* for the baseline task set. By preserving the exact phase relationships of the pre-existing tasks through precise offline analysis, analytical pessimism is confined to the runtime-added tasks (which are evaluated as sporadic).

Accepting this localized pessimism is a necessary compromise for achieving pseudo-polynomial runtime efficiency. Maintaining an exact test would require a two-dimensional matrix $\mathbf{dbf}_{\Gamma_P}(t_1, t_2)$ to preserve phase alignments. Because an exact online test must evaluate all start times t_1 across the hyperperiod $\mathrm{HP}(\Gamma_P)$, the number of online checks would still grow exponentially, incurring exponential offline memory costs without escaping exponential computational growth.

Given that an exact pseudo-polynomial test for this strongly coNP-complete problem is unlikely to exist, the sufficient test in Equation (4.3.23) represents a sound compromise. Furthermore, this decoupling provides a secondary computational advantage regarding evaluation bounds: to capture the true worst-case alignment of task releases, an exact test must evaluate all interval combinations of t_1 and t_2 up to the hyperperiod of the *combined* task set, $\mathrm{HP}(\Gamma)$. In contrast, because our sufficient test evaluates Γ_P in isolation during the offline phase, it only needs to evaluate intervals up to the hyperperiod of the baseline set, $\mathrm{HP}(\Gamma_P)$, to safely extract the maximum demand. Assuming bounded utilization, this approach circumvents the need to iterate over the potentially massive combined hyperperiod $\mathrm{HP}(\Gamma)$.

4.3.3.4 Job Migration Feasibility in Multicore Systems

As modern TT systems transition toward integrated multicore architectures, the ability to dynamically balance computational load across cores becomes essential to maintain resource efficiency and mitigate localized overloads. However, classical TT systems rely on a static task-to-core allocation, which prohibits runtime job migration. In order to safely introduce this operational flexibility, the system must verify that migrating a job to a new core will not violate the hard real-time guarantees of the tasks already executing there.

The periodic feasibility analyses developed in the previous section provide the foundation to solve this dynamic load-balancing challenge. Specifically, the permanent migration of a single job, and its subsequent periodic occurrences within a *Cause-Effect Chain (CEC)*, can be analytically reduced to the problem of admitting a single new periodic task to the candidate core.

Let τ_x be a task belonging to a given CEC E on a source core, and consider its k-th job, $j_{x,k}$, which is selected for migration. Because the baseline schedule of E strictly repeats every hyperperiod $\mathrm{HP}(E)$, a permanent migration of this job inherently shifts all its future

instances that occur at this exact relative phase in every subsequent hyperperiod.

We can define the resulting infinite sequence of migrated jobs as:

$$J_{mig} = \left\{ j_{x,k+n\cdot(\mathsf{HP}(E)/T_x)} \mid n \in \mathbb{N}_0 \right\}$$

We observe that the release times, execution requirements, and absolute deadlines of j_{mig} map perfectly to a standard four-parameter periodic task. Let this equivalent migrated task be denoted as $\tau_{mig} = (\phi_{mig}, C_{mig}, D_{mig}, T_{mig})$. By matching the temporal parameters of J_{mig}, we establish the following configuration for τ_{mig}:

$$\begin{aligned}
T_{mig} &= \mathsf{HP}(E) && \text{(Repeats every CEC hyperperiod)} \\
\phi_{mig} &= \phi_x + (k-1)T_x && \text{(Absolute release of the first migrated job)} \\
C_{mig} &= C_x && \text{(Inherited from the base task } \tau_x\text{)} \\
D_{mig} &= D_x && \text{(Inherited from the base task } \tau_x\text{)}
\end{aligned}$$

Because the temporal properties of the migrated job sequence J_{mig} is equivalent to the periodic task τ_{mig}, the migration feasibility problem reduces to a standard periodic task admission problem. Assuming the candidate core already hosts a feasible baseline asynchronous periodic task set Γ_{P}, analyzing migration feasibility is equivalent to verifying the feasibility of the extended task set $\Gamma = \Gamma_{\mathsf{P}} \cup \{\tau_{mig}\}$.

Consequently, job migration feasibility can be evaluated using either the exact test in Equation (4.3.22) or the runtime-optimized sufficient tests in Equation (4.3.23) and Algorithm 7. By enabling job migration, this approach contributes to relaxing the rigidity of classical TT assumptions and successfully introduces the operational flexibility required by modern safety-critical systems while preserving the strict temporal guarantees.

Building on this reduction, the authors in [253] demonstrate how job migration, evaluated via our exact test, can be leveraged to guarantee end-to-end latencies in systems featuring multi-rate CECs and mode changes.

4.4 Feasibility Tests Compatibility with DTT Schedulers

The feasibility tests developed in Section 4.3, targeting the runtime extension of task sets with aperiodic, sporadic, and periodic tasks, were formally derived under the assumption of a fully preemptive execution model. However, DTT schedulers, such as our DTSS method (see Section 4.2.5), enforce a fundamentally different execution paradigm by assigning non-preemptable computation slots to tasks at runtime.

Because the scheduler checks ready queues exclusively at discrete invocation points (slot boundaries), it remains completely oblivious to the exact physical arrival times of

asynchronous events that occur between these points. To maintain correctness and compatibility under DTT operation, the system designer must explicitly account for this *temporal blindness* inherent to slot-based scheduling. The required adaptations depend on the operational mode of the DTT scheduler (static vs. nonuniform invocation gaps), and the event-detection capabilities of the underlying hardware.

4.4.1 DTT Scheduler Operation Granularity

A DTT scheduler can support uniform as well as nonuniform invocation gaps as its granularity of operation, which differently affect the scheduler temporal blindness interval. For DTSS, the following applies to its supported operational modes:

- **Static mode (uniform invocation gaps):** The scheduler is invoked by a strictly periodic clock interrupt, yielding a fixed, uniform invocation gap T_{IG}. In this mode, the invocation gap and the maximum scheduler blindness are identical. Freed processor capacity resulting from early job completions before the next fixed scheduler point cannot be reclaimed, which translates into wasted utilization.

- **Dynamic mode (nonuniform invocation gaps):** The scheduler dynamically calculates the next invocation point based on the remaining execution requirement of the chosen ready job and the known release of a higher-priority job in the future. The computation slot is allocated in discrete multiples of the global time granularity, which defines a minimum invocation gap ($T_{IG_{min}}$). The invocation gap may increase up to a maximum continuous non-preemptable interval ($T_{IG_{max}}$). Because jobs can yield the processor upon early completion, slack is reclaimed immediately. In this mode, the maximum scheduler blindness is $T_{IG_{max}}$.

4.4.2 Event-Detection and Timestamping Capabilities

The extent to which the DTT scheduler's temporal blindness penalizes the feasibility analysis depends on how accurately the system can timestamp asynchronous events. Specifically, the event-detection mechanism dictates whether the temporal uncertainty must be modeled as a pessimistic *release jitter* penalty applied individually to task deadlines, or as a less severe *non-preemptive blocking* term applied globally to the analysis interval. We distinguish between software polling with and without hardware-assisted timestamping.

4.4.2.1 Software Polling without Hardware-Assisted Timestamping

In TT systems, asynchronous interrupts are disabled to prevent interrupt showers. The operating system detects the arrival of sporadic or aperiodic tasks when it polls the ready queues at the subsequent scheduler point (slot boundary). This inter-slot blindness imposes a maximum release jitter (J) on the task, which must be absorbed into the task's temporal parameters prior to applying the feasibility analysis.

4.4.2.1.1 Parameter Scaling in Static Mode

Because the invocation gap is fixed, all parameters are scaled as multiples of T_{IG}, and the worst-case release jitter is exactly $J = T_{IG}$. The following rules dictate how to scale task parameters for the different task types:

- **Periodic Tasks:** Because periodic task arrivals are synchronous with the global time base, their exact arrival times are known a priori to the scheduler. They suffer zero release jitter. Their temporal parameters are simply quantized to the invocation gap:

$$\phi_{p'} = \left\lceil \frac{\phi_p}{T_{IG}} \right\rceil T_{IG}, \qquad T_{p'} = \left\lfloor \frac{T_p}{T_{IG}} \right\rfloor T_{IG}$$
$$C_{p'} = \left\lceil \frac{C_p}{T_{IG}} \right\rceil T_{IG}, \qquad D_{p'} = \left\lfloor \frac{D_p}{T_{IG}} \right\rfloor T_{IG}$$

- **Aperiodic and Sporadic Tasks:** Since their arrivals are asynchronous, their relative deadlines must be artificially penalized by the maximum scheduler blindness (T_{IG}) to guarantee that the absolute deadline calculated by the runtime scheduler remains safe.

$$C_{s'} = \left\lceil \frac{C_s}{T_{IG}} \right\rceil T_{IG}, \qquad D_{s'} = \left\lfloor \frac{D_s - T_{IG}}{T_{IG}} \right\rfloor T_{IG}$$

In this mode, scaling execution times to a large fixed invocation gap inherently introduces significant intra-slot wasted utilization. In contrast, choosing a small invocation gap reduces the release jitter penalty but increases context-switching overheads.

4.4.2.1.2 Parameter Scaling in Dynamic Mode

The dynamic mode decouples the execution granularity from the release jitter penalty. Task parameters are quantized based on the minimum invocation gap ($T_{IG_{min}}$), while the relative deadlines of sporadic and aperiodic tasks absorb the maximum scheduler blindness ($T_{IG_{max}}$).

- **Periodic Tasks:**

$$\phi_{p'} = \left\lceil \frac{\phi_p}{T_{IG_{min}}} \right\rceil T_{IG_{min}}, \qquad T_{p'} = \left\lfloor \frac{T_p}{T_{IG_{min}}} \right\rfloor T_{IG_{min}}$$
$$C_{p'} = \left\lceil \frac{C_p}{T_{IG_{min}}} \right\rceil T_{IG_{min}}, \qquad D_{p'} = \left\lfloor \frac{D_p}{T_{IG_{min}}} \right\rfloor T_{IG_{min}}$$

- **Sporadic/Aperiodic Tasks:**

$$C_{s'} = \left\lceil \frac{C_s}{T_{IG_{min}}} \right\rceil T_{IG_{min}}, \qquad D_{s'} = \left\lfloor \frac{D_s - T_{IG_{max}}}{T_{IG_{min}}} \right\rfloor T_{IG_{min}}$$

By applying the feasibility tests using these penalized and quantized parameters, mathematical correctness and temporal safety are maintained under DTT systems without modifying the tests themselves.

4.4.2.2 Software Polling with Hardware-Assisted Timestamping

When the system is augmented with a hardware timestamping mechanism, the exact arrival time of asynchronous events is recorded at the hardware level. Consequently, when the DTT scheduler polls the ready queues, it computes the true absolute deadline, eliminating the need for release jitter deadline penalties.

The parameters for all tasks are uniformly quantized to the relevant operational granularity (T_{IG} for static mode, or $T_{IG_{min}}$ for dynamic mode) without subtracting a jitter penalty from the relative deadlines of sporadic and aperiodic tasks. However, the delay caused by waiting for the current non-preemptable slot to finish persists. Knowing the exact timestamp transforms this temporal penalty from release jitter into *non-preemptive blocking*.

The feasibility tests for mixed task sets must be modified to consider a deferred preemption EDF model to maintain compatibility under DTT operation. This is achieved by injecting an interval-specific blocking term, $b(t)$, into the total demand constraint (c.f., Algorithm 7 and Section 2.1.6.5): $h(t) + b(t) \leq t$.

In the PDA, t represents the length of the analyzed interval ($t = t_2 - t_1$). The blocking term $b(t)$ captures the maximum priority inversion delay that the demand within this interval may experience. In a DTT scheduler with EDF policy, an executing non-preemptable slot only constitutes priority inversion if it is occupied by a lower-priority task, meaning the task's relative deadline is strictly larger than the analyzed interval ($D_k > t$).

For DTSS, the maximum non-preemptable duration any task τ_k can get is constrained by its quantized worst-case execution time $C_{k'}$ and the maximum allowable invocation

gap. We define this maximum non-preemptable chunk, B_k, for each task as follows:

$$B_k = \begin{cases} T_{IG} & \text{(Static Mode)} \\ \min(C_{k'}, T_{IG_{max}}) & \text{(Dynamic Mode)} \end{cases} \tag{4.4.1}$$

Because the feasibility analysis evaluates the maximum demand over any interval of length t, it must conservatively assume that the largest non-preemptable slot began just prior to the interval. The blocking term is therefore a global penalty evaluated purely against the interval length t and the system's maximum relative deadline. If $t \geq \max(D_k)$, all tasks have deadlines within the interval, their execution is fully accounted for by the demand function $h(t)$, and priority inversion is not possible. Consequently, the interval-specific blocking term $b(t)$ is defined as:

$$b(t) = \begin{cases} \max\limits_{\tau_k \in \Gamma : D_k > t} (B_k) & \text{if } t < \max\limits_{\tau_k \in \Gamma}(D_k) \\ 0 & \text{otherwise} \end{cases} \tag{4.4.2}$$

This hardware-assisted timestamping approach significantly reduces analytical pessimism. Adding a single bounded blocking term to the interval demand is less severe than artificially shrinking the relative deadline of every individual asynchronous task in the system.

Finally, while we have established the feasibility tests necessary to safely extend TT task sets with aperiodic, sporadic, and periodic tasks at runtime under DTT operation, modern adaptive systems may require the simultaneous admission and modification of all three task types. Investigating the interplay that arises from combining these feasibility tests dynamically at runtime remains an open challenge left for future work.

4.5 Experimental Evaluation

The DTT scheduler and feasibility tests introduced in the preceding sections guarantee the safe runtime modification of TT task sets without violating the system's extra-functional properties. In this section, we evaluate their performance using both synthetic task sets and real-world automotive benchmarks. We structure our evaluation into two primary investigations:

1. **Evaluating DTSS overhead savings:** A comparative analysis quantifying the reduction in context-switching overheads (measured via scheduler invocations) achieved by the dynamic operating mode of DTSS versus the static mode common to existing DTT schedulers and the resulting effects on aperiodic task admission. Additionally, we evaluate the reduction in the memory footprint required for admitting aperiodic tasks at runtime compared to Slot-Shifting.

2. **Evaluating our feasibility tests' schedulability and runtime performance:** We evaluate our exact tests for sporadic tasks (PDA, PDA*, and QPDA*) against state-of-the-art feasibility tests, assessing their schedulability performance and execution speeds to demonstrate their suitability for online deployment.

Together, these experiments confirm that the rigid assumptions of classical TT scheduling can be safely relaxed without incurring prohibitive runtime overheads.

4.5.1 DTSS Overheads

In this section, we quantify the overhead savings introduced by DTSS when compared to state-of-the-art flexible TT scheduling methods. Note that since DTSS is tailored for the dynamic modification of task sets at runtime, a direct comparison makes sense with those methods that similarly add flexibility at runtime (e.g., Slot-Shifting [122,125]).

The first experiment assesses the reduction in context-switching overhead, measured by the total number of scheduler invocations, when transitioning from uniform to nonuniform invocation gaps. The second experiment compares DTSS against Slot-Shifting, focusing on the memory footprint required to maintain the look-ahead processor capacity information that is necessary for the admission of aperiodic tasks.

For all experiments in this section, we generated synthetic task sets using the UUniFast algorithm [254]. TT task parameters are distributed within the following ranges: offsets $[0, 50]$, periods $[15, 200]$, WCET $[1, 20]$, deadlines $[5, 200]$. Aperiodic deadline-constrained tasks are randomly generated with arrivals within the schedule cycle T_Γ and values for WCET, deadlines, and utilization within the ranges $[2, 50]$, $[5, 200]$, $[35, 70]$. For each target utilization value in the range $[35, 70]$, we generated 1000 task sets, fixing the utilization of offline tasks to 40%.

4.5.1.1 Scheduler Invocations

In this experiment, we compare the reduction of the number of scheduler invocations within task sets' schedule cycles T_Γ, when using DTSS in dynamic mode compared to DTSS in static mode (or equivalently schedulers in previous works [122,123,125,216,247, 248]). Fewer scheduler invocations correspond to reduced context switching overheads.

Let T_{IG} denote the fixed activation granularity of DTSS in the static mode. The number of scheduler invocations within T_Γ is fixed and can be calculated as follows: $\frac{T_\Gamma}{T_{IG}}$. This formulation inherently assumes that all task periods and offsets are integer multiples of T_{IG}. In this baseline configuration, we set $T_{IG} = 1$, meaning the number of scheduler invocations equals the schedule cycle length T_Γ. Consequently, the number of scheduler points remains constant unless the schedule cycle itself changes due to the addition of new periodic tasks.

As described in Section 4.2.5, in the dynamic mode, DTSS allocates ready jobs to computation slots, whose size depends on jobs' release times and deadlines. Hence, unnecessary scheduler invocations that interrupt job execution are eliminated. According to our knowledge, DTSS is the first DTT scheduler with nonuniform invocation gaps and dynamic assignment of slot sizes to jobs at runtime.

Figure 4.5.1 illustrates the relative reduction in scheduler invocations across all generated task sets when transitioning from DTSS's static to a dynamic operation mode. The results demonstrate an overall average reduction of 60%. This efficiency gain scales linearly with T_{IG}. For instance, the relative reduction doubles if T_{IG} is halved (i.e., $T_{IG} = 0.5$). Furthermore, the improvement is affected when the task set is extended or reduced since the number of scheduler points increases or decreases within T_{Γ}. However, it never exceeds that of the static mode.

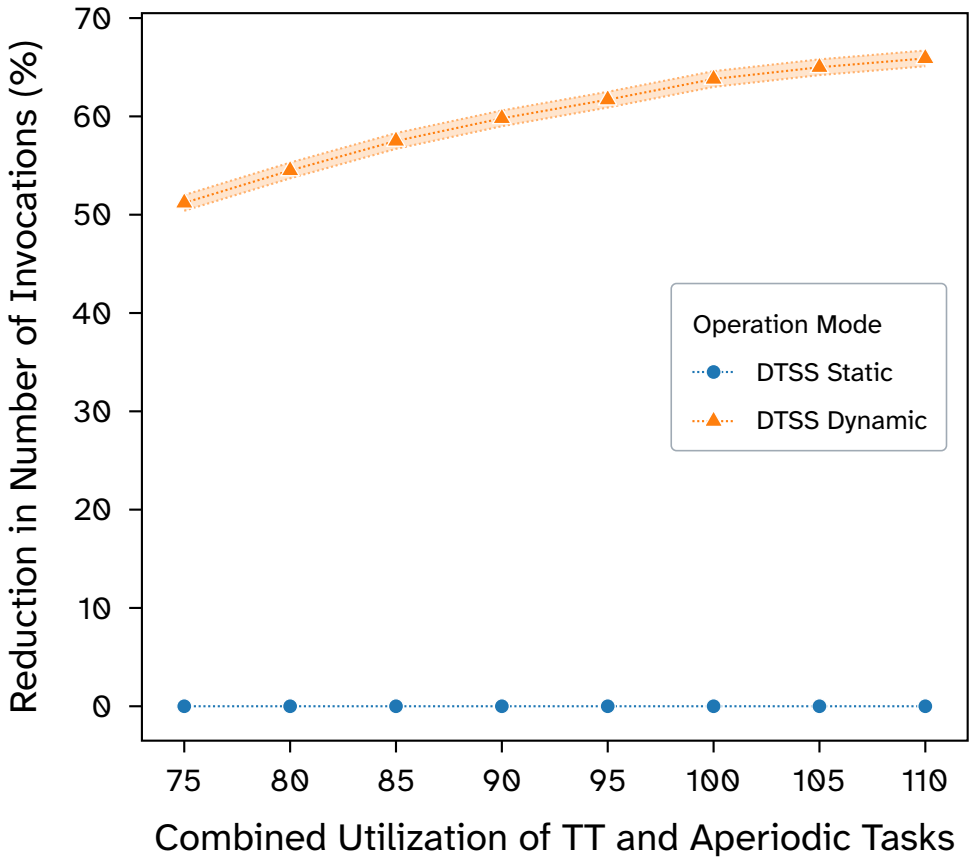


Figure 4.5.1: Relative reduction in number of invocations when using DTSS in dynamic compared to DTSS in static mode with $T_{IG} = 1$, where the utilization of TT tasks is 40%

4.5.1.2 Admission Ratio for Aperiodic Tasks

The feasibility test for admitting aperiodic tasks arriving at runtime is only performed when the node's local scheduler is invoked at the beginning of each slot. Slot-Shifting in [122,125] and DTSS in static mode have the same aperiodic task admission ratio when driven by the same defined uniform invocation gap T_{IG}. However, when DTSS is set to operate in dynamic mode, the scheduler checks less frequently for arriving jobs. Hence, in this experiment, we compare the admission ratio of aperiodic tasks when using DTSS in static mode $T_{IG} = 1$ to DTSS in dynamic mode, given that all jobs execute for their worst-case execution times. We will assume the presence of hardware-assisted time stamping to accurately calculate the absolute deadlines of arriving aperiodic tasks (see Section 4.4).

As shown in Figure 4.5.2, DTSS in dynamic mode performs worse with respect to accepting aperiodic tasks compared to DTSS in static mode. This is expected, since DTSS will be invoked less often to check for the arrival of new jobs. The overall average acceptance is 96.16% in the static mode and 77.5% in the dynamic one. Hence, the average reduction of acceptance in the dynamic compared to the static mode is 19.4%

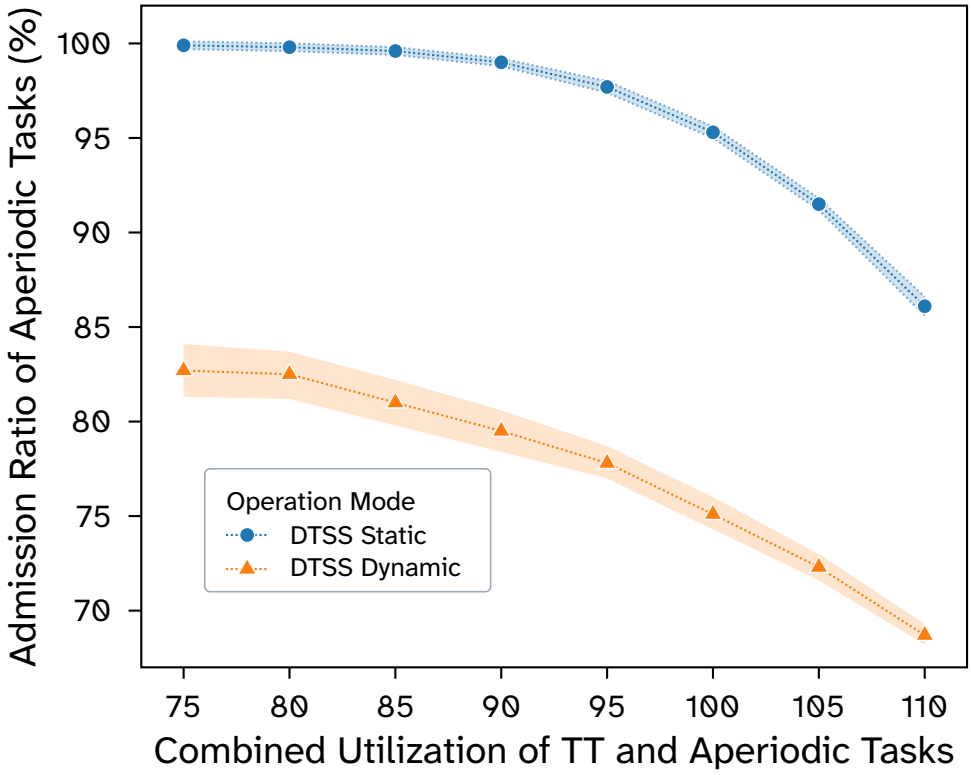


Figure 4.5.2: Relative reduction in admission ratio for aperiodic tasks when using DTSS in dynamic compared to DTSS in static mode with $T_{IG} = 1$, where the utilization of TT tasks is 40%

4.5.1.3 Memory Footprint for Processor Capacity Tracking

While Slot-Shifting does not explicitly define its spare capacity intervals in terms of an EDL schedule, these intervals function as a mechanism for tracking look-ahead capacity. However, Slot-Shifting typically generates a significantly larger number of intervals and relies on a spare-capacity borrowing mechanism. This borrowing effect is more involved and less intuitive than the concise representation provided by EDLBP deadlines in DTSS. As explained in Section 4.3.1.3, EDLBP deadlines facilitate both runtime capacity analysis and task set modifications.

Both DTSS and Slot-Shifting depend on maintaining an accurate view of available remaining processor capacity to execute their feasibility tests. The distinction lies in data representation: DTSS maintains a list of EDLBP deadlines (see Section 4.2.5), whereas Slot-Shifting requires a table of disjoint deadline intervals and their respective capacities (see Section 4.3.1.1).

By defining the duration between any two consecutive EDLBP deadlines as a singular interval, we can directly compare the memory footprint for tracking look-ahead capacity of both approaches. Figure 4.5.3 demonstrates the relative reduction in the total number of maintained intervals when utilizing DTSS instead of Slot-Shifting. Because higher system utilization naturally shapes execution demand into fewer, longer EDL busy periods, the required number of intervals decreases for an increasing number of task sets. Across all evaluated configurations, DTSS achieves an overall average interval reduction of 74.21%.

Since DTSS computes the remaining processor capacity between EDL busy periods instead of between job deadlines, every tracked interval inherently possesses a positive capacity value. This eliminates the negative-capacity borrowing effect present in Slot-Shifting, thereby bypassing the need for cascaded capacity-update routines and optimizing the runtime memory overhead.

4.5.2 Evaluating Feasibility Tests for Sporadic Task Integration

In this section, we evaluate the schedulability and runtime performance of feasibility tests designed to safely extend periodic task sets with sporadic tasks. We structure our evaluation into two distinct experiments, with the main objective of identifying the most efficient and robust feasibility test for online deployment.

The first experiment compares the execution times of the exact feasibility tests. Because these tests are mathematically exact, they yield identical schedulability results by definition. Thus, the comparison focuses exclusively on runtime performance. The evaluated exact tests include:

- **PDA**: The Processor Demand Analysis formulated in Theorem 3.

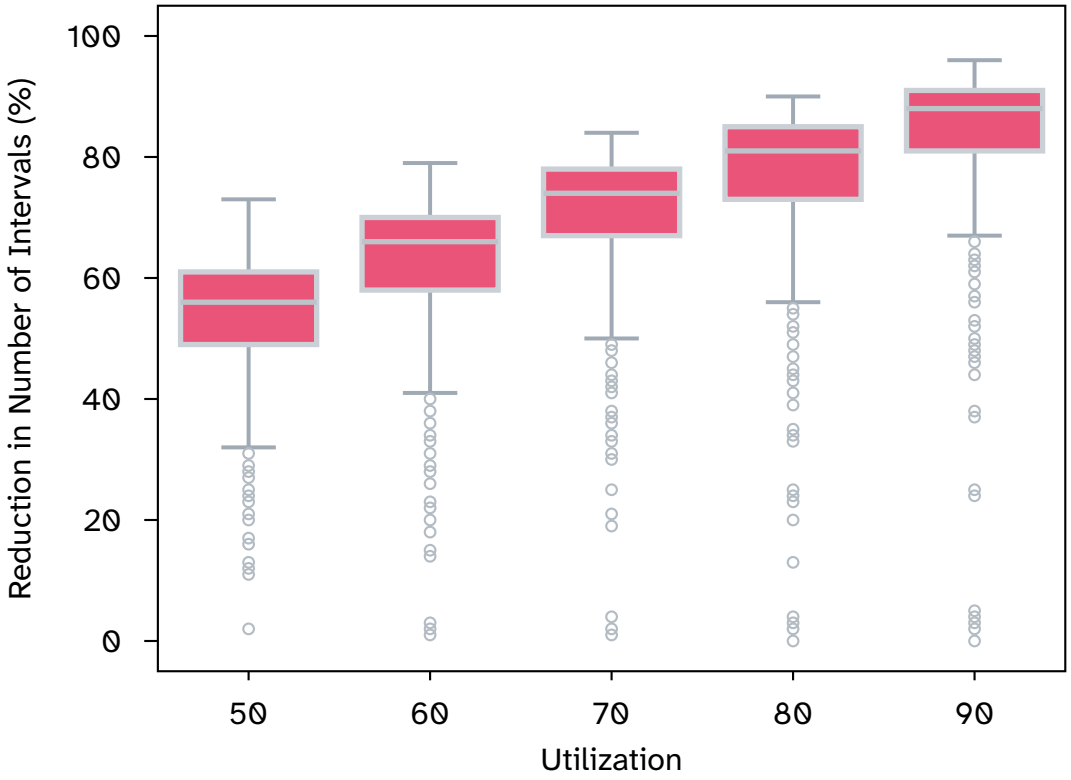


Figure 4.5.3: Reduction of created intervals by DTSS compared to Slot-Shifting for the randomly generated task sets relative to their utilization.

- **PDA***: The optimized PDA variant incorporating the offline pre-processing step defined in Theorem 4. This test utilizes a stored array of pairs $(t, \mathbf{dbf}_{\Gamma_P}(t))$, for each t where $\mathbf{dbf}_{\Gamma_P}$ in Equation (4.3.21) is discontinuous.[6]

- **QPDA***: A further refinement of *PDA** that integrates the QPA technique introduced by Zhang and Burns in [113]. The test is described in Algorithm 7.

- **ILP**: An *Integer Linear Programming* (ILP) formulation of the PDA from Theorem 3. Rather than exhaustively checking all valid interval lengths, the ILP solver directly attempts to identify a single interval where the processor demand exceeds the interval length (indicating infeasibility). The formulation is detailed in [55]. We implemented this ILP formulation using the *Gurobi* solver [255].[7]

The second experiment evaluates our proposed exact tests (*PDA*, *PDA**, and *QPDA**) against established state-of-the-art EDF-based feasibility tests, comparing both

[6]The reported execution times for *PDA** and *QPDA** exclude the offline pre-processing phase, as this overhead is incurred exclusively at design time. The computational cost of this offline phase is comparable to running the original PDA.

[7]Initial experiments were also conducted using the *SCIP* solver [256]. However, *Gurobi* consistently outperformed it across our test suites.

schedulability ratios (success rates) and runtime efficiency. A broader comparison incorporating the offline schedule modification methods discussed in Section 4.1 is deferred to future work.

