FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Composability for Fail-Safe Safety-Critical Systems

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht von

Stefan Resch
Matrikelnummer 0425306

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao. Univ. Prof. Dipl.-Ing. Dr.techn. Andreas Steininger

Diese Dissertation haben begutachtet:

(Ao. Univ. Prof. Dipl.-Ing. Dr.techn. Andreas Steininger)	(Univ. Prof. Dipl.-Ing. Dr.techn. Wilfried Elmenreich)

Wien, 03.12.2014

(Stefan Resch)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Composability for Fail-Safe Safety-Critical Systems

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Stefan Resch
Registration Number 0425306

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao. Univ. Prof. Dipl.-Ing. Dr.techn. Andreas Steininger

The dissertation has been reviewed by:

(Ao. Univ. Prof. Dipl.-Ing. Dr.techn. Andreas Steininger)	(Univ. Prof. Dipl.-Ing. Dr.techn. Wilfried Elmenreich)

Wien, 03.12.2014

(Stefan Resch)

Erklärung zur Verfassung der Arbeit

Stefan Resch
Koberweingasse 2/37, 1150 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

Above all, I want to thank Andreas Steininger, Christoph Scherrer and Heinz Kantz for the opportunity to pursue my PhD studies and their encouragement throughout. To Andreas, in particular, for his invaluable support as supervisor and for always pointing me in the right direction. Also, to Christoph for constantly challenging me and directing my focus to the practical relevance of this work. And to Heinz for always providing insightful feedback and a fresh perspective.

I would also like to acknowledge Peter Tummeltshammer for the numerous fruitful discussions and much appreciated comments. Many thanks go to Michael Paulitsch for broadening my horizons by highlighting new aspects to the topic and to Jaime de Oliveira for initially introducing me to the world of virtualization.

Furthermore, thank you to my colleagues at Thales Austria who gave me valuable pointers and support, moral or otherwise, during this process: Bernhard Kaindl, Johannes Mach, Werner Spieß, Rossen Zlatev, Oscar Medina, Mike Rumpler, Nikolaus König, Werner König, Peter Metzner, Gerhard Wipplinger, Klaus Reichl, Christoph Sünder, Christoph Plattner, Frank Riechert and Andreas Messner.

A special thank you must go to all my family for their tremendous support and understanding during this time. To my friends, thank you for always providing diversions when most needed.

Finally, I am most grateful for Cristina's encouragement, patience and help from beginning to end. I could not have completed this thesis without you.

Part of this research was funded by the ARTEMIS Joint Undertaking (nSafeCer, Grant Agreement number 295373), and the Austrian partners' national funding agency Austrian Research Promotion Agency (FFG) on behalf of the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT).

Abstract

Safety-critical systems must be carefully designed, developed and maintained in order to ensure that the threats posed by such systems are acceptably low. Certification demonstrates that these systems are fit for use. The methods of certification are applied to complete systems according to the applicable industrial standards. If parts of such a system change, substantial effort is necessary for re-certification, since the certification processes have to be repeated for the entire system.

Composability and mixed-criticality are strategies meant to support the integration and ease certification of safety-critical systems as sub-systems on one common platform, without affecting the safety or availability of the individual sub-systems. The introduction of mechanisms in order to achieve composability and mixed-criticality requires an additional layer in the architecture, responsible for the sharing of resources. This strongly affects sub-systems with strict timing requirements, such as triple-modular-redundant applications, which are widely used for fault-tolerant safety-critical computation.

This thesis investigates the requirements for achieving composability and mixed-criticality. It subsequently identifies solutions suitable for controlling the newly introduced effects. An appropriate system model and the metrics for the applications' performance are defined in order to analyze the properties of the proposed solutions. Based on the system model and the analysis results a contract concept is introduced, which allows the specification of applications, platforms and integrated systems based on provided and required resources. The validity of the analysis is evaluated with a prototype and simulation.

The results show that an out-of-the-box solution which guarantees the technical separation between applications with fast reaction time requirements is only feasible when executing at most one application per CPU-core for single and multi-core CPUs. Only when accepting changes in the architecture, applications or the applications' synchronization mechanisms, are other solutions available.

Kurzfassung

Sicherheitskritische Systeme müssen sorgfältig entworfen, entwickelt und gewartet werden, um zu gewährleisten, dass von ihnen kein Risiko ausgeht. Die Zertifizierung eines Systems zeigt, dass es geeignet für den Einsatz ist. Die Methoden der Zertifizierung werden für ganze Systeme gemäß den relevanten Industriestandards angewendet. Sollte ein Teil eines solchen Systems geändert werden, so ist erheblicher Aufwand für die Rezertifizierung nötig, da die Zertifizierungsprozesse für das gesamte System wiederholt werden müssen.

Composability und Mixed-Criticality sind Strategien, welche die Integration und Zertifizierung von sicherheitskritischen Systemen als Subsysteme auf einer gemeinsamen Plattform unterstützen, ohne die Sicherheit oder Verfügbarkeit der einzelnen Subsysteme zu beeinträchtigen. Die Einführung von Mechanismen um Composability und Mixed-Criticality zu erreichen, erfordert einen zusätzlichen Layer in der Architektur, welcher die gemeinsamen Ressourcen verwaltet. Ein solcher Layer wiederum beeinflusst Subsysteme, welche strikte zeitliche Kriterien haben, wie beispielsweise dreifach redundante Applikationen. Dreifach Redundanz ist eine weit verbreitete Technik für sicherheitskritische Anwendungen.

Diese Arbeit untersucht die Voraussetzungen um Composability und Mixed-Criticality zu erreichen. Daraus folgend werden mögliche Lösungen erarbeitet, welche den Einfluss des neuen Layers begrenzen. Ein dazugehöriges Systemmodell sowie Metriken zur Messung des Applikationsverhalten werden definiert, um die Eigenschaften der vorgeschlagenen Lösungen zu analysieren. Auf Basis des Systemmodells und der Ergebnisse der Analyse wird ein Vertragskonzept vorgestellt, mit welchem Applikationen, Plattformen und integrierte Systeme anhand von verfügbaren und erforderlichen Ressourcen spezifiziert werden können. Die Ergebnisse der Analyse werden mit Hilfe eines Prototyps und Simulationen überprüft.

Die Ergebnisse zeigen, dass es eine direkte Lösung gibt, die die technische Separierung zwischen Applikationen mit kurzen Antwortzeitanforderungen garantiert. Maximal eine sicherheitskritische Applikation darf pro CPU-Core für Single- und Multicore CPUs integriert werden. Nur wenn man Änderungen in der Architektur, den Applikationen oder dem Synchronisationsmechanismus der redundanten Applikationen zulässt, kann man auch von anderen Lösungen wählen.

List of Abbreviations

AP . . . . . Application
CBSE . . . . . Component-Based Software Engineering
CM . . . . . Communication Media
COTS . . . . . Commercial Off-The-Shelf
DMA . . . . . Direct Memory Access
EDF . . . . . Earliest Deadline First
FMEA . . . . . Failure Mode and Effects Analysis
FS . . . . . Function-Set
GSN . . . . . Goal Structuring Notation
HMI . . . . . Human-Machine Interface
IE . . . . . Integration Environment
IMA . . . . . Integrated Modular Avionics
MCH . . . . . Monitoring and Communication Handler
MILS . . . . . Multiple Independent Levels of Security
MMU . . . . . Memory Management Unit
NIC . . . . . Network Interface Controller
PH . . . . . TSM Protocol Handler
RM . . . . . Rate-Monotonic
RTC . . . . . Real-Time Clock
RTOS . . . . . Real-Time Operating System
SEU . . . . . Single Event Upset
TMR . . . . . Triple Modular Redundancy
TSM . . . . . Token Synchronized Messages
TTA . . . . . Time Triggered Architecture

Contents

CHAPTER 1

Introduction

We use services controlled by computers and software in our everyday life without even noticing or being aware of most of them [71]. Some services are used for convenience, such as vending machines, while others provide the foundation of our way of life, e.g. transportation. In the end, people develop and maintain these services with the intention of having people using them.

This thesis aims at supporting those responsible for constructing and providing such essential services.

1.1 Motivation

Failure of a safety-critical system can result in harm to humans and the environment. To ensure that the resulting threats posed by such a safety-critical system are acceptably low, it has to be certified according to the applicable industrial standards. These standards define processes in order to classify systems in levels of criticality with respect to the potential damage a failure of the system could cause. SIL, ASIL and DAL are examples of classification schemes in standards of the railway, automotive and avionics domain. These levels define different processes and methods to be followed during a system's lifetime in order to keep its probability of failure acceptably low. Here, SIL4, ASIL D and DAL - Level A require the most effort, as they apply to systems where failures can have catastrophic consequences.

The methods of certification are applied to the whole system and the system is subsequently certified in its entirety. Should parts of the system change, substantial effort is necessary for re-certification. Either the certification process is repeated for the whole system, or, with a detailed impact analysis all system parts affected by the changes are determined, after which the certification process is performed on the corresponding parts. This system-centric view logically originates from the fact that the hazard is posed by the complete system. This results in systems of systems with federated architectures, possibly utilizing only a small fraction of the underlying hardware's performance.

Composability and *mixed-criticality* promise to reduce this certification effort and increase the level of integration together with hardware utilization. This can be achieved by merging

such systems of systems on one platform, thereby creating one integrated system consisting of sub-systems, as well as by splitting existing systems in different sub-systems, which are then assigned their specific criticality level. These sub-systems are certified independently according to their criticality level, and the overall system is certified based on the evidence provided by the certification process of the sub-systems. For this approach it is crucial, that the independence of the individual sub-systems is guaranteed. Composability addresses the composition and certification of such systems and corresponding sub-systems, whereas mixed-criticality is concerned with the integration of sub-systems with different levels of criticality.

Increasing computational power and chips incorporating several CPUs enable the tight integration of numerous functions of a system in the first place. The aim of composability and mixed-criticality is to achieve this integration without affecting safety and availability of the individual sub-systems or applications. Consequently, a suitable integration approach must adhere to the relevant industrial standards and support different applications, having versatile requirements towards their integrated environment. However, not all industrial standards, e.g. IEC 61508 [33–35], explicitly support composability and mixed-criticality.

1.2 Problem Statement

The focus of this thesis is the composable integration of fail-safe triple-modular-redundant (TMR) applications subject to certification. These (existing) applications have strong requirements on the properties of the internal communication channels and reactivity of individually synchronized application instances. Together with their lower-level software, the applications assume full access to the hardware resources. With the introduction of composability this full access is inevitably removed, as in fact, the sharing of resources is the key benefit of integration. The arising problems are now twofold. First, the applications' timely behaviour is affected due to the (now) restricted access to hardware resources. Secondly, this restriction changes the basis of the applications' safety concepts for certification.

1.3 Aim of the Work

The goal of this work is to provide a technical and conceptual foundation for the (separate) certification and integration of several such triple modular redundant applications with non-critical applications. The solution shall combine the benefits of mixed-criticality and composability with that of reusing existing TMR applications. It must provide a balance between the following three different objectives:

1. Provide a technical foundation for independent certification (separation and predictability),

2. Fulfill the reaction time requirements of the applications (performance), and

3. Efficiently use the available hardware resources to take advantage of the integration (utilization).

1.4 Methodological Approach

Starting with the general requirements for composability and mixed-criticality, possible approaches for different kinds of TMR methods replicating software are evaluated. Based on this evaluation general limits and opportunities of composable architectures are outlined. Within this general solution space, concepts suitable for fulfilling the requirements imposed by the targeted safety-critical TMR applications are identified and a corresponding system model for analysis is defined. Specific software TMR synchronization mechanisms and respective metrics are presented and justified for use in the composable environment. These strategies are then evaluated for the different architectures using the metrics and system model. Slightly adapted versions of the synchronization mechanisms are proposed and analyzed, which offer beneficial properties within a specific composable setting. Based on these insights, a formal definition of contracts for the composability approach is given together with a method for evaluating feasibility of application integration and architectural changes. Simulations and measurements with prototypes are used to demonstrate the accuracy of the architectural and synchronization analysis, as well as the suitability of the contract approach.

1.5 Structure of the Work

In the next chapter an overview of the state-of-the-art of the many related research fields is given. Chapter 3 presents the general concepts for software composability, followed by the system properties of contemporary TMR architectures, when extended with composability, as well as TMR architectures leveraging composability in Chapter 4. The requirements of the chosen applications subject to integration are presented in Chapter 5, and the according system model in Chapter 6. Chapter 7 justifies and describes the three synchronization mechanisms subject to our analysis. The solutions proposed for composable integration of the applications are presented in Chapter 8, and the corresponding analysis described in Chapter 9 and Chapter 10, respectively. The contract model for independent certification for these solutions is defined in Chapter 11. Results of simulation and experimental evaluation are presented and discussed in Chapter 12. Chapter 13 concludes this thesis, and directions for future work are given in Chapter 14.

CHAPTER 2

State of the Art

Providing composability and mixed-criticality for safety-critical applications which use TMR for fault tolerance has many different aspects. We start with the support of composability in industrial standards in Section 2.1, followed by partitioning and virtualization, both of them techniques suitable for separating applications from each other, in Section 2.2. Current approaches for software abstraction with contracts are presented in Section 2.3. Section 2.4 gives an in-depth description of TMR systems and Section 2.5 is concerned with scheduling, given its impact on composable systems. Research regarding software TMR application on top of partitioned systems is presented in Section 2.6.

2.1 Standards

Concepts and methods for mixed-criticality and composability are already used in the industry.

The avionics domain has adopted the concept of integrated modular avionics (IMA) defined in DO-297 [105] for the integration of different safety-critical components on one hardware/software platform. The ARINC 653 standards [10–12] define an application software standard interface for development and integration of software functions of mixed-criticality on such a common platform following the avionics standard [99]. Different roles and tasks are used for a clean separation of function provider, platform provider and system integrator. Additionally, file formats for exchanging certification-relevant data with respect to application, platform and system behaviour are defined. These standards and guidelines are supported by industrial products for IMA, e.g. the VxWorks 653 Platform [121].

AUTOSAR is an approach to define a platform standard for the automotive domain. This includes a common interface for electronic control units and for allowing software reuse by providing a runtime environment for applications [13, 27, 45]. With the ISO 26262 automotive standard the concept of SEooC (Safety Element out of Context) can be applied to certify a safety-critical element in isolation, based on assumptions of the operational context. The final evaluation is performed when the safety-critical element is used within a specific system, and

it includes verifying the correlation of the assumed context to the specific context within the system [41].

For the railway domain, the CENELEC EN standards [28–30] provide generic safety cases for incremental certification, which are suitable to construct a safety case for composability and mixed-criticality. The CENELEC EN standards are based on the IEC 61508 standards [33–35].

2.2 Partitioning and Virtualization

Most standards address composability with a partitioning concept to isolate individual components for certification. Partitioning can be applied on board, as well as chip or software level. However, all approaches define clear boundaries between the individual partitions.

Initially, the software separation issue has been discussed by Rushby [106] with the introduction of the concept of separation kernels for security. The idea is to reach the same isolation for individual programs as with a federated architecture and ensure that the programs executed within the partitions cannot detect that they are actually located on an integrated platform. Another goal is to keep the code base of the separation kernel very small, so as to be able to formally verify its fulfillment of the separation requirements. This approach was further developed to the MILS (Multiple Independent Levels of Security) architecture [6, 7, 115] which uses such a separation kernel at the lowest level. Separation kernels in this architecture are usually based on microkernels, which also follow the design paradigm to execute most code in user mode and use partitioning [40,68,112]. This partitioning concept is also the basis of the integration concept for fail-operational systems suggested by Rushby [107] for avionics. There, the correct and timely execution of all safety-critical partitions is mandatory, and the system must remain operational even under (the hypothesized) faults. Time and space partitioning are also the core principles of the time-triggered architecture (TTA) [74]. In this architecture, clock synchronization and a system-wide static-cyclic schedule for communication and computation of safety-critical applications provide the foundation for guaranteed reaction times.

Kopetz et al. [75] developed a time-triggered System-on-Chip architecture that supports composability by partitioning the chip. Each safety-critical application is assigned its own processor, which is similar to a federated architecture, but the communication in-between the processors is provided by a time-triggered Network-on-Chip. This network is managed by a trusted network authority and only accessible for the individual applications via a trusted interface sub-system, which limits network access to predefined access patterns.

Another hardware-implemented solution for composability has been designed by Hansson et al. [59]. It achieves predictability by implementing specific schedulers for different resources, to ensure worst-case access and execution times for all requests made to the resources. The underlying scheduling strategies have been described in more detail by Akesson et al. [4]. As in the approach of the time-triggered System-on-Chip, applications do not share individual processors in the original design. This has been changed by Molnos et al. [89] by applying the technique of virtualization to successfully integrate several applications on the same processor. In the extension, a processor local static-cyclic scheduler provides constant CPU time for applications. Predictability for communication via the Network-on-Chip is achieved by computing a worst-case memory access time, since only direct memory access (DMA) requests are used for

communication.

A prominent separation approach regarding virtualization is the use of hypervisors, also called virtual machine monitors [16, 86]. A hypervisor shares the same hardware machine between different operating systems, called guests, by providing each of them a virtual representation of the hardware machine, i.e. the virtual machine. The main difference between a hypervisor and an emulator is that with a hypervisor most of the guests' instructions are executed natively and the hypervisor only interferes with certain operations, which would effect the hardware state for all guests. Current CPUs provide special hardware functions to support the isolation between virtual machines and accelerate their interaction with the hypervisor and hardware resources [96]. Performance guarantees and isolation between virtual machines is a concern for both hypervisors used in embedded systems, as well as for server virtualization [56, 86]. The Xen hypervisor, most commonly used in non-safety-critical server environments, has also been extended with an ARINC 653 CPU scheduler and I/O driver to support development of ARINC 653 applications on regular PCs [114].

As discussed by Heiser and Leslie [60], the precise border between microkernels and hypervisors is not that clear and both may use hardware-supported virtualization techniques [58,61]. A comprehensive state-of-the-art in embedded virtualization has been given by Gu and Zhao [55]. Using virtualization for implementing a primary-backup fault-tolerant system has been suggested by Bressoud and Schneider [26]. Different methods for virtualization and the fundamental concepts of hardware virtualization support to increase hypervisor performance have been discussed by Adams and Agnesen [3].

Perez at al. [95] presented a certification strategy for mixed-criticality on a single multicore chip. It is a wind power control system subject to the IEC 61508 standard. Several steps towards their targeted mixed-criticality system are outlined, in which a certified hypervisor is executed on top of diverse cores (x86 and Leon3 in the example). This hypervisor provides safety-relevant features such as fault-tolerant synchronization of clocks and safe communication between partitions. Separate diagnosis partitions are provided for each core executing a safety-critical partition as well. The diagnosis partitions periodically trigger a watchdog provided for each individual core. A static-cyclic scheduler ensures sufficient CPU time for the safety-critical partitions.

The focus of recent research on partitioning is on guarantees for isolation and performance on multi- and many-core platforms [66, 77].

2.3 Contracts

Various forms of contracts are used in software engineering, but all with the same two goals: a precise specification, and enabling software (component) reuse through interoperability.

Meyer [87] introduced contracts for object-oriented programming with the design by contract method, to explicitly expose the requirements and guarantees of objects and their clients. The contracts are expressed in terms of preconditions, postconditions and invariants of classes and methods. An exception mechanism is used when such contracts are violated.

Component-based software engineering (CBSE), applies the design by contract approach on a higher abstraction level, where applications are constructed from components within a specific

component framework. According to the fundamental idea of CBSE, contracts are provided with components to express quality of service parameters, such as availability, in addition to their functional behaviour specified in the APIs [14]. However, many industrial standards for software engineering, e.g. CENELEC EN 50128 [30], have no such notion of contracts, but use general software component specification and documentation together with tests and reviews in the development and maintenance processes to ensure software and system quality.

Conmy et al. [36] suggested to add contracts to the IMA concepts, which list matching and non-matching software components' requirements and guarantees for composition. These lists contain statements on a fine granular level and each element is related to a failure mode. The aim of these contracts is to ensure that the components always provide mitigations for failure modes. This is the same goal as for applying failure mode and effects analysis for software components and subsequent specification of safety-related application conditions for safe operation in the railway standard [29].

The concept of contracts is also used with the Goal Structuring Notation (GSN) [54]. With GSN safety of a system is documented by step-wise decomposing the overall safety goals of this system into sub-goals and linking them to elements, which support that these goals are satisfied. Contracts are used to define interfaces between elements, as well as goals, to enable local alteration of the safety argumentation when necessary, without affecting other elements, goals or contracts [43]. This is an extension of the original GSN and its tabular contracts specified in [70]. Both types of contracts are used to argue that a certain system or component property is reached and are not designed for automating the certification process.

Using contracts to enable automatic software updates for individual tasks in a certified fail-safe system has been presented by Neukircher et al. [91]. This approach was later extended for mixed-criticality [92]. The feasibility of the configuration change is evaluated based on the individual contracts before applying the upgrade. Monitoring ensures that the tasks which do not stay within the limits defined in their provided contracts are terminated. A similar contract approach has been used by Aldea et al. [5]. Each application provides a contract stating its scheduling requirements and is deployed only if the overall schedule is found to be feasible. However, contrary to the monitoring before, bandwidth servers are used to prohibit resource overconsumption by the individual applications.

Several aspects regarding contracts for system design and component-based design have been compiled and presented by Beneviste et al. [18]. A mathematical theory, based on the elements of contract theory [62] is given, for defining and composing contracts, as well as an overview of several modelling theories. An example regarding the automotive domain and AUTOSAR also considers timing guarantees in its contracts. In these contracts, minimum and maximum times between occurrences of related events and events received via interfaces are specified using a machine-readable contract specification language.

2.4 Triple Modular Redundancy

Triple modular redundancy (TMR) was originally introduced by von Neumann [118], and the general concept for software-implemented fault-tolerance by Wensley [120]. In software TMR systems, software components are replicated to provide the fault-tolerant functionality, as op-

posed to N-version programming [31], where different software components are developed to also cover faults regarding the software implementation. Schlichting and Schneider [108] presented the fail-stop processor, which stops when too many fault occur. It is constructed from several processors executing the same program and a protocol for reaching consensus. Schneider [109] also suggested the use of replicated state machines for implementing fault-tolerant services. An overview of methods and requirements for achieving fault tolerance with replication has been given by Poledna [97], covering triple modular redundant architectures with hardware lock-step, as well as software-only solutions on commercial off-the-shelf (COTS) hardware. An industrial example for a TMR system is the TAS Control Platform [48, 72], which provides TMR for the railway applications built on top of it.

The primary goal of TMR is to keep the system operational in case of a single fault with respect to the *fault hypothesis*. The principle is to have three identical modules, or replicas, and mask the output of one failed replica by the outputs of the remaining two replicas. Such a fault can have various sources, such as a single event upset or hardware wear-out. Depending on the degree of independence of these replicas, detection and fail-over can be reached for different kinds of faults. Consequently, the architecture is separated into three fault containment regions, and it is essential that only one fails at a time. There are three threats to this principle:

1. In case of *near-coincident faults* two (or all) replicas fail due to faults of independent origin, i.e. random faults that happen to occur at about the same time. In theory, this is ruled out by the single-fault assumption, and in practice, the very low fault rates make this extremely improbable.

2. In case of *common cause failures*, we again encounter failures of two (or all) replicas, this time, however, these originate in the same single fault. An example of a common-cause fault is a software bug that makes all replicas behave in the same erroneous way. That is why fault containment between the replicas is so important.

3. In case of *spare exhaustion*, one replica did not recover from a previous fault and there are therefore, too few available to mask the current fault with the remaining replicas. This makes *replacement*, in case of permanent faults, and *recovery*, for permanent and transient faults, of the affected replica essential.

Synchronization in a TMR systems keeps the individual replicas in a comparable (or close) state regarding the time and value domain [97]. Executing CPUs in lock-step operation and adding hardware voters in-between them is one option for such a mechanism [122]. Virtual synchrony, based on group communication and local timeouts, is another approach [25]. It establishes synchronization within the software layer by ordering messages. A vast number of protocols and mechanisms have been defined and used for providing virtual synchrony and fault-tolerance in distributed computing environments [8, 24, 25, 101, 113]. Fault-tolerant synchronization of hardware clocks and subsequent synchronization of input data is a another approach for achieving synchrony [76, 79, 97]. This is also the basis for TMR synchronization in the TTA [17].

TMR methods for replicating software can be classified as shown in Figure 2.1. They differ in properties of fault containment, concurrency, synchrony and resource utilization.

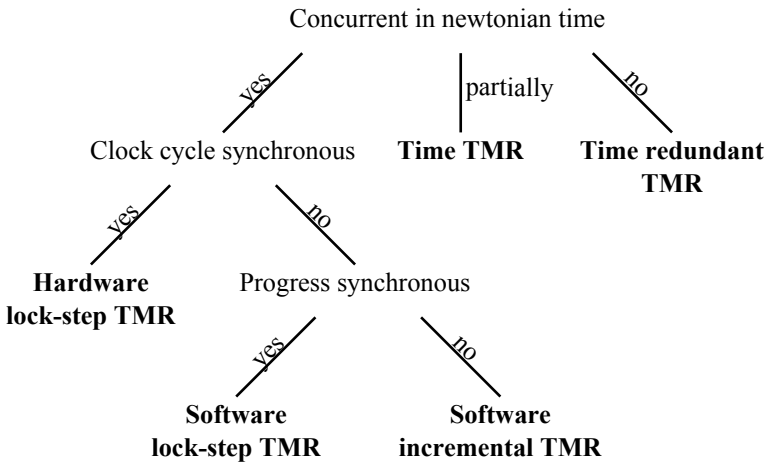


Figure 2.1: TMR classification for software triplication.

For the time redundant TMR method, instructions or software components are triplicated and executed one after the other on the same processor. Software-implemented voting over their results provides protection against transient hardware faults under the assumption that these affect only one instruction or component in the triple. Based on the same concepts as time redundant TMR, the Time TMR [37] has been developed, which executes some of the triplicated instructions in different computational units of a VLIW processor, reaching partial concurrency within the same processor. For hardware lock-step TMR, three CPUs are connected to the same clock source. The redundant voting is performed in special hardware outside the CPUs, that is connected via suitable buses. Less tightly coupled TMR systems, where voters are implemented in software, have to explicitly communicate between the replicas.[1] In software lock-step TMR, all replicated software components compute the same output, and only after voting has been performed do they continue operation. For software incremental TMR, the software components can be at different states of execution when exchanging their voting data. This full or partial synchronization can be performed periodically or sporadically. However, in both cases timeouts are needed to detect faulty replicas and guarantee reaction times. Thus, an important characteristic in these software TMR systems is the time needed for synchronization. Figure 2.2 illustrates how replicated code is executed in systems based on the presented different TMR methods.

As mentioned before, if long mission times are required, it is essential to provide recovery in the TMR approach in order to remain operational in the presence of several sporadic transient failures occurring over time, as well as during repair or replacement of a permanently faulty hardware module.

[1] A motivation for implementing redundancy mechanisms in software instead of relying on specialized hardware has been given by Bernick et al. [20].

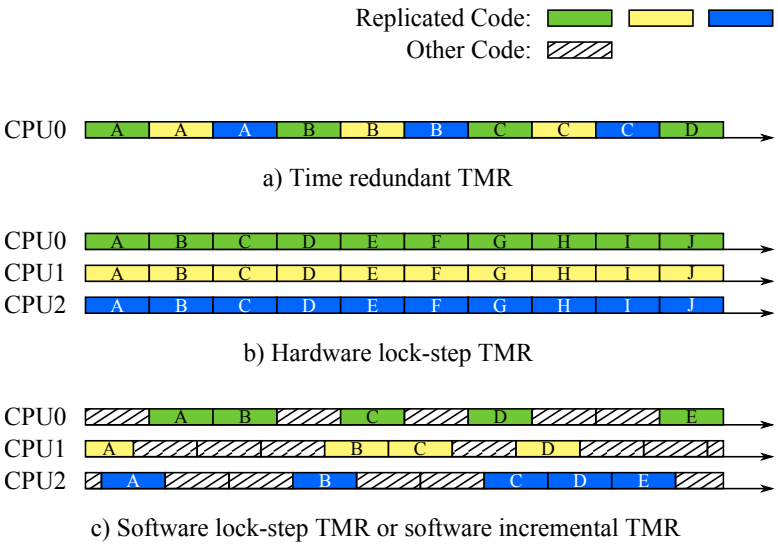


Figure 2.2: Execution of replicated code according to different TMR methods.