Specifically, we compare our methods against Slot-Shifting's offline admission test for sporadic tasks in time-triggered systems (detailed in Section 4.3.2.1), hereafter referred to as *OTSS*. We assume the parameters of the generated periodic tasks align perfectly with the target execution windows calculated during Slot-Shifting's offline phase. The available spare capacity is subsequently computed based on the ALAP execution of periodic tasks under EDF using a fixed invocation gap size, as specified in [125].

All feasibility tests were implemented in Python. While the Gurobi solver leverage Python as an interface, its core functionalities are highly optimized and written in C[8].

We conducted all experiments on an Intel Core™ i7-9700 CPU @ 3.00 GHz machine with 24 GB RAM, running Ubuntu 24.04.2 LTS. We confined the execution of the tests to a single CPU core and ran them using PyPy version 7.3.19, with the exception of the Gurobi ILP solver (version 12.0.2), which we ran with the CPython interpreter due to its current incompatibility with PyPy.

For all experiments, we restrict task parameters to natural numbers. Moreover, for any task τ_k, its deadline D_k is uniformly drawn from the upper half of the interval $[C_k, T_k]$, whereas its offset, if periodic, is uniformly drawn from the interval $[0, T_k)$.

4.5.2.1 Comparing Runtime of Exact Feasibility Tests

For this experiment, we generated synthetic task sets using *Roger Stafford's randfixedsum* algorithm [260]. For that, we extended the task generator from [261] to create mixed asynchronous periodic and synchronous sporadic task sets with specific hyperperiods. The *randfixedsum* algorithm produces rational WCET values for tasks in order to meet a predefined target utilization. Hence, to constrain WCET values to natural numbers and to prevent the occurrence of zero-valued WCET due to rounding, we begin by assigning each task an initial WCET of 1. We then generate rational WCET values using *randfixedsum*, round them to the nearest integer, and add them to the initial values. This process is repeated until the resulting total utilization matches the target within a 2% tolerance threshold.

Since input size directly affects the runtime cost of feasibility tests, we created multi-

[8] Implementing all tests in C or some other optimized compiled language would arguably provide a more balanced comparison, potentially yielding performance improvements of up to 25 to 30 times for the other tests, as indicated by previous studies on compiled language efficiency [257, 258]. However, to prioritize rapid development, we chose to forgo a native C implementation, and instead mitigate the runtime overhead by running the tests implemented in Python using PyPy's Just-in-Time (JiT) compiler [259]. We observed an average runtime improvement of approximately 6x relative to executing PDA using the CPython interpreter.

ple one-parameter-at-a-time test suites. In each test suite, we vary one input parameter and fix the rest.

Test Suite 1: Runtime vs. Task Set Size. In this test suite, we vary the number of tasks per task set, from 5 to 100 in increments of 5. For each configuration, we generate 100 task sets, resulting in a total of 2000 task sets. Each task set consists of 60% periodic tasks and 40% sporadic tasks. For all task sets, we set the target utilization to 85% and the hyper-period to 1000. The task set generation deliberately avoids configuring all periods as pseudo-harmonic to preserve the generality of the results. Therefore, we uniformly select periods from the set: {4, 8, 10, 20, 25, 40, 50, 100, 125, 200, 250}.

Figure 4.5.4 shows the runtime (logarithmic y-axis) for all feasibility tests as function of task set size (x-axis). The runtime performance of both *PDA* and *ILP* is comparable and grows considerably with task set size. However, thanks to the pre-processing step, *PDA** achieves a two to three orders of magnitude improvement over *ILP* despite its pure Python implementation. Furthermore, *QPDA** is around 2.3 times faster than *PDA** on average. However, in a few outlier cases within the test suite, *QPDA** exhibits lower performance compared to *PDA**.

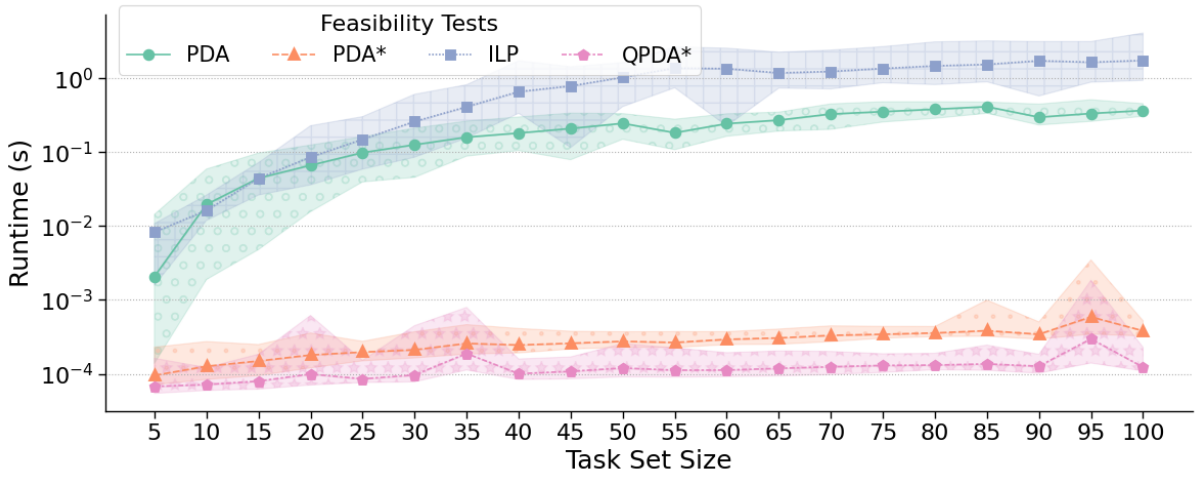


Figure 4.5.4: Mean runtime of exact feasibility tests as a function of task set size; hatched areas show min-max observed values.

Test Suite 2: Runtime vs. Hyper-period Length. In this test suite, we vary the length of the combined task set hyper-period across multiple configurations. For each value in the sets $\{0.5 \times 10^3, 10^3, 5 \times 10^3, 10^4, 2.5 \times 10^4\}$ and $\{5 \times 10^4, 10^5, 5 \times 10^5\}$, we generate 100 and 10 task sets, respectively (530 sets in total). Each task set contains 30 tasks, composed of 60% periodic and 40% sporadic tasks, with a combined utilization of 85%.

Figure 4.5.5 shows the runtime (logarithmic y-axis) for all feasibility tests as a function of hyper-period length (logarithmic x-axis). The runtime of *PDA* grows fastest with the hyper-period, whereas *ILP* maintains relatively stable runtime across different hyper-period lengths. This is expected, as *PDA* scales with the number of release times and

deadlines within $\mathrm{HP}(\Gamma_\mathrm{P}) \times \mathrm{B}(\Gamma)$ (see Theorem 3), whereas the *ILP* solver identifies solutions that satisfy the imposed model constraints without direct dependency on hyper-period length. Thanks to the pre-processing step, *PDA** and *QPDA** no longer rely on the length of $\mathrm{HP}(\Gamma_\mathrm{P})$. Hence, they are expected to be at least $\mathrm{HP}(\Gamma_\mathrm{P})$ times faster than *PDA*. Both *PDA** and *QPDA** outperform ILP by approximately one to two orders of magnitude. Notably, *QPDA** performs comparably to *PDA** at shorter hyper-periods and exceeds its performance at longer hyper-periods, thanks to its deadline skipping optimization.

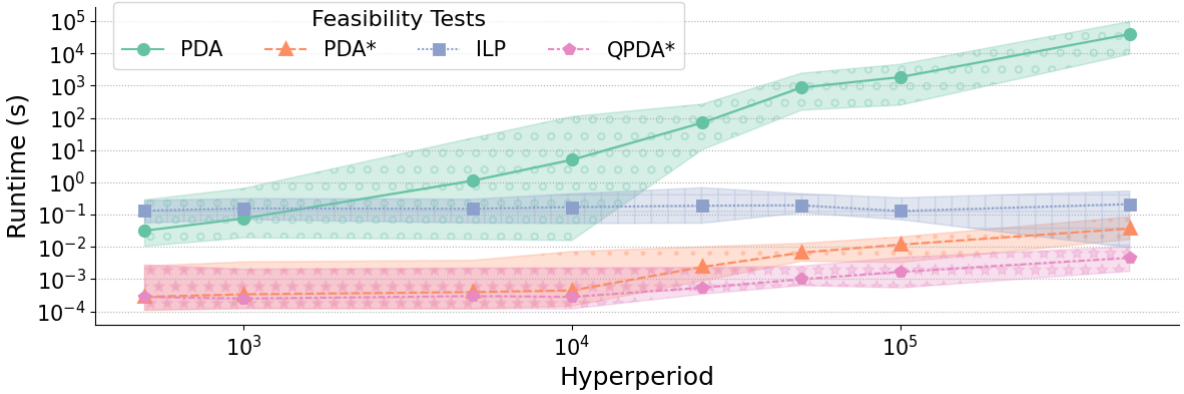


Figure 4.5.5: Mean runtime of exact feasibility tests as a function of hyper-period length; hatched areas show min-max observed values.

Memory footprint for (Q)PDA*. As demonstrated in the previous test suites, the pre-processing step employed by *PDA** and *QPDA** significantly improves runtime performance. However, as noted in Section 4.3.2, storing the demand from periodic tasks increases the memory footprint of the *PDA* test from polynomial to pseudo-polynomial complexity. Therefore, to further assess its suitability for runtime deployment, we calculate the memory requirement for storing the demand from periodic tasks considering the previous test suites and assuming that this demand is encoded in an array of integer pairs. Each pair $(t, \mathbf{dbf}_{\Gamma_\mathrm{P}}(t))$ holds the value of $\mathbf{dbf}_{\Gamma_\mathrm{P}}$ (see Eq. 4.3.21) at a discontinuity time point t, i.e., when $\mathbf{dbf}_{\Gamma_\mathrm{P}}$ changes value.

The results shown here are based on the a priori knowledge of the bound $\mathrm{B}(\Gamma)$. However, as discussed in Section 4.3.2, when the sporadic task set is not known offline, then this bound must be overestimated, which may lead to a larger memory footprint.

In Test Suite 1, the memory requirement ranged from just a few bytes to approximately 1.5 KB. By contrast, in Test Suite 2 the memory requirement exhibited more substantial increases as the hyper-period grew. Starting at less than 1 KB for $\mathrm{HP}(\Gamma) = 500$, the maximum observed memory requirement increased by roughly 2.5 KB per 1000 units of hyper-period. This trend continued up to $\mathrm{HP}(\Gamma) = 500{,}000$, where the maximum observed memory requirement dropped, with a maximum observed value of 600 KB.

Note that when feasible, opting for smaller integer types can reduce memory usage, e.g., using unsigned 16-bit integers may cut the memory footprint in half.

We did not observe a clear correlation between the maximum memory requirement and the task set size. Although hyper-period length appears to have a stronger influence, the relationship is not straightforward. The memory required for storing the pre-processed demand is primarily driven by the length of the interval $[0, B(\Gamma)]$ and the number of discontinuities in $\mathbf{dbf}_{\Gamma_P}(t)$ within that interval. These factors tend to grow with longer hyper-periods, influenced by task utilization and the difference between task periods and deadlines (see Equation (2.1.7)).

It is worth noting that many real-time systems operate with hyper-periods ranging from a few hundred to a few thousand time units. These systems typically have sufficient memory resources to accommodate the overhead, positioning *PDA** and *QPDA** as a viable feasibility test for runtime deployment.

4.5.2.2 Comparing Runtime and Schedulability of EDF-Based Tests

For this experiment, we compare EDF-based feasibility tests based on task sets generated according to the *Real-World Automotive Benchmarks* in [262]. For that, we developed a task-set generator that selects periods from the set $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}ms$ using the distribution in Table III of [262], excluding angle-synchronous activation and re-normalizing the remaining probabilities. For each chosen period, the generator samples runnables that satisfy the constraints in Tables IV and V of [262], then sums their execution times to form a single task.

Furthermore, since $OTSS$ is based on a DTT scheduler, we evaluate it considering an invocation gap of $T_{IG} = 100\ \mu s$ and scale the task set parameters accordingly (referred to as $OTSS100$). To illustrate the trade-off between runtime efficiency and schedulability, we also present results for $OTSS$ under a $1\ \mu s$ slot-size configuration ($OTSS$), although such fine granularity is generally impractical in TT systems due to the substantial scheduling overhead [123].

For all generated task sets, we fixed the utilization of periodic tasks at 30% and varied the utilization of sporadic tasks from 20% to 70% in 10% increments. We generate for each configuration 100 feasible task sets (600 in total), each consisting of 15 periodic and 15 sporadic tasks. Note that since all tests in this experiment are purely implemented in Python, the runtime performance comparison is more accurate compared to the first experiment.

Figures 4.5.6 and 4.5.7 illustrate the runtime performance (logarithmic y-axis) and schedulability success rates, respectively, for the feasibility tests across varying task set utilization values. As utilization increases, the schedulability of *OTSS* declines sharply. On average, *OTSS100* achieves a schedulability rate of only 12.76%, approximately 4%

lower than that of *OTSS*. Increasing the scheduler invocation gap size from 1 to $100\mu s$ results in a runtime reduction of one to two orders of magnitude, with higher utilization values leading to earlier violations of *OTSS*'s feasibility checks. Compared to the exact tests, *OTSS100* performs better than *PDA* in terms of runtime, but is consistently outperformed by our tests with pre-processing *PDA** and *QPDA**, which achieve two to three order of magnitude better performance.

Note that even if *OTSS* were deployed for runtime use, it would have comparable memory requirements to *QPDA**, as it needs to store Slot-Shifting's capacity intervals throughout the entire TT schedule (see Section 4.3.1.1).

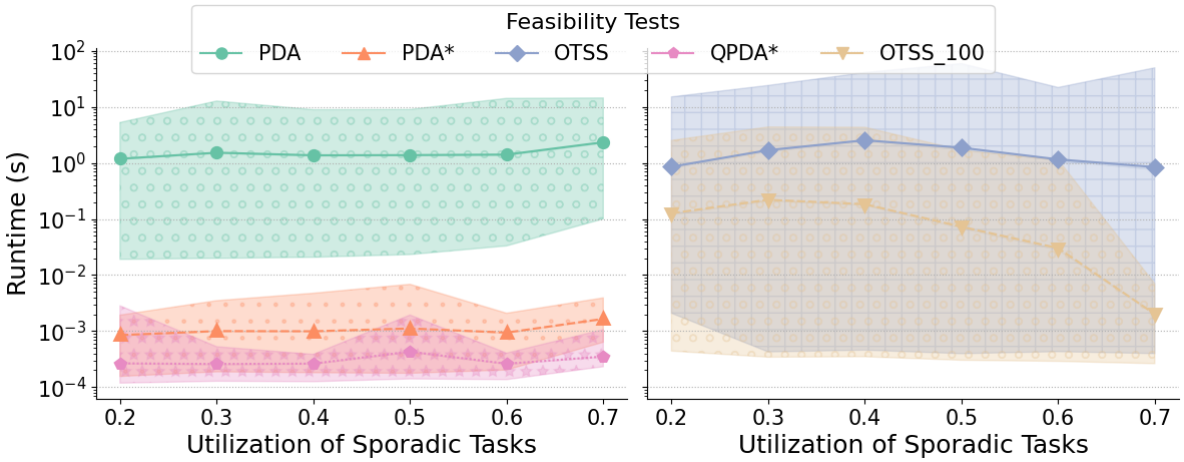


Figure 4.5.6: Mean runtime of feasibility tests as a function of task set utilization; hatched areas show min-max observed values. The utilization of periodic tasks is fixed at 30%.

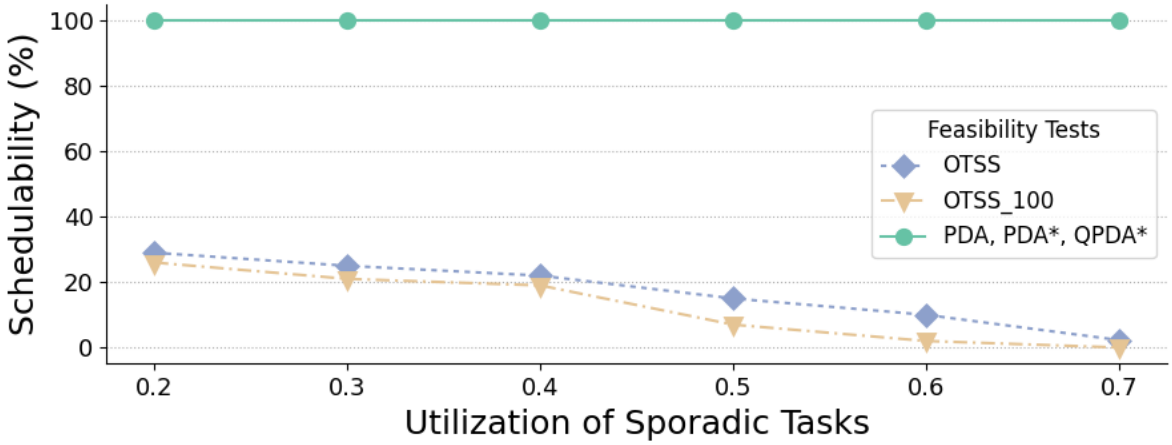


Figure 4.5.7: Schedulability success percentage of feasibility tests as a function of task set utilization. The utilization of periodic tasks is fixed at 30%

We also explored a modified variant of *OTSS*, based on the notion of simulation interval [263], that eliminates its pessimism (effectively making the test exact) by assuming as soon as possible execution of periodic tasks and choosing the release times of periodic tasks as global offsets for the synchronous arrivals of sporadic tasks. Despite this refinement, the test consistently underperformed in terms of runtime compared to the original *OTSS* and the *PDA*. Consequently, *QPDA** emerges as the most suitable option for runtime deployment, offering superior overall performance in terms of both runtime efficiency and schedulability.

4.5.3 Evaluating Feasibility Tests for Periodic Task Integration

In this section, we evaluate the schedulability and runtime performance of feasibility tests designed to safely extend a baseline asynchronous periodic task set Γ_{P} with a set of synchronous periodic tasks Γ_{PS}.

As established in Section 4.3.3, exact feasibility analysis of the combined task set $\Gamma = \Gamma_{\mathsf{P}} \cup \Gamma_{\mathsf{PS}}$ remains strongly coNP-complete, necessitating the use of sufficient pseudo-polynomial tests for online admission control. Because these tests exhibit different levels of analytical pessimism, this experiment considers the following tests to evaluate the trade-offs between their schedulability and runtime performance.

- **PDA_ASYNC**: The exact Processor Demand Analysis from [63] (see Equation (4.3.23)), which evaluates the true temporal phase relationships of periodic tasks with offsets.
- **PDA_SYNC**: The sufficient PDA from [110], which assumes all tasks in the system are synchronous (i.e., all offsets are forced to zero). This yields a computationally simpler, yet highly pessimistic, sufficient test.
- **1FT** and **2FT**: Sufficient pseudo-polynomial tests based on the M *fixed tasks* (MFT) approach from [109] for $M = 1$ and $M = 2$, respectively. These tests reduce PDA_SYNC's pessimism by evaluating all mathematically valid phase alignments among the M fixed tasks, while shifting the remaining non-fixed tasks to their minimum possible relative arrival times.
- **QPDA_PREP**: Our proposed sufficient test (Algorithm 7), which leverages the offline pre-processing of the baseline demand combined with quick-convergence techniques. It introduces analytical pessimism by treating the synchronous task set Γ_{PS} as a set of sporadic tasks[9]

[9]The reported execution times for *QPDA_PREP* exclude the offline pre-processing phase, as this overhead is incurred exclusively at design time. The computational cost of this offline phase is comparable to running the *PDA_ASYNC* on the baseline task set.

4.5.3.1 Experimental Setup

We generated mixed task sets based on the *Real-World Automotive Benchmarks* [262] to evaluate the performance of these feasibility tests under realistic conditions. We developed a custom task-set generator configured to produce a baseline task set Γ_{P} consisting of 15 tasks, alongside a synchronous extension task set Γ_{PS} comprising 10 tasks. The generator assigns periods from the set $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ ms following the distribution in Table III of [262], excluding angle-synchronous activations and re-normalizing the remaining probabilities. For each chosen period, the generator samples runnables that satisfy the constraints in Tables IV and V of [262], then sums their execution times to form a single task. For all task sets, the timing parameters are restricted to the integer domain. Furthermore, for any task τ_k, its relative deadline D_k assumes a constrained-deadline model and is drawn uniformly from the interval $[C_k, T_k]$.

The baseline asynchronous task set Γ_{P} was evaluated at fixed utilizations of 40%, 50%, 60%, 70%, and 80%. For each baseline configuration, the utilization of the synchronous extension Γ_{PS} was varied in 10% increments. For each distinct utilization configuration, we generated 100 independent task sets (2000 task sets in total).

To simulate a dynamic runtime admission request, we select the shared offset for all tasks in Γ_{PS} uniformly from the baseline hyperperiod, such that $\phi_{PS} \in [0, \mathsf{HP}(\Gamma_{\mathsf{P}}))$. Under this configuration, only the exact test (PDA_ASYNC) considers the true temporal phase relationships against the periodic arrivals of tasks in the baseline set.

All algorithms were implemented in Python and executed on a single core using PyPy version 7.3.19. The experiments were performed on an Intel Core i7-9700 CPU at 3.00 GHz with 24 GB of RAM, running Ubuntu 24.04.2 LTS.

4.5.3.2 Comparing Schedulability Performance

The schedulability success rates across the varying baseline (Γ_{P}) and extension (Γ_{PS}) utilizations are illustrated in Figure 4.5.8. As expected, the exact test *PDA_ASYNC* establishes the theoretical upper bound, yielding the highest schedulability success by evaluating the true temporal phase alignments between the baseline tasks and the synchronous extension. Conversely, the *PDA_SYNC* test exhibits the largest analytical pessimism, resulting in a sharp decline in admitted task sets, particularly as the total system utilization increases.

Instead of assuming a fully synchronous worst-case arrival pattern where all offsets are zero, the MFT tests *1FT* and *2FT* capture the exact alignments of the fixed tasks while introducing pessimism by pulling back the non-fixed tasks to their minimum possible relative differences with the fixed tasks, thereby successfully reclaiming a significant portion of this lost schedulability. At lower baseline utilizations (e.g., 40% and 50%),

2FT generally outperforms all other sufficient tests.

Our proposed sufficient test, *QPDA_PREP*, introduces pessimism exclusively to the extension task set Γ_{PS} by treating its tasks as sporadic, while preserving the exact baseline phase relationships through offline pre-processing. As illustrated in the performance results, this isolation of pessimism yields a crossover behavior. At lower baseline utilizations (40% and 50%) when the extension set constitutes a larger relative portion of the total utilization, the pessimism applied to Γ_{PS} dominates, causing the success rate of *QPDA_PREP* to fall below those of both *1FT* and *2FT*. However, as the baseline utilization grows (60% and 70%), *QPDA_PREP* leverages its zero-pessimism baseline and becomes increasingly competitive, ultimately surpassing *1FT* and achieving schedulability comparable to, or even exceeding that of *2FT*.

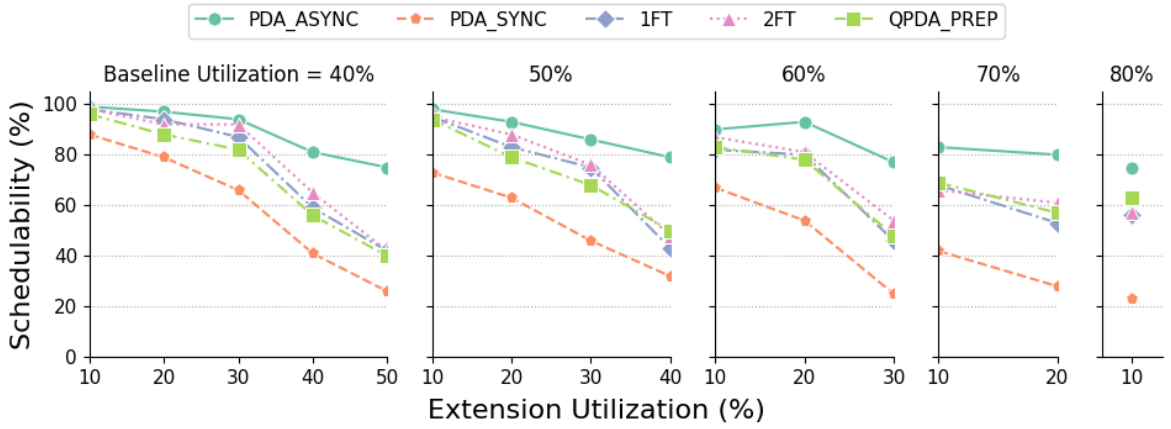


Figure 4.5.8: Schedulability success percentage of the feasibility tests as a function of Γ_{PS} utilization across different baseline Γ_P utilizations.

4.5.3.3 Comparing Runtime Performance

While some tests provide superior schedulability, their runtime performance determines their suitability for online admission control. Figure 4.5.9 illustrates the mean runtime of the evaluated feasibility tests on a logarithmic scale as a function of the extension task set (Γ_{PS}) utilization. Notably, the runtime for all considered tests remains somewhat stable across varying utilizations. This stability is a direct consequence of the fixed task counts and choice of periods, which restricts the number of steps required by these tests. Nevertheless, as these timing parameters are based on the Real-World Automotive Benchmarks [262], this bounded complexity accurately reflects actual automotive workloads, ensuring realistic runtime evaluations.

The analysis time of the exact *PDA_ASYNC* test is computationally prohibitive, averaging approximately 2×10^{-1} seconds per evaluation. Because it necessitates an exhaustive two-dimensional search across interval combinations up to the combined

hyperperiod, the approach scales poorly and is impractical for online deployment. Furthermore, although *2FT* improves schedulability over simpler alternatives, it incurs a significant computational penalty (averaging 10^{-1} seconds) due to the overhead associated with iterating over combinations of multiple fixed tasks and their relative distances. The *1FT* approach offers a more balanced trade-off, reducing the evaluation time to approximately 3×10^{-3} seconds.

In contrast, *QPDA_PREP* demonstrates exceptional runtime efficiency. Rather than repeatedly executing N distinct feasibility tests over artificially constructed busy periods, as necessitated by the MFT methods, *QPDA_PREP* offloads the computationally intensive analysis of the asynchronous baseline task set entirely to the offline design phase. Consequently, the online admission check is simplified to a one-dimensional evaluation of the synchronous extension, requiring an average analysis time of merely 10^{-4} seconds. As illustrated in the figure, the quick-convergence technique integrated into *QPDA_PREP* enables it to complete its evaluation even faster than the highly pessimistic *PDA_SYNC* test, which averages approximately 2.5×10^{-4} seconds per task set.

By achieving runtime performance that is at least three orders of magnitude faster than both *PDA_ASYNC* and *2FT*, and more than one order of magnitude faster than *1FT*, *QPDA_PREP* establishes itself as a highly practical solution for online admission control. It delivers competitive schedulability while simultaneously maintaining the lowest runtime among the evaluated methods.

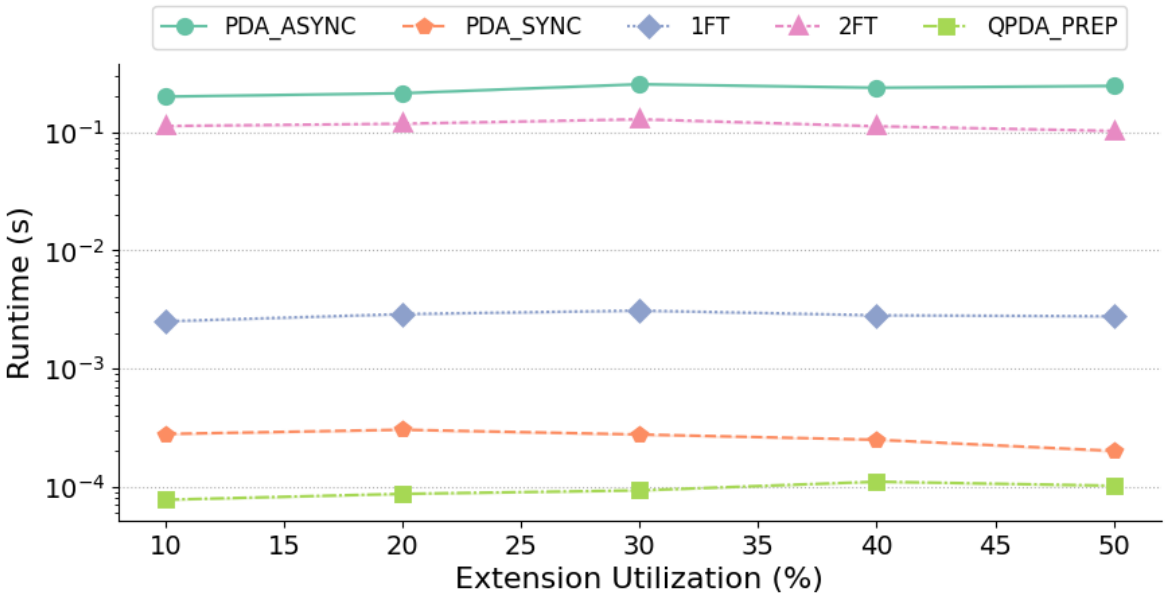


Figure 4.5.9: Mean runtime of feasibility tests as a function of Γ_{PS} utilization.

"The only **truly secure system** is one that is **powered off**, cast in a **block of concrete** and sealed in a **lead-lined room** with **armed guards**, and even then **I have my doubts**."

(Eugene H. Spafford)

Chapter 5

Intrusion Resilience in Time-Triggered Systems

Chapter 1 highlighted a growing mismatch between the closed-world assumptions of the original TTA and the realities of modern open, networked cyber-physical systems. Addressing RQ3 of this thesis, in this chapter, we investigate how TTA-based systems can be equipped with intrusion-tolerant mechanisms that preserve hard real-time guarantees and safety properties even in the presence of compromised nodes and sustained adversarial behavior.

We begin by reviewing related work on security and intrusion-resilience in TTA-based systems, followed by an overview and comparison of guardian mechanisms across different TT protocols. We then introduce our fault model and analyze how TTA-based systems can be exploited through resource-exhaustion attacks and synchrony manipulation directed at nodes and guardians. Based on these insights, we develop a replica-aware rejuvenation service that can be integrated into the TTA (Section 5.4), ensuring that nodes are restored to a safe and secure state faster than an adversary can compromise system safety.

Given the increasing heterogeneity and openness of TT networks, we revisit network-agnostic guardians as a promising architectural direction. We design novel time-domain attacks that target their synchronization behavior, derive theoretical results on their feasibility, and propose the first time-domain-attack-resilient guardian designs (Section 5.5.6) capable of enforcing rejuvenation and maintaining collision-free communication under adversarial conditions.