2.5 Scheduling

Scheduling of safety-critical real-time applications strongly influences their behaviour, especially in the time domain. Various scheduling strategies are in use for real-time applications and composable environments. Liu and Layland [84] provided the task model and initial results for analysis of dynamic fixed-priority and EDF schedulers for real-time tasks on uniprocessors. They proved the optimality of fixed-priority scheduling with rate-monotonic (RM) priority assignment and preemptive EDF for uniprocessors, i.e. if any algorithm can find a feasible schedule, then also these schedulers will find one.

ARINC 653 defines a static-cyclic scheduling scheme with major frames and minor frames for composability. The tasks are assigned slices within this schedule to fulfill their scheduling criteria regarding period and time slice duration. The main challenge during integration concerning this schedule is to find one which satisfies the scheduling requirements of all tasks. Additionally, it is possible to switch between different off-line generated schedules to accommodate various operating modes. A detailed analysis of implementing this approach on the operating system level has been conducted by Baldovin et al. [15].

A comparison of static-cyclic with fixed-priority scheduling, for periodic tasks, has been performed by Locke [85], concluding that RM will result in tasks with higher task schedule jitter but also provides a better CPU utilization. Preemptive fixed-priority scheduling, together with runtime feasibility tests and slack sharing are the key characteristics of the scheduler provided with the real-time OS DEOS for the avionics domain [23]. In the presented examples, it achieves a significantly better utilization than schedules for a static-cyclic scheduler at the cost of system scheduling optimization.

Using servers for limiting resource access is an approach to provide guarantees in systems, where tasks could exceed their designated resource share otherwise [50, 102]. Abeni and But-

tazzo presented the constant bandwidth server to provide execution share guarantees at the cost of limiting execution times for EDF scheduled tasks [2].

Vestal showed that in some scenarios other than RM priority assignment can lead to feasible schedules [116], based on the assumption that applications with a higher criticality are more pessimistic when estimating their worst-case execution time.

Having several layers of schedulers is the focus of hierarchical scheduling [82, 83]. The central idea of this approach is to provide each application with its own scheduling strategy, such that the application-specific schedule and scheduling strategy are preserved when integrating them with other applications and to provide guaranteed resource shares. Integrating hierarchical scheduling and mixed-criticality in virtual environments has been performed by Lackorzyński [78]. The virtual guests and hypervisor are adapted to provide sufficient information to the hypervisor scheduler by letting the guest choose from a set of scheduling contexts. A general solution for mapping hierarchical mixed-criticality fixed-priority schedulers to such a context-aware scheduler setup has been proposed by Völp et al. [117]. Enabling virtual machines to request different resources during runtime is the focus of the concept suggested by Groesbrink et al. [53], where conflicts on these resource requests have to be resolved and the overall schedule of the system adapted accordingly.

A summary on the research regarding real-time scheduling on multi-core systems was given by Davis and Burns [39]. Naturally, providing guarantees on resource access for multi-core platforms is important [22]. For multi-core CPUs feasibility tests exist for (global) EDF and fixed-priority schedulers [9, 21].

2.6 Partitioning and TMR

A fault-tolerant prototype combining software lock-step TMR with the partitioning provided by an ARINC 653 RTOS (real-time operating system) has been implemented by O'Connel [94]. The RTOS is modified to synchronize the static-cyclic schedules on the three redundant and fully connected hardware boards. Voting and TMR message exchange for input and output data are implemented in privileged partitions running in the kernel space. To achieve the targeted synchronization precision for the schedules, low-level network driver functionality is included in the RTOS and raw Ethernet MAC frames are used for the periodic clock synchronization. The prototype does not provide a membership service or means of recovery, but it is suggested that a partition could be reset, if it reaches a certain threshold of faults. Reintegration would take place, if it provided valid output values for a certain time after such a reset. With regard to partitioning, the approach differentiates between *Software Fault Containment Regions* established by the partitioning and *Hardware Fault Containment Regions* provided by the individual hardware boards. Integrating several triplicated and independent applications on the same hardware infrastructure was not within the scope of this prototype.

Bauer and Kopetz [17] demonstrated that transparent software-based TMR for applications can be provided within the TTA, where the TTA inherently ensures strong time partitioning and clock synchronization. Synchronizing the schedulers of microkernels (PikeOS) using a time-triggered network has been presented by Theiling [111].

Miller et al. [88] suggested an approach for synchronizing TMR applications on top of static-

cyclic scheduled partitions in an IMA compliant system. The synchronization mechanism is triggered every scheduling round and starts the replicated execution, if a sufficient number of messages have been received. The fault hypothesis of this approach assumes fail-silent nodes.

An analysis on the influence of composability when applied to various TMR methods and presented architectures leveraging this approach was conducted, which results are also included in this thesis [103].

CHAPTER 3

Software Composability

This chapter gives a general introduction to the fundamental concepts of software composability and describes the prerequisites of step-wise and independent certification. Furthermore, the challenges of composable resource sharing and the relation between composability and TMR are discussed.

Safety is a system property. Therefore, a single system component can only fulfill a safety property within the context of the whole system application [81]. The intention of composability is to allow building safe and certified systems by careful integration of components, some of which provide safe and (pre)certified functions. As an immediate advantage, this facilitates the reuse of certified components. We call such a component *function-set* (FS), to emphasize that functions are provided by one or more entities, especially in a TMR architecture. These FSs are then deployed within an *integration environment* (IE) to build the whole system. Here, the possibility of sharing the same IE for different (sub-)systems, thus saving cost, space and energy, represents another advantage. An FS provides a (sub-)service within the application context and is assigned a criticality level according to the criticality of that service. Obviously, the proper provision of this service can only be guaranteed on the condition that the IE exhibits all the properties that have been assumed in the design of the FS. While this is relatively trivial to establish in the traditional federated architectures (i.e. using a separate IE per FS), it becomes an issue in integrated approaches, since the properties of the IE, as perceived by a single FS, are (dynamically) influenced by the other FSs during their execution. Therefore, to enable composability, every FS must be associated with an appropriate *function-set contract*, specifying its requirements to the IE for correct execution. We refer to the deterministic availability of resources from the IE as *predictability*[1]. This first constituent of composability becomes crucial when FSs or elements of the IE are to be changed. Note that in the interest of a simple FS contract static guarantees (i.e. high predictability) are beneficial, while more fine-grained, even dynamic requirements usually facilitate a better resource utilization. In addition, the former is easier to enforce by technical means (see later).

[1] Unlike Akesson et al. [4], we define predictability with respect to available resources for FSs as provided by the IE and not as predictability of execution times and resource demand of FSs.

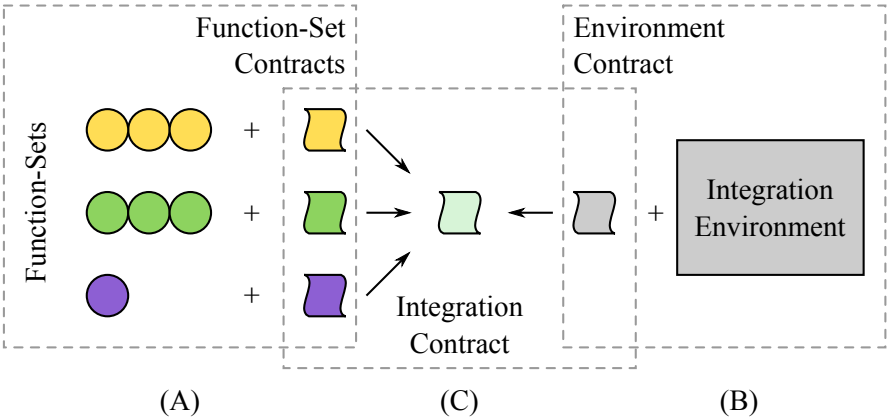


Figure 3.1: Certification strategy using contracts for mixed-criticality and composability.

The second constituent of composability, namely *non-interference*, concerns undesired effects that the execution of an FS may have on the IE and consequently on other FSs, specifically in case of failure. Again, one could, in principle, conduct a fine-grained, application-specific analysis on malign and non-malign cases to allow for the largest freedom. In practice, however, the most rigorous approach has proven most effective – a strict *failure confinement*[2]. Herein, each FS forms an individual *failure confinement region*, the failure of which remains local and has no effect on any of the others.[3] This property cannot be enforced within the FS itself, since restricting an FS's failure behavior is normally very expensive, and sometimes even impossible. Consequently, this task has to be fulfilled by the IE, which needs to implement technical provisions to *separate* the FSs from each other.

In the context of mixed-criticality, it is interesting to note that non-interference is a directional property: One may be very concerned about a non-safety-critical FS undermining the IE's assertions for a high-criticality FS, while at the same time it is – by definition – not as critical when the latter prevents the former from executing correctly. Failure confinement, however, does not appreciate this directionality; typically, the separation mechanisms are applied equally to all FSs, irrespective of their criticality.

Ultimately, the composability approach allows to split the certification of a system into three parts, as illustrated in Figure 3.1:

(A) Each safety-critical FS is certified with respect to its FS contract, which specifies all the FS's requirements for safe operation. These requirements may cover computational and networking resources, as well as separation guarantees and the availability of special services such as watchdogs or real-time clocks.

(B) An IE, e.g. hardware boards and middleware, is certified with its integration environment contract, stating the provided resources, services and failure containment properties, as

[2]Like the "Gold Standard for Partitioning" defined by Rushby [107].

[3]Please note that fault containment regions are defined with respect to the fault containment of TMR systems (see Section 3.3).

well as its feasibility evaluation method used for integration. The validity of this feasibility evaluation method is a key element when certifying the IE.

(C) The FS- and IE contracts are specified using generic properties, like network bandwidth, to enable reusing of FSs in different IEs. A concrete system is then certified by matching the IE contract with the FS contracts in an *integration contract*, using the feasibility evaluation method specified in the IE contract.

Please note that for each safety-critical FS step (A) is performed separately, as is step (B) for each specific integration environment. Furthermore, for each new or altered system step (C) must be implemented. This method needs more initial effort than certifying one system as a whole, Nonetheless, it is more efficient when building several slightly different systems, or altering existing ones. Additionally, a good utilization of the hardware resources within the IE is expected. The contracts here are based on the provided resources and features of the IE, unlike the contracts of contract-based design [19] which specify component interfaces and behaviour, and subsequently create new ones with various operations performed on other components. It is important to note that composability in our context is merely concerned with the separation issue, i.e. protecting one FS from the failure of another that shares resources with it. It is also not capable of tolerating faults in these resources, which is the goal of triple modular redundancy.

3.1 Attaining Composability

The general idea for constructing an IE is to implement a layer that provides failure confinement regions (*partitions*) within the IE, sufficiently mitigating the undesired effects of integration and independent of the specific hardware setup as shown in Figure 3.2.

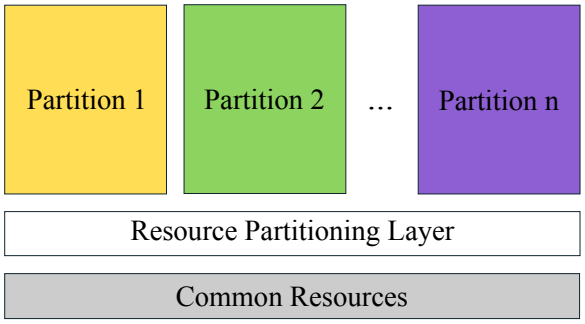


Figure 3.2: Generic resource sharing concept with partitions.

Here, the non-interference and predictability requirements of the individual safety-critical partitions are fulfilled by the partitioning layer for the hardware resources. This layer takes care of services like partition scheduling, memory management and provision of virtual I/O services. This implies that the partitioning layer has more privileges than the partitions in order to maintain full control over the system. The partitioning could be implemented within an operating system, which has to ensure that all system calls are composable, e.g. a suitable microkernel OS. In

such solutions the kernel itself offers only a very limited set of functionality, i.e. inter-partition communication and direct device access. Consequently, the other shared system services must also be composable themselves. Alternatively, it is possible to implement such a partitioning layer using virtualization techniques, e.g. hypervisors. With virtualization, most of the low-level functionality is provided by the guest OS, which is encapsulated in the same partition as the application. As already pointed out in Chapter 2, microkernels can also use virtualization technology and the boundary is not that sharp. However, our focus is composability of TMR systems and not the actual implementation. As such, we will also not go into detail with possible security aspects of methods for non-interference.

In the following, we investigate the resources that are typically shared among FSs within a common integration environment, as well as suitable separation techniques for such resources.

Processing

A typical example here is a single processor that is shared by different FSs[4]. The clear advantage of sharing is the better utilization of the processor that may be too powerful for just a single FS. As the resource sharing is performed in the time domain, the FS contract and integration environment contract will have to be concerned with providing a suitable share of the processing power to the individual FSs, in terms of total computation time, as well as the availability of such computation time at certain instances in time. The mission of separation is to enforce this time sharing and prevent a failed FS from consuming another FS's share. This is clearly a scheduling issue. In the ARINC 653 standard a static-cyclic partition scheduling with major frames and minor frames is employed [114]. For safety-critical partitions a static-scheduling scheme has to be guaranteed, whereas other partitions could be scheduled differently. The final schedules obviously have to fulfill all safety-critical FSs' timing requirements. As presented in Chapter 2, many different scheduling strategies have been proposed for composability. One may also imagine different scheduling schemes, like priority-based scheduling with additional mechanisms, such as constant bandwidth servers for ensuring predictability on a higher level.

Memory

In contrast to the case of processing treated above, memory is a stateful resource, which is why time sharing is not applicable. Instead, the physical memory (we refer to RAM in the following) must exhibit sufficient capacity to store all data from all FSs. The benefit of having one larger memory instead of several individual smaller ones is (a) saving control logic, and (b) achieving a reasonable level of occupancy even if individual FSs demand memory sizes that do not fit well to a power of 2. The mission of partitioning is to ensure that each FS can access only that portion of memory that has been exclusively assigned to it and is unable to access (in the worst case corrupt) the memory of other FSs. Typically, a Memory Management Unit (MMU) is used for this purpose.

[4]We may as well imagine a set of processors with dynamic dispatching of tasks from different FSs. This is of course the more generic concept, however, the chosen simple example is sufficient to illustrate the key issues.

Strict non-interference also requires that CPU caches are either disabled in general, or they have to be flushed and restored at each partition switch to prevent memory access jitter. This is only necessary for safety-critical partitions with very stringent timing requirements.

Devices and I/O

For this type of shared resource we choose disk I/O, networking, hardware watchdogs and real-time clocks to illustrate a wide range of different properties and behaviours.

Basically, I/O resources can either be exclusively assigned to single partitions, in which case non-interference is trivially guaranteed, or the access has to be managed by the resource partitioning layer. This is typically implemented by providing a "virtual instance" of the resource to the individual partitions and managing the synchronization of the actual resource either directly in the partition layer or within a privileged partition. In this sense, *disk sharing* can be done with actual or virtual disk drivers. Similarly, *networking* can rely on the actual network interface or on virtual network interfaces, virtual switches and virtual routers.

Note that the virtual access is similar to the processing discussed above in that there is only a single resource available that is shared in a time multiplex fashion. Therefore, the contracts have to be concerned with bandwidth and transmission time guarantees. Again, additional requirements concerning the instants of availability may apply in a real-time context – hence, the scheduling issue emerges here again.

The *real-time clock* (RTC) is a particularly difficult device to share, since it is, by definition, a stateful device – which rules out its use in a time multiplexed fashion – but at the same time it cannot be replicated like, e.g., the memory space. Depending on the FS's requirements, three different options for the emulation of a RTC are possible:

- FSs requiring only reading the RTC and alarms and timers:
  For these FSs, the RTC emulation can be restricted to multiplexed reading of the clock and handling distribution of alarms and timers.

- FSs requiring reading and writing of the RTC but no preservation of time during power off:
  Here, a clock with higher granularity than the RTC is needed for correct emulation of writing and reading the RTC. A FS writing a value to its emulated RTC expects the RTC's value to increase at a certain time after that. For example, a FS setting the RTC to 00:00:00 and reading its value 100ms later, still expects to get 00:00:00 as time value. Therefore, offsets between the emulated RTCs differ according to the value and time of occurrence of the respective FS write.

- FSs requiring reading and writing of the RTC, as well as preservation of time during power off:
  This is the most difficult case, since a clock with higher granularity than the RTC is needed for write emulation, and the RTC and non-volatile memory are needed for preservation of the time value and offsets during power off.

Sharing a *hardware watchdog* raises similar issues. To enforce non-interference a watchdog has to be emulated for each partition within the partition layer. Still, a failure in the partition layer might cause a halt of computation, so an actual hardware watchdog has to be triggered systematically by the partition layer.

In addition to the regular I/O capabilities, the partitioning layer may provide means of inter-partition communication. Here, it has to be ensured that these mechanisms are not used to undermine the composability concept of the partitioning, e.g. a partition providing non-composable I/O sharing for other partitions.

Summary

All these methods either have one or both of these demands:

- Extra hardware features, e.g., MMU or CPU support providing the partitioning layer with full control over the system[5], or
- Extra scheduling, e.g., for sharing of the CPU, virtual networking or emulation of devices.

The impact of the presented additional hardware features on TMR applications is linear with respect to execution time, e.g., setting the MMU costs a few instructions per context switch of partitions. Consequently, we focus on scheduling in our analysis.

3.2 Requirements for Function-Sets with a Safe State

In applications without a safe state, the correct and timely execution of all safety-critical FSs is mandatory and the system must remain operational even under (the hypothesized) faults. This clearly demands uncompromized predictability and strict partitioning.

Safety-critical systems with a safe state, in contrast, can handle cases where no results or outputs are provided by a safety-critical FS. The important property here is that no *incorrect* outputs are produced. This is normally ensured by fault-tolerance measures, e.g. TMR. The remaining, but very important, requirement on the partitioning layer is not to undermine the error detection and/or masking capabilities of these measures, e.g., by introducing common-mode failures.

If this aspect is ensured, the failure confinement regions might otherwise not require as strong separation as in the fail-operational case, with respect to scheduling and timing. For example, it may be tolerable to guarantee CPU scheduling with some probability, rather than cycle-wise assurance. Caches may remain enabled and do not need to be flushed on scheduling switches between partitions as memory access jitter may be tolerated. In case of overload it might, e.g., be possible to allow safety-critical partitions some CPU time in excess of the assigned slot, thus increasing the overall robustness of the system.

In the remainder of this thesis, the term *composability layer* is used rather than partitioning layer to emphasize that it is not necessarily required to achieve full partitioning in all cases.

[5]Examples for CPU features enabling full control over the system are trap-and-emulate or a special execution mode [3].

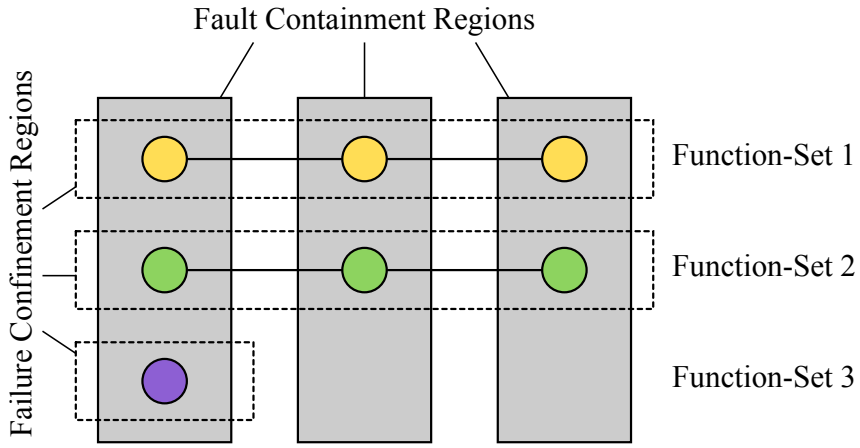


Figure 3.3: Function-sets in failure confinement regions and fault containment regions.

3.3 Failure Confinement Regions vs. Fault Containment Regions

Composability is a concept orthogonal to TMR that aims, as already outlined, at achieving better resource utilization and ease of the certification process upon integration. These benefits equally apply for composability in TMR architectures. As illustrated in Figure 3.3, the IE may comprise replicated modules, and we have two orthogonal regions in a composable TMR architecture:

- The replicated hardware modules form *fault containment regions* required to prevent single points of failure in the TMR architecture. With a properly working TMR, the safe execution of a FS can be ensured even in case of a random fault in its IE.
- Within these hardware modules, each FS forms a *failure confinement region*. This establishes the non-interference required for composability. With non-interference, the safe TMR execution of a FS in presence of other FSs is guaranteed.

In this scheme, a safety-critical FS comprises three entities, each representing a computing channel. FS 1 and FS 2 are examples of this. Assume the fault containment regions are independent hardware boards, then the failure of one is observable as fault of one entity for FS 1 and 2. In contrast, a single event upset (SEU) in the memory of one entity is only observable by the affected FS, as the failure confinement regions provide isolation within the fault containment region.[6] Furthermore, if FS 2 fails due to a software error, the failure confinement regions provide protection for FS 1 and FS 3.

Note that, in our example, FS 3 comprises one computing channel only, as it is not safety-critical. This already indicates that having three computing channels per FS only illustrates the fundamental principle of this architecture, and many variations are possible. For the fault tolerance scheme, e.g., a simplex or duplex architecture could be chosen instead of TMR as well, as

[6]This is also the reason we do not use the terms Software Fault Containment Regions and Hardware Fault Containment Regions as O'Connel [94], since composability may provide separation for software, as well as hardware faults.

is appropriate for the needs of the specific FS. More generally, there is a lot of freedom in aligning the failure confinement regions of the FSs with the modules' fault containment regions. One may, e.g., leverage the error detection capabilities of the applied fault tolerance scheme for localizing errors within partitions and only selectively stop and recover the affected partitions rather than re-booting and testing the complete system. As already described, a potential risk is that bugs in the composability layer may lead to common-mode failures, thereby undermining the fault tolerance concept. Strategies against such failures are, e.g., to use a certified composability layer, or one that is proven-in-use and incorporate suitable measures within the FSs.

CHAPTER 4

Composability for TMR Architectures

In the course of this chapter, we first take a close look at different TMR architectures and their behaviour when introducing composability, before discussing existing and novel TMR architectures enabled by the composability approach.

4.1 Contemporary TMR Architectures

In a TMR system three replicas are executed within three different fault containment regions. To keep the three replicas of a TMR system close in the time and value domain, the following three key services are necessary:

- Synchronization of the input and output data (and/or state) of the replicas,
- Voting of the synchronized data, and
- Recovery, i.e. reintegration of a new replica or one that experienced a transient fault.

Contemporary TMR architectures vary largely with respect to their fault containment regions and synchronization approaches, as already discussed in Section 2.4. Consequently, when adding composability, different strategies are needed depending on these individual properties. We outline these strategies, the incurred overhead and other implications based on the characteristics of the TMR method for time redundant, hardware lock-step and software-based TMR.

Time redundant TMR

For time redundant TMR, software instructions (or components) are triplicated at compile time and voting instructions (or components) are added automatically. The triplicated instructions use different memory, which is also assigned during compilation. The fault containment regions in this architecture are instruction sequences and corresponding memory region for the state. As time redundant TMR uses only one processor, it can only mask transient hardware faults, e.g.

SEUs. This method does not require special hardware and can be used on a COTS processor. The key services from above are established as follows:

- Synchronization is implicitly given with this method due to the order of execution on the same processor.
- Voting is performed for each instruction sequence or software component separately on the input data provided by all three replicas of the previous execution.
- For software components, recovery is performed by overwriting the memory of the faulty component with the voted memory values of the other two components. It is not necessary for instruction sequences, as all input is voted and these sequences' local state is only dependent on the input.

The overhead regarding memory and execution time introduced for voting is strongly dependent on the frequency of it being executed. Naturally, repair and replacement cannot be performed during the operational phase of the system.

Here, the potential conflict between fault tolerance and composability becomes apparent: Separation of memory and CPU can be ensured by a composable scheduler and an MMU, respectively. In this setting, however, both the scheduler, as well as the MMU represent single points of failure from the fault-tolerance point of view. While the scheduler (and potential further software-based composability services) can, just like the FSs, be protected by time redundant execution as well, the MMU remains problematic.

In general, the performance impact can be deducted from the scheduling scheme. With a static-cyclic scheduler, the reaction time can be derived from the maximum time between scheduled slices of the safety-critical FS entities and the slice duration. However, this can vary for specific FSs and also depends on the shared I/O devices. Mixed criticality can achieve good hardware utilization in this architecture, as only safety-critical FSs are triplicated, while non-critical FSs can use CPU and memory at native speed.

The *Time TMR* [37] architecture presented in Section 2.4 has the same properties regarding composability as time redundant TMR. Executing some instructions on different CPU components has no advantages regarding composability, when compared to time redundant TMR.

Hardware lock-step TMR

Hardware lock-step architectures are widely used in the industry [63]. They consist of triplicated CPUs and memory with hardware voters in-between. The software is stored and executed in two types of fault containment regions: different CPUs and different memory. This architecture tolerates both transient and permanent hardware faults. Naturally, as the fault tolerance is provided by the hardware, such special hardware boards are a prerequisite. Here, the three key services are provided as:

- Synchronization is achieved by operating the CPUs with the same clock (which is why the CPUs are sometimes located together on one hardware board), resulting in them issuing the same memory operation on the bus at the same time.

- Voting is performed in hardware voters between the CPUs and the memory on the memory operations and their value.

- Recovery is performed by stopping triple execution and reconstructing the state of the erroneous CPU from the other two.

A current industrial example for such a hardware lock-step board is the D602 from MEN Mikro Elektronik GmbH [51].

The inherent triplication of the lock-step architecture is an advantage and a drawback within the composability context at the same time. Safety critical FSs, as well as non-critical FSs, are all automatically triplicated and recovered, which degrades resource utilization. Fulfilling the composability requirements specified in Chapter 3 for each fault containment region locally, i.e. CPU, memory and I/O, is already sufficient for composability of the triplicated modular redundant system, since the triplication mechanism is not influenced by the composability layer – rather the composability layer is triplicated. As in the case of time redundant TMR, the performance impact is closely related to the composable scheduling and I/O sharing strategy.

Software-based TMR

The two software-based TMR methods discussed in this thesis, software incremental TMR and software lock-step TMR as presented in Chapter 2, share most of their properties, and so we do not need to differentiate here[1].

In software-based TMR, a middleware software is running on three fully connected hardware boards, controlling the execution of the replicated application on top of it. The three hardware boards are the three fault containment regions. Such an architecture tolerates transient and permanent hardware faults, and a set of quasi-random software faults as described in [47]. This architecture has no special requirements on the underlying hardware. Furthermore, this approach enables to layer the system in such a way that the lower level hardware and OS could be replaced without much influence on the applications themselves [98]. The three key services are provided by the middleware:

- Synchronization is achieved by exchanging the input and output data of applications and time values of the local clocks via messages between the three instances of the middleware layer.

  Defining a range in which the process of synchronization is performed periodically is the common solution for both event-based and time-triggered computation. For the latter, periodic synchronization is required by definition, whereas, for the event-based computation, a minimum period is required against overload and a maximum period to check for liveness of the individual replicas.

  The correct choice of period is crucial for the operation of the TMR system. A short period leads to lots of message exchanges, scheduling of the middleware and, consequently, less

[1] The differences of software lock-step and software incremental TMR are explained in detail at the end of Section 7.1.

CPU time for the application, while a long period results in long reaction times. It is the application developer's task to find the right balance.

The synchronization precision correlates with the reaction times of the individual middleware processes and communication latencies. Low scheduling latencies and jitter for the middleware processes on the different hardware boards are just as beneficial for the synchronization precision, as are low message transmission times and jitter between the hardware boards.

- Voting is executed on the application data and time values received via messages. Only the voted data and time values are passed to the application instances. The application must follow design constraints provided by the middleware to guarantee a replica-deterministic execution.

- Recovery is performed by the middleware without interruption of the active replicas. For this purpose, the middleware must know which replicated data needs to be recovered and has to have enough communication resources in addition to the regular bandwidth required for synchronization. With recovery and three hardware boards, maintenance actions can be performed online during system operation.

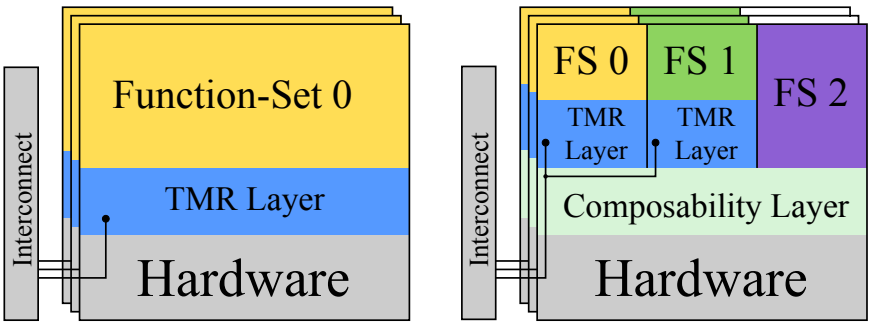
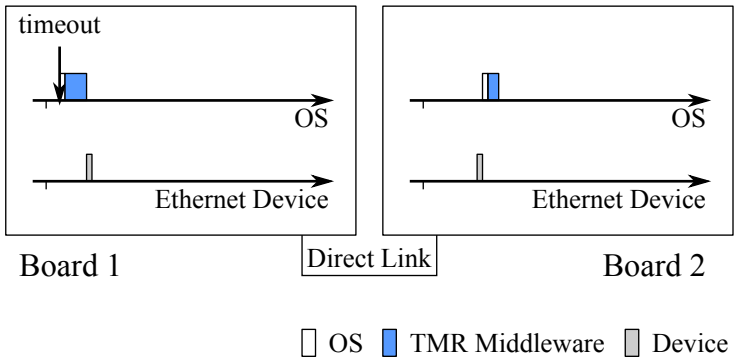


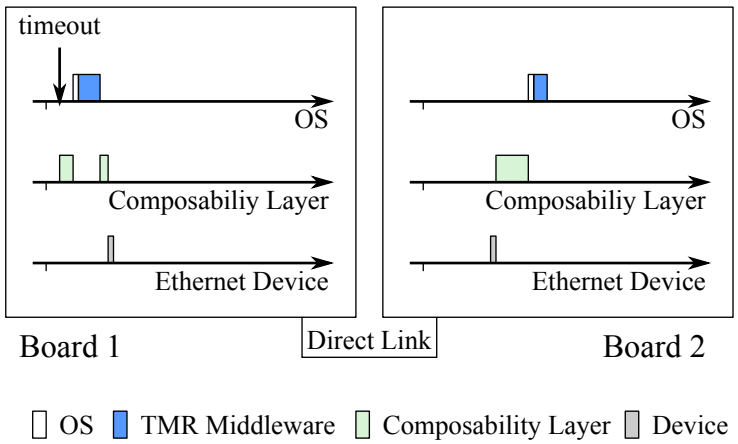
Figure 4.1: Software-based TMR without (left) and with (right) composability layer.

Figure 4.1 shows the introduction of the composability layer to software-based TMR below the triplication middleware, i.e. the TMR layer. While this enforces the desired separation of the TMR instances, the execution of the latter now relies directly on the composability layer, more specifically on its scheduling on the hardware boards and the message transmission times on the (shared!) communication links in-between. Thus, the composability layer has a direct influence on the TMR layer properties. Local non-interference for partitions, as described in Chapter 3 for the composability layer, is not sufficient for achieving good reaction times when integrating several independent triples. The execution of the triplicated applications depends on the composable scheduler, as well as on the synchronization mechanism of the corresponding TMR middleware.

The TMR synchronization process of the middleware is affected in the following two new ways in these architectures:



(a) Creating, sending and receiving a message upon a timeout with the OS executing directly on the hardware board.



(b) Creating, sending and receiving a message upon a timeout with additional scheduling of the composability layer.

Figure 4.2: Additional scheduling with composability layer.

- By resource management in the composability layer, and
- By sharing the same communication interfaces and interconnect with other TMR middleware.

The change of scheduling with the introduction of the composability layer is illustrated in Figure 4.2. In this example, one instance of the TMR middleware layer sends a message to a corresponding instance on another board. It is the scenario for the start of the synchronization process. The additional steps with the composability layer are: scheduling the sending FS instance after the timeout occurred, forwarding the message on the sending side and scheduling the FS instance forwarding the message on the receiving side. While the time necessary for scheduling processes by the OS scheduler is short, scheduling operations by the composability layer are more expensive, since more operations are required for the context switch. Measurements regarding this difference are presented in Section 12.2 on page 93. Note, that in the illustration

above no other integrated TMR replicas are active, which could delay the sending and receiving of messages even more.

The choice of scheduling strategy for the composability layer has a huge impact. Frequently scheduling the individual FS instances increases their reactivity and reduces the jitter to the point where the scheduling operations themselves introduce so much overhead, that no longer sufficient computation time is available. Also here a reasonable balance must be found between synchronization jitter, scheduling operations and computation time.

An option for reducing the impact of the scheduling by the composability layer on the TMR mechanism is to provide the synchronization service as part of the composability layer, and in doing so reduce the indeterminism introduced by another scheduling layer. This will inevitably also weaken the independence of the integrated FSs. We perform a detailed analysis of such an approach in Chapter 9.

Still, using software-based TMR in combination with composability and mixed-criticality can achieve good resource utilization, as non-critical FSs are not replicated.

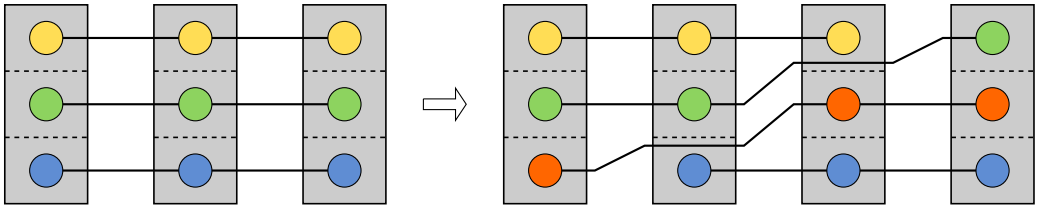
4.2 TMR Architectures Leveraging Composability

The software-based composability approach enables the creation of a set of new composable architectures which increase hardware utilization and flexibility while supporting certification and safety-critical FSs.

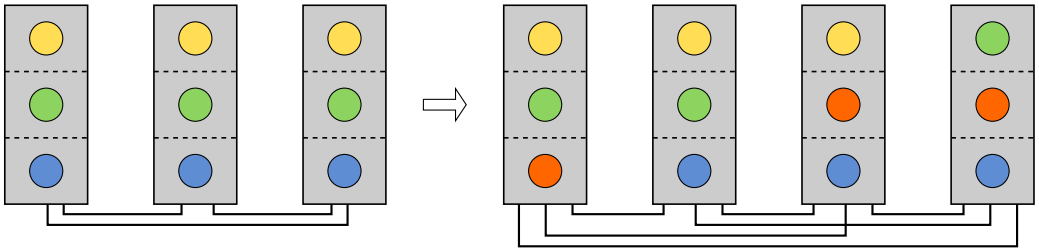
Static TMR-Composable Architecture

In the software-based TMR architecture described above, non-triplicated FSs can be added, as long as there are enough free resources on one of the hardware boards. For FSs using TMR, all three boards need free resources. If only one of the three boards has insufficient computational resources, a complete new hardware triple must be added. With composability, it is now possible to introduce a new kind of architecture, the static TMR-composable architecture. The replicas and their communication can be statically assigned to any hardware board and communication links in-between and are no longer fixed by a hardware triple. The composability and triplication mechanism are the same as for software-based TMR and also COTS hardware boards and network equipment can be used. This method improves resource utilization and scalability of the whole system. Figure 4.3(a) illustrates a possible scenario, where adding one TMR-based FS to a system can be achieved by adding one new hardware board. It depicts the change of logical interconnect, whereas Figure 4.3(b) and Figure 4.3(c) show options for the extended physical connection. In the first scenario, direct links fully connect all boards, while in the second a redundant network is constructed with two switches, each one alone already connecting all boards. The simple network structure of the second option is preferable when adding even more boards, since it requires fewer interfaces per board and can be extended easily. This change of the connection properties has to be accounted for in the composability model.

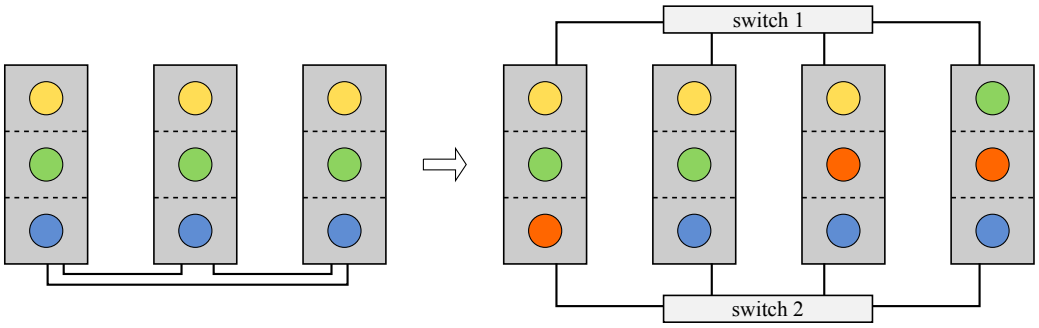
With the introduction of more hardware boards, the probability for one of them to fail increases. Since each safety-critical FS still relies only on three boards this is not an issue in the first place, but once again it has to be ensured that there are no dependencies between the



(a) Change in logical interconnect.



(b) Physical interconnect with direct links



(c) Physical interconnect with two switches

Figure 4.3: Change of static deployment when adding a new triple and hardware node.

instances of the composability layer on the hardware boards, undermining fault containment. Furthermore, the impact of the new networking components on the availability must be acceptably low in a redundant configuration. Maintenance can still be performed without system downtime. An existing example of such an architecture is the TMR on top of the time triggered architecture [73].

Dynamic TMR-Composable Architecture

In contrast to all previously presented TMR architectures, the dynamic TMR-composable architecture has no static mapping of replicas to hardware boards. An FS manager is dynamically deploying safety-critical and non-safety-critical FSs in a cluster of redundantly connected hardware boards. Ideally, this cluster can be built from COTS hardware boards and network equipment. The FS manager starts and stops partitions, establishes virtual links in-between them and

provides load balancing. Starting, stopping and interconnecting safety-critical FS entities is a safety-critical task, thus the FS manager has to be implemented as a safety-critical component within the dynamic TMR-composable architecture itself. The management of safety-critical components is actually subject to certification. For each operation altering the integrated system a new valid integration contract must be constructed from the FS and IE contracts. To automate this process, the contracts must be formalized in a machine-readable way. For fail-safe systems, it may be sufficient to implement mechanisms where the individual triples check whether the requirements specified in their contracts are fulfilled, otherwise they can trigger a safety-reaction.

The composability layer is the same as for software-based TMR. However, as the composability policies concerning bandwidth guarantees and scheduling are dynamically changed by the FS manager, it has to be ensured that such changes are covered in the IE contract. Resource usage, maintainability and availability are improved in comparison to the static TMR-composable architecture, since replicas can be moved between hardware boards during operation.

Dynamic TMR-Composable Architecture with Adaptable IE

Extending the previous approach with the ability to add and remove hardware boards during operation[2], even more flexibility is introduced in the system. Such an alteration of the system effectively changes the IE and the IE contract must be defined accordingly to also handle this variability. It affects the integration contract established between all currently running FSs and the IE before the alteration. Thus, the adaptation of the IE must be performed in two phases. First, the effects of the IE change must be determined and a corresponding new integration contract derived. Only after a contract is found where all FSs requirements are met, can the IE start to stepwise integrate or remove hardware and subsequently replace the previous integration contract. Again, this process is subject to certification and the supervision must be provided by a component developed according to the highest integrity level supported by the IE.

[2]The replacement of a faulty hardware board is not considered an alteration of the IE.

CHAPTER 5

Targeted Applications

This chapter presents the requirements and nature of the existing applications, which are going to be integrated. We consider these specific applications for three reasons. First, it narrows down the huge design space regarding applications and requirements, thus allowing for concrete analysis and solutions. Second, their requirements are still relative generic, since COTS hardware, TMR, fail-safe behaviour and round-based computation are often seen in practice. Third, they are based on actual industrial examples.

When designing a composable environment with the aim to integrate existing applications, which are already being used within certified systems, a major goal is to change these applications as little as possible so as to reduce development and certification effort for the integration and keep the trust in the applications established through the operation in the field. Therefore, the first constraint for the composability approach is:

Constraints 1 *Keep the applications unchanged.*

The targeted applications themselves must fulfill different requirements according to their provided functionality and environment. However, all are based on a middleware layer providing software lock-step TMR or software incremental TMR with the periodic synchronization algorithm described in detail in Section 7.1. Defined with the *synchronization period*, this algorithm periodically exchanges messages for synchronization and voting. The applications are all fail-safe and used within the railway domain. In general, they are rather computation than communication intensive.

The applications rely on a *safety layer*, encapsulating the applications' safety-critical functionality and providing fault tolerance with according supervision for safe execution. This encapsulation is ensured by requiring that applications only use the services of the safety layer for safety-critical communication and restrict their usage of other services. Therefore, it offers a clean interface, which includes all necessary communication primitives, to the applications. The TMR middleware together with additional middleware services, e.g. health monitoring, represents this safety layer. It is developed and maintained according to the highest criticality level of

the system's applications. Through additional monitoring of the applications' behaviour, as well as testing of the functionality used by safety-critical services, the safety layer can also mitigate faults of the triplicated lower OS and hardware layers. These lower layers are loosely coupled since the synchronization mechanism is implemented in the safety layer and only tasks of the application are replica-deterministically executed. This loose coupling of independent hardware boards and OS instances ensures strong fault containment. Consequently, the second constraint for the composable architecture is:

Constraints 2 *Embed the synchronization mechanism within the application's fault containment regions.*

Additionally, the safety layer may impose restrictions on, and require implementation of certain safety mechanisms within the application, e.g. periodically checking whether all safety-critical services are still executing or registering tasks at the health monitor after booting. All safety-critical actions are supervised with timeouts in this safety layer concept, e.g. there are timeouts for supervising the reaction time of the application or timeouts used for synchronization. The applications, together with the safety layer and underlying OS, build the FSs for our setup, which are then executed on top of the hardware infrastructure.

In the railway domain, applications or sub-systems can be classified in three different categories:

- On-board systems, located on board of the train, are limited in space, as well as power consumption as a result of restricted heat dissipation and must be resistant towards vibrations,
- Track-side systems, located along the rail tracks, are subject to harsh environmental conditions and relatively limited space, and
- General control systems, located in distant buildings equipped with server rooms for computation and possibly dedicated HMI (human-machine interface) workplaces.

We will now describe two example application tasks of the railway domain and their implications, so as to illustrate the wide range of possible application profiles.

5.1 Setting a Train Route

An example of a general control system is an interlocking system, e.g. [69]. The process of setting a train route is a typical task executed by such a system. Initially, the operator or an automatic timetabling system requests the interlocking system to set a certain train route. The interlocking system then checks the availability of all elements of the network. If all are free, it then reserves each individual element. Afterwards, it commands the switches to move into the right state and, upon success, to set the signals from red to green. If any step during this sequence fails to complete, the whole process of setting the route is aborted. This is certainly a safety-critical task, since a failure can cause a hazard with potential loss of lives, but the

timing requirements are quite relaxed from the safety point of view, since the safe system state is ensured by the sequence of operations. The required reaction time here is derived from an operability criterion and usually in the range of a few seconds.

5.2 Axle Counting

With sensors installed along the track, axle counters, e.g. [65], detect the direction of the train and, as their name implies, the numbers of axles that pass over it. Together with the topology of the railway network, it can be deduced:

- In which segments trains are on,
- How they are progressing along their defined routes, and
- Whether trains are complete, i.e. that all wagons are still in place and no axle has been lost.

Therefore, they are vital components for railway network operation. Their required reaction times, typically a few hundred milliseconds, are both a result of the operability requirement, as well as safety-related.

CHAPTER 6

System Model

The system model presented in this chapter will be extended with a detailed definition of contracts in Chapter 11.

Our architectural model is that of an integration environment (IE) consisting of IE nodes (hardware boards) and on top of each of these a composability layer is executed. This layer provides local separation of resources for FS nodes (function set nodes) deployed on the same IE node as depicted in Figure 6.1. An FS may require one or more FS nodes to provide its functionality. Each FS $\mathcal{F}_i$ is assigned an ID i, while the corresponding FS nodes are referred to as $\mathcal{F}_i^0$, $\mathcal{F}_i^1$ and $\mathcal{F}_i^2$ for FSs using TMR. Based on the targeted applications we only consider software-based TMR in our system model and subsequent analysis. Here, the hardware redundancy concept requires that per IE node at most one FS node from an FS is deployed.

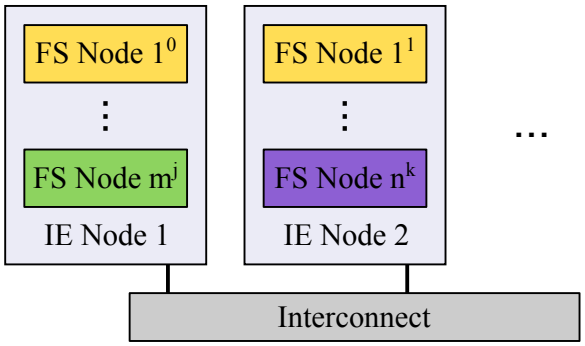


Figure 6.1: General IE node and FS node architecture.

The aim of composability is to allow more than one FS node to run on one IE node. This is in contrast to the federated architecture, where the FS nodes and IE nodes are identical, therefore not leveraging the full resources of the IE node and not providing flexibility. The resource sharing of the IE nodes introduces interferences between otherwise independent FS nodes, and specifically the need for FS scheduling within the IE as described in Chapter 4. This is a major

focus of the objectives used for the evaluation of possible solutions, which is described in the next section, followed by the definition of the scheduling parameters and the fault hypothesis. The influences of scheduling on message transmission times are presented at the end of this chapter.

6.1 Objectives

Within the system model, the objectives established in Chapter 1, predictability, performance and utilization are defined as:

1. Predictability and separation

   The predictability objective is to provide the technical basis for the feasibility evaluation method, as well as the method itself, which calculates whether or not the safety-critical applications can provide their functionality based on the resources offered by the IE. Separation (non-interference) is a pre-requisite of the technical basis.

2. Performance

   The performance objective for integrating safety-critical applications is the maximum application reaction time a_{max}. For TMR-based applications this is the time from event occurrence to reaction (output), during which the event information has to be distributed, the workload executed, the results exchanged and voting performed.

3. Utilization

   Within this domain, the utilization of the IE nodes' CPUs is most relevant, since it is the scarcest resource. The applications' bandwidth requirements for communication are typically very low compared to the available bandwidths.

6.2 IE Node Scheduling

On each IE node, the deployed FS nodes must be scheduled for sharing the CPUs. Due to the symmetric workload execution of the FSs, the FS node scheduling parameters are defined identically for all FS nodes of the same FS. Time parameters of the schedulers are defined within $\mathbb{N}_0$ and assumed to be multiples of μs. Naturally, these parameters do not contain IE-specific attributes, such as scheduling overhead. Three different scheduling algorithms are chosen for analysis: static-cyclic, preemptive fixed-priority and preemptive earliest deadline first (preemptive EDF) scheduling with constant bandwidth server. For each of these algorithms we now give a short description, motivation and their parameters in our model.

Static-Cyclic Scheduling

As presented in Section 2.5 the use of *static-cyclic scheduling* is defined in ARINC 653 for integrated modular avionics. According to the ARINC 653 definition, the scheduler cycles stepwise through a *scheduling table* to decide which partition to schedule at the current invocation. This

results in a periodic schedule, with a period equal to the duration of one full cycle though the table. In practice, this allows the defining of different (periodic) scheduling patterns for various partitions.

The available CPU time and the scheduling instances for a partition are completely predictable on an FS node local level and thereby, promise a good foundation for composability.

In our system model, static-cyclic scheduling defines that an FS node of the FS $\mathcal{F}_i$ is scheduled periodically with the period T_i for the slice duration S_i on the IE node it is deployed on. Consequently, the scheduler periodically assigns the CPU to a partition for the duration of a scheduling slice. After the designated time for this slice has been reached, the partition is preempted and the next partition scheduled. This definition is only a subset of the instance available with a scheduling table, but is sufficient for the analysis of the synchronization behaviour of our TMR mechanisms defined in the next chapter.

Preemptive Fixed-Priority Scheduling

In *preemptive fixed-priority scheduling*, scheduling decisions are taken upon arrival and completion of tasks. Each task is assigned a *priority* and the task, which is ready to execute and has the highest priority, is scheduled on the CPU. A task is preempted if another task with a higher priority arrives. The execution of the preempted task is delayed until it is again the ready task with the highest priority.

As described in Section 2.5, Liu and Layland [84] proved that fixed-priority scheduling is optimal for scheduling periodic tasks on uniprocessors, if the priorities are assigned *rate-monotonic*, i.e. tasks with shorter periods are assigned higher priorities than tasks with longer periods. Preemptive fixed-priority scheduling provides good CPU utilization and is available within most operating systems, hypervisors and microkernels. The ability to preempt is required in order to interrupt the execution of non-critical FS nodes. A composable architecture based on this scheduling strategy can choose between many different implementations for the composability layer.

In our model, P_i is the scheduling priority of all FS nodes of the FS $\mathcal{F}_i$.

Preemptive EDF Scheduling

For *preemptive EDF*, each task has a *relative deadline* and a *minimum inter-arrival time*. When a task arrives its *next deadline* is set to its arrival time plus the relative deadline. The task with the closest next deadline is executed. A task is preempted, if another task with closed next deadline arrives. After a task finishes execution it will not be scheduled until the minimum inter-arrival time has passed since its last arrival.

With the *constant bandwidth server* [2] each task is additionally assigned an *execution budget* or slice. The constant bandwidth server limits the execution time of that task within the minimum inter-arrival time to at most the execution budget. It also stops a task's execution, if it consumes more time than assigned. A task's budget is refilled after the minimum inter-arrival time has passed since the last arrival.

This scheduling approach allows to assign FS nodes a share of the CPU, which they cannot exceed. Schedulability tests for preemptive EDF on single- and multi-core CPUs exist, en-

abling to determine the feasibility of an FS node deployment on an IE node. Consequently, this scheduling strategy seems beneficial for constructing a composable system.

The scheduling parameters in our system model for all FS nodes of the FS $\mathcal{F}_i$ are defined as the minimum inter-arrival time or period $T_i^{\mathtt{E}}$, which is also the relative deadline, and the maximum execution time $S_i^{\mathtt{E}}$.

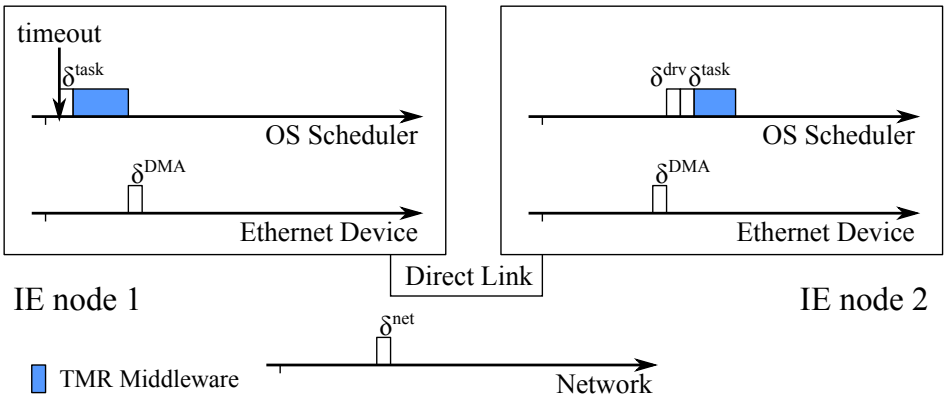
6.3 Fault Hypothesis

The safety-critical FSs in the final integrated system must be able to tolerate single transient and permanent random hardware faults on the IE node, as well as quasi-random software faults on the IE node and FS node level. These faults can exhibit arbitrary (byzantine) fault behaviour aside from being unable to forge other nodes' message signatures. Single communication link faults (transient and permanent) must also be tolerated.

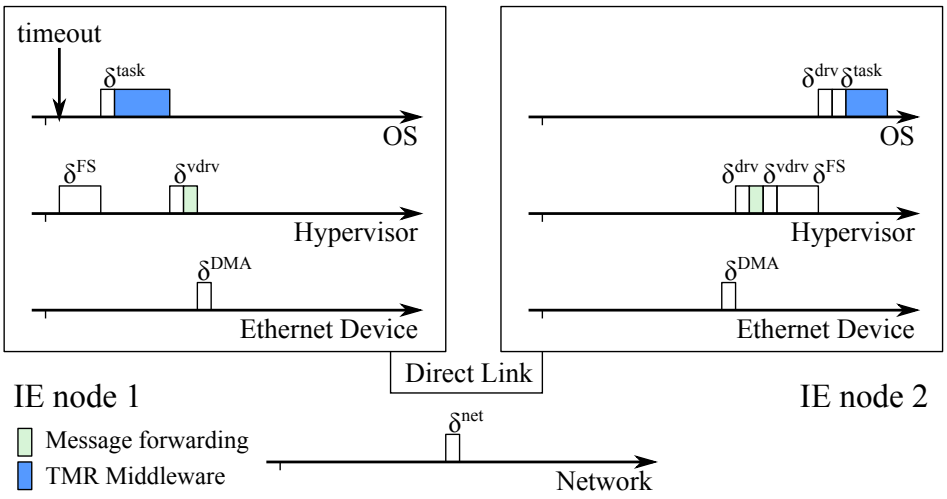
6.4 Message Transmission Times

When sending messages within an IE architecture, the duration of scheduling decisions within the IE and FS nodes are a significant contribution to the message transmission times. An example of this is sending a message from one FS node to another upon reaching a timeout for initiating the synchronization process. Regular network interface controllers (NICs) can not access the memory of virtual machines. With the exception of systems using special NICs which have this feature [52, 100], at least the following scheduling decisions must be taken:

1. $\delta^{\mathtt{FS}}$: After reaching the timeout, the sending FS node has to be scheduled by the sending side hypervisor.
2. $\delta^{\mathtt{task}}$: Within the FS, the task for sending a message has to be scheduled by the OS in the FS, which then passes the message directly to the virtual network driver.
3. $\delta^{\mathtt{vdrv}}$: The sending-side hypervisor has to schedule the virtual network driver back-end of the sending FS node, which forwards the message to the hardware network driver.
4. $\delta^{\mathtt{DMA}}$: Via direct memory access (DMA) the network driver subsequently forwards the message to the NIC.
5. $\delta^{\mathtt{net}}$: The sending NIC has to schedule the message on the network, sending it to the receiving NIC.
6. $\delta^{\mathtt{DMA}}$: The receiving NIC writes the message to the memory via DMA.
7. $\delta^{\mathtt{drv}}$: The receiving-side hypervisor has to schedule the network driver, which forwards the message to the virtual network driver back-end of the receiving side FS node.
8. $\delta^{\mathtt{vdrv}}$: The receiving-side hypervisor schedules the virtual network driver, which requests the hypervisor to map the memory containing the received message to the FS node.



(a) Creating, sending and receiving a message upon a timeout with the OS executing directly on the hardware board.



(b) Creating, sending and receiving a message upon a timeout with additional scheduling of the hypervisor layer.

Figure 6.2: Additional scheduling operations with hypervisor layer according to the system model.