5.1 Related Work

5.1.1 Security and Dependability in the TTA

There has been extensive academic and industrial research on the TTA, each strand addressing specific aspects relevant to particular application domains. Several works have explored enhancing the TTA with additional dependability or security capabilities, yet to the best of our knowledge, none have approached these extensions from the perspective of *intrusion resilience*.

For example, a fault-management protocol for TTP/C was proposed in [264], introducing FTUs with shared replicated nodes and spare nodes that can assume the roles of failed ones based on static reconfiguration tables and periodically exchanged information from the management nodes. In [153], a clique resolution algorithm was presented as a self-stabilizing mechanism to recover from transient upsets affecting multiple or even all nodes in a TTA cluster, such as those caused by severe environmental disturbances. Similarly, [206] investigated how to increase robustness against bursty random perturbations by distributing TDMA slots across the communication round to reduce the likelihood of losing all replicas of a message.

Security-oriented extensions have also been proposed. In [51], Wasicek introduced a secure start-up service to protect application software from malicious modification, a secure communication service based on time-delayed symmetric key release for message authentication and detection of injected forged messages, and a secure global-time service that combines fault-tolerant clock synchronization with continuous clock-state monitoring via a dedicated node. More recently, PCSPOOF [130] demonstrated that TTA-based systems remain susceptible to adversarial interference, achieving the first documented breach of isolation in TT Ethernet and inducing synchronization loss and critical-message omission.

While these mechanisms strengthen the security and dependability of the TTA, the architecture remains, as will be shown in Section 5.3, vulnerable to persistent attacks that violate its system and fault models, particularly those targeting synchrony and replication assumptions.

5.1.2 Real-Time Systems Attacks and Countermeasures

Driven by the need to defend real-time systems against malicious cyberattacks, a large body of work has focused on mitigating timing vulnerabilities in these systems[1]. Völp et

[1]The Aurora generator test was the first to demonstrate how cyber-attacks can cause physical damage, proving that hackers could remotely cause a generator to destroy itself [45], a principle later exploited in the Stuxnet worm [265] and in the cyberattack on the Ukrainian power grid [41]. Redirected to U.S. soil, such an attack could have caused a nationwide catastrophe, as highlighted in the U.S. Federal

al. investigated timing leakage prevention in fixed-priority schedules [267] and in real-time locking protocols [268]. Mohan et al. [269] examined leakage of the schedule itself, while Yoon et al. [270] introduced task randomization for RM scheduling. Krüger et al. [129] explored schedule randomization based on a modified version of Slot Shifting to mitigate timing-inference-based attacks in TT systems.

While these methods can hinder adversaries from gathering critical timing information and mounting successful attacks, their effectiveness diminishes over long mission durations, a characteristic of many modern safety-critical real-time systems. In this thesis, we therefore shift the focus from intrusion prevention to *intrusion resilience*, and investigate how TTA-based systems can be equipped with mechanisms such as proactive and reactive rejuvenation (see Section 2.2.8.2) that allow them to survive malicious intrusions while preserving their hard real-time guarantees.

5.1.3 Rejuvenation

Proactive recovery was first explored in the context of BFT SMR protocols (see Section 2.2.6.3) in distributed systems [162]. Sousa et al. [174, 186] were the first to recognize that, under persistent attacks, traditional fault-tolerance mechanisms become ineffective as adversaries gradually exhaust the system's ability to mask faults. They introduced the notion of *exhaustion safety* and proved that a replicated system remains exhaustion safe if it can rejuvenate all replicas faster than an adversary can compromise more than the fault threshold of F replicas (see Section 2.2.8.2).

Building on this foundation, Sousa et al. [182] investigated the interplay between proactive and reactive rejuvenation and derived conditions under which a BFT asynchronous system remains exhaustion safe, assuming that replica rejuvenation always succeeds. In contrast, we do not assume that rejuvenation of a node is always successful. Instead, we extend the rejuvenation mechanism with the ability to detect and react to unsuccessful rejuvenations, ensuring continued safety even when rejuvenation attempts fail.

Our proposed rejuvenation service is tailored to the synchronous model and therefore requires fewer replicas than approaches designed for the asynchronous case (see Section 2.2.8.2). It intertwines rejuvenation with TTA's global time and membership services, enabling the *rejuvenation service* to track which replica is currently undergoing rejuvenation (whether proactively or reactively) and to adjust message expectations accordingly, thereby making the rejuvenation service rejuvenating-replica-aware.

Energy Regulatory Commission Order RD14-6-000 [266].

5.1.4 Guardians

As discussed in Section 3.1.7.4, guardians enforce node conformance to the communication schedule (e.g., the MEDL) and convert faulty node behavior in the time domain into fail-silence as perceived by other system nodes. This is achieved by equipping the guardian with detailed knowledge of the TT protocol's operation, including the communication schedule, clock synchronization, channel coding, and protocol frames.

Since implementing a guardian ultimately requires replicating parts of the communication protocol stack, existing guardian designs differ primarily in the targeted network topology and the amount of replicated functionality. Table 5.1.1 compares several guardian implementations across protocols that provide TT communication services. Designs that realize the guardian as a fully independent networked component [23,191,192] offer strong fault-independence but are typically complex and costly. Conversely, simpler guardian designs reduce cost at the expense of fault tolerance features.

A representative example of the latter is the local bus guardian in the TTA prototype implementation [127] (see Figure 3.1.2). For simplicity and cost reduction, this guardian controls node access to both communication channels but does not independently perform clock synchronization. Instead, it relies on the controlled node's communication controller to indicate the start of each TDMA round, allowing the guardian to synchronize its local clock and determine the node's slot position. Although this design can be made network protocol and topology independent, it cannot tolerate Byzantine faults: A malicious node can desynchronize its guardian from global time and cause collisions for the duration of its allocated TDMA slot in every TDMA round. Thus, this implementation assumes fail-silent node behavior and uses the guardian primarily to support timely error detection.

Other protocols have explored alternative guardian placements. The TEA protocol [271] proposes placing two nodes on the same chip and using their communication controllers as mutual guardians. This design, however, places both nodes in the same FCR (see Section 2.2.1.4), meaning that a failure in one may induce a failure in the other. The BRAIN protocol [272] leverages its ring topology by relying on a node's direct neighbors to implement the guardian functionality, preventing the forwarding of out-of-slot messages.

In Section 5.5.6, we introduce distributed, network-agnostic guardians (Synguards) that extend the design of [127] to tolerate malicious Byzantine faults while retaining low complexity and independence from network topology and protocol details. SyncGuards are placed locally to each node and communicate through it. Beyond enforcing network access, they incorporate mechanisms to trigger and coordinate node rejuvenation, enabling them to safely re-synchronize even when nodes may fail arbitrarily.

Table 5.1.1: Overview of guardian implementation in different TT protocols

Protocol	Topology	Location of Guardian Functionality	Guardian Complexity
TTA Ultimate [190]	Dual bus	Off-chip local guardian component per node per bus	High
TTA Prototype [127]		On-chip local network guardian component that depends on the node communication controller for clock synchronization	Low
FlexRay Bus [192]		On-chip local guardian component per node per bus	High
TEA [271]		On-chip pair of communication controllers that act as guardian to each other but belong to two different nodes	Low
TTA Star [191]	Dual star	Central guardian functionality implemented at each star coupler and controls access of all directly connected nodes	High
FlexRay Star [192]			High
TTEthernet [23]	Star, tree, or hybrid	Switches are equipped with central guardian functionality and control access of all directly connected nodes	High
BRAIN [272]	Braided ring	Instead of using special guardian components, BRAIN leverages its ring topology and uses nodes' direct neighbors as guardians	Low
SyncGuard (Sec. 5.5.6)	arbitrary ($F+1$ disjoint paths)	A distributed solution of local but indirectly communicating network access and node rejuvenation guardians (*SyncGuard*s)	Low

5.2 System and Fault Models

For the system model, we assume a TTA-based system consistent with the implementations described in Section 3.1.2. However, we depart from TTA's original fault hypothesis (see Section 3.1.5), which anticipates only accidental faults and transient upsets. In the context of open safety-critical real-time systems, this assumption is no longer sufficient. We therefore extend the fault model to include malicious actors capable of exploiting latent software vulnerabilities [51]. Once a node is compromised, the adversary is assumed able to execute arbitrary code, manipulate the node's local state, and orchestrate transmissions in pursuit of specific malicious objectives.

Autonomous real-time systems typically operate over prolonged mission durations without human supervision or intervention. This extended exposure provides adversaries with ample opportunity to compromise TTA nodes or guardians and use them as infiltrated outposts. A fundamental assumption in our model is that a compromised node may continue to operate seemingly within system specifications, thereby evading standard intrusion detection mechanisms while waiting for a vulnerability window to un-

dermine safety guarantees. This assumption is justified since a malicious or arbitrarily faulty node may be impossible to diagnose as faulty under certain conditions, even when intrusion detection mechanisms (e.g., firewalls) are deployed [273].

5.3 TTA's Vulnerability to Resource Exhaustion Attacks

In open and adversarial environments, safety-critical systems become vulnerable to *resource exhaustion attacks*. Such attacks aim to compromise a sufficient number of replicated components to violate the system's resource assumptions, thereby enabling the adversary to mount more targeted attacks with the ultimate goal of compromising system safety.

Although the TTA provides several dependability services and mechanisms such as guardians, the membership service, and the FTU layer, these mechanisms are designed to tolerate accidental faults rather than frequency-controlled or intentionally provoked failures. Moreover, they rely on global time and a priori knowledge of events, properties that an attacker can exploit to gather information and craft targeted attacks that circumvent these dependability measures.

This section analyzes the vulnerabilities in TTA-based systems that adversaries may exploit to induce resource exhaustion and disrupt system synchrony.

5.3.1 Synchrony Exhaustion

The goal of this attack is to corrupt the nodes' consistent perception of global time and system state, both of which are essential for correct operation in the TTA. By gradually disturbing synchrony, an adversary can cause the formation of cliques and repeatedly trigger global system failures.

The TTA employs a specialized implementation of the Welch–Lynch algorithm [201] to synchronize node clocks. This implementation tolerates a single arbitrary fault, requiring at least four clock readings to mask one faulty value. The algorithm discards the faulty reading by computing the average of the second-largest and second-smallest values. The clock readings are computed at each receiving node, which applies the acceptance window check (see Section 3.1.4.3) to validate incoming frames and uses their reception times to compute clock differences, specifically, by averaging the differences derived from both communication channels [21].

An attacker can exploit the acceptance window mechanism to subvert synchrony by compromising a frame-relaying FCR such as a guardian in the star implementation (see Figure 3.1.3), or a communication controller in the prototype implementation (see Figure 3.1.2). The malicious FCR may delay relaying frames inconsistently across nodes

while still ensuring that each frame arrives within the respective acceptance window. Because the frames appear valid, receiving nodes cannot detect the manipulation.

By repeatedly and inconsistently injecting such delays, e.g., dragging one subset of nodes A slower and another subset B faster, the attacker can eventually partition the cluster into cliques. The clique-avoidance algorithm will then repeatedly shut down minority cliques until the entire cluster becomes inoperative. Although [190] states that central guardians in the star implementation cannot store messages, the central-guardian design in [191] explicitly performs semantic frame analysis, implying that buffering is possible. If central guardians cannot be prevented from delaying frames, tolerating this attack would require at least $3F+1$ central guardians to mask F arbitrary faulty ones (see Section 2.2.4.3).

In the ultimate implementation, a faulty guardian can only mislead the node it controls. It may gradually desynchronize the node's clock, causing transmissions to partially occur outside their assigned slot and collide with other frames. However, as long as the second channel remains correct, the attack's impact is contained. The prototype implementation is more vulnerable to this attack. Because the local guardian relies on the communication controller for clock synchronization and controls access to both communication channels, a compromised controller can drift the guardian's notion of time away from global time. This causes the guardian to open its transmission window incorrectly, enabling the controller to transmit outside its slot on both channels (a concrete version of this attack is presented in Section 5.5.4). Such a controller may corrupt other nodes' frames, execute an SoS timing attack (see Section 2.2.3), or masquerade as another node by occupying a failed node's slot or transmitting in an empty slot. Cryptographic protection does not prevent this attack, as it only protects frame contents, not timing.

An attacker may also compromise nodes responsible for sending synchronization frames. For cost reasons, only a subset of TTA nodes are equipped with accurate oscillators, and only their frames contribute to clock correction. These nodes are identifiable from the MEDL, which marks synchronization frames using the *SYF* flag. As mentioned earlier, the clock synchronization algorithm requires at least four synchronization nodes to mask the arbitrary failure of one node. In the ultimate and star implementations, the broadcast nature of the channels and the SoS protection provided by guardians prevent a compromised node from causing inconsistent reception unless all four synchronization nodes are compromised. In contrast, in the prototype implementation, compromising a single node suffices to execute an SoS timing attack by transmitting at the slot boundary, causing some nodes to accept the frame and others to reject it, thus causing the formation of cliques.

Finally, if the MEDL specifies only one or two nodes to send integration frames, compromising these nodes prevents new nodes from integrating into the cluster.

5.3.2 Fault Masking Exhaustion

The TTA implements fault tolerance based on active replication (see Section 3.1.7.7). For example, replicated nodes within an FTU are typically arranged in a TMR configuration, where all replicas receive the same inputs, execute the same tasks, and receivers vote on their output messages, allowing the system to mask the failure of one node behind the majority of two correct replicas.

Consequently, an adversary who compromises two replicas of an FTU in a TMR configuration can fully control the FTU's output and mount attacks on the control loop to which the FTU belongs. With two compromised replicas, the attacker can perform value-domain attacks by manipulating controller outputs or actuator commands, potentially causing non-specified or irreversible system behavior.

Alternatively, the attacker may perform time-domain attacks by omitting to produce messages, thereby causing deadline misses and inducing a persistent denial-of-service (DoS) within the control loop. However, such time-domain attacks require compromising all replicas of the FTU, since omitting messages from only one or two replicas would be masked by the remaining correct replicas through the voting mechanism.

5.4 Replica-Aware Rejuvenation Service

As established in Section 5.3, intelligent adversaries can exploit the static fault-tolerance mechanisms of the TTA by systematically exhausting resources and subverting synchrony. Sustaining correct operation under such adversarial conditions requires shifting the architectural focus from static fault-tolerance to *intrusion resilience* and equipping TTA-based systems with dynamic defense mechanisms capable of continuously eliminating latent compromises before they escalate into system-wide failures. In this section, we present a replica-aware rejuvenation service as a defense mechanism that is specifically tailored for the TTA.

5.4.1 Rationale

The foundational premise of intrusion resilience, discussed in Section 2.2.8, requires that the vulnerability window of each node be strictly bounded. This can be achieved through proactive rejuvenation, in which all replicas are periodically recovered faster than an adversary can compromise more than the tolerated fault threshold. By doing so, the system maintains exhaustion safety by continually resetting the time-to-failure metric of its nodes.

However, proactive rejuvenation introduces a fundamental tension between fault containment and system availability. A rejuvenating node is temporarily unavailable and be-

haves as if it has crashed during its blackout interval. Coordinating rejuvenation across replicas is therefore essential to preserve service availability. Conversely, reactive rejuvenation, which is triggered upon fault detection, removes adversarial influence and restores the system's ability to tolerate faults. However, if the interplay between proactive and reactive rejuvenation is not carefully managed, the system's fault-masking capabilities may be jeopardized.

Prior work typically assumes that node rejuvenation always succeeds, an assumption that does not hold in reality (e.g., a node can become permanently faulty). When rejuvenation attempts fail, proactive schedules that do not account for replicas rejuvenation state can inadvertently undermine continued fault masking. This motivates the need for a rejuvenation service that is aware of the rejuvenation state of replicas and can adapt its scheduling accordingly.

Furthermore, TTA's synchronous system model and its built-in dependability services and mechanisms, including global time, membership, and guardians, provide the architectural means needed to track and reach agreement on replica rejuvenation state, making it possible to design a coordinated, replica-aware rejuvenation service specifically for TTA-based systems, while also reducing the number of simultaneously recovering replicas as described in Section 2.2.8.2.

5.4.2 Proactive-Reactive Rejuvenation of TTA Nodes

In this section, we present our rejuvenation mechanism designed specifically for the TTA. The focus is on how proactive and reactive must be scheduled to preserve availability and fault masking, and how our replica-aware design leverages TTA's architectural concepts and services to efficiently achieve resource exhaustion safety.

5.4.2.1 Design Assumptions

In the context of the system and fault model described in Section 5.2, we make the following assumptions:

- **Diversification and obfuscation**: FTU replicas are diversified (e.g., using N-version programming [175]) to avoid shared vulnerabilities, and obfuscation techniques [177] (such as address-space randomization [178]) are employed to prevent adversaries from reusing knowledge gained from previous compromises (see Section 2.2.8).
- **Secure initialization**: Security measures are enforced during node startup. A secure startup service (see Section 2.2.1.5) ensures that node configuration data (e.g., the MEDL schedule) and the application image are authentic and untampered. It

verifies the cryptographic checksum of the application binary against a reference stored in a *tamper-proof memory*. If verification fails, the node is forced into a fail-silent state.

- **Heterogeneous rejuvenation times**: Nodes may have different rejuvenation durations due to hardware and software diversity. We assume that a rejuvenating node n_i is characterized by a worst-case rejuvenation time T^i_{WCR}, measured relative to the rejuvenation trigger. Since the communication controller must be powered on or restarted by the host computer to exit the freeze state [21], the worst-case rejuvenation time of a node in the entire cluster is:

$$T_{WCR} = \max_{\forall n_i \in N} \left(T^{HC_i}_{WCR} + T^{CC}_{WCR} \right),$$

  where $T^{HC_i}_{WCR}$ and T^{CC}_{WCR} are the worst-case rejuvenation time of the host computer and communication controller, respectively.

- **Rejuvenation granularity**: Communication controllers use simple restart as their rejuvenation strategy. Host computers may use either OS-level warm boot or full hardware cold boot rejuvenation (see Section 2.2.8.2).

- **Integration-frame availability**: All nodes are assumed to send integration frames (i.e., $x\text{-}frames$ or $i\text{-}frames$) in the communication schedule, i.e., frames that explicitly carry the $CC\text{-}state$ (see Section 3.1.7.6).

- **Worst-case integration time**: We denote by T_{WCI} the worst-case time required for a node to integrate into a synchronous cluster. Starting from the moment a node begins listening on the communication channels, T_{WCI} bounds the time until it may transmit in its TDMA slot.

Finally, our rejuvenation mechanism targets the ultimate and star implementations of the TTA. The prototype implementation is excluded because its guardian cannot mask the faulty behavior of a compromised node, and thus rejuvenation alone considering the above assumptions is insufficient to ensure intrusion resilience. In Section 5.5.6, we show how the guardian design of the prototype implementation can be elevated to mask compromised node behavior and thereby support intrusion-resilient operation.

5.4.2.2 Proactive Node Rejuvenation

Proactive node rejuvenation periodically recovers node replicas while ensuring the availability of their provided service to remove stealthy intruders faster than a majority quorum of replicated nodes can be compromised. As mentioned in Section 3.1.7.7, an FTU_j

comprises $N_j = 2F_j + 1$ actively replicated nodes, where F_j is the number of maskable faulty nodes and j is the unique ID of the FTU in the TTA cluster. Proactive rejuvenation requires increasing the number of replicated nodes in each FTU_j by K_j additional nodes that rejuvenate in parallel while leaving enough replicas available to provision the service and to mask faults (see Section 2.2.8).

Therefore, the proactive rejuvenation of FTU nodes is performed in groups of up to K_j replicas. The number of rejuvenation groups for FTU_j is $G_j = \left\lceil \frac{N_j}{K_j} \right\rceil$, which are recovered every proactive rejuvenation period T_{PR_j}. We refer to the number of nodes in a proactive rejuvenation group $g_{m,j}$ as $K_{m,j}$, where $m \in \{0, \ldots, G_j - 1\}$ is the index of the proactive rejuvenation group within FTU_j.

For an FTU_j, the number of proactive rejuvenation groups G_j, worst case node rejuvenation time T^j_{WCR} and worst case node integration time T_{WCI} impose a lower bound on the proactive rejuvenation period T_{PR_j}. Specifically:

$$G_j \times (T^j_{WCR} + T_{WCI}) \leq T_{PR_j}$$

This bound ensures that all rejuvenation groups of FTU_j can complete their rejuvenation and reintegration before the next proactive rejuvenation cycle begins. While T_{PR_j} may accommodate leeway, allowing the system designer to tune the rejuvenation frequency of FTU_j, this flexibility is constrained by the time adversaries require to compromise $F_j + 1$ nodes in FTU_j (see Section 2.2.8.2).

By using global time abstractions that relate TDMA rounds (see Figure 3.1.4) to the aforementioned rejuvenation parameters, we design an efficient proactive rejuvenation scheme for the TTA. Since rejuvenation considers all FTUs within the cluster, we also refer to the proactive rejuvenation period as the cluster rejuvenation cycle $crcycle$. We divide $crcycle$ into G non-overlapping intervals of fixed length, which we call proactive rejuvenation slots PRS. The length of a rejuvenation slot T_{RS} must be large enough to accommodate T_{WCR} and T_{WCI}, but can be stretched to be a multiple of the cluster's TDMA round T_{round} (e.g., $T_{RS} = Z \times T_{round}$).

A proactive rejuvenation slot PRS_m represents a place holder for recovering rejuvenation group g_m of all FTUs in the TTA cluster. As different FTUs in a TTA cluster may be planned or deployed with a different number of replicated nodes, the number of rejuvenation groups (i.e., the number of proactive rejuvenation slots) can vary from one FTU to another. Targeting efficient integration within the communicated state of the communication system (i.e., the $CC\text{-}state$), we fix the number of proactive rejuvenation slots for a TTA cluster ς and calculate it based on the largest number of rejuvenation groups in all its planned FTUs, potentially extending the rejuvenation period of FTUs with fewer replicas. The number of proactive rejuvenation slots and length of the cluster rejuvenation cycle can accordingly be calculated using equations (5.4.1) and (5.4.2)

respectively, while node allocation to these slots is determined by Equation (5.4.3).

$$G_\varsigma = \max_{\forall FTU_j \in \varsigma} \left\lceil \frac{N_j}{K_j} \right\rceil \tag{5.4.1}$$

$$T^{\varsigma}_{crcycle} = \sum_{i=0}^{G_\varsigma - 1} T_{RS_i} = G_\varsigma \times \left\lceil \frac{T_{WCR} + T_{WCI}}{T_{round}} \right\rceil \tag{5.4.2}$$

$$n_{i,j} \in g_{m,j} \; if \; i \; mod \, G_\varsigma == m \tag{5.4.3}$$

Figure 5.4.1 shows the proactive recovery scheme for a TTA cluster comprising two FTUs deployed with $N_0 = 5, K_0 = 2)$ and $(N_1 = 6, K_1 = 3)$ nodes respectively. We allocate, following Equation (5.4.3), a node $n_{i,j}$ of an FTU_j to a proactive rejuvenation group $g_{m,j}$ (i.e., to proactive rejuvenation slot PRS_m) offline based on its index i within FTU_j.

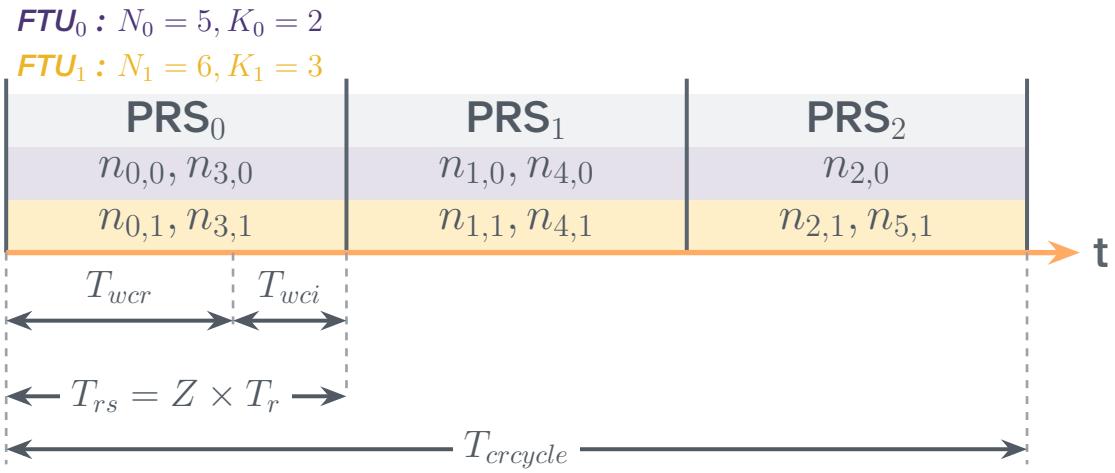


Figure 5.4.1: Example proactive rejuvenation scheme in the TTA

5.4.2.3 Reactive Node Rejuvenation

Reactive node rejuvenation recovers detected faulty nodes immediately to tolerate more faults. We use the clique avoidance mechanism of TTA's membership to exclude detected faulty nodes (see Section 3.1.7.2.3). We use this mechanism's maintained $accept$ and $reject$ counters to trigger reactive rejuvenation. When a node rejects more frames than it receives between two occurrences of its TDMA slot, it concludes that it belongs to a minority clique and has suffered a fault. The faulty node rejuvenates within a reactive rejuvenation slot RRS, which is of the same size as a proactive one and may overlap with them since reactive rejuvenation can occur at any point in time (See Figure 5.4.2).

Since a node may repeatedly undergo reactive rejuvenation and thus appear permanently faulty, we adjust the proactive rejuvenation schedule based on the global view of rejuvenation slots, the membership service, and the current cluster mode. This coordination accounts for unsuccessful rejuvenation attempts and their interaction with reactive rejuvenation. The underlying rationale is to make the rejuvenation service aware of the global system state and the number of currently available FTU replicas, ensuring that proactive rejuvenation is never triggered in a way that would render an FTU unavailable or undermine its fault-masking capability.

For that, we compute for each node $n_{i,j}$ the values N_j^{op} and $K_{m,j}^{op}$ using the node's view of the membership at the beginning of the proactive rejuvenation slot. These values indicate, respectively, the number of operational nodes within FTU_j and within proactive rejuvenation group $g_{m,j}$.

In an FTU_j, the rejuvenation of faulty nodes belonging to a proactive group $g_{p,j}$ during proactive rejuvenation slot PRS_m can delay the proactive rejuvenation of nodes in group $g_{m,j}$, which are normally scheduled for rejuvenation during PRS_m. As a result, $K_{m,j}$ is reduced in the worst case by the number of currently reactively rejuvenating faulty nodes to retain $2F_j + 1$ available nodes in FTU_j. Figure 5.4.2 illustrates how a permanently faulty node ($n_{0,0}$ in group $g_{0,0}$) trapped in an endless reactive rejuvenation loop causes the nodes of $g_{1,0} = \{n_{1,0}, n_{4,0}\}$ to be proactively rejuvenated in alternating turns every $crcycle$.

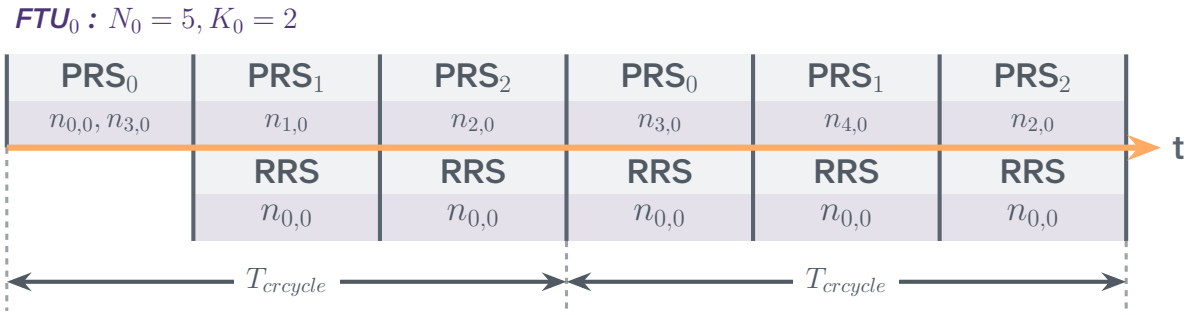


Figure 5.4.2: Example proactive-reactive rejuvenation in the TTA

We implement this behavior as follows: replicas assigned to a proactive rejuvenation slot are deterministically rejuvenated using a cyclic group rejuvenation window (grw) that may span multiple cluster rejuvenation cycles $crcycle$. A node $n_{i,m,j}^{op}$, which belongs to operational rejuvenation group $g_{m,j}^{op}$ (where $i \in \{0, \ldots, K_{m,j}^{op} - 1\}$ is the node index within group $g_{m,j}^{op}$) uses Equation (5.4.4) during its proactive rejuvenation slot to check whether it is allowed to proactively rejuvenate without hindering the fault masking capabilities of FTU_j.

The size of the group rejuvenation window may vary in each $crcycle$ and is equal to $K_{m,j}^{c}$; the number of replicas in $g_{m,j}$ that can be proactively rejuvenated within their

proactive rejuvenation slot PRS_m. When the group rejuvenation progress $prog_{pr}$ reaches $K^{op}_{m,j}$, all nodes in $g^{op}_{m,j}$ have been proactively rejuvenated.

$$grw_{n_{i,m,j}} = \left(\left\lceil \frac{\left((K^{op}_{m,j} - prog_{pr} + i) \bmod K^{op}_{m,j} \right) + 1}{K^{c}_{m,j}} \right\rceil - 1 \right) \tag{5.4.4}$$

$$K^{c}_{m,j} = \left\lfloor \frac{K^{op}_{m,j} \times (N^{op}_{j} - K^{op}_{m,j})}{2F_j + 1} \right\rfloor \tag{5.4.5}$$

5.4.2.4 Rejuvenation Algorithm

Algorithm 8 shows the main procedures of our rejuvenation mechanism that can be implemented at the communication controller of TTA nodes, and lists the required parameters and variables.

Procedures `reactive_recovery()` and `proactive_recovery()` are respectively responsible for rejuvenating the node reactively and proactively. The execution of both procedures follows the TT approach, i.e., they are triggered at cluster-wide agreed-upon points in global time. Therefore, these procedures are only effective when the synchronous operation is established in the cluster, i.e., the current cluster mode is not the cold start mode (lines 2 & 9). The `reactive_recovery()` procedure is triggered at the start of the node TDMA slot (i.e., during the PSP slot phase) and is used to rejuvenate the node if it rejects more frames than it accepts in the last TDMA round (the condition in line 3 evaluates to true). This mechanism builds on the membership and clique avoidance mechanism described in Section 3.1.7.2.3, which deems a node faulty if it is in disagreement with the majority of other nodes in the cluster.