9. $\delta^{\texttt{FS}}$: The receiving side-hypervisor has to schedule the FS node.

10. $\delta^{\texttt{drv}}$: The OS of the receiving FS node has to schedule the virtual network driver.

11. $\delta^{\texttt{task}}$: The OS of the receiving FS node has to schedule the receiving task.

Figure 6.2 illustrates these scheduling steps in the setups with and without the hypervisor layer. Five additional scheduling steps have to be taken, which once again highlights that the scheduling strategy of the hypervisor layer is an important factor. This depiction is the more detailed version of Figure 4.2 on page 27 according to the system model.

Depending on the setup, many additional scheduling decisions might have to be inserted into this list, e.g. soft interrupt request threads, virtual switches or splitting and merging of network frames. Part of this path in a Linux environment without virtualization has been described by Wu et al. [123]. Very costly operations with respect to time are the scheduling of FS nodes $\delta^{\mathtt{FS}}$, since the CPU core and MMU must be set up for executing the OS of the virtual machine and shadow page tables must be prepared. Depending on the current execution pattern on the CPUs, also the scheduling of the virtual driver back-end $\delta^{\mathtt{vdrv}}$ can take time. For our analysis, $\delta^{\mathtt{DMA}}$, $\delta^{\mathtt{net}}$ and $\delta^{\mathtt{drv}}$ will always be assumed to have no significant influence, since the message transmission time from the memory of one IE node to the other and the scheduling of the interrupt is negligibly small for the workloads of our example applications from Chapter 5. They send a limited amount of KiB-sized messages via a full-duplex gigabit Ethernet network with at most one switch between the participating nodes. Likewise, $\delta^{\mathtt{task}}$ is also omitted, since our receiving (synchronization) task in the FS node has the highest priority and the scheduling latency within the FS node is significantly smaller than scheduling the FS node on IE node level.

CHAPTER 7

Synchronization Mechanisms

In this chapter, we present the synchronization mechanisms subject to our analysis. Synchronization establishes a consistent view of the environment (input) and system state (time, membership and output). It is the foundation of replicated computation and voting. The achievable maximum reaction time a_{max} of a TMR application is strongly dependent on the used synchronization mechanism and the maximum workload computation time w_{max}.

In accordance with our targeted applications, we consider periodic synchronization for IEs with a classic TMR architecture in Section 7.1, as well as for dynamic IEs with a redundant network in Section 7.2. Additionally, a synchronization mechanism based on virtual synchrony is presented for the latter IE in Section 7.3. This is an approach used in cluster technology [38]. In Afterwards, we outline in Section 7.4 how the layer used for synchronizing influences the synchronization precision. For the general descriptions, we assume that the synchronization algorithms are executed directly on the IE node and no FS nodes are present.

7.1 Periodic Synchronization

In our railway example of Chapter 5, periodic synchronization, with a *synchronization period* T_s, is the TMR synchronization mechanism currently provided by the middleware to the targeted applications, which supports software lock-step TMR, as well as software incremental TMR. In our general description of the mechanism, we focus on periodic TMR synchronization for software lock-step TMR, since it has stronger timing requirements towards its environment and is easier to illustrate than software incremental TMR. Furthermore, we have three fully connected hardware boards as our three fault containment regions and thereby, (IE) nodes.

Software-based TMR must ensure a replica-deterministic execution for the replicated workload processed in the fault containment regions, as well as provide fault detection and isolation [97]. Using periodic synchronization, the operation is split into two phases as shown in Figure 7.1. During the periodically initiated *synchronization phase*, the three nodes exchange messages on the system state, determine the membership state of all nodes, vote on the repli-

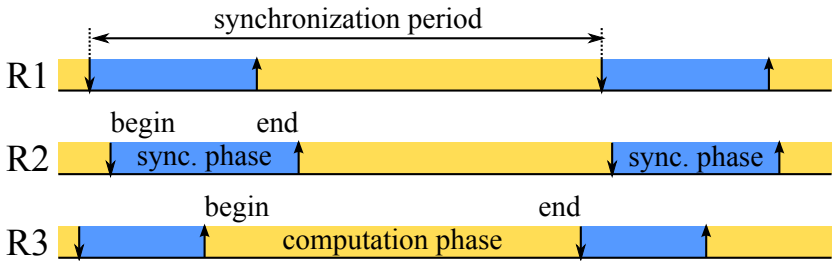


Figure 7.1: Periodic synchronization of software lock-step TMR.

cated input data and advance the logical synchronous time used for replicated computation. The *computation phase* is used for calculating the new output based on the replicated input.

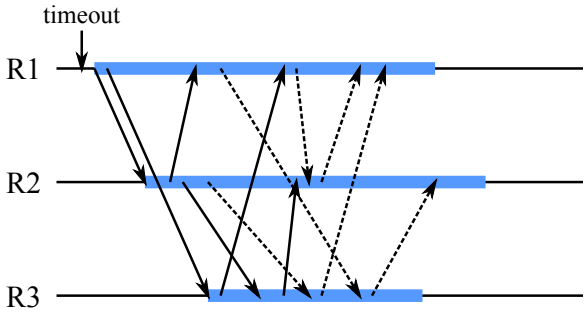


Figure 7.2: Example message exchange during synchronization phase.

More specifically, the synchronization phase for each node starts with initiation of sending a (synchronization) message[1]. This occurs either due to the synchronization phase start timeout or the reception of a message from one of the other nodes. The timeout for the next synchronization phase start is adapted accordingly, so as to account for clock drifts[2]. Each node sends out its state as signed messages to the others (solid arrows in Figure 7.2), and forwards such directly received messages (dashed arrows), with its own signature appended, to tolerate link faults and byzantine faults of individual nodes[3]. As soon as all direct and forwarded messages have been received (or a timeout is reached), the node performs membership and voting. Afterwards, it switches to the computation phase again. This phase switch can also be defined relatively to the synchronization phase start to provide constant (albeit less) computation time. However, our targeted applications do not make use of such a mechanism.

The example depicted in Figure 7.2 illustrates the influence of the message transmission times between the nodes on the synchronization phase duration. In fact, membership and voting need only a fraction of the time necessary for message exchange. For each of these mes-

[1]We consider the preceding scheduling operations as part of the synchronization phase, but not the duration between timeout occurrence and the start of this scheduling operation.

[2]The safety layer monitors local clock drift independently.

[3]The method of tolerating byzantine faults with (unforgeable) signed messages has been presented in the original paper on the byzantine generals problem [80].

sages, many scheduling steps might be necessary as outlined in Section 6.4 on page 38. These steps introduce jitter in message transmission times and subsequently increase the jitter of the synchronization phase duration. In doing so they also increase the maximum duration of the synchronization phase.

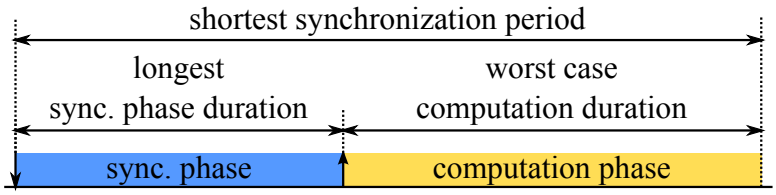


Figure 7.3: Composition of the minimal synchronization period.

The maximum duration of the synchronization phase together with the worst-case execution time of the replicated workload is the shortest possible duration of the synchronization period for software lock-step TMR, which in turn is a requirement imposed by the environment. Figure 7.3 illustrates the composition of such a minimal synchronization period.

Fault detection

Depending on the resulting behaviour, faults occurring in single nodes are detected during different tasks of the synchronization period. An erroneous output produced by a node is detected during voting in the synchronization phase. For detecting crash faults or loss of messages, a timeout for finishing the synchronization phase limits the reception of messages. This timeout is derived from the "path" taken by the longest message sequence in presence of a single link fault. In the worst-case scenario, there is a sequence of four messages, where the node at one side of the broken link starts the synchronization phase with a message sent upon a timeout and the one on the other side starts it with the reception of a message. A node is excluded, if its messages are not delivered to the other nodes before the timeout for finishing the synchronization phase has expired. The correct setting of this timeout is crucial for the stability of the periodic TMR synchronization.

Metrics

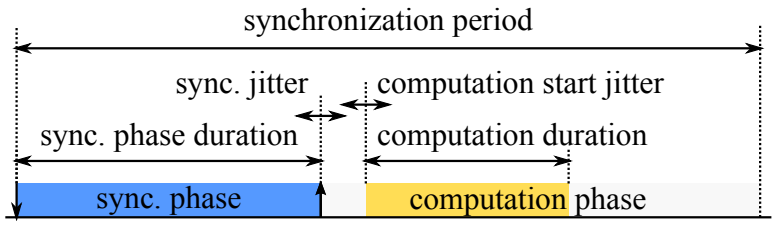


Figure 7.4: Properties used as metrics for periodic TMR synchronization.

Figure 7.4 illustrates the properties and parameters essential for the periodic synchronization service provided to the application, and consequently, its reaction time. The elements are: the synchronization phase duration s, the computation phase duration c, and the respective synchronization jitter s^j and computation start jitter c^j. This computation start jitter originates in the task switch from the TMR middleware to the application and is typically very low, since the task switch is a local OS level scheduling step. The maximum synchronization phase duration s_{max} increases with s^j ($s_{max} = s_{min} + s^j$), while both s^j and c^j reduce the minimum available computation time

$$c_{min} = T_s - c^j - s_{max},$$

where the minimum delay until the computation starts after the end of the synchronization phase is still defined as part of the synchronization phase. Contributors to the jitter s^j in an architecture where the synchronization mechanism is executed on the IE node level is the scheduling of tasks $\delta^{\texttt{task}}$, drivers $\delta^{\texttt{drv}}$, memory access $\delta^{\texttt{DMA}}$ and network transmission $\delta^{\texttt{net}}$. These scheduling steps are depicted in Chapter 6, Figure 6.2(a) on page 39 for sending and receiving of one message. The costly $\delta^{\texttt{FS}}$ and scheduling of virtual driver back-ends $\delta^{\texttt{vdrv}}$ are introduced with the composable architecture, which are illustrated just below the previous example in Figure 6.2(b) also for sending and receiving of one message.

If only one computation phase is necessary for generating the new output in response to a given input, the worst-case reaction time is two synchronization periods plus one synchronization phase

$$a_{max} = 2T_s + s_{max}.$$

The scenario here is that the input event occurs right after a synchronization phase has been initiated, and thus, the information about the event is distributed only after the first period has passed, in the next synchronization phase. The result is then voted in the subsequent synchronization phase and the new output forwarded after its completion.

The application designer's task is to find a synchronization period suitable for the required reaction time and providing sufficient computation time for the application. A high jitter s^j causes a long maximum synchronization phase duration s_{max}, and consequently, requires a long synchronization period to provide enough computation time in the worst case. Therefore, keeping s^j low is essential.

Workload Execution

Figure 7.5 compares workload execution in software lock-step TMR and software incremental TMR. In this example, the replicated workload execution has to be finished within four synchronization periods after the input event occurred in order to fulfill the reaction time requirement.

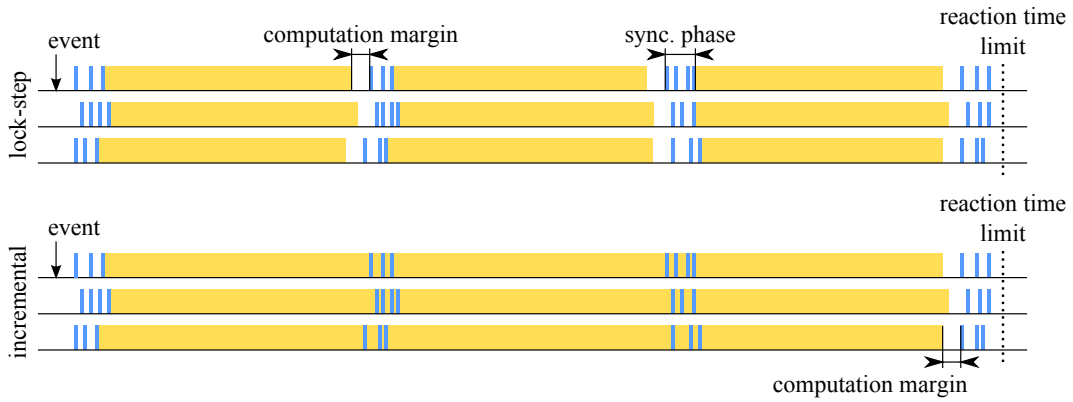


Figure 7.5: Workload execution pattern for periodic synchronization with lock-step and incremental TMR.

In the illustration, the according reaction time limit is aligned to the closest end of the synchronization phase that still satisfies the reaction time requirement. The long rectangles depict the theoretically available computation time for applications while the narrow ones illustrate the scheduling of the synchronization task in this scenario. Herein lies the main difference of both approaches, as it is possible for the incremental TMR to compute while a synchronization phase is ongoing. It may only be interrupted by the synchronization task itself, which is then handling messages and performing membership and voting. Therefore, the computation margin for ensuring completion is only required at the last synchronization period before the reaction time limit is reached in software incremental TMR. This additional computation time can only be used by the application if tasks are active, i.e. do not have to wait for voted results. Consequently, the available computation time also depends on the application's data flow. Also, the ratio of synchronization phase to computation phase and according margins vary strongly with the length of the synchronization period, such that in some cases these margins and idle times of computation are negligible for the overall performance.

Recalling the two example applications from Chapter 5, one could use software incremental TMR for setting the train route and through this enable more relaxed low-level timing requirements in the execution, while software lock-step TMR might be used for axle counting. It remains for the application developer to choose.

7.2 Periodic Synchronization for Redundant Networks

For the dynamic IE, where a redundant network is provided for periodic synchronization, the algorithm must be adapted. Two separate switches, each of which connected to all IE nodes, is a simple example of such a redundant network (see Figure 4.3(c) on page 29). The forwarding of messages is no longer required in order to tolerate single link faults with the redundant interconnect. In this case, the shift to a monitoring architecture is necessary for tolerating byzantine faults. Employing such a FS node local architectural separation of message creation and communication supervision allows the mapping of byzantine faults in one of the two components to

either fail silent or babbling idiot faults on the local node. This is a weaker fault hypothesis, as only part of the node can be byzantine faulty. Thus, a fault can result in either

- Erroneously created messages, which are filtered by the communication supervisor, or
- The supervisor itself is faulty and keeps retransmitting correct (unforgeable) signed messages or messages with invalid signatures to the network.

In the latter case, the FS bandwidth constraints enforced by the IE limits the effects of the babbling idiot for the whole network. Additionally, the other FS nodes are able to detect these duplicate or invalid messages and can trigger a safety reaction.

Using this monitoring architecture, the message forwarding step of the periodic synchronization can be eliminated, thereby shortening the duration of the synchronization phase. The resulting message exchange is that of the synchronization mechanism presented by Miller et al. [88]. Moreover, the metrics and resulting maximum reaction time a_{max} are the same as for the original periodic synchronization mechanism with a shorter synchronization phase.

7.3 Token Ring Synchronization with TSM

An alternative approach to periodic synchronization and subsequent ordering of messages for replica determinism is to define a total order on the FS's *data messages* using virtual synchrony [25]. One method for achieving virtual synchrony is to create total message ordering by constructing a logical token ring and allowing only one node at a time, the token holder, to assign IDs to its messages. Only local (unsynchronized) timeouts are used for error detection and the membership service. As opposed to the periodic synchronization, it does not require the messages to be exchanged simultaneously. This enables resending of lost messages or tokens and the network load is distributed more evenly, overall promising a more robust synchronization method in the composable environment. Examples of token ring protocols achieving such a total message order are the "Totem Single-Ring Ordering and Membership Protocol" [8], or the "TPM Protocol" [101]. Both protocols are designed to handle network partitioning, which is not tolerable in the case of fail-safe safety-critical applications. Consequently, the protocol required for our TMR synchronization can be simpler. The RTCast protocol [1] claims to have fast message distribution times, but its fault hypothesis and availability of individual nodes is not suitable for our approach. This is why we introduce a new token ring protocol, TSM (Token Synchronized Messages), which aims for short (re-)integration times of individual nodes and follows an "instant delivery" strategy. Still, it is largely based on the concepts and algorithms of the former protocols.

We start by describing the membership and message ordering algorithm, followed by the instant delivery mechanism. Then we outline the strategy for fulfilling the fault hypothesis. Finally, we show how TMR can be implemented on top of TSM and the according metrics for its performance. We will not distinguish between elements and their identifiers, if the meaning is clear from the context.

Membership

The TSM operation is separated in *membership phases* and *operational phases*. Nodes, phases and messages are all assigned identifiers. Each node has a list of all possible members. Operation starts with the initial membership phase. During a membership phase, nodes exchange *join messages* containing a suggested set of members, a *membership phase ID*[4] and their current state regarding received and delivered data messages. While in this phase, all nodes set a local *membership timeout* relative to their last sent join message, which triggers the sending of a new join message. Nodes, of which join messages have not been received before the timeout expired, are excluded from this node's suggested member set in the newly sent join message. When a node receives a join message with a higher phase ID, it either answers by sending out a join message with the same members and phase ID, or one with a new phase ID and extended member set. The latter is the case, if some of the nodes it already received join messages from within its current phase, are not included in the join message with the higher phase ID. Agreement on the member set and phase ID is reached, if a node receives the join messages exactly from all its suggested members, containing the same set of members and phase ID. It then switches to the operational mode. Receiving a *token message*, containing the same set of members and an operational phase ID equal to the node's current membership phase ID, also triggers this mode switch. Network partitioning is avoided by requiring that the member set sufficient for this mode switch consist of more than half of the nodes. When a node switches to operational mode, it checks whether it is the one with the smallest *node identifier* of the member set. In that case, it generates a token message and sends it to the node with the next higher identifier. This token contains the membership phase ID as the new operational phase ID, the member set, the next message *sequence number*, a token ID, the IDs of sender and receiver, and condensed information on all members' received messages. Membership phases are also started when a non-member node sends a message, or a *token loss timeout* occurs.

Message Ordering

During the operational phase, a node receiving the token can send out a data message containing the sequence number provided in the token. This sequence number is incremented when such a message is sent, defining a total order for all data messages. The token ID is incremented when a node forwards the token, in order to distinguish between token messages when no data messages are sent. Gaps within the data message sequence numbers occur only when nodes fail and the corresponding messages are lost. These gaps are detected during the following membership phase, where the failed nodes are excluded from the member set.

Instant Delivery

TSM nodes buffer all received data messages so as to provide their copy in case other nodes do not receive them. Each node updates its received message state in the token upon forwarding it. When receiving a data message, the node can instantly determine whether it has received all data messages preceding the current one. This is based on the total order of sequence numbers and the

[4] Identical to the ring identifier in [8].

detected gaps of data message IDs during the membership phase. In this case, the data message, and all so far received follow up messages, are immediately delivered to the application. If a node misses a message, it can request a resend from another node based on the received data message information contained in the token. These requests are sent after forwarding the token and are handled by the buffering node completely independently from the token processing. Messages, which have been received by all nodes according to the token information, are removed from the local buffer.

Supervision Architecture

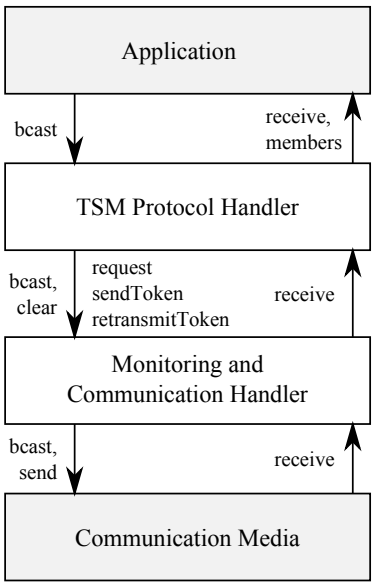


Figure 7.6: Monitoring architecture of TSM.

The concept of the supervision architecture is analogous to the supervision of periodic synchronization for redundant networks presented in Section 7.2. Figure 7.6 illustrates the TSM monitoring architecture on each node. TSM is connected to the application (AP) via an interface for message exchange and membership information, whereas only signed messages are sent via the communication media (CM). The TSM protocol itself is split into two modules: the “TSM Protocol Handler” (PH) and the “Monitoring and Communication Handler” (MCH). The PH is generating all relevant protocol messages and signs them before passing them to the MCH. The MCH, in turn, checks these signatures and adds an additional signature to the message.

Due to the simplicity of the TSM protocol, the MCH monitoring entity can check the validity of each message generated by the PH. If it is detected as incorrect, it is not forwarded to the CM, which is perceived as a “fail silent” behaviour by the other nodes. The MCH must also detect too early timeouts, e.g., to prevent a PH from continuously sending join messages and thereby, preventing a switch to the operational phase. If, on the other hand, the MCH is faulty, its only

options are to either erroneously suppress correct messages received from the PH or CM, or send multiple copies of it[5]. The latter "babbling idiot" behaviour is easily detected by the other nodes or the PH and its effects on the shared network limited through the bandwidth constraints enforced by the IE, just as for the supervision of the periodic synchronization for redundant networks.

TMR on TSM

TSM fulfills the properties of FIFO Atomic Broadcast as specified by Hadzilacos and Toueg [57]: Validity, agreement, integrity, FIFO order and total order. Together with its membership service and the aim of short interruption times when nodes fail or are reintegrated, it provides a good foundation for a TMR middleware. The remaining required features are voting and a safe synchronous time.

The synchronous time is constructed using local clocks and periodically sending out time messages as TSM data messages. The timestamps contained in these messages are used to derive a synchronous time via voting for all other messages in the ordered stream. Timeouts for the application are also derived from this synchronous time and delivered locally, when the synchronous time progresses. Consistency for the application is given, since the time information is also based on the total order message stream. An additional time supervision is required for the synchronous time to ensure that its rate is sufficiently close to the real-time. Either the synchronous time is then fed back to the local clocks to keep them relatively close together for logging, or a different service, such as NTP [90], can be used.

Voting on regular data messages is performed on the totally ordered message stream and timeouts for voting are derived from the synchronous time. The voted message is delivered when a sufficient subset of correct messages is received before the voting timeout is reached. Otherwise, a voting error is forwarded to the application.

Metrics

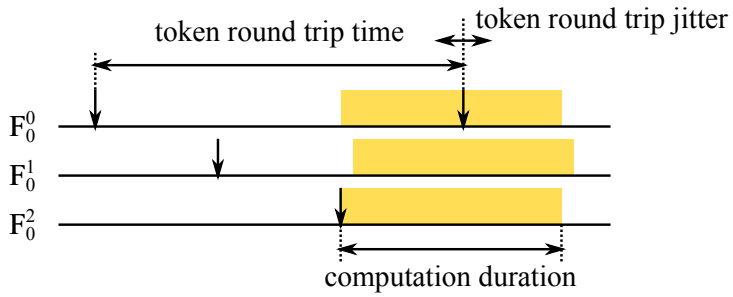


Figure 7.7: Properties of TSM-based TMR synchronization.

Figure 7.7 illustrates the key properties for the TSM-based TMR synchronization during fault-free operation. The arrows represent the reception of the token, while the messages sent

[5] Join messages with a phase ID lower than the current one are ignored, just as any other duplicate messages.

upon reception of the token are omitted in the depiction. In the example, the computation starts as soon as the FS node has received the last message, which is sent by $\mathcal{F}_0^2$ after it received the token.

The first key parameter for the reaction time achievable with this synchronization mechanism is the token round trip time r, i.e. the time it takes for the token to circle through all participating FS nodes. On a node local level it is the time between two token receive events. Also, the token round trip time jitter r^j determines the synchronization quality. Assuming that the computation time required for calculating an event reaction is feasible within one token round trip and message transmission times are symmetric, then the maximum reaction time in the fault-free case is

$$a_{max\_nf} = \frac{8}{3} r_{max}.$$

In this scenario, similar to the periodic synchronization, the event occurs just after the token has been forwarded at the node originally notified of the event. The event information is then distributed at the next token reception after one r_{max}. Distribution of results starts with the token round trip after that ($2r_{max}$) and finishes before completing the full round after two token forwards at $2r_{max} + \frac{2}{3} r_{max}$.

Another important characteristic is the regular operation interruption time g when a node fails. With TSM, this it is the time duration from the last successful token processing of a node until the next non-faulty node in the original sequence receives the new token after the membership phase. This is strongly dependent on the token loss timeout and membership timeout. The token loss timeout must be at least r_{max}, while the membership timeout has to be at least twice the message transmission time (m) plus the computation necessary to reply. In case of regular operation interruption, the worst-case scenario is when the node with the lowest ID of the member set crashes. Here, the node with the next higher ID, i.e. the next non-faulty node in the original sequence, emits the initial token after membership and receives the new token itself only after a full round (with one node less in the member set). This results in a worst-case interruption time of

$$g_{max} = r_{max} + 2m_{max} + 2m_{max}.$$

Consequently, the worst-case response time during fault occurrence is the sum of the worst case for the fault free reaction time and operation interruption

$$a_{max} = a_{max\_nf} + g_{max}.$$

Based on the maximum message transmission time m_{max} and assuming zero time computation, we can compare the reactions times of the periodic synchronization mechanisms and TSM. With $r_{max} = 3m_{max}$, we get a worst-case reaction time of

$$a_{max}^{\texttt{TSM}} = a_{max\_nf}^{\texttt{TSM}} + g_{max} = 15m_{max}$$

for TSM. While the worst-case reaction time of periodic synchronization with direct links is

$$a_{max}^{\texttt{ps}} = 2T_s^{\texttt{ps}} + s_{max}^{\texttt{ps}} = 12m_{max}.$$

Naturally, periodic synchronization is significantly faster without the message forwarding step in the redundant network, although the message transmission times themselves are longer due to the additional network infrastructure

$$a_{max}^{\texttt{rps}} = 2T_s^{\texttt{rps}} + s_{max}^{\texttt{rps}} = 3m_{max}.$$

These values should just give a first idea of the possible reaction times. We later evaluate the performance of the synchronization mechanisms in detail for our composable environments in Chapters 9 and 10.

Although interruption of message ordering and distribution caused by a node failure and subsequent membership phase is short in TSM compared to the existing token ring protocols, it still has a significant influence on the worst-case reaction time. Just as the other synchronization algorithms, TSM is strongly dependent on the message transmission times of the chosen architecture. TMR on top of TSM provides a robust middleware for triplicated applications. With the message buffering and request mechanisms, TSM is also suited for networks with high packet loss rates. Rather than immediately triggering a safety reaction when a packet is lost on the redundant interconnect, packets are resent and other high-level timeouts are used for fault detection and reaction[6]. Also, (re-)integration of a node in the membership is fast given that it requires less than one membership timeout.

[6]The comparison of reaction times above uses the shortest timeouts and thereby, has no margin for token resend.

7.4 Influence of the Layer of Synchronization on Synchronization Precision

We define the synchronization precision as the maximum offset in real time when the replicated workload's execution starts. A good synchronization precisions allows timely execution of the workload and consequently, short reaction times. When considering the definition of the synchronization precision, one factor is significant for the presented synchronization mechanisms: The message transmission times, including scheduling operations for message delivery. The drift of the local clocks has no influence in this definition, as timeouts only initiate synchronization phases for the periodic algorithms and do not trigger sending of data messages in TSM.

The layer in which the synchronization mechanism is located strongly influences the message transmission times. The synchronization precision increases with the decrease of synchronization jitter, which in turn decreases as the jitter of the message transmission times gets lower.

Implementing synchronization in a low-level kernel interrupt routine will consequently achieve a better synchronization precision than synchronizing within processes. For the latter case, the additional scheduling latency is the main contributor to the message transmission time. Synchronizing within virtual machines increases this scheduling latency even more, as illustrated with the scheduling steps defined in the system model and illustrated in Figure 6.2(b) on page 39. Generally, the lower the layer providing the synchronization mechanism is located, the better the synchronization precision is. However, implementing the synchronization mechanism within the FS provides better separation between the FS nodes of an IE node. It also ensures loose coupling between the IE nodes and thereby, fault containment regions, and allows each FS to follow its own synchronization strategy.