In contrast, the `proactive_recovery()` procedure is triggered at the start of the node proactive rejuvenation slot PRS_m. When executed, `proactive_recovery()` calculates the cross sum over the bits in the membership vector that belong to the node's FTU and proactive rejuvenation group, to respectively get the number of operational nodes within them (lines 10-11). Using the calculated variables, the node further computes the number of proactively recoverable nodes in its group (line 12). Finally, the node saves the group rejuvenation progress in permanent storage and checks using equation (5.4.4), if it is located within the cyclic group rejuvenation window. If $grw_{n_{i,m,j}}$ evaluates to zero (line 16), the node proactively recovers in the current rejuvenation slot.

5.4.2.5 Overhead and Integration Support

By leveraging TT activation in relation to already existing data structures in the TTA, the overhead of the algorithm remains small. Reactive rejuvenation replaces the action of freezing a node in the clique-avoidance mechanism with recovering it. Proactive

Algorithm 8 The proactive-reactive rejuvenation algorithm

// Parameters for node $n_{i,m,j}$

x ▷ unique node ID within cluster ς
i ▷ unique node ID within group $g_{m,j}$
F_j ▷ number of tolerable faulty nodes in FTU_j
$K_{m,j}$ ▷ number of nodes in $g_{m,j}$
$T^{\varsigma}_{crcycle}$ ▷ length of the cluster rejuvenation cycle
Φ_j ▷ set holding IDs of nodes in FTU_j
$\Gamma_{m,j}$ ▷ set holding IDs of nodes in $g_{m,j}$

// Variables for node $n_{i,m,j}$ ▷ set upon node initialization and integration
$membership_x$ ▷ node local view of the membership vector
$accept_x$ ▷ counter for accepted frames since last TDMA round
$reject_x$ ▷ counter for rejected frames since last TDMA round
N^{op}_j ▷ number of operational nodes in group FTU_j
$K^{op}_{m,j}$ ▷ number of operational nodes in group $g_{m,j}$
$K^{c}_{m,j}$ ▷ number of proactively recoverable nodes in group $g_{m,j}$
$prog_{pr}$ ▷ counter for proactively rejuvenated operational nodes in $g_{m,j}$

1: **procedure** $reactive_recovery()$ ▷ triggered at the pre-send phase of $slot_x$
2: **if** $get_cluster_mode() \neq 0$ **then** ▷ 0 = cold start mode
3: **if** $accept_x < reject_x$ **then**
4: $recover()$
5: **end if**
6: **end if**
7: **end procedure**

8: **procedure** $proactive_recovery()$ ▷ triggered at the start of PRS_m
9: **if** $get_cluster_mode() \neq 0$ **then** ▷ 0 = cold start mode
10: $N^{op}_j = get_num_op_nodes(membership_x, \Phi_j)$
11: $K^{op}_{m,j} = get_num_op_nodes(membership_x, \Gamma_{m,j})$
12: $K^{c}_{m,j} = getSizeGRW()$ ▷ see equation (5.4.5)
13: $grw_{n_{i,m,j}} = getGRW()$ ▷ see equation (5.4.4)
14: $prog_{pr} = (prog_{pr} + K^{c}_{m,j}) \mod K^{op}_{m,j}$
15: $store_progress(prog_{pr})$ ▷ in persistent storage
16: **if** $grw_{n_{i,m,j}} == 0$ **then**
17: $recover()$
18: **end if**
19: **end if**
20: **end procedure**

21: **procedure** $recover()$
22: $recover_host()$
23: $restart()$
24: **end procedure**

rejuvenation checks, in contrast, are triggered once every multiple of the communication round and introduce only a constant computation overhead to determine whether the node falls within the group rejuvenation window.

After rejuvenation or power-on, nodes must integrate into the cluster and synchronize their view of the progress within the cluster rejuvenation cycle $crcycle$. For that, we extend the communication controller's $CC\text{-}state$ by $\log_2(G_\varsigma) + \log_2(R_{PRS})$ bits. The first term encodes the current proactive rejuvenation slot, while the second encodes the current TDMA round within that slot. A one-byte extension, using 2 bits for the rejuvenation slot index and 6 bits for the TDMA round index, allows up to 4 rejuvenation slots, each with a maximum length of 64 TDMA rounds.

As with other $CC\text{-}state$ information, the proactive rejuvenation information shall be included in the computation of CRC codes and explicitly carried in integration frames, i.e., $i\text{-}frames$ and $x\text{-}frames$.

5.4.3 Rejuvenation Enforcement

Guardians determine which node is currently undergoing rejuvenation based on the progress of global time and the current $CC\text{-}state$. By executing a procedure analogous to that in Figure 8, guardians can therefore prevent a node from transmitting on the communication channels during its rejuvenation slot.

While blocking write access is necessary, relying solely on a node to rejuvenate itself is insufficient to guarantee that rejuvenation has actually been performed. Enforcing node rejuvenation therefore requires an external mechanism, which can be implemented at guardians. We distinguish two approaches depending on the chosen rejuvenation granularity for the node (see Section 2.2.8.2):

- **Simple hardware reset line.** When rejuvenation is performed at the node level, the guardian asserts a hardware reset line to the controlled node, forcing a complete cold boot. This approach is straightforward to implement, requires no additional software support, and provides strong guarantees because the guardian directly controls the reset line.

- **Rejuvenation proof.** When rejuvenation is performed at the OS level, the guardian must verify that the node has executed the secure-startup sequence correctly. This requires the node to transmit an authentic rejuvenation certificate (e.g., an authentication tag) generated by the secure startup service. The guardian must validate this certificate before allowing the node to rejoin the cluster. This approach requires additional software support and time to verify tag authenticity.

At the beginning of a node's proactive or reactive rejuvenation slot (whichever occurs first), the guardian sets a timeout of length T_{RS}. During this interval, the guardian blocks

the node's outgoing messages and verifies, if rejuvenation proof is required, the received authentication tag to confirm that rejuvenation has completed successfully. We do not assume that secure startup always succeeds. In contrast to the closed secure-startup model of [51], we adopt an *open* secure-startup model in which guardians externally enforce correct behavior. This approach increases fault independence because guardians reside in a different FCR than the node and can therefore contain its faults.

If a node fails to present a valid rejuvenation certificate before the timeout expires, the guardian blocks its access to the communication channels and triggers reactive rejuvenation. A node that repeatedly fails to complete rejuvenation within its slot is thus perceived as permanently faulty.

Hence, in order to support OS-level node rejuvenation enforcement, guardians must be extended to process and verify rejuvenation certificates. We distinguish three possible locations within guardians for executing the rejuvenation enforcement procedure:

1. **Communication-controller-based enforcement.** The guardian's communication controller performs certificate verification. This option is suitable when guardians are deployed without host computers, as in the ultimate implementation.

2. **Host-computer-based enforcement.** A host computer attached to the guardian verifies certificates and may perform more sophisticated analysis.

3. **Dedicated hardware enforcement.** The guardian is extended with a hardware accelerator that performs certificate verification in parallel and routes messages accordingly during rejuvenation slots. This option provides the highest performance and is more suitable for central guardian solutions since they control multiple nodes.

All three options require the guardian to maintain memory for storing rejuvenation certificates of the nodes it protects. They also require real-time scheduling of certificate messages and of the rejuvenation-enforcement procedure itself. A simple scheduling strategy is to transmit the rejuvenation certificate during the node's first transmission slot after rejuvenation and integration. This requires extending the rejuvenation slot by one additional TDMA round (certificate verification must complete by then).

Guardian failures due to crashes or blocking nodes do not compromise the correctness of the rejuvenation algorithm. However, if a guardian colludes with faulty nodes and forwards their messages during rejuvenation slots, other nodes may accept these messages, violating correctness. The TTA fault hypothesis tolerates the failure of either a node or its guardian, but not both simultaneously (see Section 3.1.5). We adopt this assumption, while noting that masking colluding guardian behavior would require triplicated guardians with majority-voting-based reception.

5.5 Network-Agnostic Guardians

The rejuvenation service introduced in Section 5.4 bounds the vulnerability window of individual nodes by periodically restoring them to a safe and secure state faster than adversaries can compromise a sufficient number of nodes to induce failures in the system. However, this service fundamentally relies on guardians to enforce rejuvenation and to guarantee collision-free TT network access. While the local guardian design of the TTA prototype can, in principle, be made *network-agnostic*, that is, independent of the network topology and communication protocol, it cannot contain the faulty behavior of its controlled node if that node behaves maliciously and deliberately desynchronizes it from global time (see Section 3.1.5).

In this section, we examine network-agnostic guardians in detail and analyze their susceptibility to timing attacks launched by the very nodes they monitor. We further derive theoretical results on synchronizing network-agnostic guardians to global time under such conditions. Based on these insights, we introduce a distributed architecture of cooperating network-agnostic guardians, which we call *SyncGuards*. SyncGuards form a distributed wormhole solution that operates independently of the underlying communication protocol (see Section 2.2.7). They not only ensure collision-free TT network access but also coordinate and enforce the rejuvenation of their controlled nodes.

To the best of our knowledge, SyncGuards constitute the *first* solution that simultaneously provides (i) protocol-agnostic enforcement of collision-free TT communication, (ii) adversary-resilient synchronization of guardians to global time, and (iii) distributed enforcement of rejuvenation for compromised nodes.

5.5.1 Rationale

The failure to prevent out-of-slot communication in safety-critical systems can have severe consequences, including extended fail-safe shutdowns intended to avoid even more catastrophic outcomes caused by missed distributed deadlines. Network guardians have long proven valuable in such systems because they convert the behavior of timing-faulty nodes into fail-silent behavior from the perspective of the rest of the system. In particular, they protect healthy nodes by blocking any transmission that occurs outside a node's allocated TDMA slots.

However, contemporary guardian designs are inherently *network-aware*: they are built for a specific network interface, communication protocol, and topology (see Section 5.1.4). They participate in sending and receiving messages, most importantly to resynchronize their local clocks with the maintained global time base, and therefore include the full functionality of a networked component, including protocol stacks, drivers, etc. Such a design approach increases system complexity, particularly in systems that

employ multiple TT networks, since the guardian functionality must be re-implemented and re-certified whenever the network or topology changes.

This motivates the need for *network-agnostic* guardians that avoid protocol- and topology-specific dependencies while still enforcing strict TT behavior in modern time-domain–attack-resilient safety-critical systems. Instead of replicating the entire communication infrastructure already present in the node, network-agnostic guardians leverage this infrastructure to synchronize with the global time base, thereby significantly reducing system complexity and certification effort.

Accordingly, the central question becomes: *How can network-agnostic guardians enforce compliance with the TT schedule and prevent out-of-slot activities despite the presence of malicious nodes?*

5.5.2 System Model

We consider real-time distributed systems, as shown in Figure 5.5.1, which are comprised of N nodes connected via a TT network consisting of C independent communication channels. Although we do not constrain the assumed network to a specific TT protocol and topology, we assume that access to the network follows a TDMA strategy based on an offline-defined schedule and a global notion of time.

We assume that this global notion of time is established on top of approximately synchronized local clocks, whether synchronized internally or externally, with a precision of π_{nn}. Furthermore, we assume that clock drift remains bounded by a known value ρ, and that nodes periodically adjust their perception of global time by recalibrating the macrotick duration (*MT*) every resynchronization period.

The role of a node n_i (where $i \in \{0, \ldots, N-1\}$) is to participate in the execution of the distributed real-time application. Access to the network and rejuvenation of node n_i is coordinated through a novel guardian component g_i, which we call SyncGuard. The guardian SyncGuard is *network-agnostic*; in contrast to a network-aware guardian, it has no understanding of the utilized network protocol and topology. Consequently, guardian g_i interfaces with n_i through a memory-mapped I/O interface and must involve this node for the purpose of exchanging messages, in particular those for resynchronizing its own clock to global time.

We assume that SyncGuards perform a rudimentary acceptance window check on messages relayed from their node, considering a message valid only if it arrives within a time window centered at the expected arrival time from the SyncGuard's perspective. The acceptance window has a minimum duration of $2 \times \pi_{ng}$, accounting for potential clock drift in both directions (see Section 3.1.4.3). Here, π_{ng} denotes the synchronization precision between the node and its SyncGuard.

Additionally, we assume that each g_i maintains its own copy of n_i's proactive rejuve-

nation schedule and communication schedule, which g_i uses to track the system state in relation to global time and force rejuvenate n_i or enable/disable n_i's send interface to a given communication channel through the *transmission controller* (TC) to enforce n_i's compliance to its allocated TDMA slots.

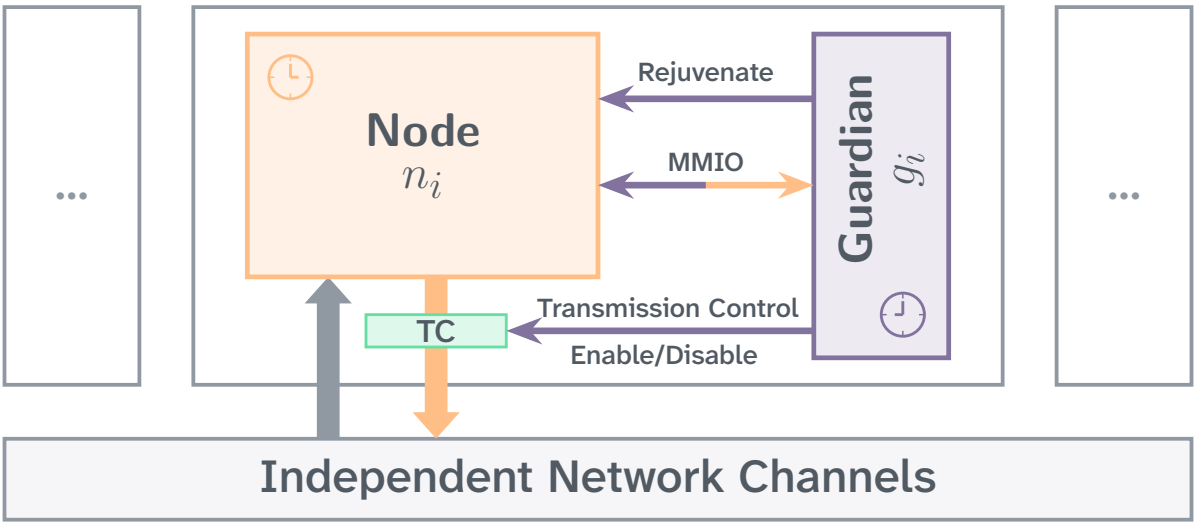


Figure 5.5.1: Distributed system of N nodes communicating over a TT network, whose access and rejuvenation are coordinated through a per-node *network-agnostic* guardian that can only resynchronize through messages received via their node.

5.5.3 Node-SyncGuard Clock Precision

Now, we turn to deriving a safe lower bound for the synchronization precision between a node and its SyncGuard (π_{ng}), under the assumptions stated in Section 5.5.2. For simplicity, we further assume that the node synchronizes the SyncGuard's clock through a direct synchronization message or signal that indicates the start of the next TDMA communication round.

We adopt a similar approach of formalizing the notion of time and clocks as in [274]. We distinguish the following two time notions: *real-time* $\mathbb{R}$ as an abstract global time quantity, and *local-time* $\mathbb{C}$ as node or guardian local time. A clock is formally a function $C(t) : \mathbb{R} \rightarrow \mathbb{C}$, such that $C_{g_x}(t)$ returns the local clock value of g_x at real-time instant t.

There are two primary sources that contribute to the clock deviation between a node and its SyncGuard:

1. **Clock drift**: Refers to the gradual divergence between the node's and SyncGuard's clocks due to the differences in oscillator frequencies. Let c_{drift} represent the clock deviation between SyncGuard and its node due to clock drift within a single

resynchronization round R_{int}. Since drift may occur in both directions, the maximum value of c_{drift} is $2\rho R_{int}$. This scenario occurs when the node experiences the maximum negative drift rate and the guardian the maximum positive drift rate. Conversely, the minimum value of c_{drift} is $-2\rho R_{int}$, representing the scenario in which the node has the maximum positive drift rate and the guardian the maximum negative drift rate.

2. **Node clock correction**: The synchronization of the node's clock to global time affects the difference between its clock and that of its SyncGuard. Let c_{ncorr} denote the clock correction term that the node applies to its clock during a resynchronization round R_{int}. This value is necessarily upper bounded by the assumed precision between nodes π_{nn} to ensure clock convergence [275]. c_{ncorr} reaches the boundary values in scenarios where the node experiences the maximum possible clock drift in the opposite direction relative to all other correct node clocks, and consequently, relative to global time.

Lemma 6. *For any clock synchronization round, the correction value c_{gcorr} applied to SyncGuard's clock is determined by:*

$$c_{gcorr} = c_{ncorr} - c_{drift} \tag{5.5.1}$$

Proof. After applying the correction term c_{ncorr}, the start of the next TDMA round according to the node's clock $C_n(rs)$ becomes $C_n(rs') = C_n(rs) + c_{ncorr}$. From the perspective of SyncGuard, the start of the TDMA round is then $C_g(rs) = C_n(rs) + c_{drift}$

The correction value c_{gcorr} is computed as the difference between the corrected time received from the node and the expected start of the next TDMA round at SyncGuard:

$$\begin{aligned} c_{gcorr} &= C_n(rs') - C_g(rs) \\ &= (C_n(rs) + c_{ncorr}) - (C_n(rs) + c_{drift}) \\ &= c_{ncorr} - c_{drift} \end{aligned}$$

□

From Lemma 6, a safe clock precision for the externally synchronized network-agnostic guardians must be equal to or greater than the maximum possible value of c_{gcorr}, which is bounded from above by π_{nn} as demonstrated by Theorem 5.

Theorem 5. *For any resynchronization round R_{int}, the correction value c_{gcorr} applied to SyncGuard's clock satisfies:*

$$|c_{gcorr}| \leq \pi_{nn} \tag{5.5.2}$$

Proof. Considering Equation 5.5.1, the maximum value of c_{gcorr} occurs when c_{ncorr} reaches its maximum positive value and c_{drift} reaches its maximum negative value.

The maximum c_{ncorr} occurs when the node's clock has the maximum negative drift rate relative to all other nodes (representing global time), which simultaneously exhibit the maximum positive drift rate. In addition, messages between nodes are subjected to the worst-case jitter and faults, leading to a maximum acceptable node clock correction value $c_{ncorr} = \pi_{nn}$ [275]. In contrast, the minimum c_{drift} occurs when the node's clock exhibit the maximum positive drift rate relative to SyncGuard's clock, while SyncGuard's clock simultaneously experiences the maximum negative drift rate, yielding: $c_{drift} = -2\rho R_{int}$ Since the node's clock cannot possibly have the maximum positive and negative drift rates simultaneously, i.e., these scenarios are mutually exclusive, we have $|c_{gcorr}| < 2\rho R_{int} + \pi_{nn}$.

In Table 5.5.1, we present the extreme cases of clock drift, illustrating opposing deviations among the node, SyncGuard, and the global time (collectively representing all other nodes in the system). For each case, the corresponding values of c_{drift}, c_{ncorr}, and c_{gcorr} are derived. In *Case C*, where the node and global time exhibit drift in the same direction, the value of c_{ncorr} is determined solely by additional sources of uncertainty in the network such as communication jitter and faults, whose effects are bounded by the interval $|\pi_{nn} - 2\rho R_{int}|$ based on the design assumptions of clock synchronization.

Table 5.5.1: Extreme cases of clock drift among the node, SyncGuard, and global time, and corresponding values for c_{drift}, c_{ncorr} and c_{gcorr}.

Cases	Clock	Clock Drift	c_{drift}	c_{ncorr}	c_{gcorr}
A	Node	$\mp\rho$	0	$\pm\pi_{nn}$	$\pm\pi_{nn}$
	SyncGuard	$\mp\rho$			
	Global time	$\pm\rho$			
B	Node	$\mp\rho$	$\pm 2\rho \cdot R_{int}$	$\pm\pi_{nn}$	$\pm\pi_{nn} \mp 2\rho \cdot R_{int}$
	SyncGuard	$\pm\rho$			
	Global time	$\pm\rho$			
C	Node	$\pm\rho$	$\mp 2\rho \cdot R_{int}$	$\pm\pi_{nn} \mp 2\rho \cdot R_{int}$	$\pm\pi_{nn}$
	SyncGuard	$\mp\rho$			
	Global time	$\pm\rho$			

Since all the extreme drift cases satisfy $-\pi_{nn} \leq c_{gcorr} \leq \pi_{nn}$, Equation (5.5.2) holds. □

Corollary 1. *The synchronization precision between a node and its SyncGuard satisfies:*

$$\pi_{ng} \geq \pi_{nn} \tag{5.5.3}$$

The proof follows directly from Theorem 5, as the maximum clock correction value applied by SyncGuards establishes this precision bound. Note that $\pi_{ng} = \pi_{nn}$ represents the best-case scenario for network-agnostic guardians. Depending on the physical deployment, however, additional factors such message relay delays and processing jitter

can result in $\pi_{ng} > \pi_{nn}$. Consequently, the reasonableness condition for global time must be formulated using this larger precision, i.e., $|MT| > \pi_{ng}$. This granularity must then be used in the design rules of transmission window alignment and in the insertion of inter-slot gaps in the TDMA schedule (see Section 3.1.4.2). In the remainder of this chapter, we drop the subscript and use π to denote the assumed synchronization precision for the entire cluster.

5.5.4 Timing Attacks Against Network-Agnostic Guardians

In this section, we discuss potential attacks that hamper the synchrony of network agnostic guardians. We assume that a node n_x and its *network-agnostic* guardian g_x are initially synchronized with other nodes and guardians in the system and are in agreement with other nodes and guardians on global time. Two guardians g_x and g_w are considered synchronized if their clocks at any real-time instant t are no more than the precision π apart, i.e.,

$$\forall t \in \mathbb{R},\ |C_{g_x}(t) - C_{g_w}(t)| \leq \pi \tag{5.5.4}$$

We further assume that an adversary successfully compromised n_x, and now attempts to prevent g_x from re-aligning its clock with global time by mounting timing attacks on it. We consider the resynchronization problem under the assumption that resynchronization must not interfere with any message healthy replicas send over the network. Such interference may be fatal in highly-critical scenarios where messages must be transmitted before their deadline to avoid system failure, in particular if resend-after-collision is not possible.

5.5.4.1 Omission Attacks

Since clocks drift away from each other, resynchronizing to a remote guardian g_w requires g_x to capture information about the state of g_w's clock by receiving a synchronization message from it. However, g_x is network-agnostic, i.e., it must rely on n_x to relay g_w's message to it. The malicious n_x can, nevertheless, intentionally drop any such messages to g_x, which can lead to the violation of the relation in Equation 5.5.4. Hence, without realigning g_x's clock to global time, g_x cannot rule out collisions if it were to allow n_x to transmit. Under such scenario, g_x can reactively rejuvenate n_x to regain an opportunity to resynchronize. Note that g_x can not discern whether the sender n_w did not send synchronization messages, whether n_w crashed or whether n_x omitted relaying the message to g_x. In the presence of omission faults, the situation where n_w crashed or did not send the message can be compensated by resynchronizing to global time using any one of $F+1$ synchronization messages, where F is a parameter and an upper bound on the number of such omission faults that the system can tolerate.

5.5.4.2 Replay Attacks

Replay may cause the repeated or deferred delivery of the most recent synchronization message, the synchronization messages from previous synchronization rounds or any combination of the two. Thus, malicious n_x can present g_x with an outdated but otherwise correct view of global time by replaying synchronization messages. If g_x adopts this view, it may allow n_x to send outside the slots allocated to it.

To mount a replay attack, n_x injects an arbitrary delay $T_{INJECTED}$ alongside the relay delay T_{RELAY} before delivering g_w's synchronization message to g_x as follows:

$$C_{g_x}(t^w_{rcv}) = C_{n_x}(t^w_{rcv}) + T_{RELAY} + T_{INJECTED}$$

In other words, n_x can influence the value of $C_{g_x}(t^w_{rcv})$, which denotes the reception time of g_w's message according to g_x's clock and which g_x may use to correct its clock (see Section 3.1.4.3).

Since the initially synchronized g_x can detect the omission of synchronization messages as well as the replay of an old sequence of authentic messages (by leveraging cryptographic means and knowledge of communication schedule and global time), it can indefinitely block n_x from transmitting or trigger its rejuvenation. Therefore, instead of exhausting nodes by causing them to fail silently, we consider a replay attack with the main goal of allowing n_x to transmit partially outside its allocated slot, which leads to collisions and/or asymmetric reception.

Node n_x systematically delays the delivery of synchronization messages to g_x to undetectably skew g_x's local view away from global time. It exploits uncertainties that enlarge the synchronization precision, such as the variance between best-case and worst-case communication latencies and faults, while still ensuring that each delayed message remains within g_x's acceptance window. We refer to this timing attack as an *acceptance window attack*.

Suppose that g_x is scheduled to receive a synchronization message from a remote guardian g_w at global time t^w_{at}, and that g_x uses the reception time of this message to adjust its clock toward g_w's. As per the TT approach (see Section 3.1.4.3), g_x will accept g_w's message only if its actual reception time $C_{g_x}(t^w_{rcv})$ satisfies the following acceptance window condition:

$$C_{g_x}(t^w_{dat}) - \pi \leq C_{g_x}(t^w_{rcv}) \leq C_{g_x}(t^w_{dat}) + \pi \tag{5.5.5}$$

Now, consider the following two scenarios:

- ***Scenario 1***: n_x delivers all synchronization messages to its guardian g_x in a timely manner such that they satisfy Equation 5.5.5 and hence, g_x's clock is corrected to

tick at a rate that keeps it in agreement with the global notion of time.

- ***Scenario 2***: n_x starts a virtual time base V, initialized with its own clock, and delays the delivery of g_w's synchronization message (and all other synchronization messages) to g_x by $T_{Injected} = C_{n_x}(t^w_{rcv}) - C_{n_x}(t^w_{dat})$. Since the clocks of n_x and g_x still satisfy Equation 5.5.4 at this point, all delayed synchronization messages continue to satisfy the acceptance window condition in Equation 5.5.5. Meanwhile, n_x computes the true correction terms from the received synchronization messages to determine the direction in which global time is drifting (i.e., whether it advances faster or slower).

  For every subsequent resynchronization round, n_x updates both its virtual clock V and the clock of g_x using the maximum covert injection delay κ, applied with opposite polarity to the direction of global-time drift. That is, n_x replays synchronization messages to g_x with an added delay of $C_V(t_{rcv}) - C_V(t_{dat}) \pm \kappa$, where the first two terms are derived from the virtual clock V, and where the maximum covert injection delay within a resynchronization round R_{int} is given by $\kappa = \pi_{nn} - 2\rho R_{int}$, following Theorem 5. Over multiple resynchronization rounds, the injected delays accumulate as the virtual time base diverges from global time, eventually violating Equation 5.5.4 and causing g_x to lose synchrony with the rest of the system.

Since g_x cannot distinguish between the above scenarios, the safe resynchronization of g_x cannot be guaranteed without risking network interference. The situation becomes even more problematic if g_x loses synchrony or attempts to join an existing network cluster: in this setting, g_x can neither rely on an acceptance window check nor prevent n_x from injecting arbitrary delays before the defined sequence of messages in the schedule. As a result, the synchronization of network-agnostic guardians becomes impossible under such adversarial conditions.

The work presented in [276] extends our acceptance window attack to target a specific TDMA slot in the schedule, enabling a compromised node to corrupt messages transmitted within that slot. The authors evaluate both the maximum covert injection delay and the targeted slot attack using a FlexRay fieldbus model implemented within the OMNeT++ network simulation framework [277]. Their results show that the attack remains fully undetected while reliably disrupting all message transmissions within the targeted slot.

From the discussed attacks, it is clear that a node has complete control over the delivery time of synchronization messages sent by other nodes to its network-agnostic guardian. It is therefore more efficient for the node itself to synchronize its guardian's clock to global time, eliminating the overhead of storing, for every synchronization message of every remote node on every channel, all parameters required to compute the acceptance window at each guardian.

Furthermore, the initial clock offset synchronization of network-agnostic guardians requires that the controlled node be guaranteed to behave correctly. Node rejuvenation combined with secure startup can establish such an initial interval of trust during which malicious activity on the controlled node is absent. Additionally, because network-agnostic guardians cannot proactively deter acceptance-window attacks, they may establish a virtual control network and exchange messages among themselves (via their nodes) to assess the timeliness of received synchronization messages with respect to their acceptance windows, and thereafter reactively rejuvenate nodes whose faulty behavior is detected. In this setting, network-agnostic guardians form a distributed wormhole solution, consistent with the architectural concepts described in Section 2.2.7.

5.5.4.3 Asymmetric Reception Attacks

A new attack emerges when network-agnostic guardians exchange messages that rely on local inputs provided by their controlled nodes. Since meeting an acceptance window is a purely local property, correct nodes and guardians cannot distinguish between two cases: a faulty transmitter that intentionally causes asymmetric reception (after first desynchronizing its guardian via acceptance window attacks), or a faulty receiver that falsely reports whether a message met its acceptance window.

Consequently, although network-agnostic guardians may still enforce non-interference between TDMA slots, they cannot alone ensure that messages are received within the acceptance window of all correct receivers. However, because an acceptance window attack requires injecting delays across multiple rounds, reducing the period of proactive node rejuvenation limits the time available to an adversary on the transmitting node to cause asymmetric message receptions. A detailed study of the interplay between the rejuvenation period and this attack is left for future work.

5.5.5 Fault Model

We adopt the fault model introduced by Sousa et al. [174] in assuming the presence of strong adversaries (and accidental faults). More precisely, we assume nodes can become repeatedly faulty due to accidental or intentionally malicious reasons. However, during any sliding window of length T_{Rejuv} at most F new faults occur. T_{Rejuv} defines the periodicity of proactively rejuvenating replicated nodes in the system. As described in Section 2.2.8.2, the choice of T_{Rejuv} is inferred from the maximum time needed to rejuvenate replica groups as well as the minimum time to compromise $F + 1$ nodes.

We further assume that the minimal time adversaries need to compromise an individual replicated node $T_{CoprInd}$ (e.g., the time to again mount an attack) is not arbitrarily small. This can be achieved by using techniques such as diversification [185, 278] and

obfuscation [177,178,279]. The former strengthens independence between replicas and the latter helps with rendering any gained knowledge from successful previous infiltrations futile. We further assume that the deployed obfuscation techniques suffice for protecting replicas diversity throughout the mission time.

We assume critical functionality to be replicated and conclude, following the argument in [174], that replicated nodes, continue to be able to mask faults, provided that all replicas are recurrently rejuvenated faster than the time to compromise more than $F + 1$ replicas, and that rejuvenated replica nodes can safely re-join the system, which requires resynchronizing their guardians.

Whereas nodes can fail in an arbitrary and possibly intentionally malicious and Byzantine manner, we assume SyncGuards and TCs to only fail by silencing the node in the process. The simplicity of SyncGuard defines its trustworthiness and as such the coverage of the assumption that it will not fail arbitrarily, which turns SyncGuards into trusted-trustworthy components (see Section 2.2.7). We consider the failure of SyncGuards and TCs as well as permanent node damage to be rare events. We make no further timeliness or availability claims once the system's (spare) resources are exhausted due to such failures.

We make the usual assumptions regarding the strength of the applied cryptographic algorithms and the protection of all involved credentials. In particular, adversaries cannot obtain keys actively used by trusted components (including the guardians), nor can they break the employed ciphers without such keys. Finally, we assume that attackers can influence the forwarding of messages to guardians only by compromising nodes.

5.5.6 SyncGuard Design

In this section, we describe the design of our network-agnostic guardian (SyncGuard) and how it can be safely resynchronized and integrated with nodes in the system.

A SyncGuard contains, as shown in Figure 5.5.2, the following components:

1. **State machine**: coordinates and enforces node rejuvenation and manages network access control.

2. **Memory and interfaces (NGI and CSI)**: store SyncGuard configurations as well as messages received from peer SyncGuards and from the controlled node.

3. **Oscillator and timer**: provide SyncGuard with its own notion of time progression and drive the state machine.

4. **Cryptographic and arithmetic unit**: analyzes and ensures the authenticity and integrity of exchanged SyncGuard messages.

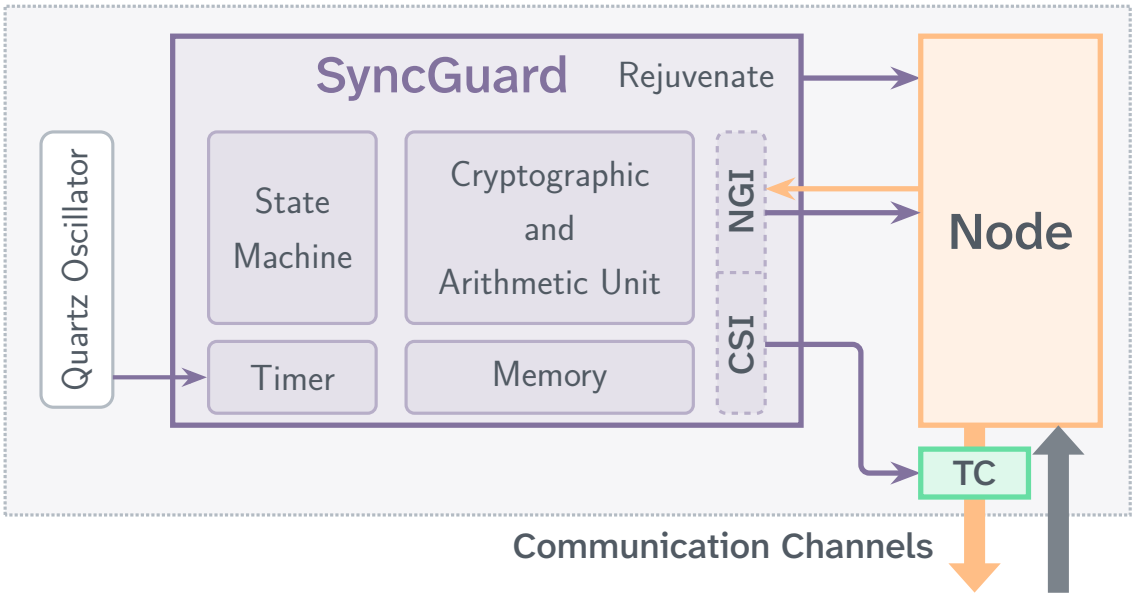


Figure 5.5.2: SyncGuard structure

5.5.6.1 SyncGuard Cycle and State Machine

As discussed in Section 2.1.5, clock synchronization is essential to maintain a consistent view about global time. Therefore, nodes must periodically synchronize the clocks of their SyncGuards for the latter to remain within the assumed clock precision (π). Additionally, SyncGuards have to remain in synchrony w.r.t. system state development in terms of network communication and node rejuvenation progress, which necessitates the exchange of messages.

As illustrated in Figure 5.5.3, SyncGuards operate in a cycle of four-phases:

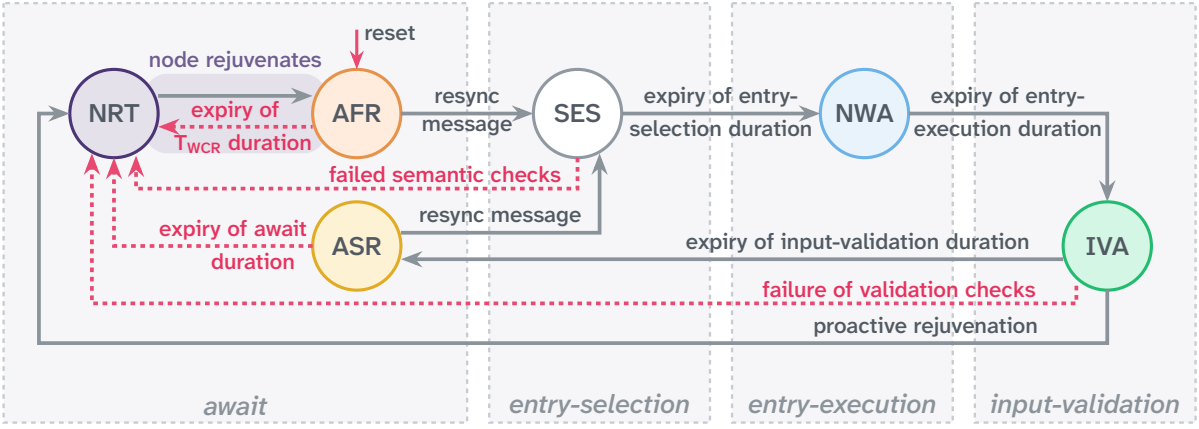


Figure 5.5.3: SyncGuard's abstract state machine. Solid black lines indicate normal progression through SyncGuard cycle phases. Dashed red lines denote breaking out of the SyncGuard cycle.

1. The *await* phase is entered by a SyncGuard g_x upon reset, node rejuvenation or after completing a SyncGuard cycle. Upon rejuvenation or reset, g_x is put into the *Await-First Resynchronization* (AFR) state, where it sets a timeout that equals the worst-case node rejuvenation time T_{WCR}, and waits for the first synchronization message from n_x. Failing to present g_x with the synchronization message before the timer expiry is considered a resynchronization error and results in g_x reactively rejuvenating n_x by transitioning its state machine into the *Node Rejuvenation Trigger* (NRT) state. An endless succession of such failures is thus perceived by other nodes as a permanent node failure.

   On the other hand, when g_x enters the *await* phase after the completion of the previous cycle, its state machine transitions into the *Await Subsequent Resynchronization* (ASR) state, where it waits for a scheduled synchronization message from n_x. If n_x fails to present the synchronization message, or relays it outside its acceptance window, n_x is reactively rejuvenated (see Section 3.1.4.3).

2. The *entry-selection* phase is entered from the *AFR* or *ASR* states upon the valid reception of n_x's synchronization message, triggering the start of a new SyncGuard cycle and transitioning g_x's state machine into the *Schedule Entry Selection* (SES) state, During this phase, g_x uses the relayed synchronization message from its node to select a valid entry in its schedule (i.e., updates its view on system state). If the delivered message content matches g_x's knowledge of the current system state (e.g., schedule entry is immediate successor of previous), g_x creates a message holding its system view and authenticates it using the cryptographic unit, which n_x then relays to other nodes. Otherwise, g_x reactively rejuvenates n_x by transitioning to the *NRT* state.

3. The *entry-execution* phase is entered after the expiry of the *entry-selection* phase duration, where g_x transitions into the *Network Access* (NWA) state. During this phase, the guardian g_x grants its node n_x access to the network channels, by enabling transmission control for the duration of n_x's allotted TDMA slots on the corresponding network channels. This is the only phase where g_x grants its node write access to the network channels.

4. The *input-validation* phase is entered upon the timeout of n_x's TDMA slot end from the perspective of g_x. This moves g_x's state machine into the *Input-Validation* (IVA) state. Under our fault model (Section 5.5.5), nodes may behave maliciously only after $T_{ComprInd}$ has elapsed since the completion of their rejuvenation. This implies that, before $T_{ComprInd}$ is reached, a SyncGuard can trust the temporal information relayed by its node, provided that the messages pass the rudimentary semantic checks performed during the *entry-selection* phase. Note that $T_{ComprInd}$ cannot

be set to zero, since g_x would not be able to prevent malicious n_x from causing collisions (see Section 5.5.4.2). Practically, a re-initialized SyncGuard needs to trust its node for at least its first SyncGuard cycle.

During this short interval of trust, the behavior of g_x in the *IVA* state is restricted to passively receiving system view messages from remote SyncGuards, for the duration of the *input-validation* phase. Once $T_{ComprInd}$ elapses, g_x also executes a deterministic input-validation algorithm (see Section 5.5.6.4) in the *IVA* state. The algorithm counteracts timing attacks by using the received and authenticated system view messages from remote SyncGuards to evaluate the correctness of the relayed information by its node. If any check fails, SyncGuard reactively rejuvenates its node.

At the end of the *input-validation* phase, g_x transitions to the *NRT* state if proactive rejuvenation of n_x is scheduled (i.e., the start of n_x's proactive rejuvenation slot). Otherwise, it transitions to the *ASR* state. The duration of this phase depends on the maximum number of messages that must be received in a SyncGuard cycle, as well as the capabilities of the cryptographic unit.

The length of the SyncGuard cycle $T_{Cycle}^{SyncGuard}$, and thereby the frequency of exchanging SyncGuard messages, can be determined based on the gap intervals between TDMA slots. When slots in the TDMA schedule are separated by a minimum gap of 4π, following the guardian design rules for node transmission windows [191], the local clocks of SyncGuards and nodes must be synchronized at the same rate. It is also possible to stretch the SyncGuard cycle, thereby further worsening the clock precision of SyncGuards relative to that of the nodes, in order to reduce the frequency of SyncGuard resynchronization and message exchange. However, as indicated in Section 5.5.3, the system designer must choose the gaps between TDMA slots based on the worse clock precision (i.e., that of the SyncGuards) to rule out message collisions.

Reducing the frequency of SyncGuard message exchange also affects the freshness of each SyncGuard's knowledge about the operational state of its peers. Therefore, system designers must weigh these trade-offs carefully, while ensuring that $T_{Cycle}^{SyncGuard}$ remains sufficient for the reception and authentication of all scheduled messages within the SyncGuard cycle.

Finally, apart from the bandwidth required to transmit and relay feedback messages, as well as the additional gaps inserted between TDMA slots, the SyncGuard cycle has no further influence on the effective network bandwidth, since the cycle phases do not overlap across different SyncGuards.

5.5.6.2 SyncGuard Messages

SyncGuards implement two types of messages:

1. **Feedback messages**: represent the sending SyncGuard's view of the system state and contain the following fields:

ID	*Position*			**Membership**	*HMAC*
	Current mode	*Requested mode*	*Cycle ID*		

   - ***ID***: uniquely identifies the sending SyncGuard among its communicating peers.
   - ***Position***: encodes the position within the TDMA schedule via *current mode* (index of the active TDMA schedule), *requested mode* (index of the schedule to switch to after the current one completes), and *cycle ID* (a counter, large enough to prevent replay attacks, denoting both the SyncGuard's cycle modulo the TDMA schedule length and the current proactive rejuvenation slot modulo the rejuvenation period).
   - ***Membership***: a bit vector indicating the operational status of each node.
   - ***HMAC***: contains the keyed hash of the message, computed using the symmetric key shared exclusively among SyncGuards.

2. **Synchronization messages**: relayed from a node to its SyncGuard. They contain only the *position* field of feedback messages. Additionally, their arrival time at the receiving SyncGuard is used to correct its local clock.

We allow SyncGuards to change the mode of operation, including the TDMA schedule. However, we restrict the time when such a change can happen to the last schedule entry (i.e, last SyncGuard's cycle) of a given operational mode to ensure that first the request is stable and cannot be changed anymore and that all SyncGuards receive the same request from their nodes and agree on it. Nodes that requests to change the mode of operation earlier are considered faulty in their attempt to create a mode synchronization error and are reactively rejuvenated by their guardians.

5.5.6.3 SyncGuard Interfaces

With respect to communication interfaces, a SyncGuard provides two interfaces:

1. **Node Guardian Interface (NGI)**: a dual-port memory with bidirectional data flow used to relay messages between SyncGuards through their nodes. The NGI is

divided into two sections: the first stores outgoing messages (i.e., SyncGuard's own feedback message), while the second stores incoming messages (i.e., the node's synchronization message as well as feedback messages from remote SyncGuards). All SyncGuards enforce strict read/write windows for accessing the NGI during each cycle. For example, the synchronization message from the node must be received during the *await* phase, and the node must read its SyncGuard's feedback message during the *entry-execution* phase. SyncGuard clears these messages at the end of each cycle. Note that SyncGuards receive relayed feedback messages for a given cycle from the beginning of the *input-validation* phase of the previous cycle until the beginning of the same phase in the current cycle.

2. **Channel State Interface (CSI)**: a minimal interface through which the SyncGuard controls access to communication channels. The *transmission controller* (TC) monitors changes in the CSI and configures the node's transmission path accordingly. For each communication channel, the CSI exposes a signal that, when asserted, causes the TC to grant the node write access to that channel.

In addition to the memory required for storing the dynamic data structures represented by the above messages, each SyncGuard must also include sufficient memory to store its configuration data. These configurations include the symmetric cryptographic key and the network-access and rejuvenation schedules.

5.5.6.4 The Input-Validation Algorithm

The purpose of the input-validation algorithm is to establish a consensual view of the system state across the communicating SyncGuards despite the presence of participating faulty nodes, which is necessary to ensure the correct delivery of their services. Algorithm 9 presents the pseudocode of the input-validation algorithm (implemented by the procedure `ValidateIn()`). It also lists the procedure `selectEntry()` (lines 1–5) to indicate the phase in which SyncGuards update their view and construct their feedback messages. SyncGuards store feedback messages according to node IDs. Therefore, duplicates or messages with invalid IDs are discarded. Furthermore, SyncGuards use their cryptographic unit to verify the authenticity and integrity of received feedback messages based on the appended HMACs. This is done by recomputing the HMAC from the message content using the secret SyncGuard key and comparing it with the appended value.

Each SyncGuard keeps a counter, denoted $nack$, which records the number of SyncGuards that disagree with its current system view. It also maintains the *expected* set, a configuration parameter that specifies, for each SyncGuard cycle, the IDs of the SyncGuards from which it is supposed to receive feedback messages.

Property 1. *The input-validation algorithm causes a SyncGuard g_x to reactively rejuvenate its node n_x by the end of the input-validation phase if $O + R + A > F$ holds, where:*

- *O is the number of omitted feedback messages.*
- *R is the number of replayed feedback messages.*
- *A is the number of out-of-slot receivers during n_x's last transmission.*

By the end of the *input-validation* phase, the `checkForOmissions()` procedure at g_x must have updated the value of the $nack$ counter with the difference between the received and expected messages in the *expected* set, which equals O (lines 18–22). Expected messages that are manipulated in the value domain fail authentication and are therefore dropped by g_x; consequently, they are treated as omitted and contribute to O. Afterwards, the `checkForReplays()` procedure compares the *position* field in the received authentic feedback messages with the corresponding field in n_x's synchronization message (line 26). Thus, g_x must have incremented the $nack$ counter R times for the R replayed authentic feedback messages provided by n_x.

Finally, if n_x manages to desynchronize g_x from A other SyncGuards, thereby transmitting in a way that causes an out-of-slot reception at A nodes in the preceding SyncGuard cycle, or, indistinguishably, if A remote nodes accuse n_x of missing their acceptance window (see Section 5.5.4.3), then g_x reacts by executing the `checkForTimelyAccess()` procedure. This procedure increments the $nack$ counter and resets the bit position in g_x's membership field for each of the A remote SyncGuards that disacknowledge n_x's transmission by equally resetting g_x's bit in their expected feedback message (lines 34–35). Note that this action is not performed if a remote SyncGuard sends a feedback message with an empty membership field (line 33), which occurs during the first network access of that SyncGuard's node after rejuvenation.

The algorithm then executes the `rejuvenateIfFaulty()` procedure, which causes the SyncGuard to reactively rejuvenate its node if the value of the $nack$ counter exceeds the tolerable threshold F (line 40). In other words, n_x proceeds unrejuvenated only if, by the end of the validation phase, the condition $nack = O + R + A > F$ does *not* hold.

Property 2. *The input-validation algorithm causes all communicating SyncGuards to reactively rejuvenate their nodes by the end of the validation phase if more than F faults occur in any SyncGuard cycle.*

A situation in which more than F faults occur implies that the condition $nack > F$ (line 40) evaluates to true at all SyncGuards. Following Property 1, this causes every SyncGuard to reactively rejuvenate its node by the end of the *input-validation* phase. This behavior is consistent with the fault assumptions stated in Section 5.5.5. Moreover,

when these assumptions are violated, the algorithm ensures that the system is driven back into a safe and stable state.

5.5.6.5 SyncGuard Rejuvenation Scheme

The precise mechanism used to coordinate node rejuvenation is not central to SyncGuard design itself. For this reason, we leave the concrete implementation of this mechanism open. System designers may adopt the rejuvenation mechanism presented in Section 5.4, or they may employ the simpler proactive rejuvenation scheme shown in Algorithm 10, which follows the same structural principles as the example illustrated in Figure 2.2.5.

All SyncGuards maintain a local copy of the parameters required for proactive rejuvenation, including the assigned proactive rejuvenation slot PRS, its duration T_{RS}, and the total number of proactive rejuvenation slots G within the rejuvenation period T_{Rejuv}. Rejuvenation slots are measured in multiples of the SyncGuard cycle. Each SyncGuard uses the variable *cycle_ID* to determine when its proactive rejuvenation turn occurs (lines 2–3). As discussed in Section 2.2.8.2, availability and fault masking are guaranteed as long as no more than F faults occur within any rejuvenation period.

5.5.7 SyncGuard Overhead

The simplicity of the input-validation algorithm in Algorithm 9 demonstrates that it can be turned into a hardware-near implementation, on a simple microcontroller, within an FPGA, or even as custom logic. As a result, message authentication, and in particular the cost of HMAC computation, becomes the dominant contributor to SyncGuard overhead. This is unavoidable in settings where network-agnostic guardians must communicate: without authentication, faulty nodes could intercept, modify, or even forge messages and impersonate other SyncGuards. In contrast, safety-critical systems with a security focus typically already employ message authentication.

Assuming message frames with 240 bytes reserved for application data (e.g., as in TTP and FlexRay), SyncGuard feedback messages consume approximately 13% of the available application data bandwidth when transmitting the allocation ID (5 bits), position information (42 bits), membership information (32 bits), and an HMAC (recommended size 160 bits [280]) in every TDMA slot. In contrast, TT protocols for Ethernet-based networks such as TTEthernet and TSN provide substantially larger payload capacities of up to 1500 bytes per frame. With the same allocation, the SyncGuard message overhead in these systems is reduced to roughly 2% of the available bandwidth for TT traffic.

Algorithm 9 SyncGuard's Input-validation algorithm

// Parameters and variables for a SyncGuard g_x ▷ set upon integration
$expected$ ▷ a set holding SyncGuard IDs, from which g_x expects feedback
$feedback$ ▷ a set of messages, carrying the local system views of SyncGuards
$myFeedback$ ▷ g_x's feedback message to other SyncGuards
$nack$ ▷ variable holding the number of disagreeing feedback messages
$syncMsg$ ▷ synchronization message from n_x in the current SyncGuard cycle

1: **procedure** $selectEntry()$ ▷ triggered at the beginning of entry-selection phase
2: $checkSemantics(syncMsg)$ ▷ e.g., whether schedule entry exists and succeeds the previous
3: $myFeedback.position = syncMsg$
4: $createFeedbackMsg(myFeedback)$ ▷ forms and places g_x's authenticated feedback message to be relayed by n_x
5: **end procedure**

6: **procedure** $validateIn()$ ▷ triggered at the beginning of input-validation phase
7: **if** (n_x `is potentially compromised, i.e.,` $globalTime \geq lastRejuvTime + T_{ComprInd}$) **then**
8: $checkForOmissions()$
9: $checkForReplays()$
10: $checkForTimelyAccess()$
11: $recoverIfFaulty()$
12: **end if**
13: $feedback.clear()$ ▷ removes all entries from the feedback set
14: $update(parameters)$ ▷ e.g., expected feedback messages and position from next entry in g_x's schedule
15: $checkProactiveRejuvenate()$ ▷ g_x proactively rejuvenates n_x according to Algorithm 10
16: **end procedure**

17: **procedure** $checkForOmissions()$ ▷ updates nack with the number of omitted feedback messages
18: **if** $feedback.size() < expected.size()$ **then**
19: $nack = expected.size() - feedback.size()$
20: **else**
21: $nack = 0$
22: **end if**
23: **end procedure**

24: **procedure** $checkForReplays()$ ▷ updates nack with the number of replayed feedback messages
25: **for** $(j = 0; j < feedback.size(); j++)$ **do**
26: **if** ($feedback[j].position \neq myFeedback.position$) **then**
27: $nack += 1$
28: **end if**
29: **end for**
30: **end procedure**

31: **procedure** $checkForTimelyAccess()$ ▷ updates nack with the number of out-of-slot receivers
32: **for** $(j = 0; j < feedback.size(); j++)$ **do**
33: **if** ($!feedback[j].membership.empty()$ *and* $feedback[j].membership[i] == 0$ *and* $feedback[j].position == myFeedback.position$) **then**
34: $nack += 1$
35: $myFeedback.membership[j] = 0$
36: **end if**
37: **end for**
38: **end procedure**

39: **procedure** $recoverIfFaulty()$
40: **if** $nack > F$ **then**
41: $reactiveRejuvenate()$
42: **end if**
43: **end procedure**

Algorithm 10 A simple proactive rejuvenation scheme for SyncGuards

```
// Parameters and variables for a SyncGuard g_x                ▷ set upon initialization and integration
PRS_i                                                          ▷ proactive rejuvenation slot to which g_x is assigned
T_RS                                                           ▷ length of rejvuenation slot in terms of SyncGuard cycles
G                                                              ▷ number of proactive rejuvenation slots in a rejuvenation period
cycle_ID                                                       ▷ counter variable holding current SyncGuard cycle number

1: procedure checkProactiveRejuvenate()
2:     if (cycle_ID mod T_RS == 0) then                        ▷ beginning of a PRS
3:         if ((cycle_ID / T_RS) mod G == PRS_i) then          ▷ PRS_current=PRS_i
4:             rejuvenateNode()                                ▷ proactively trigger rejuvenation of node n_x
5:         end if
6:     end if
7: end procedure
```

Software-based HMAC generation and validation can introduce non-negligible runtime costs, which may affect synchronization accuracy, especially when the execution time to generate HMACs varies significantly. Measurements in [57] show that, on a 1.4 GHz Raspberry Pi 3B+, HMAC generation and validation require 6310 and 6477 cycles, respectively (4.5 μs and 4.6 μs). Comparable magnitudes have been reported for software implementations on deeply embedded systems [281]. In contrast, optimized hardware implementations [282] achieve throughput rates of up to 875.22 Mbps, corresponding to only 3 cycles for 240-byte messages. The processing cost of such hardware-accelerated solutions is therefore negligible, although they require the use of FPGAs or custom hardware.

5.5.8 SyncGuardLite

The systems considered so far exhibit node rejuvenation times that are significantly longer than the SyncGuard cycle $T_{Cycle}^{SyncGuard}$, largely due to their architectural complexity. In order to broaden the applicability of SyncGuard to less complex and more resource-constrained platforms, we introduce *SyncGuardLite*, a streamlined guardian variant designed for systems whose nodes have comparatively short rejuvenation times.

5.5.8.1 Communication-Free Resynchronization

The basic idea of SyncGuardLite is to remove the need for message exchange and the input-validation algorithm, while still guaranteeing collision-free network access. Communication-free resynchronization provides savings on two fronts: it eliminates the network bandwidth previously consumed by feedback messages, and it further reduces guardian implementation complexity because no cryptographic units are required.

Since SyncGuardLites do not exchange messages, they trade off coordinated reactive

rejuvenation and mode-change support for a simpler design. The former can be mitigated through more frequent proactive rejuvenation, while the latter can be handled as in previous work by making the guardian mode-change insensitive (see Section 3.1.7.8).

Preventing collisions in the presence of timing attacks, particularly acceptance window attacks, requires widening the gaps between TDMA slots. These gaps must account not only for the clock precision π, but also for the number of SyncGuardLite cycles during which a node may remain compromised before being proactively rejuvenated in the next rejuvenation period T_{Rejuv}. As discussed in Section 5.5.4.2, a malicious node can desynchronize its network-agnostic guardian in each resynchronization round R_{int} by the maximum covert injection delay κ. Equation 5.5.6 gives the minimum gap between TDMA slots required to rule out collisions caused by timing attacks. This gap can be reduced either by shortening the proactive rejuvenation period T_{Rejuv} or by increasing $T_{ComprInd}$ through fortifying nodes against adversaries.

$$gap^{min}_{inter\text{-}slot} = 3 \times \pi + \max\left(\pi, \left\lceil \frac{T_{Rejuv} - T_{ComprInd}}{R_{int}} \right\rceil \times \kappa\right) \tag{5.5.6}$$

"Now this is **not the end**. It is **not even the beginning of the end**. But it is, perhaps, the **end of the beginning**."

(Winston Churchill)

Chapter 6

Conclusion and Future Directions

The *Time-Triggered Architecture* (TTA) has long stood as the cornerstone for designing dependable distributed safety-critical real-time systems. However, its rigid, closed-world design assumptions are becoming increasingly incompatible with the demands of modern cyber-physical environments. As explored throughout this work, the static nature of time-triggered schedules and the assumption of purely accidental faults create an architectural tension with the modern necessity for operational flexibility and resilience against malicious intrusions.

This thesis contributes to resolving this tension by systematically deconstructing the core dependencies of the TTA and demonstrating that dynamic task set management and intrusion-tolerant mechanisms (e.g., node rejuvenation and network-agnostic guardians) can be integrated into time-triggered systems. Ultimately, this work demonstrates that time-triggered systems can achieve the flexibility and security required by open environments while preserving the extra-functional properties of the TTA: determinism, temporal predictability, replica determinism, fault and error containment, timely online agreement, diagnosability, constructive testability, temporal composability, and the ability to meet complex temporal constraints.

6.1 Summary of Contributions

This thesis hypothesizes that the TTA can evolve to meet the requirements of modern safety-critical systems operating in open and adversarial environments. This evolution requires systematically identifying and relaxing implementation-specific constraints, introducing dynamic scheduling mechanisms, and fortifying dependability mechanisms against malicious intrusions. We validate this hypothesis through the following contributions:

Dependency Analysis of TTA Services and Properties

Introducing flexibility into the rigid TTA while preserving its core extra-functional properties requires a systematic analysis of how the architecture achieves these properties. In Chapter 3, we investigated and classified the dependencies between the dependability services and extra-functional properties of the TTA. We examined how mechanisms such as sparse global time, offline schedules, atomic multicast, and guardians directly influence the realization of properties such as determinism, fault containment, replica determinism, and composability.

We introduced a classification framework that organizes the dependencies between these services and properties into three distinct categories: fundamental, implementation-based, and application-dependent. This framework establishes a theoretical foundation for distinguishing which aspects of the architecture represent implementation-specific choices that can be safely adapted, relaxed, or substituted. For instance, while deterministic behavior fundamentally requires common knowledge across distributed nodes, we demonstrated that the reliance on rigid offline schedules to simulate such knowledge is largely an implementation-based dependency that can be effectively replaced with global task parameters or execution windows. This insight enabled us to integrate flexible scheduling into time-triggered operations without violating the strong guarantees provided by the TTA.

Advancing Flexibility in Time-Triggered Scheduling

Addressing the limitations imposed by the rigid periodic workload model and the fixed task-set assumption, Chapter 4 builds upon the insight that deterministic behavior can be retained by simulating common knowledge of task parameters instead of scheduling decisions. By leveraging this principle, we introduced the *Dynamic Task Set Scheduler* (DTSS), which effectively replaces the rigid offline schedule with a dynamic time-triggered scheduler. DTSS is specifically designed to accommodate dynamically changing task sets. It equips the system with the capability to admit, execute, skip, and remove event-triggered tasks at runtime. Such a high degree of runtime flexibility is essential for emerging paradigms like autonomous driving and real-time cloud computing, where workloads continuously change in response to unpredictable environmental stimuli or dynamic service requests. Furthermore, unlike classical schedulers that are restricted to uniform invocation gaps, DTSS can also work with nonuniform ones. This capability results in considerable savings in context-switching overhead and enables better reclamation of freed processor capacity. Additionally, DTSS enforces strict execution-time budgets during operation, effectively preventing overrun-induced interference. This ensures the fault containment and temporal isolation between tasks required for certifi-

cation, while unlocking the operational flexibility necessary for modern open environments.

Permitting dynamic modifications to the system task set necessitates runtime verification to ensure that newly admitted tasks do not violate the stringent timing constraints of other tasks in the system, particularly the critical ones. Addressing this requirement, we developed and integrated exact feasibility tests based on the *Processor Demand Analysis* (PDA) that enable the safe modification of task sets at runtime. We designed the *Remaining Processor Capacity Analysis* (RPCA) and integrated it into DTSS to support the safe integration of aperiodic tasks. We subsequently investigated the integration of periodic and sporadic tasks, deriving exact feasibility tests specifically tailored to avoid the pessimism inherent in previous approaches.