CHAPTER 8

Proposed Concepts for Composable Solutions

The introduction of composability affects most aspects of safety-critical systems. Not only the certification process and system architecture, but also test procedures, configuration methods and many other elements and tasks have to be adapted accordingly. Thus, it is of the utmost importance that the benefit of switching to a composable approach justifies the overall effort. Based on our targeted applications, we first suggest strategies for the composability layer and the respective safety concept and certification approach. Afterwards, we present two possible architectural solutions. The technical aspects of these solutions are evaluated in detail in the Chapters 9 and 10, and certification based on contracts in Chapter 11.

8.1 Virtualization as Composability Layer

Virtualization is a technique for providing separate virtual machines to several operating systems, called guests deployed on the same hardware. As described in Section 2.2 on page 6, hypervisors provide these virtual machines by only interfering with a few of the guest operating systems' instructions, i.e. those that would change the hardware state for all guests. As for composability, isolation between the guests and performance guarantees are both central features of hypervisors. Adopting this technology as a composability layer allows the use of a wide range of hypervisors, some of which have a large user base and are thereby, extensively tested. Also, bundling the existing applications with their operating systems together in virtual machines leads to requiring only a confined set of functionalities of the composability layer, since most low-level services are still implemented within the guest operating systems.

This approach naturally introduces a discrepancy between the IE node's and FS node's view of the system and of its state. Using virtualization for composability, the context of applications and services executed on an FS node are encapsulated within the virtual machine. The host system and guest operating systems of the virtual machines interact via abstractions of hardware

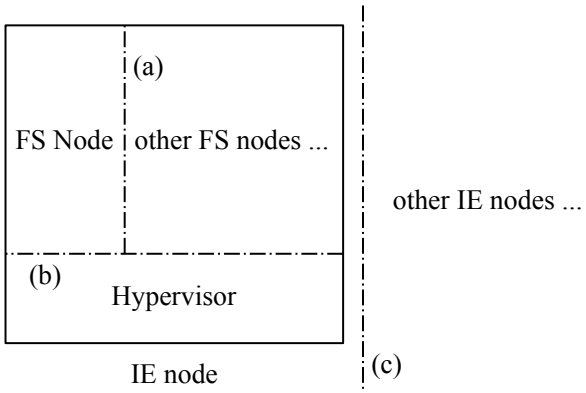


Figure 8.1: Limits of separation in a composable system with hypervisor.

mechanisms. This introduces a semantic gap, since neither the host, nor the guest can infer the "full" system state [32]. Figure 8.1 illustrates this with the border (b) between hypervisor and FS nodes. This is a very beneficial property for isolation, with a strong restriction of the possible influence of the hypervisor, but makes scheduling of such synchronized FSs very difficult. A virtual machine going to sleep might just be waiting for the next synchronization message within a few μs or it has just finished the replicated workload computation and is waiting for the start of the next synchronization phase in several ms. Alternatively, it might also have another timeout set for non-replicated I/O. This is not visible to the scheduler in the IE node.

Closing a part of this semantic gap has been achieved in the case of memory usage for server virtualization with the ballooning technique [119]. Using a special interface to the hypervisor, guests can request and deallocate memory as needed. Through this mechanism, the host knows which memory pages are not needed, and consequently, will not swap them out to the disk in case of memory overcommitment. Naturally, memory overcommitment and ballooning have unfavourable properties regarding non-interference to begin with, and are, therefore, avoided for safety-critical FSs. Other methods have been proposed for scheduling and exposing the guest state to the host [78, 117], as already outlined in Section 2.2

Not only the semantic gap hides information in this setup, as illustrated in Figure 8.1. Ideally FS nodes cannot influence each others behaviour in any way, which is depicted as border (a), being the limit of failure confinement. Border (b) is that of the semantic gap, and the boundary for fault containment is border (c) between the IE nodes.

The key criteria for selecting the right hypervisor are the performance and predictability the TMR workload execution and synchronization can attain on top of it. Synchronizing the schedulers of the hypervisors [94] is functional tightly coupled to the used hardware, system architecture, and naturally, the hypervisor. This limits the overall flexibility of the integrated system. Additionally, it might compromise the general separation properties offered by the hypervisor and the fault containment properties achieved with the loose coupling between the IE nodes, i.e. border (c).

Preemptive fixed-priority scheduling is provided by all commonly available virtualization environments, thus, constructing an IE on this basis allows to choose between a wide range of

preexisting solutions. Furthermore, priority-based scheduling is also preferable for the utilization of CPU resources by non-critical FSs, since they can access resources which would be kept idle with other (composable) scheduling strategies, such as static-cyclic scheduling. Nevertheless, static-cyclic, as well as EDF scheduling are widely used in embedded systems with hard real-time requirements due to their predictable behaviour. Consequently, we explore solutions with static-cyclic, priority-based and EDF scheduling, which we also stated in Section 6.2.

8.2 Safety Layer

There are two options for safely adopting a (COTS) hypervisor as composability layer. The first is to have a certified hypervisor, which already provides a safety concept and clearly states its requirements for safe operation. The second is to be able to tolerate all faults introduced by the hypervisor.

The safety concept of our targeted applications, described in Chapter 5 on page 31, uses the second option with the safety layer strategy for integrating the operating system and hardware. Therefore, reapplying this concept is a continuation of the philosophy followed until now and is the concept suggested for the solution. It requires that the faults introduced by the hypervisor must be detectable and partly tolerable by the (distributed) middleware, just as faults for any other layer below the safety layer with the current approach. This includes random, as well as systematic faults. One approach here is to conduct a rigorous failure mode and effects analysis (FMEA) [110] regarding the generic features and properties of the hypervisor. If, based on this FMEA, measures are found to mitgate all possible hypervisor faults, the existing safety layer concept can be extended for the hypervisor as well. The task of performing such an FMEA gets simpler or even just feasible, if both the complexity of the functions required from the hypervisor layer and the possible influence between the hypervisor instances in different fault containment regions are low. As mentioned above, synchronizing the hypervisor schedulers is an approach that weakens the fault containment regions due to the introduction of common cause scenarios. This most likely results in a system where the safety layer strategy is not applicable, and a certified hypervisor needs to be used.

A general strategy for the safety layer to detect faults in the composability layer can be to always twofoldly exchange information through it via different interfaces. This is an approach similar to diversity [46]. The key assumption here is that a fault will not affect the two information paths in the same way. An example would be a FS manager instructing the composability layer to start an FS node on a certain hardware board with a virtual disk containing its configuration and application. After deployment and booting, the FS node registers itself, its configuration and application at the FS manager with a network message. Thereby, two very different interfaces are used to exchange the FS information through the composability layer, the interface for starting FSs and the networking interface. Similar approaches are applicable for clocks, watchdogs and other resources.

Another class of faults is introduced with composability due solely to the existence of data, code and configuration of several safety-critical FSs within the same system, which are protected via the same safety mechanisms. One example scenario is that a FS's application might recognize another FS's configuration as valid input. The new safety concept must guarantee that such

a situation cannot occur. One approach is to use unique identifiers and signatures to ensure that within an FS only the corresponding application, configuration and data are identified as correct by the safety layer.

Provided the mean time between failure is acceptably high, the availability guarantees for individual FSs and the overall integrated system can be reached. However, the safety layer alone is only one part of the overall safety concept. The other is contract-based certification.

8.3 Contract-based Certification

By splitting the certification process in three phases, as presented at the beginning of Chapter 3, FS, IE and the integrated system are certified independently. This approach is essential for our composable solution. The goal is that the contracts are machine-readable and the composition contract autonomously constructable by a safety-critical entity, i.e. the FS manager, using the individual FS contracts, as well as the IE contract. For IEs which support dynamic adaptations, additionally alterations of the IE contracts must be supported to adapt the IE contracts in accordance with the changes.

The resource representation and feasibility evaluation of the final composed system must safely reflect the behaviour of the actual IE and FSs, which is a central aspect of the analysis conducted in the next three chapters.

To support application developers, methods for FSs must additionally be defined within the contract-based certification concept so as to evaluate whether the resources specified in the FS contract are sufficient for their execution. The safety layer presented above can detect insufficient resources and violations of the contract only during execution.

For the machine-readable contract format, we suggest the use of normalized quantities for resource specifications regarding CPU, Memory, Disk and I/O, and predicates for individual features provided or properties fulfilled by IEs and FSs alike. Chapter 11 contains a description of our contract concept.

8.4 Composable Architectures

Within the range of possible solutions we look at two extremes. The first one is to keep the classic TMR architecture and deploy several FSs within it, with as little adaptations as possible. In this approach, the hardware setup and synchronization method are kept the same or very close to the original mechanisms. This architecture and its corresponding behaviour of the middleware are the focus of the detailed analysis in Chapter 9.

Moving away from the classic TMR architecture to architectures with several nodes requires a change of the network structure as discussed in Section 4.2 on page 28. As can be seen, the use of direct links for fully connecting all nodes is not efficient for many nodes. Since the fault tolerance concept of the synchronization mechanism used so far relies on these direct links, it must be adapted. This is a major change within the middleware layer, thus, our second proposed architecture – at the other end of the scale – is the dynamic TMR-composable architecture with adaptable IE to justify such an extensive change. The analysis regarding this architecture and middleware is described in Chapter 10.

CHAPTER 9

IE with Classic TMR Architecture

The classical software-based TMR architecture consists of the three hardware boards with directly connected dedicated links for synchronization, as presented in Section 4.1 on page 23. Providing composability and mixed-criticality in this architecture with static FS node to IE node assignment enables the reuse of existing infrastructure with higher utilization while getting the benefits of the step-wise certification process as already outlined. It is suitable for systems with space limitations and no need for dynamic changes. Considering the targeted applications from Chapter 5 on page 31, on-board systems and track-side systems are potential candidates.

We now discuss the influence of different IE node-scheduling strategies on the synchronization and application performance in this architecture. For comparing the impact on CPU utilization we consider two applications. The first one, deployed in the FS $\mathcal{F}_{10}$, requires at most 10ms of replicated workload execution, 2ms for processing synchronization messages (including voting) and a synchronization period of 100ms. The second application, deployed in FS $\mathcal{F}_{20}$, requires 20ms for the worst-case replicated workload, 2ms for the synchronization process and a synchronization period of 50ms. Based on the measurements in Section 12.2 on page 93 the cost of scheduling an FS node is assumed as 1ms. Without composability, the application of $\mathcal{F}_{10}$ has a utilization of less than 15 percent of the hardware boards' CPUs, and the application of $\mathcal{F}_{20}$ of 44 percent.

Since direct links are used for communication, only periodic synchronization is considered in the analysis.

9.1 Static-Cyclic Scheduling

The schedule for an FS node of $\mathcal{F}_i$ is defined for static-cyclic scheduling with the scheduling period T_i and scheduling slice S_i. Static-cyclic scheduling minimizes the variability of resource access times on a local level. Due to predetermined static scheduling slots provided each period to an FS, cache load and similar effects can be determined and bounded with suitable FS tests simulating an IE under maximum load.

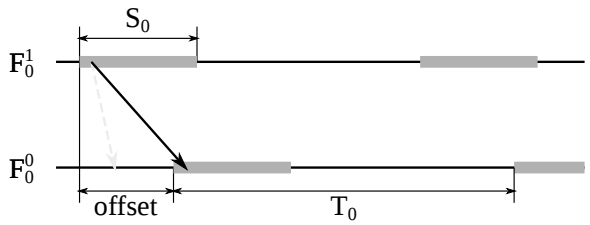


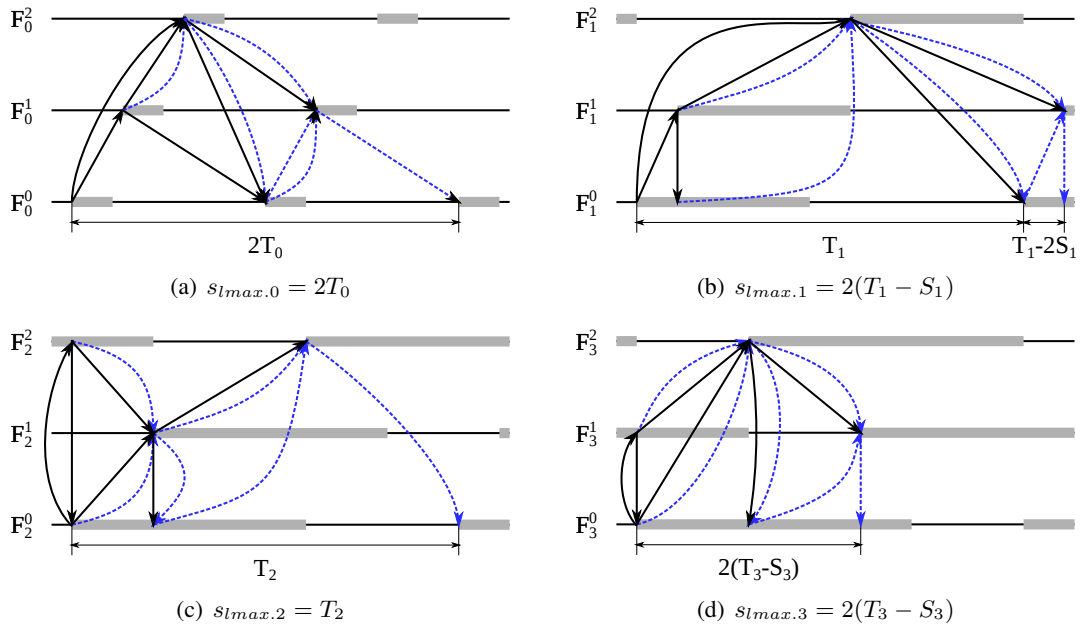
Figure 9.1: Properties of static-cyclic schedulers affecting message transmission times.

Three properties of the static-cyclic schedulers now affect the message transmission times between the FS nodes: the scheduling period, scheduling slot and the relative offsets between the slots (as we do not synchronize the schedulers due to Constraint 2 from Chapter 5). An example is depicted in Figure 9.1, where the dashed arrow illustrates the message transmission, in case message reception is scheduled immediately. The solid arrow shows the message transmission as perceived with the static-cyclic scheduled FS nodes. Since the synchronization phase duration depends on these message transmission times (see Section 7.1), static-cyclic scheduling limits the minimum synchronization period achievable with the periodic TMR synchronization algorithm. Depending on the ratio between T_i and S_i, the lower bound for the maximum (local) synchronization phase duration $s_{lmax,i}$, with the assumption that message transmission and computation times are negligibly small and the full slice is available for computation and communication, is

$$s_{lmax,i} = \begin{cases} 2T_i & \text{if } 3S_i \leq T_i \\ 2(T_i - S_i) & \text{if } 2S_i \leq T_i < 3S_i \\ T_i & \text{if } \frac{3}{2}S_i \leq T_i \leq 2S_i \\ 2(T_i - S_i) & \text{if } T_i < \frac{3}{2}S_i \end{cases} .$$

This is a result of the different offsets of the local scheduling periods (as we do not synchronize the schedulers due to Constraint 2 from Chapter 5), and the forwarding of messages to tolerate link faults and byzantine faults of replicas.

Figure 9.2 illustrates the different scenarios for the worst case with zero computation and message transmission times, where solid lines represent direct messages and dashed lines forwarded ones. In the worst scheduling scenarios with $3S_i \leq T_i$, on the node observing this synchronization phase duration, the first scheduling period is needed for receiving all but the last forwarded message, which is received after the second period (Figure 9.2(a)). For the worst-case schedules with $T_i < 3S_i$, we can distinguish three cases. In the first two, with $\frac{3}{2}S_i \leq T_i < 3S_i$, one scheduling period is needed for scheduling all FS nodes to send the initial messages and $T_i - 2S_i$ is the additional worst-case delay for forwarding the last messages (Figure 9.2(b). If $T_i - 2S_i < 0$ all forwarding steps also happen during the first scheduling round and only receiving is delayed until T_i (Figure 9.2(c)). This results in a maximum synchronization phase duration of $2(T_i - S_i)$ when $2S_i \leq T_i < 3S_i$ and T_i when $\frac{3}{2}S_i \leq T_i \leq 2S_i$. For the last



(a) $s_{lmax.0} = 2T_0$ (b) $s_{lmax.1} = 2(T_1 - S_1)$

(c) $s_{lmax.2} = T_2$ (d) $s_{lmax.3} = 2(T_3 - S_3)$

Figure 9.2: Worst case scenarios for periodic synchronization under static-cyclic scheduling.

case, with $T_i < \frac{3}{2}2S_i$, the synchronization phase finishes in the worst-case scenario after two consecutive scheduling gaps of duration $T_i - S_i$, resulting in $2(T_i - S_i)$ (Figure 9.2(d)).

Figure 9.3 shows s_{lmax} over the ratio of scheduling slice to scheduling period. We can deduce that short synchronization phase durations are only achievable, if either a large part of the scheduling period is assigned to the replicated FS, or the scheduling periods are very short with respect to the synchronization period.

Sched. Period [ms]	Sched. Slice [ms]	CPU for δ^{FS}	Idle CPU	Remaining CPU
100	71	1.0%	58.0%	29.0%
90	65	1.1%	59.1%	27.8%
80	32	1.3%	25.5%	61.3%
70	28	1.4%	26.6%	60.0%
60	25	1.7%	28.0%	58.3%
50	21	2.0%	28.0%	58.0%
40	13	2.5%	23.0%	62.5%
30	7	3.3%	8.0%	76.7%
20	5	5.0%	8.0%	75.0%
10	3	10.0%	8.0%	70.0%

Table 9.1: Scheduling parameters and subsequent CPU usage for the example application of $\mathcal{F}_{10}$ under static-cyclic scheduling with periodic synchronization.

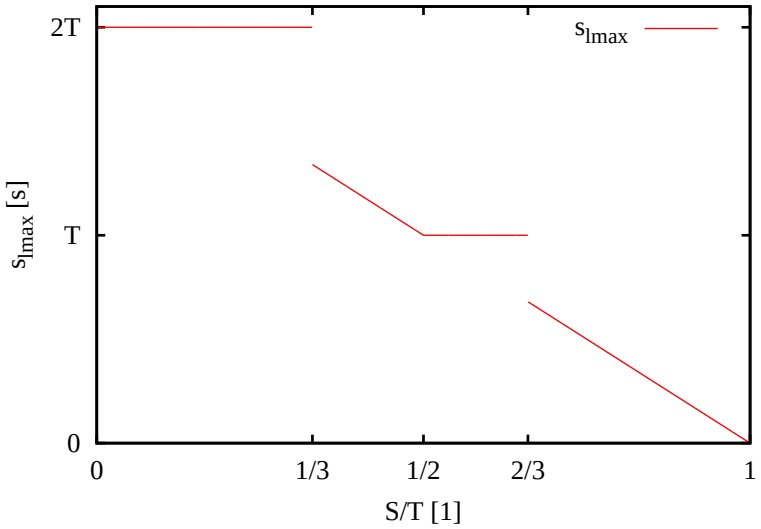


Figure 9.3: The lower bound of the worst-case synchronization phase duration s_{lmax} over the ratio of static-cyclic scheduling slice S to period T.

Sched. Period [ms]	Sched. Slice [ms]	CPU for δ^{FS}	Idle CPU	Remaining CPU
50	41	2.0%	36.0%	24.0%
40	35	2.5%	41.0%	12.5%
30	28	3.3%	26.0%	6.7%
20	16	5.0%	31.0%	20.0%
10	7	10.0%	16%	30.0%

Table 9.2: Scheduling parameters and subsequent CPU usage for the example application of $\mathcal{F}_{20}$ under static-cyclic scheduling with periodic synchronization.

We also see this effect with the example applications. The parameters for the static-cyclic scheduler are selected such that the requirements of our example application are still met, while at the same time using the least CPU share. Table 9.1 contains these parameters and resulting CPU shares on the IE node level for the scheduling operations of one $\mathcal{F}_{10}$ FS node (CPU for δ^{FS}), the dedicated idle time within these FS node (idle CPU) and the time available for other FS nodes (remaining CPU). Table 9.2 shows the same for $\mathcal{F}_{20}$. The dedicated idle time is the time assigned to an FS node, which it will not use – even when experiencing the worst-case workload.

Exemplary, the scheduling slices for the scheduling period of 100ms and 10ms are determined for $\mathcal{F}_{10}$ as follows. After the synchronization phase finishes in the first case, 10ms of the slice time must be left for computation to fit in the scheduling period of 100ms. Since each FS node is at most scheduled once within the synchronization period (also 100ms), our worst-case scenario must be that of Figure 9.2(d). The synchronization phase duration can only be smaller than the scheduling period, as is required, if the slice duration is at least $\frac{2}{3}T$. In the worst case,

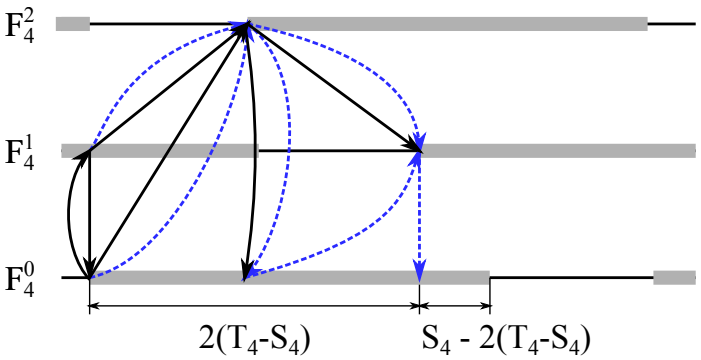


Figure 9.4: Example worst-case schedule for $\mathcal{F}_{10}$ with a scheduling period of 100ms.

the slice-time left after $s_{l,max}$ is $S - 2(T - S)$. This is illustrated in Figure 9.4. The remaining time needs to be at least the 10ms workload computation and an additional 2ms for computation required by the synchronization process[1]. Another 1ms is required for scheduling the FS node at the beginning of the slice ($\delta^{\mathtt{FS}}$). Consequently, the shortest feasible slice duration is 71ms, when scheduling with a granularity of milliseconds. In the case, with a scheduling period of 10ms, it is sufficient if the synchronization phase finishes after two scheduling periods as in the scenario of Figure9.2(a). This leaves eight slices for computation of the 12ms. Together with the scheduling overhead, this results in 20ms, which must be scheduled within the remaining eight slices. This yields a scheduling slice duration of 3ms.

Intuitively, there are more scheduling options for applications with low workloads and long synchronization phases, that leave a substantial share for integrating additional FS nodes, than with shorter synchronization phases and higher workload. This is also reflected in the two tables above where for $\mathcal{F}_{10}$ many parameters yield remaining CPU shares of 60 percent or more, while 24 percent is the maximum with $\mathcal{F}_{20}$. Consequently, integrating several FSs with various requirements on synchronization period and workload can be a difficult, if not impossible task. In fact, if one integrates $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$, only a scheduling period of 10ms is feasible, with a scheduling overhead for FS nodes of 20 percent.

Regarding the integration of other FSs, we need to use a more fine grain scheduler (microseconds) to reduce the slices of $\mathcal{F}_{10}$ to 2.500ms and $\mathcal{F}_{20}$ to 6.667ms. Consequently, we have 0.833ms left for other tasks every 10ms. This is too few for integrating another FS, since it requires at least 1ms for loading. Assuming now it would fit and the integrated FS is computationally intensive and executed whenever the others are not, scheduling overhead would rise to 30 percent.

Violating Constraint 2 from Chapter 5 by synchronizing the schedulers of the IE nodes reduces the overall scheduling overhead. This comes at the cost of a weaker border (c) between the IE nodes according to the illustration of limits in Figure 8.1 on page 54. Provided a suitable scheduling pattern can be found in the integrated system, an FS node needs only one scheduling operation per synchronization period. The slice duration then consists of the time required for message exchange, membership, voting, computation and an additional margin to compensate

[1]This includes signature checks, voting, membership and other tasks.

for the slice offsets between the corresponding FS nodes, due to imprecision of the synchronized schedulers. This margin is strongly dependent on the used synchronization mechanism and may range from a few microseconds to milliseconds, depending on the layer in which it is implemented, as described in Section 7.4 on page 52.

For illustration with our integration example, we assume a margin of 1ms, as well as 1ms of processing time required for the periodic synchronization of the schedulers. Integrating $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$ on top of such a scheduler yields an overhead for $\delta^{\mathtt{FS}}$ of 3 percent. The remaining CPU share is 40 percent, and a computationally FS node as above needs 2 percent for scheduling and can utilize 38 percent of the CPU for computation.

However, the incurred overhead with composability using (unsynchronized) static-cyclic scheduling is strongly dependent on the integrated FSs. Non-safety-critical FSs may get only a very reduced share of the CPU, and subsequently, only part of the systems' resources are utilized.

9.2 Preemptive Fixed-Priority Scheduling

Naturally, in preemptive fixed-priority scheduling, lower priority tasks are affected by those with higher priority.

Execution Guarantees

With fixed-priority scheduling of workloads, a very beneficial strategy to assert completion of the individual FSs is to consider three priority regions within the system:

- The highest priority band is reserved for system essential tasks, such as network interface handling, scheduling, etc.

- A priority band in the middle range is dedicated to the safety-critical FSs. In principle, one may consider assigning the highest priority within this band to the most critical FS, but this is not a prerequisite – after all, a completion guarantee must be given to all critical FSs requiring it, even the ones with the lowest criticality.

  Still, assigning different priority levels assists in enforcing a deterministic sequence of FS node scheduling, thereby keeping task switches and jitter at a minimum for this scheduling scheme. This helps maintaining most of the FS nodes' memory in cache and shadow page tables available. Apart from assigning the correct priorities to FSs, a supervisor must keep track of each FS's resource usage and limit or even stop execution of a misbehaving FS. This is necessary, since the priority-based approach in principle allows for unbounded execution time, which distinguishes it most from static-cyclic scheduling.

- The lowest priority band may be dedicated to "best effort" services whose execution is not critical and may, therefore, be performed according to the availability of remaining computation time.

Performance of Synchronization in FS

The direct approach to integrate different replicated FSs is to let each of them use the synchronization service within their failure confinement region, which is a significant advantage since it does not require more specific protection than an ordinary FS. Furthermore, all FSs can configure and handle their synchronization completely independently. For the FS nodes where execution is guaranteed, the lower bound for the worst-case synchronization phase duration, considering only FS node scheduling, is

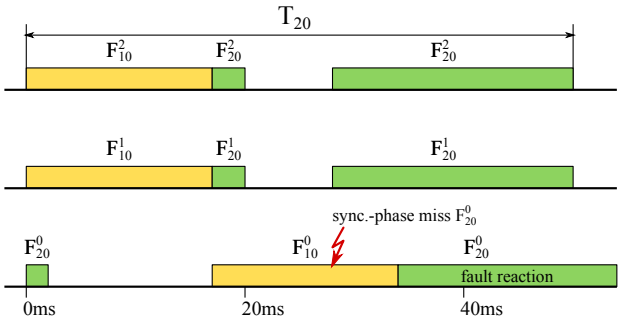
$$s_{lmax,i} = 5\delta_{max}^{\mathtt{FS}}.$$

The longest sequence of scheduling delays for this $s_{lmax,i}$ scenario starts with scheduling the fastest FS node due to the synchronization timeout, followed by only one FS node receiving its direct message due to a communication fault. Thus, the third FS node receives the first message after $3\delta_{max}^{\mathtt{FS}}$ and it takes two additional scheduling delays until its message is forwarded to the fastest FS node. With our example applications, the incurred maximum overhead is 5 percent CPU load for $\mathcal{F}_{10}$ and 10 percent for $\mathcal{F}_{20}$.