When integrating sporadic tasks into time-triggered systems, we examined a closely related state-of-the-art offline acceptance test, identifying and resolving specific sources of analytical pessimism within it. Building upon these insights, we derived a PDA-based exact feasibility test and subsequently optimized it for runtime usage. By pre-processing the periodic demand offline, our optimized test relocates the vast majority of the computational effort to the design phase. This design choice lowers the required runtime complexity of the test from necessarily exponential to pseudo-polynomial under bounded task set utilization assumptions. Furthermore, employing a quick convergence technique during online evaluation demonstrated additional improvements in runtime performance. Ultimately, our exact feasibility tests yield runtime performance improvements of two to three orders of magnitude over existing methods, bringing exact schedulability analysis into the realm of practical runtime deployment for safety-critical systems.

Enhancing Intrusion Resilience in TTA-based Systems

Focusing on the vulnerability of time-triggered systems to targeted resource-exhaustion attacks, Chapter 5 analyzes how stealthy adversaries can strategically bypass standard fault-tolerance mechanisms. In long-lived autonomous missions, adversaries actively exploit vulnerabilities to covertly compromise a critical threshold of replicas, thereby subverting the system's voting quorum without triggering any alarms.

Counteracting this persistent threat, we designed a proactive and reactive node rejuvenation service tailored specifically for TTA-based systems. This service periodically restores processing nodes to a secure and pristine state, effectively outpacing adversaries before they can establish a persistent foothold and violate established exhaustion-safety bounds. Avoiding the need for extensive architectural modifications, our replica-aware rejuvenation service leverages the existing dependability services of the TTA for recovery coordination. Specifically, it utilizes global time abstractions to

schedule proactive recovery across the cluster, integrates with the membership service to reactively recover detected faulty nodes, and uses guardians to externally enforce node rejuvenation.

Furthermore, we addressed the critical role of network-agnostic guardians in modern topologies. These guardians are highly desirable because they enforce network access according to a defined schedule while remaining entirely independent of the underlying network protocol and topology, effectively reducing design and integration complexity. However, because classical implementations must rely on their controlled node's networking infrastructure to maintain clock synchrony, they operate under the fragile assumption that nodes only fail silently. We established an impossibility result demonstrating that the safe re-synchronization of network-agnostic guardians cannot be guaranteed when the nodes they monitor exhibit malicious behavior. This vulnerability becomes particularly evident when compromised nodes execute time-domain acceptance-window attacks to covertly desynchronize the guardians from global time.

Addressing this fundamental synchronization limitation, we introduced *SyncGuard*, the first network-agnostic and time-domain attack-resilient guardian. SyncGuard extends the traditional guardian's responsibility from merely coordinating network access to actively coordinating both the proactive and reactive rejuvenation of the nodes themselves. By operating as a trusted distributed wormhole subsystem that securely exchanges authenticated feedback messages, SyncGuard maintains system synchrony, ensures resource-exhaustion safe operation, and enforces collision-free time-triggered communication across highly heterogeneous network topologies.

6.2 Future Directions

While the contributions presented in this thesis establish a theoretical foundation for next-generation intrusion-resilient and flexible time-triggered systems, they represent a milestone rather than an endpoint. The increasing complexity of modern hardware and the continuously evolving nature of cyber-threats present new research opportunities. In this section, we describe promising avenues for extending this foundation.

Dynamic Time-Triggered Scheduling in Multicore Architectures

The assumption made in this thesis is that each node in the distributed system comprises a single uniprocessor host computer. Modern safety-critical systems, however, increasingly rely on multicore architectures to meet growing computational demands. Transitioning dynamic time-triggered scheduling to these architectures requires investigating the effects of dynamic slot assignment on task *execution times*. Tasks exe-

cuting in parallel across multiple cores inevitably contend for shared microarchitectural resources, such as main memory. Under table-driven time-triggered scheduling, the exact set of concurrently executing tasks is strictly determined by the fixed slot assignments within the offline scheduling table, resulting in predictable interference intervals. In contrast, dynamic time-triggered scheduling assigns computation slots dynamically within task execution windows. Although the set of potentially interfering tasks remains identifiable through their overlapping windows, this runtime flexibility dictates that the execution time analysis must account for the absolute worst-case overlap of concurrently contending tasks, leading to more pessimistic WCET bounds.

Consequently, admission tests for multicore platforms demand more than merely verifying remaining processor capacity. They must also account for this dynamic contention over shared resources, such as main memory bandwidth and cache interference, which directly influence the temporal behavior of parallel workloads. Formulating admission tests for multicore architectures that model these interference bounds, while incorporating runtime knowledge to limit analytical pessimism, constitutes a challenge left for future work.

Time-Triggered Systems under Uncertain Execution Times

The timing guarantees provided by the time-triggered paradigm are intrinsically dependent on the assumption of a known, tight, and safe WCET for every task in the system. This strict WCET assumption is becoming increasingly untenable. Modern embedded platforms are driven by the need for high performance, which, in turn, results in the adoption of multicore processors, GPUs, complex cache hierarchies, shared resources, and AI accelerators. The timing behavior of these architectures is optimized for average-case performance rather than strict temporal analyzability and exhibits large variability due to shared-resource contention, asynchronous execution, and complex micro-architectural behavior. As a result, deriving tight and trustworthy WCET bounds is becoming exceedingly difficult and often intractable in practice. Furthermore, even when conservative worst-case bounds can be established, designing entire systems around rare execution scenarios tends to lead to excessive over-provisioning and poor cost-efficiency, particularly for less-critical software components. Therefore, even in cases where the use of sound WCET bounds is technically feasible, it is not necessarily desirable or economically viable for all tasks.

Consequently, future time-triggered systems cannot be engineered under the illusion that tight WCET bounds exist for all components; a strictly WCET-centric design paradigm either precludes the use of modern functionality or forces an unacceptable trade-off between predictability and efficiency. An important research question, which we leave for future work, is how to provision time-triggered systems on modern hetero-

geneous and multicore platforms in the presence of incomplete or uncertain execution time information, and crucially, without incurring prohibitive resource over-provisioning.

Because our DTSS algorithm (see Chapter 4) natively enforces execution-time budgets and maintains an accurate view of the remaining processor capacity, it provides a foundation for handling execution overruns reactively. Future research can extend DTSS to incorporate principled, criticality-aware mitigation strategies. For example, upon detecting an overrun, an enhanced DTSS could pause the overrunning task, evaluate the look-ahead EDL schedule for available slack, and apply a targeted resolution. Depending on the task's assigned criticality and parameters, the scheduler could dynamically grant a budget extension, gracefully degrade system performance by skipping lower-priority or soft aperiodic workloads to free up capacity, or safely abort the overrunning task entirely. By leveraging these reactive capabilities, the requirements of strict WCET can be relaxed, while embracing modern platforms and maintaining temporal isolation for high criticality tasks.

Distributed Wormhole Subsystems for Flexible Time-Triggered Systems

As demonstrated in Chapter 5, SyncGuards represent a valuable and robust solution for maintaining synchrony and ensuring collision-free time-triggered network access across modern safety-critical systems with heterogeneous networks. By operating as isolated and trustworthy components within a potentially compromised payload environment, they embody the design paradigm of architectural hybridization.

A compelling path for future work involves exploring the applicability of similar distributed wormhole solutions to improve architectural flexibility in time-triggered systems. For instance, as discussed in Chapter 3, the central node approach represents a viable and efficient strategy to synchronize a cluster of nodes and flexibly adjust message scheduling at runtime. However, a central node inherently constitutes a single point of failure, making it an unacceptable vulnerability in adversarial and safety-critical environments.

Future research can explore the idea of delegating the responsibility of dynamic system-wide scheduling to distributed local wormhole components (i.e., trusted, highly predictable subsystems embedded within untrusted, highly complex payload system). Through secure bounded-latency consensus protocols running exclusively among these trusted local wormholes, the system could autonomously adjust the schedule to accommodate dynamic workloads or hardware reconfigurations. Although this distributed architecture inherently eliminates the central single point of failure, reaching agreement on the global schedule while strictly adhering to real-time deadlines presents a significant and compelling research challenge.

Interplay Between Obfuscation, Diversification, and Time-Triggered Schedulers

As established in Chapter 5, our rejuvenation service operates under the assumption that the implementation of nodes (including application tasks) is diversified and obfuscated. This architectural diversity is crucial for preventing stealthy adversaries from weaponizing a single vulnerability across all replicas simultaneously. By ensuring that an exploit successful against one replica ultimately fails against another, diversification significantly strengthens the statistical independence of the fault-containment regions.

Although this thesis leverages these assumptions to guarantee system resilience, the interplay between diversified task variants and time-triggered schedulers remains a largely unexplored and vital frontier. Crucially, different obfuscated versions of the same task can have distinct execution times and resource requirements. Future research must investigate how admission tests and runtime scheduling mechanisms can be made *diversity-aware* to manage this variability. For instance, when a new task awaits admission but the remaining processor capacity is insufficient, the scheduler could dynamically select a computationally lighter, functionally equivalent variant of that task. Alternatively, the scheduler could transition some existing system tasks to their lighter variants to dynamically free up the necessary capacity. Formulating such scheduling mechanisms and their corresponding admission tests provides a vital roadmap for next-generation intrusion-tolerant time-triggered systems.

Bibliography

[1] H. Kopetz and W. Steiner, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 3rd ed. Springer, 2022.

[2] *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Inc. RTCA Standard DO-178C, Dec. 2011.

[3] *ISO 26262: Road vehicles - Functional safety*, International Organization for Standardization Std., 2018.

[4] E. A. Lee, "Determinism," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5, May 2021. [Online]. Available: https://doi.org/10.1145/3453652

[5] S. Poledna, A. Burns, A. Wellings, and P. Barrett, "Replica determinism and flexible scheduling in hard real-time dependable systems," *IEEE Transactions on Computers*, vol. 49, no. 2, pp. 100–111, 2000.

[6] R. Obermaisser, *Time-Triggered Communication*, 1st ed. USA: CRC Press, 2011.

[7] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," in *International conference on fundamentals of computation theory*. Springer, 1983, pp. 127–140.

[8] S. Sallinger, G. Weissenbacher, and F. Zuleger, "A Formalization of Heisenbugs and Their Causes," in *Software Engineering and Formal Methods*. Springer, Cham, 2023, pp. 282–300.

[9] H. Kopetz, "The temporal specification of interfaces in distributed real-time systems," in *Embedded Software (EMSOFT)*. Springer, 2001.

[10] A. Minaeva and Z. Hanzálek, "Survey on periodic scheduling for time-triggered hard real-time systems," *ACM Computing Surveys (CSUR)*, vol. 54, no. 1, pp. 1–32, 2021.

[11] P. Bieber, F. Boniol, M. Boyer, E. Noulard, and C. Pagetti, "New Challenges for Future Avionic Architectures." *Aerospace Lab*, no. 4, pp. p. 1–10, May 2012. [Online]. Available: https://hal.science/hal-01184101

[12] C. M. Fuchs and A. S. Schneele, "The evolution of avionics networks from ARINC 429 to AFDX," *Innovative Internet Technologies and Mobile Communications (IITM), and Aerospace Networks (AN)*, vol. 65, pp. 1551–3203, 2012.

[13] J. Murdock and J. R. Koenig, "Open systems avionics network to replace MIL-STD-1553," in *19th DASC. 19th Digital Avionics Systems Conference. Proceedings (Cat. No. 00CH37126)*, vol. 1. IEEE, 2000, pp. 4E5–1.

[14] H. Butz, "Open integrated modular avionic (ima): State of the art and future development road map at airbus deutschland," *Signal*, vol. 10, p. 1000, 2010.

[15] T. Gaska, C. Watkin, and Y. Chen, “Integrated modular avionics-past, present, and future,” *IEEE Aerospace and Electronic Systems Magazine*, vol. 30, no. 9, pp. 12–23, 2015.

[16] *Avionics Application Software Standard Interface*, Aeronautical Radio, Inc. Std. ARINC Specification 653, 1997.

[17] *AUTOSAR (AUTomotive Open System ARchitecture) Standard*, AUTOSAR Development Partnership Std., 2015.

[18] H. Kopetz, “The Time-Triggered Model of Computation,” *Real-Time Systems*, 2003.

[19] H. Kopetz and G. Bauer, “The time-triggered architecture,” *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[20] H. Kopetz and G. Grunsteidl, “TTP - A time-triggered protocol for fault-tolerant real-time systems,” in *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, 1993, pp. 524–533.

[21] TTTech, *TTP Communication Protocol Specification*, Std., FEB 2011.

[22] A.-D. D. Ethernet and U. Networking, *Time-Triggered Ethernet*, Nov. 2016.

[23] W. Steiner, “TTEthernet: Time-triggered services for Ethernet networks,” in *2009 IEEE/AIAA 28th Digital Avionics Systems Conference*, 2009, pp. 1.B.4–1–1.B.4–1.

[24] D. Tămaş–Selicean, P. Pop, and W. Steiner, “Design Optimization of TTEthernet-Based Distributed Real-Time Systems,” *Real-Time Systems*, vol. 51, pp. 1–35, 2015. [Online]. Available: https://link.springer.com/article/10.1007/s11241-014-9214-8

[25] IEEE Computer Society, *IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications*, 2020. [Online]. Available: https://standards.ieee.org/standard/802_1AS-2020.html

[26] V. Ghotra, M. Helm, and B. Jaeger, “TSN Qbv and Schedule Generation Approaches,” *Technical University of Munich*, 2023. [Online]. Available: https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2023-06-1/NET-2023-06-1_06.pdf

[27] M. Broy, “Challenges in Automotive Software Engineering,” in *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. ACM, 2006, pp. 33–42.

[28] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, “Trends in automotive communication systems,” *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1204–1223, 2005.

[29] P. Puschner and R. Kirner, “From time-triggered to time-deterministic real-time systems,” in *From Model-Driven Design to Resource Management for Distributed Embedded Systems*, B. Kleinjohann, L. Kleinjohann, R. J. Machado, C. E. Pereira, and P. S. Thiagarajan, Eds. Boston, MA: Springer US, 2006, pp. 115–124.

[30] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM transactions on embedded computing systems (TECS)*, vol. 7, no. 3, pp. 1–53, 2008.

[31] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor System-on-Chip (MPSoC) Technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1701–1713, 2008.

[32] R. Obermaisser, C. E. Salloum, B. Huber, and H. Kopetz, "The time-triggered System-on-a-Chip architecture," in *2008 IEEE International Symposium on Industrial Electronics*, 2008, pp. 1941–1947.

[33] C. E. Salloum, M. Elshuber, O. Höftberger, H. Isakovic, and A. Wasicek, "The ACROSS MPSoC – A New Generation of Multi-core Processors Designed for Safety-Critical Embedded Systems," in *2012 15th Euromicro Conference on Digital System Design*, 2012, pp. 105–113.

[34] S. Chattopadhyay, A. Roychoudhury, J. Rosén, P. Eles, and Z. Peng, "Time-predictable embedded software on multi-core platforms: Analysis and optimization," *Foundatiosn and Trends in Electronic Design Automation*, vol. 8, no. 3-4, pp. 199–356, 2014.

[35] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of field robotics*, vol. 37, no. 3, pp. 362–386, 2020.

[36] A. Carrio, C. Sampedro, A. Rodriguez-Ramos, and P. Campoy, "A review of deep learning methods and applications for unmanned aerial vehicles," *Journal of Sensors*, vol. 2017, no. 1, p. 3296874, 2017.

[37] J. Rushby, "Bus architectures for safety-critical embedded systems," *EMSOFT 2001*, pp. 306–323, 2001.

[38] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[39] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive experimental analyses of automotive attack surfaces," in *20th USENIX security symposium (USENIX Security 11)*, 2011.

[40] A. Greenberg, "Hackers Remotely Kill a Jeep on the Highway—With Me in It," wired, 2015, last accessed: 12-12-2022. [Online]. Available: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

[41] R. M. Lee, M. J. Assante, and T. Conway, "Analysis of the Cyber Attack on the Ukrainian Power Grid," March 2016. [Online]. Available: https://ics.sans.org/media/E-ISAC_SANS_Ukraine_DUC_5.pdf

[42] J. Slay and M. Miller, "Lessons Learned from the Maroochy Water Breach," in *Critical Infrastructure Protection*, E. Goetz and S. Shenoi, Eds. Boston, MA: Springer US, 2008, pp. 73–82.

[43] D. Shepard, J. Bhatti, and T. Humphreys, "Drone Hack: Spoofing Attack Demonstration on a Civilian Unmanned Aerial Vehicle," *GPS World*, vol. 23, pp. 30–33, 08 2012.

[44] T. M. Chen and S. Abu-Nimeh, "Lessons from Stuxnet," *Computer*, vol. 44, no. 4, pp. 91–93, 2011.

[45] J. Meserve, "Mouse click could plunge city into darkness, experts say," CNN, 2007, last accessed: 12-12-2022. [Online]. Available: https://edition.cnn.com/2007/US/09/27/power.at.risk/index.html

[46] S. Scoles, "The Feds Want These Teams to Hack a Satellite—From Home," wired, 2020, last accessed: 12-12-2022. [Online]. Available: https://www.wired.com/story/the-feds-want-these-teams-to-hack-a-satellite-from-home/

[47] B. Sari, "Fail-operational safety architecture for ADAS/AD systems," in *Fail-operational Safety Architecture for ADAS/AD Systems and a Model-driven Approach for Dependent Failure Analysis*. Springer, 2020, pp. 31–75.

[48] G. M. Uchenick and W. M. Vanfleet, "Multiple independent levels of safety and security: high assurance architecture for MSLS/MLS," in *MILCOM 2005-2005 IEEE Military Communications Conference*. IEEE, 2005, pp. 610–614.

[49] ISO/SAE, *ISO/SAE 21434:2021 Road vehicles — Cybersecurity engineering*, International Organization for Standardization, Std., 2021.

[50] RTCA, Inc., *DO-326A: Airworthiness Security Process Specification*, RTCA DO-326A, Std., 2014.

[51] A. R. Wasicek, "Security in Time-Triggered Systems," Ph.D. dissertation, Technische Universität Wien, 2011.

[52] M. I. Alkoudsi, M. Völp, and G. Fohler, "Dependency Analysis of the Extra-Functional Properties of the Time-Triggered Architecture," *IEEE Access*, 2026, accepted for publication.

[53] M. I. Alkoudsi and G. Fohler, "Scheduling Dynamic Task-Sets in Time-Triggered Real-Time Systems," in *Proceedings of the 32nd International Conference on Real-Time Networks and Systems*, ser. RTNS '24. New York, NY, USA: Association for Computing Machinery, 2025, p. 48–58. [Online]. Available: https://doi.org/10.1145/3696355.3699699

[54] M. I. Alkoudsi, D. Isovic, and G. Fohler, "Revisiting Slot-Shifting's Offline Acceptance Test for Sporadic Tasks: A Technical Note," *Leibniz Transactions on Embedded Systems*, vol. 10, no. 1, pp. 4:1–4:6, 2025. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LITES.10.1.4

[55] M. I. Alkoudsi, S. Baruah, P. Ekberg, G. Fohler, J. Goossens, and M. Moslehi, "Extending Periodic Task Sets with Sporadic Tasks: Computational Complexity and

Exact Feasibility Tests," in *Proceedings of the 33rd International Conference on Real-Time Networks and Systems (RTNS 2025)*, Pisa, Italy, 2025.

[56] M. I. Alkoudsi, G. Fohler, and M. Völp, "Tolerating Resource Exhaustion Attacks in the Time-Triggered Architecture," in *2022 XII Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2022, pp. 1–8.

[57] M. I. Alkoudsi, G. Fohler, and M. Völp, "A Network-Agnostic Approach to Enforcing Collision-Free Time-Triggered Communication," in *2023 IEEE 28th Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2023, pp. 71–77.

[58] P. Verissimo and L. Rodrigues, *Distributed Systems for System Architects*. USA: Kluwer Academic Publishers, 2001.

[59] K. Ramamritham, "Where Do Time Constraints Come From? Where Do They Go?" *Journal of Database Management*, vol. 7, no. 2, pp. 4–10, 1996.

[60] G. C. Buttazzo, "Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition," in *Real-Time Systems Series*, 2011.

[61] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[62] Y. Tang, N. Guan, and W. Yi, "Real-time task models," in *Handbook of real-time computing*. Springer, 2022, pp. 469–487.

[63] S. Baruah, R. Howell, and L. Rosier, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Systems: The International Journal of Time-Critical Computing*, vol. 2, pp. 301–324, 1990.

[64] M. Stigge and W. Yi, "Graph-Based Models for Real-Time Workload: A Survey," *Real-Time Systems*, 2015. [Online]. Available: https://user.it.uu.se/~wangyi/pdf-files/2015/sy-survey-rtsj15.pdf

[65] J.-J. Chen, G. Nelissen, and W.-H. Huang, "A Review of Self-Suspending Tasks in Real-Time Systems," *Real-Time Systems*, 2019. [Online]. Available: https://eprints.whiterose.ac.uk/id/eprint/135393/1/2019_JRTS.pdf

[66] D. Powell, G. Bonn, D. T. Seaton, P. Verissimo, and F. Waeselynck, "The Delta-4 Approach to Dependability in Open Distributed Computing Systems." in *FTCS*, vol. 18, 1988.

[67] E. D. Jensen, C. D. Locke, and H. Tokuda, "A time-driven scheduling model for real-time operating systems." in *Rtss*, vol. 85, 1985, pp. 112–122.

[68] P. Veríssimo, A. Casimiro, and C. Fetzer, "The Timely Computing Base model and architecture," *IEEE Transactions on Computers*, vol. 49, no. 8, pp. 822–837, 2000.

[69] S. Vestal, *Preemptive Scheduling of Multi-Criticality Systems with Varying Degrees of Execution Time Assurance*, Honeywell Laboratories Std., 2007.

[70] A. Burns and R. Davis, "A survey of research into mixed criticality systems," *ACM Computing Surveys*, vol. 50, no. 6, pp. 1–37, 2017.

[71] J. Lundelius and N. Lynch, “An upper and lower bound for clock synchronization,” *Information and Control*, vol. 62, no. 2, pp. 190–204, 1984.

[72] J. Hartmanis, “Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson),” *Siam Review*, vol. 24, no. 1, p. 90, 1982.

[73] S. Rai and G. Vairaktarakis, “NP-complete Problems and Proof Methodology,” 01 2008.

[74] J. D. Ullman, “NP-complete scheduling problems,” *Journal of Computer and System sciences*, vol. 10, no. 3, pp. 384–393, 1975.

[75] J. K. Lenstra and A. H. G. Rinnooy Kan, “Complexity of Scheduling under Precedence Constraints,” *Operations Research*, vol. 26, no. 1, pp. 22–35, 1978.

[76] M. R. Garey and D. S. Johnson, “Complexity Results for Multiprocessor Scheduling under Resource Constraints,” *SIAM*, vol. 4, pp. 397–411, 1975.

[77] M. Völp, M. I. Alkoudsi, A. B. Asl, K. Krüger, J. R. M. da Neto, and G. Fohler, “Defending Event-Triggered Systems against Out-of-Envelope Environments,” 2025. [Online]. Available: https://arxiv.org/abs/2512.06331

[78] J. Y.-T. Leung and M. L. Whitehead, “On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks,” in *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*. ACM, 1982, pp. 279–290.

[79] R. Barry, *FreeRTOS Reference Manual*, Real Time Engineers Ltd., 2024, available at https://www.freertos.org/.

[80] Zephyr Project Contributors, “Zephyr Project Documentation,” https://docs.zephyrproject.org/, 2024, linux Foundation.

[81] “AUTOSAR Specification of Operating System,” Standard R19-11, 2019, accessed: 2026-04-08. [Online]. Available: https://www.autosar.org/fileadmin/standards/R19-11/CP/AUTOSAR_SWS_OS.pdf

[82] J. P. Lehoczky, “Fixed Priority Scheduling of Sporadic Tasks,” in *Fixed Priority Scheduling Theory for Hard Real-Time Systems*, J. P. Lehoczky and L. Sha, Eds. Kluwer Academic, 1990, pp. 1–24, also appears in collected works on fixed-priority theory.

[83] E. Bini and G. C. Buttazzo, “Rate Monotonic Analysis: The Hyperbolic Bound,” in *Proceedings of the 24th IEEE Real-Time Systems Symposium*. IEEE, 2003, pp. 71–79.

[84] K. Jeffay, D. F. Stanat, and C. U. Martel, “On Non-Preemptive Scheduling of Periodic and Sporadic Tasks,” in *Proc. 12th IEEE Real-Time Systems Symposium (RTSS)*, 1991, pp. 129–139.

[85] G. Brüggen, J.-J. Chen, and W.-H. Huang, “Schedulability and Optimization Analysis for Non-Preemptive Static Priority Scheduling Based on Task Utilization and Blocking Factors,” in *Proc. 27th Euromicro Conference on Real-Time Systems (ECRTS)*, 2015, pp. 90–101.

[86] A. Burns, B. F. Niu, M. J. Mercer, and S. M. Petters, “Criticality-Based Priority Assignment for Real-Time Systems,” in *Proc. 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2005.

[87] R. Ghattas and A. G. Dean, “Preemption Threshold Scheduling: Stack Optimality, Enhancements and Analysis,” in *Proc. 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2007, pp. 147–157.

[88] J. P. Lehoczky, L. Sha, and J. K. Strosnider, “Enhanced Aperiodic Responsiveness in Hard Real-Time Environments,” in *Proc. 8th IEEE Real-Time Systems Symposium (RTSS)*, 1987, pp. 261–270.

[89] J. K. Strosnider, J. P. Lehoczky, and L. Sha, “The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments,” *IEEE Transactions on Computers*, vol. 44, no. 1, pp. 73–91, 1995.

[90] B. Sprunt, L. Sha, and J. P. Lehoczky, “Aperiodic Task Scheduling for Hard-Real-Time Systems,” *Real-Time Systems*, vol. 2, no. 1, pp. 27–60, 1990.

[91] L. Spuri and G. C. Buttazzo, “Scheduling Aperiodic Tasks in Dynamic Priority Systems,” *Real-Time Systems*, vol. 10, no. 2, pp. 179–210, 1996.

[92] L. Abeni and G. C. Buttazzo, “Integrating Multimedia Applications in Hard Real-Time Systems,” in *Proc. 19th IEEE Real-Time Systems Symposium (RTSS)*, 1998, pp. 4–13.

[93] H. Chetto and M. Chetto, “Some Results of the Earliest Deadline Scheduling Algorithm,” *IEEE Transactions on Software Engineering*, vol. 15, no. 10, pp. 1261–1269, 1989.

[94] I. Ripoll, A. Crespo, and A. García-Fornes, “An Optimal Algorithm for Scheduling Soft Aperiodic Tasks in Dynamic-Priority Preemptive Systems,” *IEEE Transactions on Software Engineering*, vol. 23, no. 6, pp. 388–400, 1997.

[95] A. Biondi, A. Melani, and M. Bertogna, “Hard constant bandwidth server: Comprehensive formulation and critical scenarios,” in *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*. IEEE, 2014, pp. 29–37.

[96] D. Casini, L. Abeni, A. Biondi, T. Cucinotta, and G. Buttazzo, “Constant bandwidth servers with constrained deadlines,” in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, 2017, pp. 68–77.

[97] L. Sha, R. Rajkumar, and J. P. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronization,” *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, sep 1990.

[98] T. P. Baker, “A Stack-Based Resource Allocation Policy for Realtime Processes,” in *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 1990, pp. 191–200, note: The journal version appeared in 1991, but the foundational conference paper is often cited as 1990.

[99] R. I. Davis and A. Burns, “A Survey of Hard Real-Time Scheduling for Multiprocessor Systems,” *ACM Computing Surveys*, vol. 43, no. 4, pp. 35:1–35:44, oct 2011.

[100] S. Baruah, M. Bertogna, and G. Buttazzo, *Multiprocessor Scheduling for Real-Time Systems*, ser. Embedded Systems. Springer International Publishing, 2016.

[101] S. K. Dhall and C. L. Liu, "On a Real-Time Scheduling Problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[102] N. Fisher, J. Goossens, and S. Baruah, "Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible," *Real-Time Systems*, vol. 45, no. 1, pp. 26–71, jun 2010.

[103] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," in *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, 1993, pp. 345–354.

[104] M. Moir and S. Ramamurthy, "Pfair Scheduling of Fixed and Migrating Periodic Tasks on Multiple Resources," Department of Computer Science, University of Pittsburgh, Pittsburgh, PA, USA, Tech. Rep. TR-99-18, 1999.

[105] A. Srinivasan and J. H. Anderson, "Optimal Rate-Based Scheduling on Multiprocessors," *Journal of Computer and System Sciences*, vol. 68, no. 3, pp. 555–586, 2004.

[106] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah, "The case for fair multiprocessor scheduling," in *Proceedings International Parallel and Distributed Processing Symposium*, 2003, pp. 10 pp.–.

[107] B. Andersson and E. Tovar, "Multiprocessor Scheduling with Few Preemptions," in *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE Computer Society, jul 2006, pp. 322–334.

[108] I. Ripoll, A. Crespo, and A. K. Mok, "Improvement in feasibility testing for real-time tasks," *Real-Time Systems: The International Journal of Time-Critical Computing*, vol. 11, pp. 19–39, 1996.

[109] R. Pellizzoni and G. Lipari, "Feasibility analysis of real-time periodic tasks with offsets," *Real-Time Systems*, vol. 30, no. 1, pp. 105–128, 2005.

[110] S. Baruah, A. Mok, and L. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Proceedings of the 11th Real-Time Systems Symposium*. Orlando, Florida: IEEE Computer Society Press, 1990, pp. 182–190.

[111] P. Ekberg and W. Yi, "Uniprocessor feasibility of sporadic tasks with constrained deadlines is strongly coNP-complete," in *2015 27th Euromicro Conference on Real-Time Systems*, 2015, pp. 281–286.

[112] S. K. Baruah, R. R. Howell, and L. E. Rosier, "Feasibility problems for recurring tasks on one processor," *Theoretical Computer Science*, vol. 118, no. 1, pp. 3–20, 1993.

[113] F. Zhang and A. Burns, "Schedulability Analysis for Real-Time Systems with EDF Scheduling," *IEEE Transactions on Computers*, vol. 58, no. 9, pp. 1250–1258, 2009.

[114] L. George, N. Rivierre, and M. Spuri, “Preemptive and non-preemptive real-time UniProcessor scheduling,” no. RR-2966, 1996, technical Report, INRIA.

[115] T. Baker, “An analysis of EDF schedulability on a multiprocessor,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 8, pp. 760–768, 2005.