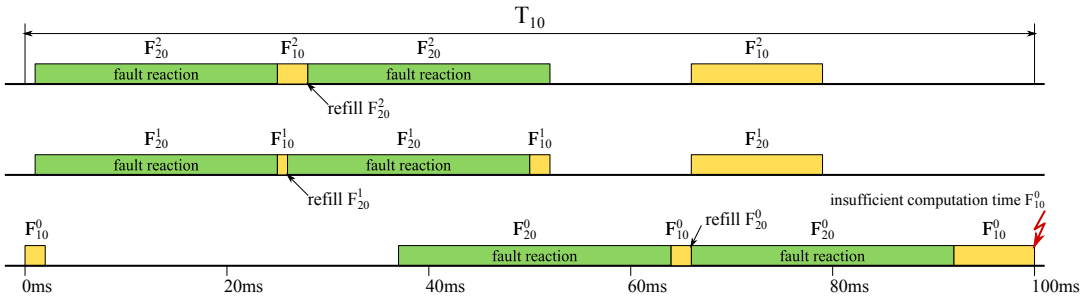
However, this lower bound is only valid for the highest priority FS, e.g. with a setup using a dual-core CPU for IE nodes, only the two highest priority FS nodes are provided with sufficient resources for a reasonable synchronization period. FS nodes with lower priority can be blocked by (misbehaving) higher ones at any time, causing them to miss their synchronization phase or have insufficient time for computation even with resource usage supervision.

Figure 9.5(a) illustrates such a scenario with $\mathcal{F}_{10}$ having a higher priority than $\mathcal{F}_{20}$. The case with inversed priority assignment is depicted in Figure 9.5(b). Supervision of CPU usage, e.g. with a constant bandwidth server, would not prevent either case from occurring. In the first example, the FS nodes of $\mathcal{F}_{10}$ are all within the limits of their CPU share as they require at least 12ms for computation and synchronization-related tasks plus 5ms for scheduling in the synchronization phase ($s_{lmax,10}$). Here, F_{10}^0 has a fault and is already excluded by F_{10}^1 and F_{10}^2. F_{10}^0 starts its execution such that F_{20}^0 fails to receive the other nodes' messages, misses the end of its synchronization phase and has to trigger a fault reaction. The second example is unlikely but possible. Here, each IE node uses a constant bandwidth server locally to limit the execution time for the FS nodes. The FS nodes are provided with a budget of CPU time to execute. If an FS node exceeds its budget, its execution is stopped. The constant bandwidth server now periodically refills these budgets so that the stopped FS nodes can continue execution. In the depicted scenario, the FS nodes of $\mathcal{F}_{20}$ are triggering a fault reaction, and thereby, exceeding their regular CPU budgets at some point. There execution is subsequently resumed when the local refill events occur as indicated. Also, $\mathcal{F}_{20}^0$ is scheduled later than the other nodes, e.g. due to waiting for a timeout or log-server response. This sequence of events causes $\mathcal{F}_{10}^0$ to have insufficient computation time.

If the IE reserves a large portion of resources for limiting FS node interference, i.e. one CPU core per FS node, the FSs' requirements can indeed be fulfilled by using fixed-priority scheduling. The benefit here, as opposed to the static-cyclic scheduling, is that a non-critical FS can



(a) $\mathcal{F}^0_{10}$ causes $\mathcal{F}^0_{20}$ to miss its synchronization phase.



(b) Safety reaction and subsequent processing pattern of $\mathcal{F}_{20}$ causes $\mathcal{F}^0_{10}$ to have insufficient computation time.

Figure 9.5: Violation of FS guarantees due to interfering higher priority FS.

still use the CPU when the critical FSs are not scheduled, and therefore, have a better utilization. Consequently, caches are most likely empty when FS nodes are scheduled and memory access is slowed down due to parallel requests. An analysis regarding these influences must be conducted before certification of the IE, as well as of the FSs. Such an analysis may be based on a suitable benchmark stressing the IE nodes memory [44, 93].

Synchronization as Integration Environment Service

As discussed in Section 7.4 on page 52 and illustrated for static-cyclic scheduling in the previous section, implementing the synchronization mechanism on the level of the hypervisor scheduler provides a better precision than on higher layers of the system. With fixed-priority scheduling a compromise between synchronization of the hypervisor schedulers and synchronization within the FS nodes is available: a single synchronization task per IE node. Based on the findings presented up to now, this approach promises a good synchronization precision, while at the same time avoids to change the hypervisor itself.

This task executes the periodic synchronization algorithm and, upon completion of the synchronization phase, starts distributing the corresponding data to the individual FS nodes. Voting, membership and computation is performed within the FS node. When finishing its computation, the FS node returns the result to the synchronization task. Figure 9.6 illustrates the scheduling pattern of such a service and the FS nodes on one IE node of our example applications. This

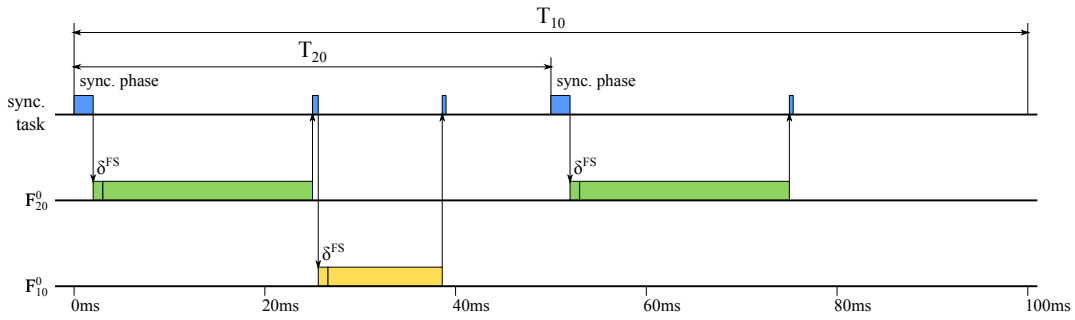


Figure 9.6: Example schedule of the FS $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$ FS nodes on IE node 0 with a single synchronization task per IE node.

single synchronization task for all integrated FS nodes provides a more predictable solution. The synchronization task is assigned a higher priority than all safety-critical FSs, achieving a better precision of synchronization. This is because message transmission times and subsequent synchronization phase durations are shortened, while avoiding the additional overhead of the context switches into the FS nodes during the synchronization phase. In fact, only one $\delta^{\mathtt{FS}}$ is required per FS node during the FS's synchronization period, as also depicted in the figure above for $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$. The synchronization task can also provide the supervision of the FSs' execution times, since it determines when the FS node starts execution and receives the FS node's result as soon as computation is completed.

This setup is a kind of time-triggered architecture above the hypervisor. It enables the determination of an upper bound for the synchronization phase and computation duration for several FSs, and it is, therefore, a composable TMR synchronized architecture, which reuses the already well-established synchronization mechanism.

The first disadvantage of this solution is the tight coupling between FS nodes of the same IE node. This violates Constraint 2 from Chapter 5 and may limit the robustness of the failure confinement regions provided by the IE. Also, a fault in the synchronization task itself now affects all FS nodes of the respective IE node. Another drawback is the reduced flexibility of the integrated system, given that the synchronization period of each FS must be a multiple of the synchronization period of the IE service.

However, this approach still enables integration of more safety-critical FSs than the method of synchronizing within the individual FS nodes as presented above.

Regarding the performance of non-safety-critical FSs, we again assume one FS with a single FS node executing a computation intensive task. This FS node can use all remaining CPU time minus its FS scheduling operations caused by being interrupted by the safety-critical FS nodes and synchronization task. This occurs once per shortest scheduling period of the integrated FSs. With our example applications, the scheduling overhead experienced by the non-safety-critical FS node is 2 percent CPU share.

9.3 Preemptive EDF Scheduling

A different problem arises for preemptive EDF scheduling. With the EDF's minimum task inter-arrival time $T^{\mathtt{E}}$, scheduling an FS node for initially sending or forwarding a message only, causes a gap of $T^{\mathtt{E}}$ until it is scheduled the next time. This actually results in the same maximum synchronization phase duration as for static-cyclic scheduling

$$s_{lmax,i} = \begin{cases} 2T_i^{\mathtt{E}} & \text{if } 3S_i^{\mathtt{E}} \leq T_i^{\mathtt{E}} \\ 2(T_i^{\mathtt{E}} - S_i^{\mathtt{E}}) & \text{if } 2S_i^{\mathtt{E}} \leq T_i^{\mathtt{E}} < 3S_i^{\mathtt{E}} \\ T_i^{\mathtt{E}} & \text{if } \frac{3}{2}S_i^{\mathtt{E}} \leq T_i^{\mathtt{E}} \leq 2S_i^{\mathtt{E}} \\ 2(T_i^{\mathtt{E}} - S_i^{\mathtt{E}}) & \text{if } T_i^{\mathtt{E}} < \frac{3}{2}S_i^{\mathtt{E}} \end{cases} .$$

In principle, the same solution as for fixed-priority scheduling is available, with applying synchronization on the IE level. For such a setup the synchronization tasks have a very short deadline compared to the FS node deadlines and the constant bandwidth server limits the execution time of the FS nodes.

The problem with EDF in this case is that it is strongly dependent on the job release time since this specifies the next deadline. If an FS node $\mathcal{F}_0^0$ wakes up without having all data for workload execution available, it will not be scheduled for another $T_0^{\mathtt{E}}$. This subsequently causes $\mathcal{F}_0^0$ to not be executed as soon as the synchronization phase is complete. It is assumed faulty and shutdown. Consequently, the FS nodes themselves are not able to use timeouts for internal supervision, such as for health monitoring, other than for triggering a safety reaction. Thus, the safety layer and applications within the FSs must be adapted so that they are executed according to the synchronization process and no longer have their independent timeouts. This is an extensive (certification) effort for the expected integration benefit and a violation of Constraint 1 from Chapter 5. This approach will be further discussed for the dynamic IE, where the advantages may justify such changes in the safety layer and applications.

9.4 Comparison of Scenarios Integrating the Example Applications

The two possible solutions for integrating our example applications are static-cyclic scheduling and preemptive fixed-priority with synchronization on IE node level. We assume 2ms execution for the synchronization task within the period of T_{20} for our comparison. Table 9.3 shows how the shares of the CPU are used, also by an additional non-critical FS. It illustrates that the scheduling overhead in the static-cyclic case, together with the dedicated idle time, consumes most of the CPU share, leaving not enough for the non-critical FS. With fixed-priority scheduling and providing the synchronization service within the IE, the FS scheduling operations are reduced to the minimum and no such idle time occurs.

FS type	Property	Static-cyclic	Fixed-priority
Safety-critical FSs	FS scheduling CPU share	20.0%	3.0%
	Computing CPU share	56.0%	60.0%
	Idle CPU share	15.7%	0.0%
	Remaining CPU share	8.3%	37.0%
Non-critical FS	FS scheduling CPU share	8.3%	2.0%
	Computing CPU share	-	35.0%

Table 9.3: CPU usage with static-cyclic and fixed-priority scheduling (with IE node synchronization service) when integrating the example safety-critical applications $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$ and one computational intensive non-critical FS.

CHAPTER 10

Dynamic IE

The adaptable dynamic IE is an extensible architecture. The additional functionalities here are the dynamic deployment of FSs and the ability of on the fly changing the structure of the IE. The latter includes adding and removing IE nodes and adapting interconnect during operation. These features affect the execution of safety-critical services and must, therefore, be implemented as a service with highest criticality level, i.e. an extended FS manager. Inputs for such a FS manager are the IE contract and FS contracts, on the basis of which it decides whether or not requested deployments and IE alterations are feasible. In fact, it makes a new integration contract on the fly, providing a kind of online certification.

Within the dynamic IE, point to point connections no longer provide sufficient network capabilities as already illustrated in Section 4.2 on page 28 for the static TMR-composable architecture. Consequently, the direct connections are replaced by a redundant interconnect, e.g. two switches connect five IE nodes. This architecture alone requires an adaptation of the periodic synchronization method in the TMR middleware in order to support redundant networks.

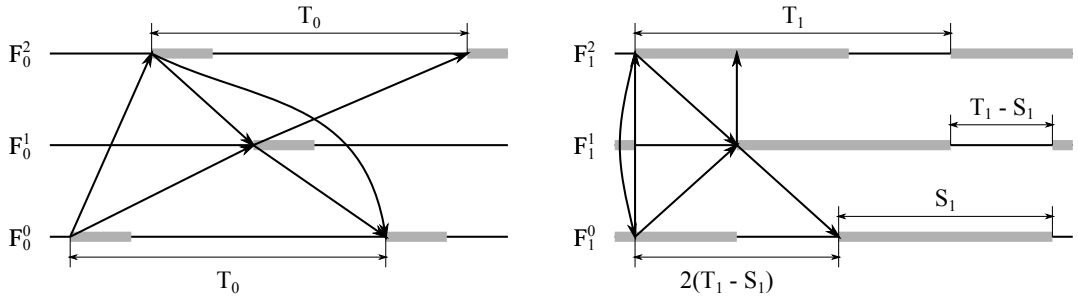
We investigate the synchronization behaviour of periodic synchronization and TSM-based TMR for the different IE node scheduling strategies. For illustration purposes, we use the example applications $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$ described at the beginning of the previous chapter on page 57.

10.1 Periodic Synchronization with Redundant Networks

The message forwarding step, still necessary in the classic TMR architecture, is not required for periodic synchronization with redundant networks. Consequently, a better synchronization performance is to be expected for all scheduling strategies of the IE nodes.

Static-Cyclic Scheduling

Figure 10.1(a) shows the (redundant) messages exchanged for the periodic synchronization under static-cyclic scheduling. The synchronization phase starts with the sending of the first message and ends with reception of the last, thus, for the depicted (worst-case) scenario each replica



(a) $s_{lmax.0} = T_0$ (b) $s_{lmax.1} = 2(T_1 - S_1)$

Figure 10.1: Worst-case synchronization phase duration of periodic synchronization in redundant network using static-cyclic scheduling.

uses one local scheduling slice for synchronization. The lower bound of the worst-case synchronization phase duration for static-cyclic scheduling is

$$s_{lmax,i} = \begin{cases} T_i & \text{if } 2S_i \leq T_i \\ 2(T_i - S_i) & \text{if } T_i < 2S_i \end{cases}$$

With $2S_i \leq T_i$, the worst case is T_i as illustrated in Figure 10.1(a), where $\mathcal{F}_0^0$ and $\mathcal{F}_0^2$ receive the message of $\mathcal{F}_0^1$ after waiting a full scheduling period. For $T_i < 2S_i$, an example scenario is that the synchronization phase is started by $\mathcal{F}_1^0$ right at the beginning of a scheduling gap at the FS node $\mathcal{F}_1^1$ as shown in Figure 10.1(b). $\mathcal{F}_1^1$ sends its message when it is scheduled after $T_1 - S_1$, at which point the FS node $\mathcal{F}_1^0$ has the start of its scheduling gap. Consequently, $\mathcal{F}_1^0$ receives the message of $\mathcal{F}_1^1$ after $2(T_1 - S_1)$.

As expected, the synchronization phase is shorter than when using message forwarding. However, it remains still T_i in the scenario where less than half a CPU core is assigned to the FS node.[1]

For illustration, the integration of our example applications results in a scheduling period of 10ms and slice durations of 2.3ms for $\mathcal{F}_{10}$ and 6.5ms for $\mathcal{F}_{20}$. The idle time is 12 percent, and consequently, 3 percent lower than in the classic TMR architecture with the other periodic synchronization mechanism. Integration of a non-safety-critical FS with computationally intensive workload is now possible with 12 percent remaining CPU time. However, it only gets a CPU share of 2 percent for computation and needs the other 10 percent for scheduling.

Synchronizing the static-cyclic schedulers in this architecture not only weakens fault containment between IE nodes, but also introduces correlations on scheduling level between FSs, which are not even integrated on the same IE nodes. The CPU shares are those of the classic TMR architecture with synchronized schedulers, with a shorter synchronization phase duration.

[1] If we allowed the scheduling slice to be executed anytime during the scheduling period, we would have the same two scheduling periods as presented by Miller et al. [88].

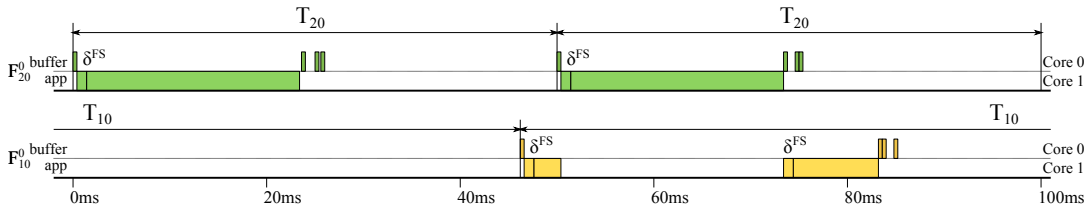


Figure 10.2: Example schedule of integrated FSs $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$ under EDF with constant bandwidth server.

Preemptive Fixed-Priority Scheduling

Providing the periodic synchronization for redundant networks as an IE service, leads to many synchronization tasks receiving and processing message data, for which they have no local FS node. Therefore, this synchronization method does not scale when the number of participating (IE) nodes increases. Consequently, only one solution for the dynamic IE with fixed-priority scheduling is beneficial: one critical FS node per CPU core and providing synchronization within the FS, as shown in the last chapter. Since the forwarding step of the algorithm is omitted, the new lower bound for the worst-case synchronization phase duration is

$$s_{lmax,i} = 3\delta^{\mathtt{FS}}_{max}.$$

Therefore, our example applications $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$ experience a scheduling overhead of 3 and 6 percent of CPU time, respectively.

Preemptive EDF-based Solution

As we have seen for the classic TMR architecture, exposing part of the FS internal scheduling information by providing the synchronization on the IE level, enabled the increase of the overall predictability at the cost of tighter coupling of unrelated FSs and a weaker separation. The move to a dynamic IE enables a similar approach with preemptive EDF scheduling, since the extensive development and certification costs can now be justified by the advantages it brings. With the preemptive EDF scheduler, the CPU can, unlike in the static-cyclic scheme, be used by lower critical FSs. At the same time, EDF enables the use of existing and well researched EDF schedulability tests [21] as reliable basis for the feasibility evaluation method provided by the IE contract. Also the definitions of EDF with scheduling period and slice fits well to the execution pattern of the periodically synchronized applications.

The idea is to provide for each FS node a very lightweight FS buffer above the composability layer, which receives *all* communication sent to the FS node and has low scheduling overhead. These buffers periodically deliver the received messages to their FS nodes, ensuring that the FS nodes are only scheduled when they have enough messages for progress or triggering a safety reaction. The FS nodes are then scheduled with an EDF scheduler and constant bandwidth

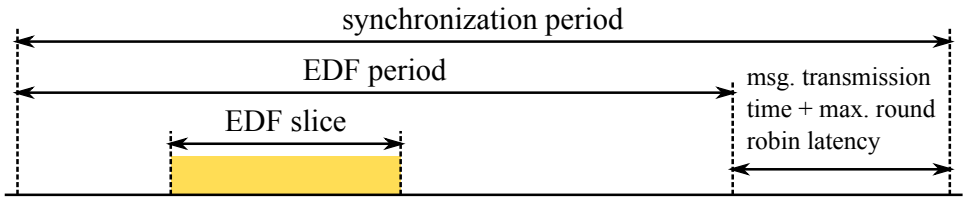


Figure 10.3: Relation of synchronization parameters for EDF-based scheduling.

server, which ensures that all get their designated share of the CPU before their synchronization periods finish. For this strategy to work, it is important that the lightweight buffers are not blocking too long when sending their data to the FS nodes, otherwise the deadline for FS node execution can be set after the end of the current synchronization period. Also, the FS nodes execution must not be interrupted by such lightweight buffers. Consequently, this approach is only feasible when applied on a multi-core CPU. The FS buffers are then scheduled round robin on a dedicated core, while FS nodes are scheduled with EDF and a constant bandwidth server on the other available cores. Figure 10.2 illustrates the execution of FS nodes and corresponding buffers on one IE node for $\mathcal{F}^0_{10}$ and $\mathcal{F}^0_{20}$. The scheduling of the buffers on core 1 is depicted by the boxes directly above the according dotted line, while the FS nodes are scheduled on core 0 below this line.

Figure 10.3 illustrates how the EDF scheduling period must be defined in relation to the synchronization period and round robin scheduling delay of the FS buffer.

This method reduces the required number of $\delta^{\mathtt{FS}}$ operations to one per synchronization period per FS node, if all FS nodes deployed on the IE node have the same synchronization period. Otherwise, the number of FS nodes with shorter synchronization phase determines the maximum additional $\delta^{\mathtt{FS}}$ operations, if one CPU core is available for EDF, with:

$$i_{max} = \delta^{\mathtt{FS}}_{max} \sum_{\forall T^{\mathtt{E}}_j : T^{\mathtt{E}}_j < T^{\mathtt{E}}_i} \left\lceil \frac{T^{\mathtt{E}}_j - T^{\mathtt{E}}_i}{T^{\mathtt{E}}_i} \right\rceil .$$

This bound has been presented by Ju et al. [67] for cache related preemption analysis of EDF scheduled processes. Based on this result, a simple division by the number of cores n provides a (non-tight) upper bound, if the FSs' can be executed on more than one CPU core:

$$i_{umax} = \delta^{\mathtt{FS}}_{max} \left\lceil \frac{1}{n} \sum_{\forall T^{\mathtt{E}}_j : T^{\mathtt{E}}_j < T^{\mathtt{E}}_i} \left\lceil \frac{T^{\mathtt{E}}_j - T^{\mathtt{E}}_i}{T^{\mathtt{E}}_i} \right\rceil \right\rceil .$$

Such an overhead must be included in the FS node's slice. The scheduling overhead incurred by a non-critical, computationally intensive FS node can be bounded with considering all safety-critical FSs. In the example from above, it is at most 3 percent of the non-critical FS node's CPU time, when it is pinned to the same core as the safety-critical FS nodes.

Some additional restrictions are required when implementing this approach. The periodic synchronization mechanism for redundant networks within the FS node must be adapted such that it sends out the synchronization messages as soon as the local computation has finished. Also, the timebase for periodic message delivery by the buffer partitions must be provided by the local FS node. This ensures that the functionality provided by the buffers and the resulting failure modes for these buffers are limited. As already discussed, only one timeout, which detects the misbehaviour of the FS buffer, must be used within the FS nodes. This requires an adaptation of the safety layer within the FS node to trigger all safety-related tasks according to the receiving of messages.

Once the above prerequisites are met, a regular EDF scheduling test can be used for testing the feasibility of integrating the FS nodes. Also, the worst-case round robin scheduling time of the FS buffers has to be evaluated in order to determine the synchronization jitter and thereby, the required margins.

The significant drawback to this approach is the necessity to not only change the safe middleware, but even adapt the applications themselves, since all application specific timeouts and I/O must be based on the synchronization phase, resulting in a huge development and re-certification effort for applications. Therefore, this approach effectively violates Constraint 1, not to change the application. However, since the FS buffer only provides periodic delivery of messages according to the time it gets from the FS node and the synchronization mechanism itself is still implemented in the safety layer of the FS node, Constraint 2 (provide the synchronization mechanism within the FSs) is not violated.

Comparison of Scenarios Integrating the Example Applications

FS type	Property	Static-cyclic	EDF
Safety-critical FSs	FS scheduling CPU share	20.0%	3.0%
	Computing CPU share	56.0%	60.0%
	Idle CPU share	12.0%	0.0%
	Remaining CPU share	12.0%	37.0%
Non-critical FS	FS scheduling CPU share	10.0%	3.0%
	Computing CPU share	2.0%	34.0%

Table 10.1: CPU usage with static-cyclic and EDF-based scheduling when integrating the example safety-critical applications $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$ and one computationally intensive non-critical FS.

Table 10.1 shows the CPU shares when integrating the example applications $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$ in a dynamic IE with IE nodes having single-core CPUs. Assuming an overhead of 4 percent for

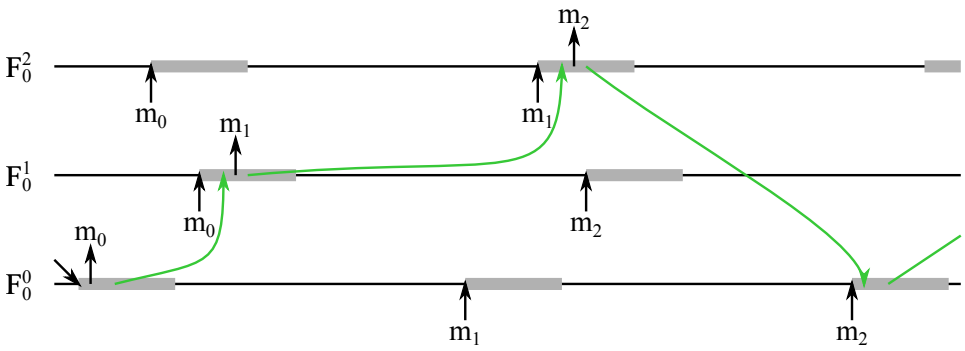


Figure 10.4: TSM message exchange under static-cyclic scheduling.

the FS node buffers, the EDF approach has, apart from one percent scheduling overhead for the non-critical FS, the same CPU shares as the fixed-priority approach with synchronization service for the IE with classic TMR architecture. The additional interruption of the non-critical FS is a result of the unsynchronized execution of safety-critical FS nodes for EDF, possibly leaving a gap in-between them.

In the static-cyclic case, again, the scheduling overhead and dedicated idle time use a significant amount of the CPU. This results in only 2 percent of CPU time effectively being usable by the non-critical FS for execution.

10.2 TSM

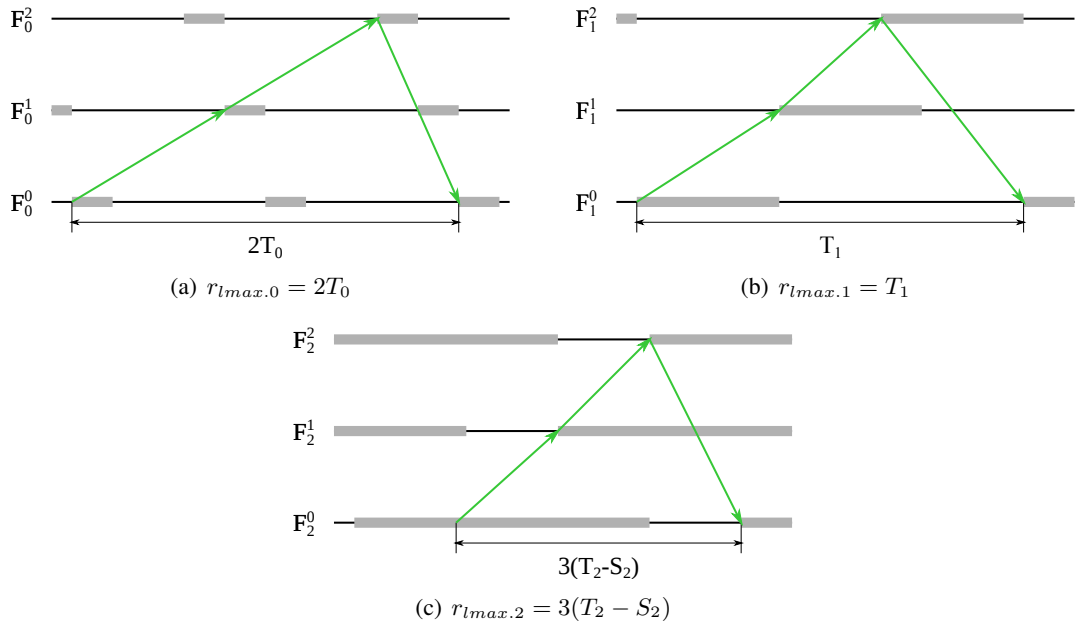
TSM is based on message exchange for message ordering during fault-free operation. Timeouts are only used for fault detection and the membership service. Thus, TSM has at message-ordering level no notion of synchronized time itself, as opposed to the periodic synchronization algorithms. This prevents an EDF-based solution for TSM, as presented in the previous section. It is also the reason why TSM can not be used as a single synchronization service for all FSs integrated on an IE node.

Regarding fixed-priority scheduling, the same restriction applies in case of TSM-synchronized FSs as for those using the periodic synchronization algorithm: At most one safety-critical FS node per CPU core can be integrated. Also, here, a higher priority FS node can interrupt a lower priority one at any time, if they are not provided with different cores.

Consequently, static-cyclic and fixed-priority are the options available for composable scheduling when using TSM for synchronization.