[116] S. Baruah and T. Baker, “Schedulability analysis of global EDF,” *Real-Time Systems*, vol. 38, no. 3, pp. 223–235, 2008.

[117] G. Fohler, “Predictable Flexible Real-Time Systems,” in *Advances in Real-Time Systems*. Springer, 2012, pp. 207–218.

[118] G. Fohler, “Flexibility in statically scheduled hard realtime systems,” *Ph. D. Thesis, Technisch Universitaet Wien*, 1994.

[119] D.-T. Peng, K. G. Shin, and T. F. Abdelzaher, “Assignment and Scheduling Communicating Periodic Tasks in Distributed Real-Time Systems,” *IEEE Trans. Software Eng.*, vol. 23, pp. 745–758, 1997. [Online]. Available: https://api.semanticscholar.org/CorpusID:13537985

[120] F. Heilmann, A. Syed, and G. Fohler, “Mode-Changes in COTS Time-Triggered Network Hardware without Online Reconfiguration,” *SIGBED Review*, vol. 13, no. 4, pp. 55–60, 2016.

[121] G. Fohler, “Changing operational modes in the context of pre run-time scheduling,” *IEICE Transactions on Information and Systems*, vol. 76, no. 11, pp. 1333–1340, 1993.

[122] G. Fohler, “Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems,” in *Proceedings of the 16th IEEE Real-Time Systems Symposium*, Dec 1995, pp. 152–161.

[123] S. S. Craciunas, R. Serna Oliver, and V. Ecker, “Optimal static scheduling of real-time tasks on distributed time-triggered networked systems,” *IEEE ETFA*, pp. 1–8, 2014.

[124] K. Ramamritham, “Allocation and scheduling of precedence-related periodic tasks,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 4, pp. 412–420, 1995.

[125] D. Isovic and G. Fohler, “Handling mixed sets of tasks in combined offline and online scheduled real-time systems,” *Real-time systems*, vol. 43, no. 3, pp. 296–325, 2009.

[126] G. Fohler, “Flexibility in statically scheduled real-time systems,” Ph.D. dissertation, Technische Universität Wien, 1994.

[127] C. Temple, “Avoiding the babbling-idiot failure in a time-triggered communication system,” in *28th International Symposium on Fault-Tolerant Computing*, 1998, pp. 218–227.

[128] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla, “Cache side-channel attacks and time-predictability in high-performance critical real-time systems,” in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[129] N. Vreman, R. Pates, K. Krüger, G. Fohler, and M. Maggio, "Minimizing side-channel attack vulnerability via schedule randomization," in *2019 IEEE 58th Conference on decision and control (CDC)*. IEEE, 2019, pp. 2928–2933.

[130] A. Loveless, L. T. X. Phan, R. Dreslinski, and B. Kasikci, "Pcspoof: Compromising the safety of time-triggered ethernet," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 3193–3208.

[131] H. Kopetz, "Fault containment and error detection in the time-triggered architecture," in *Sixth International Symposium on Autonomous Decentralized Systems (ISADS)*, 2003, pp. 139–146.

[132] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No. 97CB36097)*. IEEE, 1997, pp. 65–71.

[133] A. Regenscheid, "Platform firmware resiliency guidelines," National Institute of Standards and Technology, Tech. Rep., 2017.

[134] J. Rushby *et al.*, "A trusted computing base for embedded systems," Tech. Rep., 1984.

[135] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[136] P. Verissimo, M. Correia, N. F. Neves, and P. Sousa, "Intrusion-resilient middleware design and validation," *Information Assurance, Security and Privacy Services*, vol. 4, pp. 615–678, 2009.

[137] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[138] H. Kopetz, "The time-triggered model of computation," in *19th IEEE Real-Time Systems Symposium*, 1998, pp. 168–177.

[139] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[140] M. Ben-Or, "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols," *PODC*, pp. 27–30, 1983.

[141] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[142] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[143] E. Brewer, "CAP twelve years later: How the" rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[144] E. A. Lee, S. Bateni, S. Lin, M. Lohstroh, and C. Menard, "Quantifying and generalizing the CAP theorem," *CoRR*, vol. abs/2109.07771, 2021. [Online]. Available: https://arxiv.org/abs/2109.07771

[145] P. Peti, “The Concepts behind Time, State, Component, and Interface - A Literature Survey,” Institute of Computer Engineering, Vienna University of Technology, Tech. Rep., 2002. [Online]. Available: http://www.vmars.tuwien.ac.at/php/pserver/extern/docdetail.php?DID=1047

[146] G. Kahn, “The Semantics of a Simple Language for Parallel Programming,” *IFIP Congress*, pp. 471–475, 1974.

[147] F. Boussinot and R. de Simone, “The Esterel language,” *Proceedings of the IEEE*, vol. 79, pp. 1293–1304, 1991.

[148] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, “The synchronous languages 12 years later,” *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[149] P. Panangaden and K. Taylor, “Concurrent common knowledge: a new definition of agreement for asynchronous systems,” in *Proceedings of PODC ’88*. ACM, 1988, pp. 197–209.

[150] J. Y. Halpern and Y. Moses, “Knowledge and common knowledge in a distributed environment,” 2000. [Online]. Available: https://arxiv.org/abs/cs/0006009

[151] L. Rodrigues, P. Veríssimo, and A. Casimiro, “Priority-Based Totally Ordered Multicast,” *IFAC Proceedings Volumes*, vol. 28, no. 5, pp. 351–359, 1995.

[152] H. Kopetz, “Sparse time versus dense time in distributed real-time systems,” in *Proceedings of the 12th International Conference on Distributed Computing Systems*, 1992, pp. 460–467.

[153] W. Steiner, M. Paulitsch, and H. Kopetz, “The TTA’s Approach to Resilience after Transient Upsets,” *Real-Time Systems*, vol. 32, pp. 213–233, 2006.

[154] F. P. Mathur and P. T. de Sousa, “Reliability models of NMR systems,” *IEEE Transactions on Reliability*, vol. 24, no. 2, pp. 108–113, 1975.

[155] L. Lamport, R. Shostak, and M. Pease, “The Byzantine generals problem,” in *Concurrency: the works of leslie lamport*, 2019, pp. 203–226.

[156] K. P. Birman and T. A. Joseph, “Reliable Communication in the Presence of Failures,” *ACM Transactions on Computer Systems*, vol. 5, no. 1, pp. 47–76, 1987.

[157] X. Défago, A. Schiper, and P. Urbán, “Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey,” *ACM Computing Surveys*, vol. 36, no. 4, pp. 372–421, 2004.

[158] L. Lamport, “Paxos made simple,” *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

[159] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *USENIX ATC 2014*, 2014, pp. 305–319.

[160] H. Howard and R. Mortier, “Paxos vs Raft: Have we reached consensus on distributed consensus?” in *PaPoC 2020*, 2020, pp. 1–9.

[161] M. Castro and B. Liskov, “Practical Byzantine fault tolerance,” in *OSDI 1999*, 1999, pp. 173–186.

[162] M. Castro and B. Liskov, “Practical Byzantine Fault Tolerance and Proactive Recovery,” in *ACM Transactions on Computer Systems (TOCS)*, vol. 20. ACM, November 2002.

[163] Y. Amir, B. Coan, J. Kirsch, and J. Lane, “Prime: Byzantine replication under attack,” *IEEE transactions on dependable and secure computing*, vol. 8, no. 4, pp. 564–577, 2010.

[164] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, “Spin one’s wheels? Byzantine fault tolerance with a spinning primary,” in *2009 28th IEEE International Symposium on Reliable Distributed Systems*. IEEE, 2009, pp. 135–144.

[165] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, “EBAWA: Efficient Byzantine agreement for wide-area networks,” in *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*. IEEE, 2010, pp. 10–19.

[166] P. Verissimo, “Travelling through wormholes: a new look at distributed systems models,” *SIGACT News*, vol. 37, no. 1, pp. 66–81, 2006.

[167] P. Veríssimo, “Uncertainty and predictability: Can they be reconciled?” in *Future Directions in Distributed Computing: Research and Position Papers*. Springer, 2003, pp. 108–113.

[168] F. Cristian and C. Fetzer, “The timed asynchronous distributed system model,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 642–657, 1999.

[169] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, “Efficient byzantine fault-tolerance,” *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2011.

[170] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, “CheapBFT: Resource-Efficient Byzantine Fault Tolerance,” in *EuroSys ’12*, 2012, pp. 295–308.

[171] M. Correia, N. F. Neves, and P. Verissimo, “Bft-to: Intrusion tolerance with less replicas,” *The Computer Journal*, vol. 56, no. 6, pp. 693–715, 2013.

[172] M. Correia, P. Veríssimo, and N. F. Neves, “The design of a COTS real-time distributed security kernel,” in *European Dependable Computing Conference*. Springer, 2002, pp. 234–252.

[173] P. E. Veríssimo, N. F. Neves, and M. P. Correia, “Intrusion-Tolerant Architectures: Concepts and Design,” in *Architecting Dependable Systems*, R. de Lemos, C. Gacek, and A. Romanovsky, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 3–36.

[174] P. Sousa, N. F. Neves, and P. Verissimo, “Proactive resilience through architectural hybridization,” in *Proceedings of the 2006 ACM Symposium on Applied Computing*. ACM, 2006, pp. 686–690.

[175] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, vol. 1, 1978, pp. 3–9.

[176] A. Avizienis and L. Chen, "On the implementation of N-version programming for software fault-tolerance during program execution," in *Proceedings of IEEE COMPSAC*, 1977, pp. 149–155.

[177] T. Roeder and F. Schneider, "Proactive Obfuscation," *ACM Transactions on Computer Systems*, vol. 28, 2010.

[178] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Transparent runtime randomization for security," in *22nd International Symposium on Reliable Distributed Systems*, 2003, pp. 260–269.

[179] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert, "Proactive Management of Software Aging," *IBM Journal of Research and Development*, vol. 45, no. 2.3, pp. 311–332, 2001.

[180] J. Alonso, R. Matias, E. Vicente, A. Maria, and K. S. Trivedi, "A comparative experimental study of software rejuvenation overhead," *Performance Evaluation*, vol. 70, no. 3, pp. 231–250, 2013.

[181] J. Zhao, Y. Hu, and K. S. Trivedi, "A comprehensive approach to optimal software rejuvenation," *Performance Evaluation*, vol. 70, no. 11, pp. 917–933, 2013.

[182] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo, "Highly Available Intrusion-Tolerant Services with Proactive-Reactive Recovery," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, pp. 452–465, 2010.

[183] A. Matovic, R. Graczyk, F. Lucchetti, and M. Völp, "Consensual Resilient Control: Stateless Recovery of Stateful Controllers," in *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2023, pp. 14–1.

[184] R. Pietrantuono and S. Russo, "A survey on software aging and rejuvenation in the cloud," *Software Quality Journal*, vol. 28, no. 1, pp. 7–38, 2020.

[185] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro, "OS diversity for intrusion tolerance: Myth or reality?" in *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, 2011, pp. 383–394.

[186] P. Sousa, N. F. Neves, and P. Veríssimo, "How Resilient are Distributed f Fault/Intrusion-Tolerant Systems?" in *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN)*, 2005, pp. 98–107.

[187] S. S. Craciunas and R. S. Oliver, "SMT-Based Task- and Network-Level Static Schedule Generation for Time-Triggered Networked Systems," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, ser. RTNS '14. New York, NY, USA: ACM, 2014, pp. 45–54.

[188] A. Minaeva, B. Akesson, Z. Hanzálek, and D. Dasari, "Time-Triggered Co-Scheduling of Computation and Communication with Jitter Requirements," *IEEE Transactions on Computers*, vol. 67, no. 1, pp. 115–129, 2018.

[189] R. Kirner and P. Puschner, "A Quantitative Analysis of Interfaces to Time-Triggered Communication Buses," *IEEE/ACM Transactions on Networking*, pp. 1–12, 2021.

[190] G. Bauer, H. Kopetz, and P. Puschner, "Assumption coverage under different failure modes in the time-triggered architecture," in *ETFA 2001. 8th International Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No.01TH8597)*, 2001, pp. 333–341 vol.1.

[191] G. Bauer, H. Kopetz, and W. Steiner, "The central guardian approach to enforce fault isolation in the time-triggered architecture," in *ISADS 2003*, 2003, pp. 37–44.

[192] M. Rausch, *FlexRay: Grundlagen, Funktionsweise, Anwendung*, ser. Hanser eLibrary. Hanser, 2008.

[193] J. Rushby, "Formal Verification of Transmission Window Timing for the Time-Triggered Architecture," 01 2001.

[194] H. Kopetz, "Event-triggered versus time-triggered real-time systems," in *Operating Systems of the 90s and Beyond*. Springer, 2005, pp. 86–101.

[195] R. Kirner and P. Puschner, "A Quantitative Analysis of Interfaces to Time-Triggered Communication Buses," *IEEE/ACM Transactions on Networking*, vol. 29, no. 4, pp. 1786–1797, 2021.

[196] H. Kopetz and R. Nossal, "Temporal firewalls in large distributed real-time systems," in *Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, 1997, pp. 310–315.

[197] H. Pfeifer, *Formal Verification of the TTP Group Membership Algorithm*. Boston, MA: Springer US, 2000, pp. 3–18.

[198] G. Bauer and M. Paulitsch, "An Investigation of Membership and Clique Avoidance in TTP/C," in *19th Symposium on Reliable Distributed Systems*, 2000, pp. 118–124.

[199] G. Bauer and H. Kopetz, "Transparent redundancy in the time-triggered architecture," in *DSN 2000*, 2000, pp. 5–13.

[200] H. Kopetz, R. Nossal, R. Hexel, A. Krüger, D. Millinger, R. Pallierer, C. Temple, and M. Krug, "Mode handling in the time-triggered architecture," in *IFAC Workshop on Distributed Computer Control Systems (DCCS97)*, Seoul, Korea, July 1997.

[201] J. Lundelius Welch and N. Lynch, "A new fault-tolerant algorithm for clock synchronization," *Information and Computation*, vol. 77, no. 1, pp. 1–36, 1988.

[202] A. Bouajjani and A. Merceron, "Parametric Verification of a Group Membership Algorithm," *CoRR*, vol. abs/cs/0505033, 2005.

[203] Z. Stojanovic, A. Dahanayake, and H. Sol, "Modelling for Distributed Network Systems: The Client-Server Model," in *Modeling and Design of Service-Oriented Architecture*. Springer, 2005, pp. 17–38.

[204] P. Puschner and R. Kirner, "Asynchronous vs. synchronous interfacing to time-triggered communication systems," *Journal of Systems Architecture*, vol. 103, p. 101690, 2020.

[205] N. Lynch, "A Hundred Impossibility Proofs for Distributed Computing," in *Proceedings of PODC '89*. New York, NY, USA: ACM, 1989, pp. 1–28.

[206] B. Gaujal and N. Navet, "Maximizing the Robustness of TDMA Networks with Applications to TTP/C," *Real-Time Systems*, vol. 31, no. 1-3, pp. 5–31, 2005.

[207] M. Ashjaei, P. Pedreiras, M. Behnam, L. Almeida, and T. Nolte, "Dynamic Reconfiguration in Multi-Hop Switched Ethernet Networks," *SIGBED Review*, vol. 11, no. 3, pp. 62–65, 2014.

[208] A. A. Tsonis and J. B. Elsner, "Chaos, strange attractors, and weather," *Bulletin of the American Meteorological Society*, vol. 70, no. 1, pp. 14–23, 1989.

[209] W. Schütz, "Fundamental issues in testing distributed real-time systems," *Real-Time Systems*, vol. 7, no. 2, pp. 129–157, 1994.

[210] C. Shelton and P. Koopman, "Improving system dependability with functional alternatives," in *DSN 2004*, 2004, p. 295.

[211] A. Loveless, R. Dreslinski, B. Kasikci, and L. T. X. Phan, "IGOR: Accelerating Byzantine Fault Tolerance for Real-Time Systems with Eager Execution," in *RTAS 2021*, 2021, pp. 360–373.

[212] A. Loveless, L. T. X. Phan, L. Erickson, R. Dreslinski, and B. Kasikci, "CrossTalk: Making Low-Latency Fault Tolerance Cheap by Exploiting Redundant Networks," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 5s, 2023.

[213] H. Kopetz, C. El-Salloum, B. Huber, and R. Obermaisser, "Periodic Finite-State Machines," in *ISORC 2007*, 2007, pp. 10–20.

[214] P. Pedreiras and L. Almeida, "The Flexible Time-Triggered (FTT) paradigm: an approach to QoS management in distributed real-time systems," in *International Parallel and Distributed Processing Symposium*, 2003, pp. 1–9.

[215] Y. Wang, E. Anceaume, F. Brasileiro, F. Greve, and M. Hurfin, "Solving the Group Priority Inversion Problem in a Timed Asynchronous System," *IEEE Trans. Comput.*, vol. 51, no. 8, pp. 900–915, 2002.

[216] S. Baruah and G. Fohler, "Certification-Cognizant Time-Triggered Scheduling of Mixed-Criticality Systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2011, pp. 3–12.

[217] L. Almeida, P. Pedreiras, and J. A. G. Fonseca, "The FTT-CAN protocol: why and how," *IEEE Transactions on Industrial Electronics*, vol. 49, no. 6, pp. 1189–1201, 2002.

[218] P. Pedreiras, L. Almeida, and P. Gai, "The FTT-Ethernet protocol: merging flexibility, timeliness and efficiency," in *14th Euromicro Conference on Real-Time Systems*, 2002, pp. 134–142.

[219] L. Zhao, F. He, E. Li, and J. Lu, "Comparison of Time Sensitive Networking (TSN) and TTEthernet," in *DASC 2018*, 2018, pp. 1–7.

[220] S. S. Craciunas, R. Serna Oliver, M. Chmelik, and W. Steiner, "Scheduling real-time communication in IEEE 802.1Qbv time sensitive networks," in *RTNS 2016*, 2016, pp. 183–192.

[221] J. Jiang, S. Jin, X. Li, K. Zhang, and B. Sun, "A Zero-Touch Dynamic Configuration Management Framework for Time-Sensitive Networking (TSN)," *Entropy*, vol. 27, no. 6, p. 584, 2025.

[222] L. Maile, K. Hielscher, and R. German, "Combining Static and Dynamic Traffic with Delay Guarantees in Time-Sensitive Networking," vol. 539, pp. 117–132, 2023.

[223] A. Gujarati, S. Bozhko, and B. B. Brandenburg, "Real-Time Replica Consistency over Ethernet with Reliability Bounds," in *2020 IEEE RTAS*, 2020, pp. 376–389.

[224] A. Naghavi and N. Navet, "Total Execution Order in Fault-Tolerant Real-Time Systems," in *Proceedings of the 32nd International Conference on Real-Time Networks and Systems*, ser. RTNS '24, 2025, pp. 12–24.

[225] T. Nikolic, G. Djordjevic, and M. Stojcev, "Simultaneous data transfers over peripheral bus using CDMA technique," in *26th International Conference on Microelectronics*, 2008, pp. 437–440.

[226] J. Mendonça, A. Bayrami Asl, F. Lucchetti, and M. Völp, "Confirmed-location group membership for intrusion-resilient cooperative maneuvers," in *IEEE VTC 2024-Spring*, 2024, pp. 01–07.

[227] M. Paulitsch and B. Hall, "Insights into the Sensitivity of the BRAIN (Braided Ring Availability Integrity Network)–On Platform Robustness in Extended Operation," in *DSN 2007*, 2007, pp. 154–163.

[228] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *IEEE RTAS*, 2013.

[229] G. Heiser and G. Klein, "The seL4 microkernel: An introduction," in *Trustworthy Systems*, 2012.

[230] J. Perez-Cerrolaza, P. Balbastre, E. Mezzetti, A. Crespo, J. González Eguiluz, M. Martin Acosta, and C.-A. Lefebvre, "Virtualization Technology for Dependable Embedded Systems: A Survey for Transportation and Industrial Domains," *ACM Comput. Surv.*, vol. 58, no. 4, 2025.

[231] G. Gala, C. Rodriguez, G. Monaco, J. Castillo, G. Fohler, V. Falico, and S. Tverdyshev, "Monitoring framework to support mixed-criticality applications on multi-core platforms," in *DSD 2022*, 2022, pp. 649–656.

[232] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Springer, 2019, pp. 1–20.

[233] H. Perez, J. J. Gutiérrez, S. Peiro, and A. Crespo, "Distributed architecture for developing mixed-criticality systems in multi-core platforms," *Journal of Systems and Software*, vol. 123, pp. 145–159, 2017.

[234] U. Steinberg and B. Kauer, “NOVA: A microhypervisor-based secure virtualization architecture,” in *EuroSys*, 2010.

[235] M. Masmano, Y. Valiente, P. Balbastre, I. Ripoll, A. Crespo, and J. J. Metge, “LithOS: an ARINC-653 guest operating for XtratuM,” in *12th Real-Time Linux Workshop*, 2010.

[236] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and Linux containers,” *IEEE ISPASS*, 2015.

[237] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, Omega, and Kubernetes,” in *ACM Queue*, 2016.

[238] M. Barletta, M. Cinque, L. De Simone, and R. Della Corte, “Criticality-aware monitoring and orchestration for containerized industry 4.0 environments,” *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 1, pp. 1–28, 2024.

[239] G. Monaco, G. Gala, and G. Fohler, “Extensions for shared resource orchestration in Kubernetes to support RT-cloud containers,” *arXiv preprint arXiv:2301.07479*, 2023.

[240] S. S. Craciunas, R. S. Oliver, and W. Steiner, “Container-based orchestration for mixed-criticality real-time systems,” in *IEEE ETFA*, 2019.

[241] S. Resch, “Composability for fail-safe safety-critical systems,” Ph.D. dissertation, TU Wien, 2014.

[242] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, “A survey of timing verification techniques for multi-core real-time systems,” in *Verimag Research Report TR-2018-9*, 2018.

[243] A. Agrawal, G. Fohler, J. Freitag, J. Nowotsch, S. Uhrig, and M. Paulitsch, “Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study,” in *ECRTS 2017*, 2017, pp. 2:1–2:22.

[244] L. Almeida and P. Pedreiras, “Scheduling within temporal partitions: response-time analysis and server design,” in *Proceedings of the 4th ACM International Conference on Embedded Software*, ser. EMSOFT ’04. New York, NY, USA: Association for Computing Machinery, 2004, pp. 95–103. [Online]. Available: https://doi.org/10.1145/1017753.1017772

[245] I. Shin and I. Lee, “Compositional real-time scheduling framework with periodic model,” *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, may 2008. [Online]. Available: https://doi.org/10.1145/1347375.1347383

[246] C. Meroni, S. S. Craciunas, A. Finzi, and P. Pop, “Mapping and integration of event- and time-triggered real-time tasks on partitioned multi-core systems,” in *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2023, pp. 1–8.

[247] T. Pop, P. Eles, and Z. Peng, “Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems,” in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, ser. CODES

'02. New York, NY, USA: Association for Computing Machinery, 2002, pp. 187–192. [Online]. Available: https://doi.org/10.1145/774789.774828

[248] A. Finzi, S. S. Craciunas, and M. Boyer, "Integrating sporadic events in time-triggered systems via affine envelope approximations," in *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2024, pp. 15–28.

[249] E. Bini, "Cutting the unnecessary deadlines in edf," in *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2019, pp. 1–8.

[250] M. Maggio, A. Hamann, E. Mayer-John, and D. Ziegenbein, "Control-System Stability Under Consecutive Deadline Misses Constraints," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Völp, Ed., vol. 165. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, pp. 21:1–21:24. [Online]. Available: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2020.21

[251] N. Vreman, A. Cervin, and M. Maggio, "Stability and Performance Analysis of Control Systems Subject to Bursts of Deadline Misses," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), B. B. Brandenburg, Ed., vol. 196. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 15:1–15:23. [Online]. Available: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2021.15

[252] S. Even, A. L. Selman, and Y. Yacobi, "The complexity of promise problems with applications to public-key cryptography," *Information and Control*, vol. 61, no. 2, pp. 159–173, 1984. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S001999588480056X

[253] L. Maia, M. I. Alkoudsi, and G. Fohler, "Ensuring End-to-End Latencies of Multi-Rate Cause-Effect Chains in Multi-Mode Systems Through Migration," in *Proceeding of ERTS 2026 on Embedded Real Time Systems*, ser. Proceeding of ERTS 2026 on Embedded Real Time Systems, Toulouse, France, Feb. 2026. [Online]. Available: https://hal.science/hal-05513753

[254] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-time systems*, vol. 30, no. 1, pp. 129–154, 2005.

[255] Gurobi Optimization, Inc., "Gurobi Optimizer 12.0 Documentation," https://docs.gurobi.com/current/, 2025, last accessed July 3, 2025.

[256] S. Bolusani, M. Besançon, and K. Bestuzheva, "The SCIP Optimization Suite 9.0," arXiv preprint arXiv:2402.17702, 2024. [Online]. Available: https://arxiv.org/abs/2402.17702

[257] N. Tamimi, "How fast is c++ compared to python?" https://docs.gurobi.com/current/, 2020, last accessed July 3, 2025.

[258] IBM Corporation. (2020) Compiled versus interpreted languages. Last accessed July 3, 2025. [Online]. Available: https://www.ibm.com/docs/en/zos-basic-skills?topic=zos-compiled-versus-interpreted-languages

[259] "Pypy documentation," https://doc.pypy.org/en/latest/, last accessed July 3, 2025.

[260] R. Stafford, "Random Vectors with Fixed Sum," https://www.mathworks.com/matlabcentral/fileexchange/9700-random-vectors-with-fixed-sum, 2006, last accessed July 3, 2025.

[261] P. Emberson, R. Stafford, and R. Davis, "A Taskset Generator for Experiments with Real-Time Task Sets," http://retis.sssup.it/waters2010/tools.php, 2010, includes Python implementation of randfixedsum. Released under a permissive license. Developed for WATERS 2010. Last accessed June 18, 2025.

[262] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmarks for free," in *Proceedings of the 6th International Workshop on Analysis Tools and Methodologies for Embedded Real-time Systems (WATERS)*, Jul. 2015. [Online]. Available: https://rtn.ecrts.org/forum/download/WATERS15_Real_World_Automotive_Benchmark_For_Free.pdf

[263] J. Goossens, E. Grolleau, and L. Cucu-Grosjean, "Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms," *Real-time systems*, vol. 52, no. 6, pp. 808–832, 2016.

[264] J. R. Pimentel and T. Sacristan, "A fault management protocol for TTP/C," in *IECON'01: 27th Annual Conference of the IEEE Industrial Electronics Society*, vol. 3, 2001, pp. 1800–1805.

[265] G. Keizer, ""Is Stuxnet the 'best' malware ever?"," Infoworld, 2010, last accessed: 24-05-2022. [Online]. Available: http://www.infoworld.com/article/2626009/malware/is-stuxnet-the--best--malware-ever-.html

[266] U. F. E. R. Commission, "Reliability Standards for Physical," RD14-6-000, March 2016.

[267] M. Völp, C.-J. Hamann, and H. Härtig, "Avoiding timing channels in fixed-priority schedulers," 2008, pp. 44–55.

[268] M. Völp, B. Engel, C.-J. Hamann, and H. Härtig, "On confidentiality-preserving real-time locking protocols," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 153–162.

[269] S. Mohan, M.-K. Yoon, R. Pellizzoni, and R. B. Bobba, "Integrating security constraints into fixed priority real-time schedulers," *Real-Time Systems*, vol. 52, pp. 644–674, 2016.

[270] M.-K. Yoon, S. Mohan, C.-Y. Chen, and L. Sha, "TaskShuffler: A Schedule Randomization Protocol for Obfuscation against Timing Inference Attacks in Real-Time Systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–12.

[271] J. C. Lisner, “Scheduling in a time-triggered protocol with dynamic arbitration,” in *IEEE International Symposium on Industrial Electronics (ISIE)*, vol. 4, 2005, pp. 1399–1404.

[272] K. Driscoll, B. Hall, and S. Varadarajan, “Maximizing fault tolerance in a low-swap data network,” in *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, 2012, pp. 7A2–1–7A2–16.

[273] C. J. Walter, P. Lincoln, and N. Suri, “Formally Verified On-Line Diagnosis,” *IEEE Transactions on Software Engineering*, vol. 23, pp. 684–721, 1997.

[274] L. Lamport and P. M. Melliar-Smith, “Synchronizing clocks in the presence of faults,” *J. ACM*, vol. 32, no. 1, p. 52–78, jan 1985. [Online]. Available: https://doi.org/10.1145/2455.2457

[275] F. B. Schneider, “Understanding Protocols for Byzantine Clock Synchronization,” Cornell University, USA, Tech. Rep., 1987.

[276] S. V. Kulhalli, M. I. Alkoudsi, and G. Fohler, “Exploring and exploiting synchrony limitations of time-triggered network-agnostic guardians,” in *2026 29th International Symposium on Real-Time Distributed Computing (ISORC)*, 2026, pp. 1–10.

[277] OpenSim Ltd, *OMNeT++ Simulation Manual*, https://omnetpp.org/documentation/, OpenSim Ltd.

[278] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro, “Analysis of operating system diversity for intrusion tolerance,” *Softw. Pract. Exper.*, vol. 44, no. 6, p. 735–770, jun 2014. [Online]. Available: https://doi.org/10.1002/spe.2180

[279] R. Pucella and F. B. Schneider, “Independence from obfuscation: A semantic framework for diversity,” in *19th IEEE Workshop on Computer Security Foundations*, 2006, pp. 230–241.

[280] J. R. Vacca, Ed., *Computer and Information Security Handbook*, 3rd ed. USA: Morgan Kaufmann, 2017.

[281] H. Bühler, A. Walz, and A. Sikora, “Benchmarking of symmetric cryptographic algorithms on a deeply embedded system,” *IFAC-PapersOnLine*, vol. 55, no. 4, pp. 266–271, 2022, 17th IFAC Conference on Programmable Devices and Embedded Systems PDES 2022 — Sarajevo, Bosnia and Herzegovina, 17-19 May 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896322003597

[282] Z. He, L. Wu, and X. Zhang, “High-speed pipeline design for hmac of sha-256 with masking scheme,” in *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, 2018, pp. 174–178.