Static-Cyclic Scheduling

Figure 10.4 illustrates the operation of TSM under static-cyclic scheduling. The grey thick boxes indicate when the slice of an FS node using TSM is scheduled. Token messages are represented with the arrows connecting the time lines of the replicas. The sending and receiving of data messages is indicated by arrows pointing from message identifiers to the time line and vice-



(a) $r_{lmax.0} = 2T_0$ (b) $r_{lmax.1} = T_1$

(c) $r_{lmax.2} = 3(T_2 - S_2)$

Figure 10.5: Worst-case token round trip time of TSM under static-cyclic scheduling.

versa. All sent data messages are instantly received by the sending replica and omitted in the diagram for simplicity. In the example schedule, the replicated application, executed on top, can continue computation as soon as the last sent data message, m_2, is received.

The timeouts of TSM must be adapted according to the static-cyclic scheduling. The token loss timeout is still at least the token round trip time, while the membership timeout is increased to at least the minimum of the maximum token round trip time and the scheduling period, i.e. $\min(r_{max,i}, T_i)$.

The token round trip time r_i, indirectly determining the reaction time of the replicated FS, depends on the scheduling period T_i, the scheduling slice S_i and the relative offset of the three schedulers. The lower bound for the maximum token round trip time $r_{lmax,i}$, again only considering the effect of FS node scheduling, is

$$r_{lmax,i} = \begin{cases} 2T_i & \text{if } 3S_i \leq T_i \\ T_i & \text{if } \frac{3}{2}S_i \leq T_i < 3S_i \\ 3(T_i - S_i) & \text{if } T_i < \frac{3}{2}S_i \end{cases}$$

This is also the token round trip jitter, since the round trip time is zero in our model, when all FS nodes are simultaneously scheduled.

Figure 10.5 illustrates three different examples of worst-case scenarios for the lower bound $r_{lmax,i}$ of the token round trip time. The solid lines are token messages. In the first one, with $3S_0 \leq T_0$, the scheduling slice of $\mathcal{F}_0^2$ occurs just before that of $\mathcal{F}_0^1$, which itself is just before

that of $\mathcal{F}_0^0$. Subsequently, the first two token messages each need a time of $T_0 - S_0$, while the last one is $2S_0$ long. This results in a round trip time of $2T_0$. In the second scenario, with $\frac{2}{3}S_1 \leq T_1 < 3S_1$, the two token-forwarding events of $\mathcal{F}_1^1$ and $\mathcal{F}_1^2$ occur during the scheduling gap of $\mathcal{F}_1^0$, which then receives the token again after T_1. In the last scenario, with $T_2 < \frac{3}{2}S_2$, the three scheduling gaps of the FS nodes occur right after each other in the order in which the token is forwarded. This causes a token round trip time of three times the scheduling gap, i.e. $3(T_2 - S_2)$.

The worst-case reaction time of the FS is also a result of the regular operation interruption time g_i on fault occurrence. Assuming the lowest possible values for these timeouts, the lower bound in our zero computation and message transmission time model is

$$g_{lmax,i} = \begin{cases} 5T_i & \text{if } 3S_i \leq T_i \\ 4T_i & \text{if } \frac{3}{2}S_i \leq T_i < 3S_i \\ 5(T_i - S_i) & \text{if } \frac{4}{3}T_i \leq \frac{3}{2}S_i \\ 2T_i - S_i & \text{if } \frac{5}{4}S_i \leq T_i < \frac{4}{3}S_i \\ 5(T_i - S_i) & \text{if } T_i < \frac{5}{4}S_i \end{cases}$$

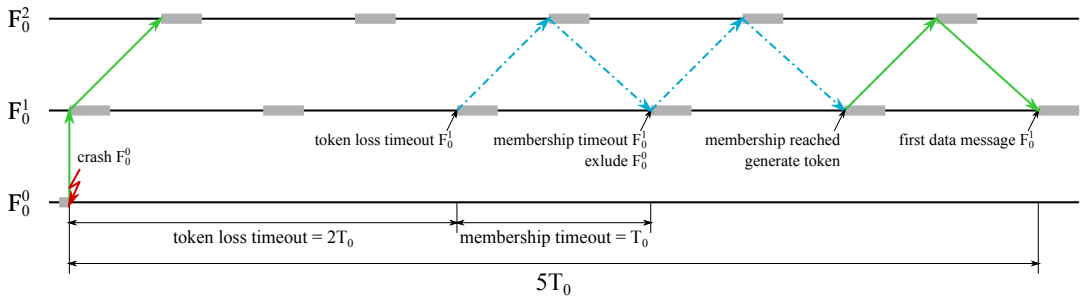
For the last three cases, the ratio between token loss timeout and slice duration is the defining parameter, as we now see in the examples.

Figure 10.6 illustrates the example scenarios for the worst-case interruption time with $\frac{3}{2}S_i \leq T_i$. The dashed lines are membership messages. For the first case with $3S_0 \leq T_0$, the first two scheduling periods are required for detecting the token loss, the third to identify the crashed FS node and the fourth to generate a token. When receiving the token after the fifth scheduling period, $\mathcal{F}_0^1$ can also resume sending application data. In the scenario with $\frac{3}{2}S_i \leq T_1 < 3S_1$, the token loss timeout is one scheduling period shorter than for the previous scenario and thus, the interruption is $4T_1$. The examples for $T_i < \frac{3}{2}S_i$ are shown in Figure 10.7. In the scenario depicted for $\frac{4}{3}S_2 \leq T_2 < \frac{3}{2}S_2$, $\mathcal{F}_2^2$ detects the token loss after $3(T_2 - S_2)$ and just misses the membership timeout to propose excluding $\mathcal{F}_2^0$ by being interrupted at $4(T_2 - S_2)$. Regular operation is resumed when $\mathcal{F}_2^2$ is scheduled again at $5(T_2 - S_2)$. In the scenario for $frac54S_3 \leq T_3 < \frac{4}{3}S_3$, $\mathcal{F}_3^2$ receives the token after $T_3 - S_3$ and has the token loss timeout at $3(T_3 - S_3)$. Afterwards, $\mathcal{F}_3^2$ is interrupted and $\mathcal{F}_3^1$ sends the join message to exclude $\mathcal{F}_3^0$. $\mathcal{F}_3^2$ resumes execution at $T_3 + T_3 - S_3$, at which point membership is reached and the token created. The last scenario with $T_4 < \frac{5}{4}S_4$ has the same sequence of events as that of T_2 and S_2.

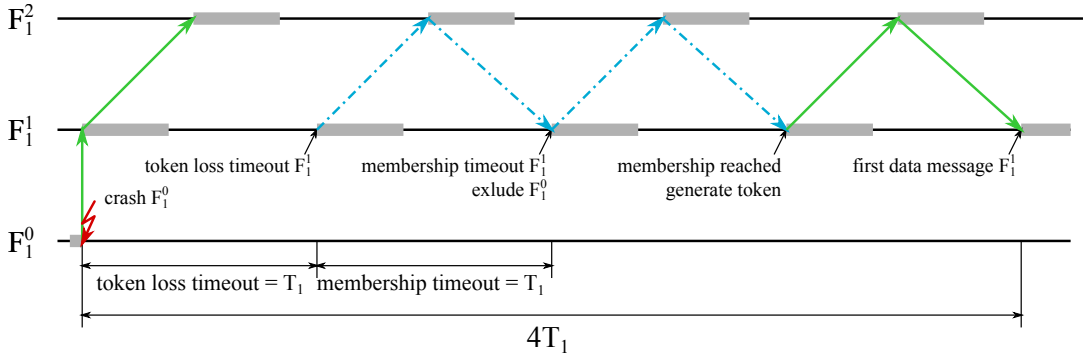
For illustration purposes, let us assume a scheduling period of 10ms and a slice of 3ms. With this configuration, the regular protocol operation is interrupted for 50ms when a node fails.

Compared with the periodic synchronization algorithms, where the time it takes for detecting a faulty FS node is the maximum synchronization phase duration[2], this is a very large interruption of operation. Under the assumption that the computation of the reaction fits into the single scheduling slice available during token round trip, we can deduce the lower bound for the worst-case reaction time without fault occurrence under static-cyclic scheduling as

[2]The maximum synchronization phase duration for periodic synchronization is T in the worst case for the dynamic IE and static-cyclic scheduling.



(a) $g_{lmax.0} = 5T_0$



(b) $g_{lmax.1} = 4T_1$

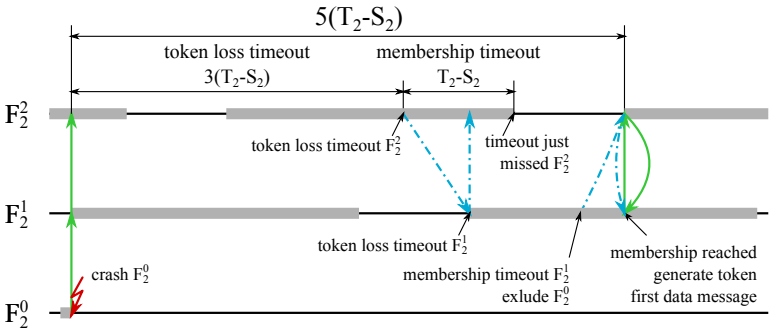
Figure 10.6: Worst-case regular operation interruption of TSM under static-cyclic scheduling with a slice duration of up to two thirds of the scheduling period.

$$a_{lmax\_nf,i} = 3r_{lmax,i}.$$

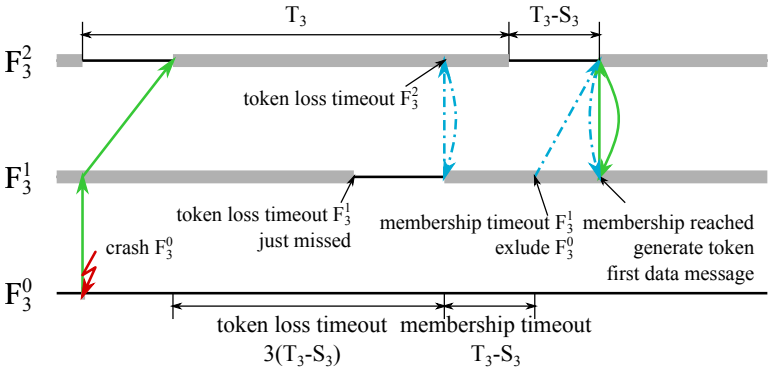
Here, the scenario is similar to the one described in Section 7.3. Again, the first token round trip is a result of just missing the event at a token forward. Distributing the event and one slice computation takes another token round trip. The results are sent and voted during the third token round trip. Compared to the fault-free reaction time in a fixed-priority scheduled system with sufficient CPU cores, the token round trip time, as well as the relative reaction time are increased; the latter by $\frac{1}{3}r_{lmax,i}$.

Since faults must be tolerated during TMR execution, the lower bound of the worst-case operation interruption time has to be added to the lower bound of the worst-case response time of the fault free case

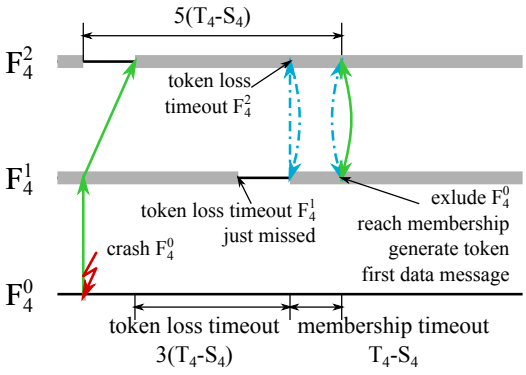
$$a_{lmax,i} = a_{lmax\_nf,i} + g_{lmax,i}.$$



(a) $g_{lmax.2} = 5(T_2 - S_2)$



(b) $g_{lmax.3} = 2T_3 - S_3$



(c) $g_{lmax.4} = 5(T_4 - S_4)$

Figure 10.7: Worst-case regular operation interruption of TSM under static-cyclic scheduling with a slice duration of more than two thirds of the scheduling period.

Although such a long response time can only be observed when a fault occurs, the application must be able to tolerate it. TSM is consequently only suitable for applications having relaxed timing requirements in this scenario. However, TSM still provides a robust synchronization service, if the application requirements fit.

Sched. Period [ms]	Sched. Slice [ms]	CPU for $\delta^{\mathtt{FS}}$	Idle CPU	Remaining CPU
100	78	1.0%	65.0%	22.0%
90	71	1.1%	65.8%	21.2%
80	64	1.3%	66.8%	20.0%
70	56	1.4%	66.6%	20.0%
60	49	1.7%	68.0%	18.3%
50	17	2.0%	20.0%	66.0%
40	14	2.5%	32.5%	65.0%
30	11	3.3%	21.3%	63.3%
20	5	5.0%	8.0%	75.0%
10	3	10.0%	8.0%	70.0%

Table 10.2: Scheduling parameters and subsequent CPU usage for the example application of $\mathcal{F}_{10}$ under static-cyclic scheduling with TSM for synchronization.

Sched. Period [ms]	Sched. Slice [ms]	CPU for $\delta^{\mathtt{FS}}$	Idle CPU	Remaining CPU
50	44	2.0%	42.0%	12.0%
40	36	2.5%	43.5%	10.0%
30	29	3.3%	49.3%	3.3%
20	17	5.0%	36.0%	15.0%
10	7	10.0%	16.0%	30.0%

Table 10.3: Scheduling parameters and subsequent CPU usage for the example application of $\mathcal{F}_{20}$ under static-cyclic scheduling with TSM for synchronization.

Table 10.2 shows the CPU usage by $\mathcal{F}_{10}$ and Table 10.3 by $\mathcal{F}_{20}$. These are the shares during fault-free operation and under the assumption that the example applications can tolerate the interruptions caused by faults. An overhead of 2ms for processing the data and time messages is also included. Apart from the scheduling period of 10ms, both applications synchronized with TSM require a few more milliseconds slice time than the periodic synchronization in the classic TMR architecture presented in Section 9.1 on page 57. Also, in the scenario with TSM only a scheduling period of 10ms is suitable for integrating $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$.

Preemptive Fixed-Priority Scheduling

With preemptive fixed-priority scheduling the analysis of synchronization performance in Section 7.3 on page 49 can be reused. Again, under the assumption that FS scheduling is the most time intensive and neglecting message transmission times and computation times, the lower bound for the worst-case token round trip time is

$$r_{lmax,i} = 3\delta^{\mathtt{FS}}.$$

Therefore, the token loss timeout must be at least $3\delta^{\mathtt{FS}}$ long. The membership timeout has to be at least $2\delta^{\mathtt{FS}}$, with one scheduling operation for each message transmission.

The overhead incurred by the scheduling operations for one safety-critical FS node can be as high as one third of the CPU time with TSM in this setting. This share can be reduced by delaying the token forwarding at the FS node for a certain time, at the cost of extending the token round trip time.

CHAPTER 11

Contracts for Certification

In our approach, contracts are the basis for the independent certification of FSs, IEs and integrated systems. In this chapter, we define a contract formalism suitable for abstracting the elements' behaviour and safely deriving properties of such integrated systems, i.e. whether the integrated FSs are provided with sufficient resources. This requires that the contracts explicitly contain all necessary (abstract) information for certification. Although the semantic gap, as described in Section 8.1, hinders the fine grain composition of FSs, it is exactly this abstraction that at the same time enables the definition of contracts which are mainly based on computational resources.

The IE architecture states additional requirements towards the contracts, as described in Chapter 4. Refining the certification strategy presented in Chapter 3 with these additional requirements, results in three types of contracts with the following properties:

1. The FS contract specifies the feature and resource requirements for safe operation of a specific FS.

2. The IE contract defines the available resources within the IE and the feasibility evaluation method for checking whether a specific configuration of FSs can be safely integrated on top of the IE. If supported, also the IE alteration operation, which enables changing the available resources of the IE, is specified in the IE contract.

3. The integration contract defines the integrated system with a mapping between the FSs contracts and the IE contract.

Such a contract is *valid*, if it correctly reflects the behaviour of the actual system. Validity must be shown during the certification processes of the IE and FSs by appropriate means, e.g. analysis of worst-case execution times. If the stated resource requirements and feasibility evaluation method are close to the actual requirements, the utilization of resources can be high for safety-critical FSs. However, as presented in the previous two chapters, the execution time

available for non-critical FSs is strongly dependent on the scheduler of the IE nodes. The feasibility evaluation method itself may treat non-critical FSs differently and assume less margin for fulfilling their requirements[1].

In the next section, we give a formal definition of the contracts, followed by a section refining parts of this contract model for TMR-based FSs. We then outline the requirements and effects regarding operations on these contracts, when altering the IE. At the end of this chapter, we discuss aspects for finding a feasible integration contract and the limitations of the approach presented here.

11.1 Formal Contract Definition

Our concepts of component and contracts follow the general ideas presented by Beneviste et al. [18]. Notations and definitions are tailored and extended for our integration- and resource-focused setting. Assumptions and guarantees are specified at a high abstraction level. It is uncommon to include the feasibility evaluation method and operations for contract alteration within contracts themselves, as they are usually part of the formalism. However, it is necessary for certification in our solution. We will not argue about the correct implementation of FSs, only about their required resources.

Figure 11.1 and 11.2 give an overview of the elements of the contract model, which we now describe in detail.

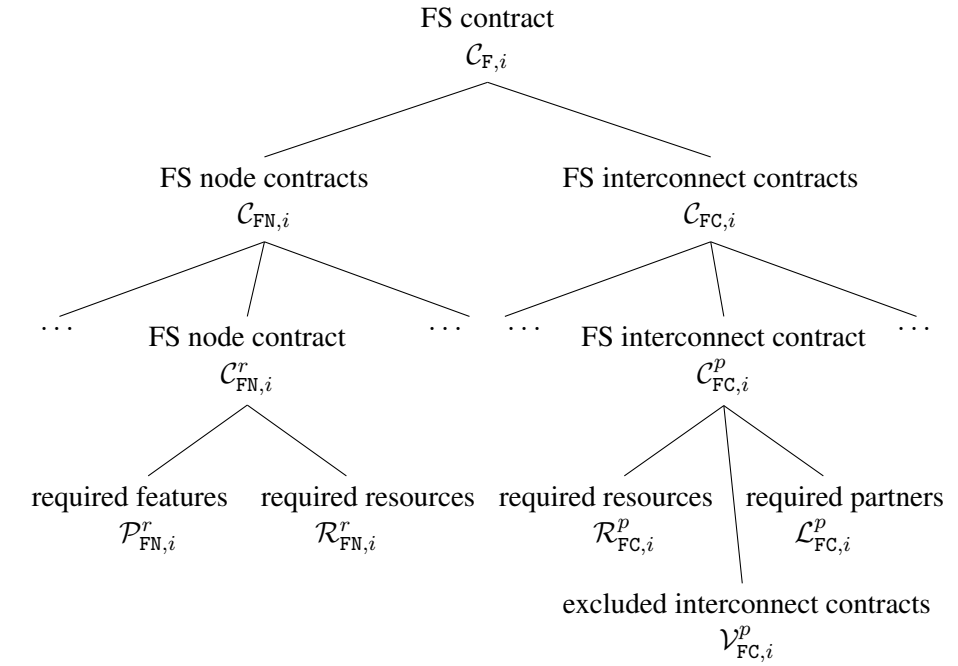
FS Contract

For each FS $\mathcal{F}_i$ with ID $i \in \mathbb{N}_0$, we define the *FS contract* $\mathcal{C}_{\mathtt{F},i} = (\mathcal{C}_{\mathtt{FN},i}, \mathcal{C}_{\mathtt{FC},i})$ as tuple of the set of corresponding *FS node contracts* $\mathcal{C}_{\mathtt{FN},i}$ and *FS interconnect contracts* $\mathcal{C}_{\mathtt{FC},i}$.

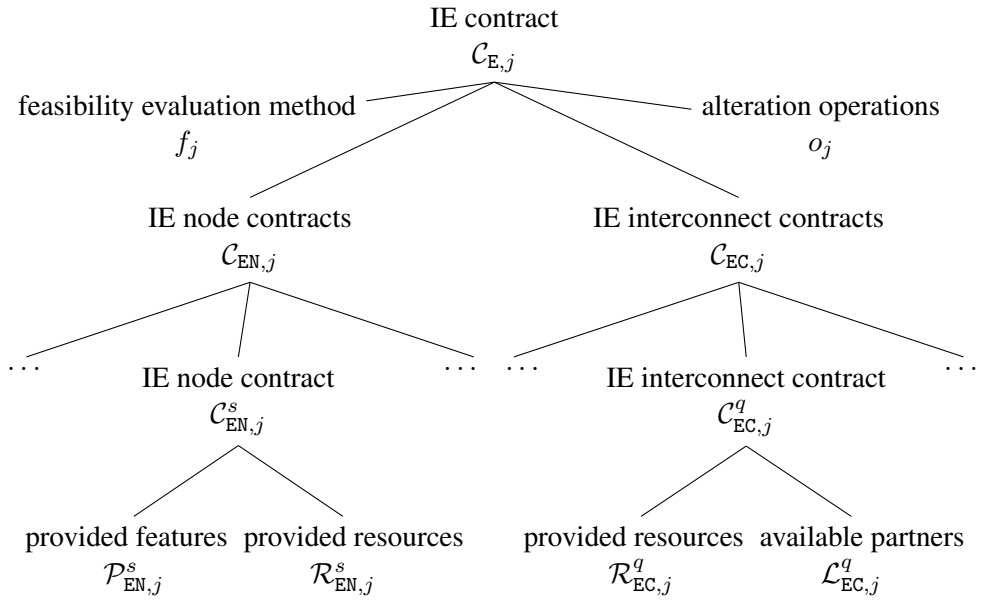
Each FS node contract is a tuple $\mathcal{C}^r_{\mathtt{FN},i} = (\mathcal{P}^r_{\mathtt{FN},i}, \mathcal{R}^r_{\mathtt{FN},i})$, with ID $r \in \mathbb{N}_0$ of the FS node and is part of the set $\mathcal{C}_{\mathtt{FN},i}$. It consists of the node's *required node features* $\mathcal{P}^r_{\mathtt{FN},i}$ and *required node resources* $\mathcal{R}^r_{\mathtt{FN},i}$. The required features are a subset of the set of all *feature identifiers* $\mathbf{F}$. Features are used twofold in this approach. The first is to provide the FS nodes with a certain functionality, e.g. a watchdog. The second are architectural requirements such as different_ie_nodes, with which an FS requires that all its FS nodes are integrated on different IE nodes. The required resources are a mapping $\mathcal{R}^r_{\mathtt{FN},i} : \mathbf{R}_{\mathtt{FN}} \rightarrow \mathbf{W} \cup \perp$ of the *required resource identifiers* $\mathbf{R}_{\mathtt{FN}}$ to the *quantity or type of resource* $\mathbf{W}$. Only if the FS node has no requirement regarding a specific resource, does the corresponding resource identifier map to $\perp$.

An FS interconnect contract is defined as tuple $\mathcal{C}^p_{\mathtt{FC},i} = (\mathcal{R}^p_{\mathtt{FC},i}, \mathcal{L}^p_{\mathtt{FC},i}, \mathcal{V}^p_{\mathtt{FC},i})$, has an ID $p \in \mathbb{N}_0$ and is included in the set $\mathcal{C}_{\mathtt{FC},i}$. It consists of the *required connection resources* $\mathcal{R}^p_{\mathtt{FC},i}$, the set of FS node contracts $\mathcal{L}^p_{\mathtt{FC},i}$ as *required communication partners* on this link and the set of *excluded interconnect contracts* $\mathcal{V}^p_{\mathtt{FC},i}$ which must not be deployed on the same IE communication links. The required connection resources are a mapping $\mathcal{R}^p_{\mathtt{FC},i} : \mathbf{R}_{\mathtt{FC}} \rightarrow \mathbf{W} \cup \perp$ of *connection resource identifiers* $\mathbf{R}_{\mathtt{FC}}$ to the quantity or type of resource $\mathbf{W}$. Again, if the FS has no requirement regarding a specific communication property, the corresponding property identifier maps to $\perp$. This contract definition reflects the communication requirements between the FS nodes of a

[1] This approach is similar to that of mixed-criticality scheduling based on worst-case execution times [116].



(a) Structure of FS contracts.



(b) Structure of IE contracts.

Figure 11.1: Structure of the FS and IE contract model.

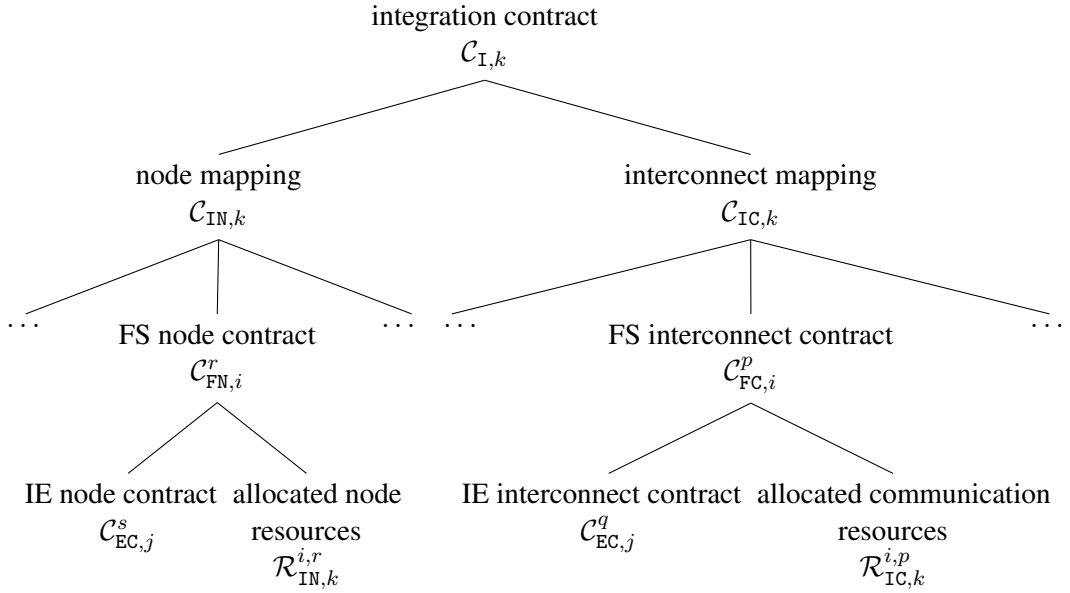


Figure 11.2: Structure of the integration contract model.

single FS, as well as between different FSs. External communication to sensors or other systems, must be modelled using local resources.

IE Contract

For each IE $\mathcal{E}_j$ with ID $j \in \mathbb{N}_0$, the associated *IE contract* $\mathcal{C}_{\mathtt{E},j} = (\mathcal{C}_{\mathtt{EN},j}, \mathcal{C}_{\mathtt{EC},j}, f_j, o_j)$ is defined as tuple of the corresponding set of *IE node contracts* $\mathcal{C}_{\mathtt{EN},j}$, the set of IE interconnect contracts $\mathcal{C}_{\mathtt{EC},j}$, the feasibility evaluation method f_j and the integration environment alteration method o_j.

An IE node contract, with the ID $s \in \mathbb{N}_0$ and which is an element of the set $\mathcal{C}_{\mathtt{EN},j}$, is a tuple $\mathcal{C}^s_{\mathtt{EN},j} = (\mathcal{P}^s_{\mathtt{EN},j}, \mathcal{R}^s_{\mathtt{EN},j})$ of the *available node features* $\mathcal{P}^s_{\mathtt{EN},j}$ and *available node resources* $\mathcal{R}^s_{\mathtt{EN},j}$. The available features are a subset of the set of all feature identifiers $\mathbf{F}$. The IE node contract offers the features of the IE node to all its integrated FS nodes without consideration of issues related to its shared use. Consequently, a limited resource must be modeled as an available resource and not a feature. The available resources are a mapping $\mathcal{R}^s_{\mathtt{EN},j} : \mathbf{R}_{\mathtt{EN}} \rightarrow \mathbf{W} \cup \perp$ of the *available resource identifiers* $\mathbf{R}_{\mathtt{EN}}$ to the quantity or type of resource $\mathbf{W}$. If the IE node does not provide a specific resource, the corresponding resource identifier maps to $\perp$.