Summary

Flexibility and Intrusion Resilience for Time-Triggered Real Time Systems in Open Safety-Critical Environments

The *Time-Triggered Architecture* (TTA) has long proven highly suitable for distributed safety-critical real-time systems by providing strong guarantees regarding extra-functional properties such as determinism, temporal predictability, replica determinism, fault containment, and composability. Traditionally, TTA-based systems operated under a closed-world assumption, where physical isolation served as the primary security perimeter and task sets, mainly originating from the periodic requirements of control loops, were fixed at design time. However, modern safety-critical systems are rapidly transitioning toward open, highly interconnected, and complex platforms. This evolution introduces the necessity for dynamically changing task sets, requiring event-driven workloads to safely coexist in a mixed-criticality fashion on shared hardware alongside fixed, high-criticality tasks. Additionally, this open connectivity exposes the system to intelligent adversaries over long mission durations.

The static nature of classical time-triggered schedules hinders operational flexibility when systems must adapt to varying environmental demands. Furthermore, the original fault model of the TTA fails to address persistent resource-exhaustion and time-domain attacks orchestrated by stealthy adversaries. This thesis examines the combined challenges of operational flexibility and intrusion resilience in open time-triggered systems. We propose a suite of novel scheduling mechanisms and architectural extensions that successfully accommodate event-driven workloads and mitigate malicious threats, all while preserving the extra-functional properties and safety guarantees required by high-assurance domains.

Chapter 1 - Introduction

This chapter establishes the context for the research presented in this thesis, examining the technological and economic trends driving the paradigm shift from isolated, federated architectures to open, integrated platforms. The core problem statement is defined

by highlighting four critical assumption gaps between the design of the classical TTA and modern operational realities:

- **Assumption gap 1 (The periodic workload model):** Forcing non-periodic activities into rigid periodic schedules inherently leads to wasteful resource overprovisioning.
- **Assumption gap 2 (The fixed task set):** Static configurations scale poorly in modern systems that require dynamically changing task sets and adaptive quality-of-service strategies.
- **Assumption gap 3 (The known WCET):** Deriving tight and trustworthy *Worst-Case Execution Time* (WCET) bounds for tasks on contemporary, heterogeneous multicore platforms is becoming increasingly intractable.
- **Assumption gap 4 (Non-adversarial environments):** The original fault hypothesis of the TTA considers only faults of accidental origin. Its dependability mechanisms thus fail to counteract intelligent adversaries capable of executing sophisticated time-domain and resource-exhaustion attacks over extended mission durations.

Arising from these gaps, the chapter formulates three central research questions (**RQ1–RQ3**) investigating how to accommodate dynamically changing task sets, efficiently schedule mixed workloads, and equip time-triggered architectures with intrusion-resilience mechanisms. We subsequently summarize the primary contributions of the thesis, organized into two overarching objectives: advancing scheduling flexibility without sacrificing the stringent extra-functional properties of the TTA, and enhancing intrusion resilience against malicious and persistent attacks. The chapter concludes by outlining the structural organization of the remainder of the thesis.

Chapter 2 - Background: Fundamental Concepts and Theory

This chapter provides an overview of the background concepts required to contextualize the contributions of this thesis. Reflecting the dual challenges of flexible time-triggered scheduling and intrusion resilience, the chapter is divided into two main sections:

- **Real-time systems theory:** This section reviews the origins and modeling of temporal requirements and then presents scheduling paradigms for meeting those requirements, together with analyses that verify feasibility under each scheduling method. This establishes the baseline required for the dynamic scheduling mechanisms and feasibility analyses developed in Chapter 4.

- **Distributed systems theory:** This section reviews the main concepts of dependability, security, and intrusion resilience. It then covers fundamental theoretical limits in distributed computing and the principles of fault tolerance, state machine replication, architectural hybridization, and rejuvenation, laying the theoretical foundation for the dependency analysis presented in Chapter 3 and the resilience mechanisms introduced in Chapter 5.

Chapter 3 - The Time-Triggered Architecture

This chapter focuses on the TTA, providing an overview of its system model, its node and network structure, and the real-time and dependability services it offers. We then introduce TTA's extra-functional properties from a hierarchical perspective that highlights their interrelationships. Building on this foundation, we present our dependency analysis, which classifies how these properties rely on specific TTA services and mechanisms, and discuss alternative solutions together with their trade-offs.

This analysis organizes the architectural dependencies into three distinct categories:

- **Fundamental**: The property cannot be achieved by any other means across all applications. This represents a true architectural requirement where alternatives do not exist or would fundamentally compromise the property.

- **Implementation-based**: The property can be realized through alternative approaches, potentially with different complexity, performance, or cost characteristics. These reflect TTA's specific design choices rather than fundamental requirements.

- **Application-dependent**: The property can be realized through alternative approaches, but the feasibility of these approaches depends on the timing requirements of the target application.

By demonstrating that the majority of these dependencies are implementation-based, the analysis reveals a substantial design space for architectural adaptation. This insight allows us to pinpoint exactly which design choices or constraints can be safely modified to introduce scheduling flexibility, ensuring that the stringent dependability guarantees demanded by high-assurance domains are not compromised.

Chapter 4 - Flexibility in Time-Triggered Scheduling

This chapter addresses the challenge of supporting dynamically evolving task sets by relaxing the rigid assumptions inherent in static offline scheduling. We introduce a

novel dynamic time-triggered scheduler together with runtime-optimized exact feasibility tests, demonstrating how these mechanisms enable the safe and efficient modification of task sets at runtime. Crucially, our approach preserves temporal guarantees and maintains the extra-functional properties of the TTA.

The primary contributions of this chapter include:

- **The Dynamic Task Set Scheduler (DTSS):** Representing our primary scheduling solution, DTSS resolves the fundamental tension between predictability and flexibility identified in the introduction. By demonstrating that slot-based execution does not require static offline allocation, DTSS enables runtime task admission, removal, and controlled skipping. It enforces strict execution budgets to prevent temporal interference from misbehaving tasks and supports dynamic operation granularity to reduce context-switching overhead.

- **Exact feasibility tests:** We develop exact feasibility tests to safely integrate aperiodic, sporadic, and periodic tasks alongside existing time-triggered task sets. For aperiodic tasks, we introduce the *Remaining Processor Capacity Analysis* (RPCA), which uses a minimal set of constraints to efficiently verify task set extension feasibility in polynomial time. For sporadic tasks, we analyze a state-of-the-art acceptance test, identify the analytical pessimism within it, and show how it can be resolved. Based on these insights, we design an exact test for extending time-triggered task sets with sporadic tasks and optimize it for runtime usage by relocating the majority of the computational effort to the design phase. This optimization leverages bounded utilization and offline pre-processing to reduce the runtime complexity of the online feasibility analysis from exponential to pseudo-polynomial.

- **Experimental evaluation:** The results of our experimental evaluations, utilizing both synthetic and real-world automotive benchmarks, confirm the superiority of our proposed mechanisms. Utilizing dynamic slot sizes in DTSS reduces scheduler invocations and context-switching overheads by an average of 62%, while decreasing the memory overhead associated with storing the look-ahead EDL schedule by 74% compared to closely related state-of-the-art methods. For admission control, while the aforementioned pessimistic test from previous work drops to a 12% schedulability success rate under high sporadic utilization, our runtime-optimized exact test maintains optimal schedulability and executes two to three orders of magnitude faster than previous work and our non-optimized variant, successfully transitioning runtime time-triggered task set extension with sporadic tasks from a theoretical concept into a practical tool for runtime deployment.

Chapter 5 - Intrusion Resilience in Time-Triggered Systems

This chapter addresses the transition from static fault tolerance to dynamic intrusion resilience in open, adversarial environments. While traditional dependability models assume that simultaneous faults are rare and independent, intelligent adversaries invalidate this assumption by silently exploiting vulnerabilities over extended mission durations to compromise a majority quorum.

This chapter presents three main contributions to counteract these persistent threats:

- **Vulnerability analysis:** We analyze the susceptibility of TTA nodes to resource-exhaustion attacks, and we show that an adversary who compromises frame-relaying components such as central guardians can deliberately disrupt synchrony among nodes, thereby undermining the temporal coordination on which the architecture relies. Furthermore, we show that by violating the design assumptions that support TTA's fault-tolerance, an adversary can circumvent TTA's mechanisms, cause errors to propagate in the value domain, and ultimately trigger system-level failures.

- **Replica-aware rejuvenation service:** We develop a replica-aware rejuvenation service that constrains the vulnerability window of individual nodes, thereby counteracting stealthy and long-term intrusions. The service leverages the existing sparse global time base and membership information, and coordinates both proactive and reactive recovery without introducing single points of failure. By restoring nodes to a secure state more rapidly than an adversary can compromise them, the service ensures node-exhaustion safety and the continued provisioning of the replicated service.

- **SyncGuard:** We analyze the limitations of network-agnostic guardians. While these guardians are highly desirable because their independence from underlying network protocols and topologies significantly reduces design and certification efforts, we establish an impossibility result showing that their safe resynchronization cannot be guaranteed when the nodes they monitor behave maliciously. We overcome this limitation by introducing a distributed wormhole of trustworthy network-agnostic guardians, which not only control nodes' access to the network but also coordinate and enforce their proactive and reactive rejuvenation. We refer to this improved network-agnostic guardian as SyncGuard. By enforcing rejuvenation and controlling network access, SyncGuard preserves system synchrony and maintains collision-free time-triggered communication across heterogeneous networks, thereby ensuring dependable operation even in adversarial environments.

Chapter 6 - Conclusion and Future Directions

This chapter summarizes the main contributions of this thesis and outlines compelling directions for future work.

Zusammenfassung

Flexible und intrusionsresiliente zeitgesteuerte Echtzeitsysteme für offene sicherheitskritische Umgebungen

Die *Time-Triggered Architecture* (TTA) hat sich für verteilte sicherheitskritische Echtzeitsysteme längst als äußerst geeignet erwiesen, da sie starke Garantien bezüglich extrafunktionaler Eigenschaften wie Determinismus, zeitlicher Vorhersagbarkeit, Replika-Determinismus, Fehlereingrenzung und Komponierbarkeit bietet. Traditionell operierten TTA-basierte Systeme unter der Annahme einer geschlossenen Welt (Closed-World-Annahme), bei der physische Isolation als primäre Sicherheitsgrenze diente und Task Sets, die hauptsächlich aus den periodischen Anforderungen von Regelkreisen stammten, zur Entwurfszeit festgeschrieben waren. Moderne sicherheitskritische Systeme entwickeln sich jedoch rasch in Richtung offener, stark vernetzter und komplexer Plattformen. Diese Entwicklung macht dynamisch wechselnde Task Sets erforderlich, sodass ereignisgesteuerte Workloads unterschiedlicher Kritikalität (Mixed-Criticality) sicher auf gemeinsam genutzter Hardware neben festgelegten, hochkritischen Tasks koexistieren können. Darüber hinaus setzt diese offene Konnektivität das System über lange Missionsdauern hinweg intelligenten Angreifern aus.

Die statische Natur klassischer zeitgesteuerter Schedules behindert die operative Flexibilität, wenn sich Systeme an wechselnde Umgebungsanforderungen anpassen müssen. Darüber hinaus wird das ursprüngliche Fehlermodell der TTA anhaltenden Ressourcenerschöpfungs- und Zeitbereichsangriffen durch verdeckt operierende Angreifer nicht gerecht. Diese Dissertation untersucht die kombinierten Herausforderungen von operativer Flexibilität und Intrusionsresilienz in offenen zeitgesteuerten Systemen. Wir schlagen eine Reihe neuartiger Scheduling-Mechanismen und architektonischer Erweiterungen vor, die ereignisgesteuerte Workloads erfolgreich integrieren und böswillige Bedrohungen abwehren, während sie gleichzeitig die von hochsicheren Domänen geforderten extrafunktionalen Eigenschaften und Sicherheitsgarantien aufrechterhalten.

Kapitel 1 - Einleitung

Dieses Kapitel stellt den Kontext für die in dieser Arbeit präsentierte Forschung her und untersucht die technologischen und wirtschaftlichen Trends, die den Paradigmenwechsel von isolierten, föderierten Architekturen hin zu offenen, integrierten Plattformen vorantreiben. Die zentrale Problemstellung wird definiert, indem vier kritische Annahmelücken (Assumption Gaps) zwischen dem Design der klassischen TTA und modernen operativen Realitäten aufgezeigt werden:

- **Annahmelücke 1 (Das periodische Workload-Modell):** Das Aufzwingen nichtperiodischer Aktivitäten in starre periodische Schedules führt unweigerlich zu einer verschwenderischen Ressourcenüberprovisionierung.
- **Annahmelücke 2 (Das statische Task Set):** Statische Konfigurationen skalieren schlecht in modernen Systemen, die dynamisch wechselnde Task Sets und adaptive Quality-of-Service-Strategien erfordern.
- **Annahmelücke 3 (Die bekannte WCET):** Die Ableitung enger und verlässlicher *Worst-Case Execution Time* (WCET), Schranken für Tasks auf zeitgemäßen, heterogenen Multicore-Plattformen wird zunehmend unlösbar.
- **Annahmelücke 4 (Nicht-adversarielle Umgebungen):** Die ursprüngliche Fehlerhypothese der TTA berücksichtigt nur Fehler zufälligen Ursprungs. Ihre Zuverlässigkeitsmechanismen versagen daher bei der Abwehr intelligenter Angreifer, die in der Lage sind, komplexe Zeitbereichs- und Ressourcenerschöpfungsangriffe über längere Missionsdauern hinweg auszuführen.

Ausgehend von diesen Lücken formuliert das Kapitel drei zentrale Forschungsfragen (**RQ1**–**RQ3**), die untersuchen, wie dynamisch wechselnde Task Sets integriert, gemischte Workloads effizient geplant und zeitgesteuerte Architekturen mit intrusionsresilienten Mechanismen ausgestattet werden können. Anschließend fassen wir die Hauptbeiträge der Arbeit zusammen, die in zwei übergeordnete Ziele unterteilt sind: die Weiterentwicklung der Scheduling-Flexibilität ohne Einbußen bei den strengen extrafunktionalen Eigenschaften der TTA sowie die Verbesserung der Intrusionsresilienz gegen böswillige und anhaltende Angriffe. Das Kapitel schließt mit einem Überblick über die strukturelle Gliederung der restlichen Arbeit.

Kapitel 2 - Grundlagen: Grundlegende Konzepte und Theorie

Dieses Kapitel bietet einen Überblick über die Hintergrundkonzepte, die erforderlich sind, um die Beiträge dieser Arbeit in einen Kontext zu setzen. Die zweifachen Heraus-

forderungen von flexiblem zeitgesteuerten Scheduling und Intrusionsresilienz widerspiegelnd, ist das Kapitel in zwei Hauptabschnitte unterteilt:

- **Echtzeitsystemtheorie:** Dieser Abschnitt behandelt die Ursprünge und Modellierung zeitlicher Anforderungen und stellt dann Scheduling-Paradigmen zur Erfüllung dieser Anforderungen vor, zusammen mit Analysen, die die Machbarkeit unter jeder Scheduling-Methode verifizieren. Dies liefert die theoretische Basis, die für die in Kapitel 4 entwickelten dynamischen Scheduling-Mechanismen und Machbarkeitsanalysen erforderlich ist.

- **Theorie verteilter Systeme:** Dieser Abschnitt bespricht die Hauptkonzepte von Verlässlichkeit (Dependability), Sicherheit und Intrusionsresilienz. Anschließend werden grundlegende theoretische Grenzen in der verteilten Datenverarbeitung sowie die Prinzipien von Fehlertoleranz, Zustandsautomaten-Replikation (State Machine Replication), architektonischer Hybridisierung und Rejuvenation behandelt, was die theoretische Grundlage für die in Kapitel 3 präsentierte Abhängigkeitsanalyse und die in Kapitel 5 eingeführten Resilienzmechanismen legt.

Kapitel 3 - Die Time-Triggered Architecture

Dieses Kapitel konzentriert sich auf die TTA und bietet einen Überblick über ihr Systemmodell, ihre Knoten- und Netzwerkstruktur sowie die Echtzeit- und Zuverlässigkeitsdienste, die sie anbietet. Wir stellen dann die extrafunktionalen Eigenschaften der TTA aus einer hierarchischen Perspektive vor, die ihre Wechselbeziehungen hervorhebt. Darauf aufbauend präsentieren wir unsere Abhängigkeitsanalyse, die klassifiziert, wie diese Eigenschaften auf spezifischen TTA-Diensten und -Mechanismen beruhen, und diskutieren alternative Lösungen zusammen mit ihren Trade-offs.

Diese Analyse gliedert die architektonischen Abhängigkeiten in drei unterschiedliche Kategorien:

- **Fundamental**: Die Eigenschaft kann anwendungsübergreifend auf keine andere Weise erreicht werden. Dies stellt eine echte architektonische Anforderung dar, für die es keine Alternativen gibt oder die die Eigenschaft grundlegend gefährden würden.

- **Implementierungsbasiert**: Die Eigenschaft kann durch alternative Ansätze realisiert werden, potenziell mit unterschiedlichen Komplexitäts-, Leistungs- oder Kostenmerkmalen. Diese spiegeln eher spezifische Designentscheidungen der TTA wider als fundamentale Anforderungen.

- **Anwendungsabhängig**: Die Eigenschaft kann durch alternative Ansätze realisiert werden, aber die Machbarkeit dieser Ansätze hängt von den zeitlichen Anforderungen der Zielanwendung ab.

Indem demonstriert wird, dass die Mehrheit dieser Abhängigkeiten implementierungsbasiert ist, offenbart die Analyse einen erheblichen Gestaltungsspielraum für architektonische Anpassungen. Diese Erkenntnis ermöglicht es uns, genau zu bestimmen, welche Designentscheidungen oder Einschränkungen sicher modifiziert werden können, um Scheduling-Flexibilität einzuführen, und stellt so sicher, dass die strengen Zuverlässigkeitsgarantien, die von hochsicheren Domänen gefordert werden, nicht beeinträchtigt werden.

Kapitel 4 - Flexibilität im zeitgesteuerten Scheduling

Dieses Kapitel befasst sich mit der Herausforderung, dynamisch evolvierende Task Sets zu unterstützen, indem die in statischem Offline-Scheduling verankerten starren Annahmen gelockert werden. Wir stellen einen neuartigen dynamischen zeitgesteuerten Scheduler zusammen mit laufzeitoptimierten, exakten Machbarkeitsanalysen vor und demonstrieren, wie diese Mechanismen eine sichere und effiziente Modifikation von Task Sets zur Laufzeit ermöglichen. Entscheidend dabei ist, dass unser Ansatz zeitliche Garantien bewahrt und die extrafunktionalen Eigenschaften der TTA aufrechterhält.

Die Hauptbeiträge dieses Kapitels umfassen:

- **Der Dynamic Task Set Scheduler (DTSS):** Als unsere primäre Scheduling-Lösung löst DTSS das in der Einleitung identifizierte fundamentale Spannungsfeld zwischen Vorhersagbarkeit und Anpassungsfähigkeit auf. Indem bewiesen wird, dass eine slotbasierte Ausführung keine statische Offline-Zuweisung erfordert, ermöglicht DTSS die Zulassung, Entfernung und das kontrollierte Überspringen von Tasks zur Laufzeit. Er setzt strenge Ausführungsbudgets durch, um zeitliche Interferenzen durch sich fehlerhaft verhaltende Tasks zu verhindern, und unterstützt dynamische Slot-Größen, um den Kontextwechsel-Overhead zu reduzieren.

- **Exakte Machbarkeitsanalysen:** Wir entwickeln exakte Machbarkeitsanalysen, um aperiodische, sporadische und periodische Tasks sicher in bestehende zeitgesteuerte Task Sets zu integrieren. Für aperiodische Tasks führen wir die *Remaining Processor Capacity Analysis* (RPCA) ein, die eine minimale Menge von Einschränkungen verwendet, um die Machbarkeit von Task Set-Erweiterungen effizient in polynomieller Zeit zu verifizieren. Für sporadische Tasks analysieren wir einen hochmodernen Akzeptanztest, identifizieren den darin enthaltenen analytischen Pessimismus und zeigen, wie dieser behoben werden kann. Basierend

auf diesen Erkenntnissen entwerfen wir einen exakten Test zur Erweiterung zeitgesteuerter Task Sets um sporadische Tasks und optimieren ihn für die Laufzeitnutzung. Diese Optimierung nutzt eine beschränkte Systemauslastung sowie Offline-Vorverarbeitung, um die Laufzeitkomplexität der Online-Machbarkeitsanalyse von exponentiell auf pseudo-polynomiell zu reduzieren.

- **Experimentelle Evaluation:** Die Ergebnisse unserer experimentellen Evaluationen, bei denen sowohl synthetische als auch reale automobile Benchmarks herangezogen wurden, bestätigen die Überlegenheit unserer vorgeschlagenen Mechanismen. Die Verwendung dynamischer Slot-Größen in DTSS reduziert Scheduler-Aufrufe und Kontextwechsel-Overheads um durchschnittlich 62%, während der Speicher-Overhead, der mit der Speicherung des vorausschauenden EDL-Schedules verbunden ist, im Vergleich zu eng verwandten State-of-the-Art-Methoden um 74% sinkt. Bei der Zulassungskontrolle fällt der zuvor erwähnte pessimistische Test aus früheren Arbeiten bei hoher sporadischer Auslastung auf eine Schedulability-Erfolgsrate von 12% ab, während unser laufzeitoptimierter exakter Test die optimale Planbarkeit aufrechterhält und zwei bis drei Größenordnungen schneller ausgeführt wird als frühere Arbeiten und unsere nicht-optimierte Variante. Dies überführt die zeitgesteuerte Task Set-Erweiterung zur Laufzeit mit sporadischen Tasks erfolgreich von einem theoretischen Konzept in ein praktisches Werkzeug für den Laufzeiteinsatz.

Kapitel 5 - Intrusionsresilienz in zeitgesteuerten Systemen

Dieses Kapitel behandelt den Übergang von statischer Fehlertoleranz zu dynamischer Intrusionsresilienz in offenen, adversariellen Umgebungen. Während traditionelle Zuverlässigkeitsmodelle davon ausgehen, dass gleichzeitige Fehler selten und unabhängig voneinander auftreten, entkräften intelligente Angreifer diese Annahme, indem sie lautlos Schwachstellen über lange Missionsdauern hinweg ausnutzen, um ein Mehrheitsquorum zu kompromittieren.

Das Kapitel präsentiert die folgenden drei Hauptbeiträge, um diesen anhaltenden Bedrohungen entgegenzuwirken:

- **Schwachstellenanalyse:** Wir analysieren die Anfälligkeit von TTA-Knoten für Ressourcenerschöpfungsangriffe und zeigen, dass ein Angreifer, der Nachrichten weiterleitende Komponenten wie zentrale Guardians kompromittiert, absichtlich die Synchronität zwischen Knoten stören und damit die zeitliche Koordination untergraben kann, auf die sich die Architektur stützt. Darüber hinaus zeigen wir, dass der Angreifer durch die Verletzung der Designannahmen, die die Fehlertol-

eranz der TTA unterstützen, diese Schutzmechanismen umgehen, Fehler in den Wertebereich propagieren und letztendlich systemweite Ausfälle auslösen kann.

- **Replika-bewusster Rejuvenation-Dienst:** Wir entwickeln einen Replika-bewussten Rejuvenation-Dienst, der das Verwundbarkeitsfenster einzelner Knoten einschränkt und so verdeckten und langfristigen Intrusionen entgegenwirkt. Der Dienst nutzt die bestehende spärliche globale Zeitbasis sowie Membership-Informationen und koordiniert sowohl die proaktive als auch die reaktive Wiederherstellung, ohne Single Points of Failure einzuführen. Indem Knoten schneller in einen sicheren Zustand zurückversetzt werden, als ein Angreifer sie kompromittieren kann, gewährleistet der Dienst die Sicherheit vor Knotenerschöpfung und die kontinuierliche Bereitstellung des replizierten Dienstes.
- **SyncGuard:** Wir analysieren die Einschränkungen netzwerkagnostischer Guardians. Obwohl diese Guardians höchst wünschenswert sind, da ihre Unabhängigkeit von zugrunde liegenden Netzwerkprotokollen und -topologien den Design- und Zertifizierungsaufwand erheblich reduziert, stellen wir ein Unmöglichkeitsresultat auf, das zeigt, dass ihre sichere Resynchronisation nicht garantiert werden kann, wenn sich die von ihnen überwachten Knoten bösartig verhalten. Wir überwinden diese Einschränkung, indem wir ein verteiltes Wormhole aus vertrauenswürdigen lokalen, netzwerkagnostischen Guardians einführen, die nicht nur den Netzwerkzugriff der Knoten kontrollieren, sondern auch deren proaktive und reaktive Rejuvenation koordinieren und erzwingen. Wir bezeichnen diesen verbesserten netzwerkagnostischen Guardian als SyncGuard. Durch die Erzwingung von Rejuvenation und die Kontrolle der Kommunikation über heterogene Netzwerke hinweg bewahrt SyncGuard die Systemsynchronität und hält eine kollisionsfreie zeitgesteuerte Kommunikation aufrecht, wodurch ein zuverlässiger Betrieb selbst in adversariellen Umgebungen gewährleistet wird.

Kapitel 6 - Fazit und zukünftige Forschungsrichtungen

Dieses Kapitel fasst die Hauptbeiträge dieser Arbeit zusammen und skizziert vielversprechende Richtungen für zukünftige Arbeiten.

Mohammad Ibrahim Alkoudsi

Real-Time Systems Researcher

RPTU Kaiserslautern, Postfach 3049, 67663 Kaiserslautern, Germany
alkoudsi@rptu.com | ORCID: 0000-0002-1787-3284

Education

2020 – 2026 **Ph.D. in Electrical and Computer Engineering**
RPTU Kaiserslautern-Landau, Germany
Thesis: *Flexibility and Intrusion Resilience for Time-Triggered Real-Time Systems in Open Safety-Critical Environments*

2017 – 2020 **M.Sc. European Master in Embedded Computing Systems**
TU Kaiserslautern (Germany) & NTNU (Norway)
GPA: 1.3 | Thesis: *Control Flow Graph Optimizations for Property Driven Design Methodology*

2011 – 2016 **B.Sc. Electronics and Communications Engineering**
Damascus University, Syria
GPA: 83% | Special Merit Award for Best Graduate Student in the Department (2016)

Research Experience

2020 – Present **Research Assistant**
RPTU Kaiserslautern-Landau (Chair of Real-Time Systems)
- Investigating flexibility and resilience in distributed time-triggered real-time systems.
- Developing combined online and offline scheduling methodologies.
- Supervising and mentoring graduate students throughout their master's thesis research.

2019 – 2020 **Student Research Assistant (HiWi)**
TU Kaiserslautern (Chair of Electronic Design Automation)
- Developed software for the Design from SystemC Abstract Models (DeSCAM) tool.
- Created an extension plugin for the Atom text editor to support the DeSCAM tool workflow.

2019 – 2020 **Student Research Assistant (HiWi)**
Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI)
- Developed Zephyr RTOS embedded applications and validated hardware for the BIONIC IoT project.
- Maintained and extended a KiCAD hardware components library.

Teaching Experience

2020 – Present **Lecturer & Teaching Assistant**
RPTU Kaiserslautern-Landau (Chair of Real-Time Systems)
- **Lectures:** Dependability and multiprocessor scheduling (Real-Time Systems 1); Time-triggered architecture (Real-Time Systems 2).
- **Exercises:** System startup, processes and threads, security, memory management, clock synchronization, multiprocessor scheduling, aperiodic task scheduling.

Publications

- **M. I. Alkoudsi**, M. Völp, and G. Fohler, "Dependency Analysis of the Extra-Functional Properties of the Time-Triggered Architecture," *IEEE Access*, accepted for publication, 2026.
- S. V. Kulhalli, **M. I. Alkoudsi**, and G. Fohler, "Exploring and Exploiting Synchrony Limitations of Time-Triggered Network-Agnostic Guardians," *29th International Symposium on Real-Time Distributed Computing (ISORC)*, 2026.
- L. Maia, **M. I. Alkoudsi**, and G. Fohler, "Ensuring End-to-End Latencies of Multi-Rate Cause-Effect Chains in Multi-Mode Systems Through Migration," *13th European Congress of Embedded Real Time Systems (ERTS)*, 2026.
- **M. I. Alkoudsi**, S. Baruah, P. Ekberg, G. Fohler, J. Goossens, and M. Moslehi, "Extending Periodic Task Sets with Sporadic Tasks: Computational Complexity and Exact Feasibility Tests," *33rd Int. Conf. on Real-Time Networks and Systems (RTNS)*, 2025.
- **M. I. Alkoudsi**, D. Isovic, and G. Fohler, "Revisiting Slot-Shifting's Offline Acceptance Test for Sporadic Tasks: A Technical Note," *Leibniz Transactions on Embedded Systems (LITES)*, vol. 10, no. 1, pp. 4:1–4:6, 2025.
- M. Völp, **M. I. Alkoudsi**, A. Bayrami Asl, K. Krüger, J. R. M. da Neto, and G. Fohler, "Defending Event-Triggered Systems Against Out-of-Envelope Environments," *2nd Workshop on Real-Time Autonomous Systems Security (RT-AutoSec)*, 2025.
- **M. I. Alkoudsi** and G. Fohler, "Scheduling Dynamic Task-Sets in Time-Triggered Real-Time Systems," *32nd Int. Conf. on Real-Time Networks and Systems (RTNS)*, 2024.
- G. Gala, T. Unte, L. Maia, J. Kühbacher, I. Kadusale, **M. I. Alkoudsi**, G. Fohler, and S. Altmeyer, "Safety-Critical Edge Robotics Architecture with Bounded End-to-End Latency," *Real-Time Cloud (RT-Cloud) Workshop (co-hosted with 36th ECRTS)*, 2024.
- **M. I. Alkoudsi**, G. Fohler, and M. Völp, "A Network-Agnostic Approach to Enforcing Collision-Free Time-Triggered Communication," *28th Pacific Rim Int. Symposium on Dependable Computing (PRDC)*, 2023.
- **M. I. Alkoudsi**, G. Fohler, and M. Völp, "Tolerating Resource Exhaustion Attacks in the Time-Triggered Architecture," *XII Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2022.

Honors & Awards

2025	**Best Student Paper Award** – *RTNS Conference*
2024, 2025	**Outstanding Paper Award** – *RTNS Conference* (Two-time recipient)
2017	**Erasmus+ Scholarship** – *European Master in Embedded Computing Systems (EMECS)*
2013 – 2016	**Bassel Award for Academic Excellence** – *Damascus University* (Three-time recipient)

Technical Skills & Languages

Programming	C++, C#, Python, JavaScript
Software	Zephyr, FreeRTOS, Git, Unity3d
Languages	Arabic (Native), English (Fluent / IELTS: 7), German (Intermediate / B2.2)