An IE interconnect contract represents a communication link of the IE with two or more communication partners. The IE interconnect contract, having an ID $q \in \mathbb{N}_0$ and being an element of the set $\mathcal{C}_{\mathtt{EC},j}$, is defined as a tuple $\mathcal{C}^q_{\mathtt{EC},j} = (\mathcal{R}^q_{\mathtt{EC},j}, \mathcal{L}^q_{\mathtt{EC},j})$ of the *available communication resources* $\mathcal{R}^q_{\mathtt{EC},j}$ and the set of *available IE nodes* $\mathcal{L}^q_{\mathtt{EC},j}$ on this link. The available communication resources are a mapping $\mathcal{R}^q_{\mathtt{EC},j} : \mathbf{R}_{\mathtt{EC}} \rightarrow \mathbf{W} \cup \perp$ of *connection resource identifiers* $\mathbf{R}_{\mathtt{EC}}$ to quantity or type of resource $\mathbf{W}$. Again, $\perp$ defines that a communication specific resource is not available.

The feasibility evaluation method is a mapping $f_j : \mathcal{C}_{\mathtt{I}} \rightarrow \top \cup \bot$ of integration contracts to $\top \cup \bot$, where a system with a valid integration contract that maps to $\top$ provides all integrated FSs with sufficient resources. Note that, if the integration contract maps to $\bot$, it does not necessarily mean that the FSs will have insufficient resources. The feasibility evaluation method is part of the IE contract, since feasibility strongly depends on the properties of the actual components and mechanisms used within the IE and its validity must be ensured during the certification of the IE.

The integration environment alteration method o_j is a mapping from alteration operations to IE contracts.

Integration Contract

An integration contract $\mathcal{C}_{\mathtt{I},k}$ for an integrated system $\mathcal{I}_k$ with ID $k \in \mathbb{N}_0$ is defined as a tuple $\mathcal{C}_{\mathtt{I},k} = (\mathcal{C}_{\mathtt{IN},k}, \mathcal{C}_{\mathtt{IC},k})$ of *node mapping* $\mathcal{C}_{\mathtt{IN},k}$ and *interconnect mapping* $\mathcal{C}_{\mathtt{IC},k}$ with $\mathcal{C}_{\mathtt{FN}}$ as the set of all FS node contracts, $\mathcal{C}_{\mathtt{EN}}$ as the set of all IE node contracts and $\mathcal{R}_{\mathtt{IN},k}$ as the set of all *resource allocation mappings*. The node mapping is defined as $\mathcal{C}_{\mathtt{IN},k} : \mathcal{C}_{\mathtt{FN}} \rightarrow (\mathcal{C}_{\mathtt{EN}}, \mathcal{R}_{\mathtt{IN},k}) \cup \bot$. Mapping an FS node contract to $\bot$ indicates that the corresponding FS node is not part of the integrated system. Each mapping $\mathcal{R}^{i,r}_{\mathtt{IN},k} : \mathbf{R}_{\mathtt{IN}} \rightarrow \mathbf{W} \cup \bot$ defines the allocated node resources for the FS node r of FS i with assigning the *node resource identifiers* $\mathbf{R}_{\mathtt{IN}}$ a quantity or type of resource $\mathbf{W}$. Here, $\bot$ specifies that the corresponding resources are not allocated.

The interconnect mapping is defined as $\mathcal{C}_{\mathtt{IC},k} : \mathcal{C}_{\mathtt{FC}} \rightarrow (\mathcal{C}_{\mathtt{FC}}, \mathcal{R}_{\mathtt{IC},k}) \cup \bot$, with $\mathcal{C}_{\mathtt{FC}}$ as the set of all FS interconnect contracts, $\mathcal{R}_{\mathtt{IC},k}$ as the set of all *allocated communication resources mappings* and $\mathcal{C}_{\mathtt{EC}}$ as the set of all IE interconnect contracts. Again, a mapping to $\bot$ indicates that the FS interconnect is not part of the integrated system. The allocated communication resource mapping $\mathcal{R}^{i,p}_{\mathtt{IC},k} : \mathbf{R}_{\mathtt{IC}} \rightarrow \mathbf{W} \cup \bot$ specifies quantities or types of resources $\mathbf{W}$ for the allocated resources $\mathbf{R}_{\mathtt{IC}}$ on the communication link p for the FS i. As before, $\bot$ specifies that the according resources are not allocated.

An integration contract is *fulfilled*, if the contracts of all elements it integrates are valid and the feasibility evaluation method of the corresponding IE contract evaluates to $\top$ for this integration contract.

Integration contracts do not contain elements such as the feasibility evaluation method, to ensure that they are not providing safety-critical functionality themselves. As we will later see, this eases the creation of such contracts.

Figure 11.3 illustrates the elements of contracts for the example FSs $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$ integrated on an IE $\mathcal{E}_0$ with the example node mapping $\mathcal{C}_{\mathtt{IN},0}$. It omits the depiction of allocated resources and the mapping of communication contracts to provide a simple overview.

11.2 Refinement for our TMR-based FSs

We now refine the general contract definitions for our TMR-based FSs regarding node resources, node features, networking and the feasibility evaluation method for illustration and integrating the analysis of the previous chapter within the contract formalism.

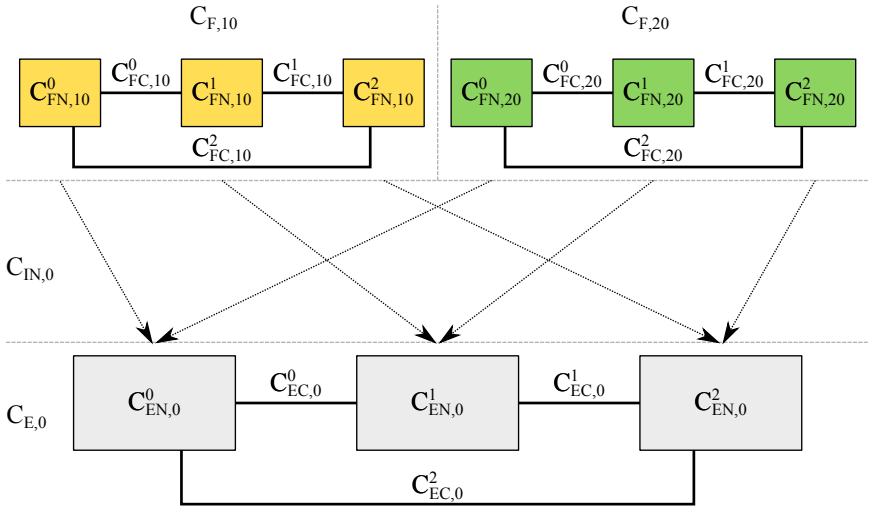


Figure 11.3: Elements of contracts for the example FSs $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$ integrated on IE $\mathcal{E}_0$ with the example node mapping $\mathcal{C}_{\texttt{IN},0}$.

Resources

$r \in \mathbf{R}_{\texttt{FN}}$	Description	Unit
disk	Disk size of FS node.	Byte
memory	Memory size of FS node.	Byte
replicated_load	Replicated workload duration of FS.	μs
period	Minimum inter-arrival time of the replicated workload.	μs
sync_algorithm	Type of used synchronization algorithm $\in \mathbf{M}$.	1

Table 11.1: Example list of elements of the resource identifier set $\mathbf{R}_{\texttt{FN}}$ for FS node contracts.

Table 11.1 lists the node resource identifiers for FS node contracts with a description and unit. As our focus is on TMR-based FSs, this definition only covers the replicated workload, as well as the minimum inter-arrival time for this workload. $\mathbf{M}$ is the set of synchronization algorithms provided by the middleware layer $\mathbf{M} = \{\texttt{ps}, \texttt{rps}, \texttt{tsm}\}$, where `ps` provides periodic synchronization, `rps` periodic synchronization on top of a redundant network and `tsm` the TSM-based synchronization with TMR extension, as described in Chapter 7.

The node resource identifiers for the IE nodes are listed with a description in Table 11.2. The set of possible schedulers is $\mathbf{S}_{\texttt{EN}} = \{\text{prio}, \text{cyclic}, \text{edf}\}$.

Table 11.3 lists the allocated resource identifiers for the integration contract. In consistency with the definitions used in earlier chapters, we define the static-cyclic scheduling period as $T_i = \mathcal{R}^{i,0}_{\texttt{IN},k}(\text{sched\_cyclic\_period})$ and $S_i = \mathcal{R}^{i,0}_{\texttt{IN},k}(\text{sched\_cyclic\_slice})$ as the static-cyclic scheduling slice of a FS $\mathcal{F}_i$ for the integrated system $\mathcal{I}_k$. The definition of the EDF scheduling is similar with the parameters $T^{\texttt{E}}_i = \mathcal{R}^{i,0}_{\texttt{IN},k}(\text{sched\_edf\_period})$ and $S^{\texttt{E}}_i = \mathcal{R}^{i,0}_{\texttt{IN},k}(\text{sched\_edf\_slice})$.

$r \in \mathbf{R}_{\mathtt{EN}}$	Description	Unit
disk	Disk size of IE node	Byte
memory	Memory size of IE node	Byte
cpu	Count of available CPU cores on IE node	1
sched	Scheduler type of IE node $\in \mathbf{S}_{\mathtt{EN}}$.	1

Table 11.2: Example list of elements of the resource identifier set $\mathbf{R}_{\mathtt{EN}}$ for IE node contracts.

$r \in \mathbf{R}_{\mathtt{IN}}$	Description	Unit
sched_prio	Scheduling priority for priority scheduling.	1
sched_cyclic_period	Scheduling period for static-cyclic scheduling.	μs
sched_cyclic_slice	Static-cyclic scheduling slice available for FS computation.	μs
sched_edf_period	Scheduling period for EDF scheduler.	μs
sched_edf_slice	Scheduling slice for EDF scheduler for FS computation.	μs

Table 11.3: Example list of elements of the resource identifier set $\mathbf{R}_{\mathtt{IN}}$ for integration contracts.

For the resource identifier sets for communication, $\mathbf{R}_{\mathtt{FC}}$, $\mathbf{R}_{\mathtt{EC}}$ and $\mathbf{R}_{\mathtt{IC}}$ are defined as identical sets consisting of only the bandwidth identifier, as we focus mainly on the CPU in our example.

Features

For illustration purposes, we define the features providing functionality on node level as watchdog and mmu. The example architectural feature used in the feasibility evaluation method presented in the following is different_ie_nodes, with which an FS requires that all its FS nodes are integrated on different IE nodes. Consequently, our set of feature identifiers is $\mathbf{F} = \{$watchdog, mmu, different_ie_nodes$\}$.

Feasibility Evaluation Method

The feasibility evaluation method we partly specify here covers integration of safety-critical FSs only and demonstrates the feature feasibility evaluation. It consists of three parts: feasibility on nodes (fn), feasibility on interconnect (fi) and consistency of the integration contract (fc):

$$f_j(\mathcal{C}_{\mathtt{I},k}) = fn_j(\mathcal{C}_{\mathtt{IN},k}) \wedge fi_j(\mathcal{C}_{\mathtt{IC},k}) \wedge fc_j(\mathcal{C}_{\mathtt{I},k}).$$

Feasibility on nodes, is again split in two parts: feasibility of node features (fnf) and feasibility of node resources (fnr):

$$fn_j(\mathcal{C}_{\mathtt{IN},k}) = fnf_j(\mathcal{C}_{\mathtt{IN},k}) \wedge fnr_j(\mathcal{C}_{\mathtt{IN},k}).$$

Likewise feasibility on interconnect and consistency of the integration contract can be broken down.

Let $integ\_fn(\mathcal{C}_{\mathtt{IN},k}, \mathcal{C}^s_{\mathtt{EN},j}) = \{e | e \mapsto (\mathcal{C}^s_{\mathtt{EN},j}, b) \in \mathcal{C}_{\mathtt{IN},k}\}$ be the set of FS nodes integrated on the IE node. Feature feasibility for the integrated system is satisfied, if it is satisfied for the FS nodes on each IE node locally. The function $fnfn$ checks the **f**easibility of **n**ode **f**eatures for a single **n**ode:

$$fnf_j(\mathcal{C}_{\mathtt{IN},k}) = \bigwedge_{\mathcal{C}^s_{\mathtt{EN},j} \in \mathcal{C}_{\mathtt{EN},j}} fnfn_j(\mathcal{C}_{\mathtt{IN},k}, \mathcal{C}^s_{\mathtt{EN},j}).$$

This local check is separated in the functions for checking the availability of features providing functionality ($fnfn\_func$) and the fulfillment of architectural features ($fnfn\_arch$):

$$\begin{aligned}
fnfn_j(\mathcal{C}_{\mathtt{IN},k}, \mathcal{C}_{\mathtt{EN},j}) &= fnfn_func(\mathcal{C}_{\mathtt{IN},k}, \mathcal{C}_{\mathtt{EN},j}) \wedge fnfn_arch(\mathcal{C}_{\mathtt{IN},k}, \mathcal{C}_{\mathtt{EN},j}) \\
fnfn_func_j(\mathcal{C}_{\mathtt{IN},k}, \mathcal{C}^s_{\mathtt{EN},j}) &= \bigwedge_{\mathcal{C}^r_{\mathtt{FN},i} \in integ_fn(\mathcal{C}_{\mathtt{IN},k}, \mathcal{C}^s_{\mathtt{EN},j})} \mathcal{P}^r_{\mathtt{FN},i} \subseteq \mathcal{P}^s_{\mathtt{EN},j} \\
fnfn_arch_j(\mathcal{C}_{\mathtt{IN},k}, \mathcal{C}^s_{\mathtt{EN},j}) &= \bigwedge_{\mathcal{C}^r_{\mathtt{FN},i} \in integ_fn(\mathcal{C}_{\mathtt{IN},k}, \mathcal{C}^s_{\mathtt{EN},j})} indep_j(\mathcal{C}_{\mathtt{IN},k}, \mathcal{C}^s_{\mathtt{EN},j}, \mathcal{C}^r_{\mathtt{FN},i}) \\
indep_j(\mathcal{C}_{\mathtt{IN},k}, \mathcal{C}^s_{\mathtt{EN},j}, \mathcal{C}^r_{\mathtt{FN},i}) &= \text{different_ie_nodes} \in \mathcal{P}^r_{\mathtt{FN},i} \implies \\
&\quad \forall \mathcal{P}^b_{\mathtt{FN},a} \big(\mathcal{P}^b_{\mathtt{FN},a} \in integ_fn(\mathcal{C}^s_{\mathtt{EN},j}) \wedge a = i \implies r = b \big).
\end{aligned}$$

A violation of the feature different_ie_nodes by the integrated system $\mathcal{I}_k$ is detected via the $indep$ function. In case of such a violation, more than one FS node r of the FS i is then integrated on the IE node s of the IE j. Consequently, not all $\mathcal{P}^b_{\mathtt{FN},a}$ of the deployed nodes of the FS i have the ID r and subsequently the corresponding implication evaluates to $\bot$.

Note that, with the definition of $fnfn\_func$, the IE node contracts must also list that they provide the architectural features. Likewise, feasibility of resources and interconnect can be broken down and checked. The scheduling overheads for the CPU assignment can be derived from the analysis of the previous chapter together with a timing analysis of the actual IE. This shows that the feasibility evaluation method is strongly dependent on the IE itself.

Since the certified feasibility evaluation method decides whether an integrated system can satisfy all the FSs' resource requirements, it must also include a check of the consistency of the integration contract. With the definition based on mappings, the consistency check consists of two parts. The first part, checking the consistency of the mapping (fcm), is to to determine whether all FS nodes ($fcmin$) and corresponding interconnections ($fcmic$) are included and mapped ($fcmn$, $fcmc$) only to elements of the associated IE:

$$fcm_j(\mathcal{C}_{\mathtt{I},k}) = fcmin_j(\mathcal{C}_{\mathtt{IN},k}) \wedge fcmic_j(\mathcal{C}_{\mathtt{IC},k}) \wedge fcmn_j(\mathcal{C}_{\mathtt{I},k}) \wedge fcmc_j(\mathcal{C}_{\mathtt{I},k}).$$

Ensuring that all nodes of the integrated FSs are included, is checked with asserting that if one node or interconnect contract of an FS is mapped to an IE element, no other node or interconnect contract of this FS is mapped to $\perp$:

$$\begin{aligned}
fcmin_j(\mathcal{C}_{\mathtt{IN},k}) = \forall \mathcal{C}^b_{\mathtt{FN},a} \Big(& \mathcal{C}^b_{\mathtt{FN},a} \mapsto c \in \mathcal{C}_{\mathtt{IN},k} \wedge c \neq \perp \\
& \implies \nexists \mathcal{C}^d_{\mathtt{FN},a} \big(\mathcal{C}^d_{\mathtt{FN},a} \mapsto \perp \in \mathcal{C}_{\mathtt{IN},k} \big) \wedge \nexists \mathcal{C}^d_{\mathtt{FC},a} \big(\mathcal{C}^d_{\mathtt{FC},a} \mapsto \perp \in \mathcal{C}_{\mathtt{IC},k} \big) \Big) \\
fcmic_j(\mathcal{C}_{\mathtt{IC},k}) = \forall \mathcal{C}^b_{\mathtt{FC},a} \Big(& \mathcal{C}^b_{\mathtt{FC},a} \mapsto c \in \mathcal{C}_{\mathtt{IC},k} \wedge c \neq \perp \\
& \implies \nexists \mathcal{C}^d_{\mathtt{FN},a} \big(\mathcal{C}^d_{\mathtt{FN},a} \mapsto \perp \in \mathcal{C}_{\mathtt{IN},k} \big) \wedge \nexists \mathcal{C}^d_{\mathtt{FC},a} \big(\mathcal{C}^d_{\mathtt{FC},a} \mapsto \perp \in \mathcal{C}_{\mathtt{IC},k} \big) \Big)
\end{aligned}$$

Testing whether the integration contract only maps the elements of the integrated FS only to elements of the IE for which the feasibility evaluation function is defined, is performed by checking that the ID of the associated IE elements are the same as that of the IE of the feasibility evaluation method:

$$\begin{aligned}
fcmn_j(\mathcal{C}_{\mathtt{IN},k}) &= \forall a \big(a \mapsto (\mathcal{C}^c_{\mathtt{EN},b}, d) \in \mathcal{C}_{\mathtt{IN},k} \implies b = j \big) \\
fcmc_j(\mathcal{C}_{\mathtt{IC},k}) &= \forall a \big(a \mapsto (\mathcal{C}^c_{\mathtt{EC},b}, d) \in \mathcal{C}_{\mathtt{IC},k} \implies b = j \big)
\end{aligned}$$

The second part of evaluating consistency is to check whether the allocated resources are in accordance with the required resources. For this check, the analysis of the previous chapters enables us to determine whether the replicated workload can be executed with the allocated CPU resources. For most other properties it is sufficient to check equivalence.

Provided that the FS contracts and IE contracts of the integrated system are valid, the feasibility evaluation method ensures that no incorrect integration contract can be applied.

11.3 Alterations of the IE

Altering the IE and associated IE contract is a safety-critical task. The risk here is that an operation on a valid IE contract may result in an invalid IE contract. This may lead to an integrated system in which the FSs' requirements are not fulfilled, and therefore, has an increased risk of causing a hazard. To prevent such a case, these operations must be certified together with the IE contract and executed by a safety-critical entity, i.e. the FS manager, as already discussed at the end of Chapter 4 on page 23.

Based on the definition above, two strategies for certification can now be applied. The first is to have a limited set of possible IE configurations and associated contracts, using o_j to switch in-between them. During the certification process this limited set of IE configurations and contracts must then be evaluated. In the second option, a set of operations is defined with the mapping of o_j resulting in new IE contracts. These operations are required to have predictable and bounded effects on the IE. Here, the certification process must ensure that such IE contracts are valid with respect to the behaviour of the new IE. This can be achieved defining a range in which such operations can be applied and by determining the behaviour of the integrated systems at the boundaries of this range.

11.4 Applying Integration Contracts

Finding an integration contract can be accomplished using any suitable planning or optimization algorithm [49]. It is neither time- nor safety-critical, since the safety-critical FS manager checks the feasibility of the integration contract with the method defined in the IE contract before it starts deployment. Consequently, the validity of the IE and FS contracts provided to the FS manager is crucial for safety, while non-optimal or incorrect integration contracts must by definition have no impact on safety. However, as the FS manager is the checking instance it must not be used as rating function by a planning algorithm itself. Still, fast and correct creation of integration contracts is beneficial for availability as it ensures that FSs can provide their services soon after boot or deployment.

11.5 Limitations of the Presented Contract Concept

In our model, we assumed that message latency through the network has only a minor effect and thus, the contract definition does not include it. Furthermore, only one (undefined) type of CPU is used. Also, representing the features required by the FSs and provided by the IE only with unique identifiers is a direct, but ultimately impracticable, approach. Especially with the adaptable dynamic IE, the IE nodes most likely will not have the same versions of their provided features. Also, the FS nodes will need a specific or compatible version of a feature. Consequently, a productive system must extend the concept of feature identifiers with version numbers. This additionally requires a (safety-critical) mechanism which determines whether the versions of the features provided by the IE node are compatible to the ones required by the FSs.

CHAPTER 12

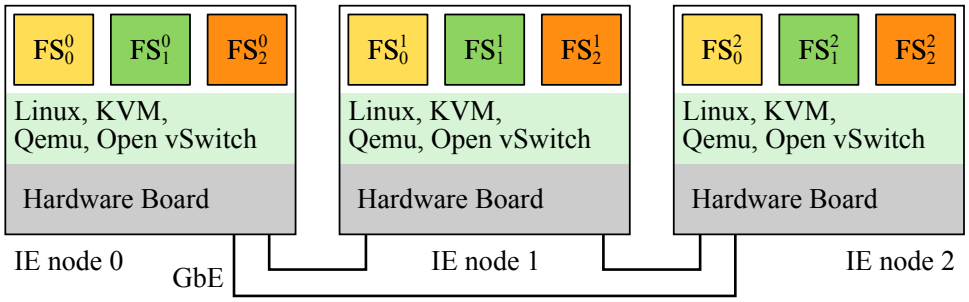
Evaluation

This chapter presents the evaluation of the proposed concepts for composable solutions. It covers both approaches, namely the classic TMR IE, as well as the dynamic IE. The full solution space, as presented in Chapter 9 and Chapter 10, is summarized in Table 12.1. Also, examples for the contracts presented in the previous chapter are provided. Simulations and a setup with a prototype were used for the evaluation.

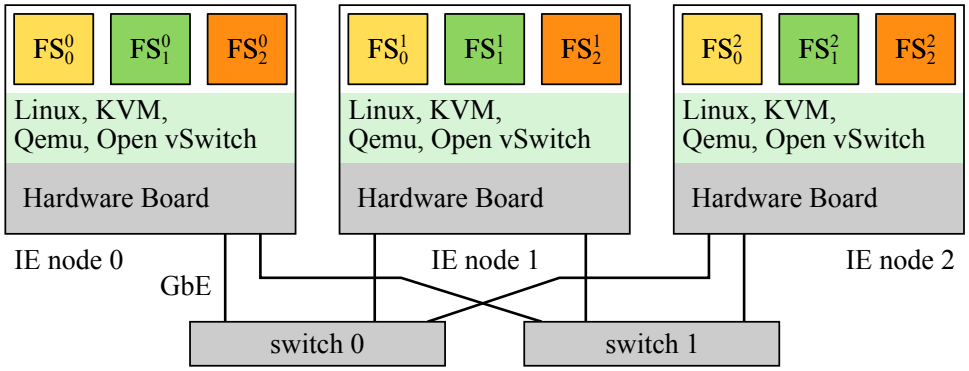
Scheduler	Classic TMR IE	Dynamic TMR IE
static-cyclic	inefficient CPU usage	inefficient CPU usage
fixed-priority	dedicated core solution, or synchronization as IE service solution (violating Constraint 2)	dedicated core solution
EDF	expensive to implement (violating Constraint 1)	FS buffer solution (violating Constraint 1)

Table 12.1: Solution space for IE architecture and scheduler.

The next section specifies the setup of the prototype, followed by a section on the measurement results of the scheduling operations' duration within the prototype. The simulation results regarding IEs providing static-cyclic scheduling are described in Section 12.3. The measurement results obtained with the preemptive fixed-priority-based prototype and corresponding IE network architectures are presented in Section 12.4. Finally, the possibilities for implementing an EDF-based IE and corresponding FSs are evaluated in Section 12.5.



(a) Setup for the static IE with direct links between IE nodes.



(b) Setup for the dynamic IE with redundant network.

Figure 12.1: Prototype setups for different IEs integrating three replicated FSs.

12.1 Prototype Setup

The prototype is based on three P4080 hardware boards. On top of each of these boards Linux, KVM, Qemu and Open vSwitch provide the composability layer. The recommendations given by IBM [64] for virtual environments with KVM were followed when applicable. All virtual machines execute exactly one FS node each. The individual test scenarios ran for approximately three hours, unless the synchronization algorithm failed during execution.

Communication is based on UDP messages and the hardware boards are connected through gigabit Ethernet links. These links are either directly connected in the configuration for the classic TMR IE, or via switches for the dynamic IE as illustrated in Figure 12.1. No bandwidth restriction is applied in the prototype, given that the applications restrict themselves for the tests, but VLANs ensure that only the specified communication partners are reachable.

For the various tests the number of available CPU cores was defined using the Linux boot parameter “maxcpus”. The CPU cores used by the system tasks and FS nodes were not separated and the FS nodes were not pinned to certain CPU cores. For the purpose of restricting the scheduling behaviour in this way, the Linux kernel provides the control group mechanism. However, enabling this feature lead to unpredictable delays for the scheduling operations and therefore, control groups were not used.

The TMR applications executed within the prototype are based on the example application $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$ of Chapter 9 and Chapter 10. Instances of $\mathcal{F}_{10}$ execute for at most 10ms every 100ms and produce an output of 512Bytes each period. The instances of $\mathcal{F}_{20}$ execute for at most 20ms every 50ms and distribute 512Bytes each period, as well.

This specification of the FSs can also be given within the contract model. For the purpose of illustration, the FS contract $\mathcal{C}_{\mathtt{F},20}$ for the FS $\mathcal{F}_{20}$ when using the periodic synchronization algorithm, is defined as:

$$
\begin{aligned}
\mathcal{C}_{\mathtt{FN},20} &= \{\mathcal{C}^0_{\mathtt{FN},20}, \mathcal{C}^1_{\mathtt{FN},20}, \mathcal{C}^2_{\mathtt{FN},20}\} \\
\mathcal{P}^0_{\mathtt{FN},20} &= \mathcal{P}^1_{\mathtt{FN},20} = \mathcal{P}^2_{\mathtt{FN},20} = \{\text{different\_ie\_nodes}\} \\
\mathcal{R}^0_{\mathtt{FN},20} &= \mathcal{R}^1_{\mathtt{FN},20} = \mathcal{R}^2_{\mathtt{FN},20} \\
\mathcal{R}^0_{\mathtt{FN},20}(r) &= \begin{cases} 100\text{MiB} & \text{if } r = \text{disk} \\ 210\text{MiB} & \text{if } r = \text{memory} \\ 20\text{ms} & \text{if } r = \text{replicated\_load} \\ 50\text{ms} & \text{if } r = \text{period} \\ \mathtt{ps} & \text{if } r = \text{sync\_algorithm} \\ \bot & \text{otherwise} \end{cases} \\
\mathcal{C}_{\mathtt{FC},20} &= \{\mathcal{C}^0_{\mathtt{FC},20}, \mathcal{C}^1_{\mathtt{FC},20}, \mathcal{C}^2_{\mathtt{FC},20}\} \\
\mathcal{L}^0_{\mathtt{FC},20} &= \{\mathcal{C}^0_{\mathtt{FN},20}, \mathcal{C}^1_{\mathtt{FN},20}\}; \mathcal{L}^1_{\mathtt{FC},20} = \{\mathcal{C}^1_{\mathtt{FN},20}, \mathcal{C}^2_{\mathtt{FN},20}\}; \mathcal{L}^2_{\mathtt{FC},20} = \{\mathcal{C}^0_{\mathtt{FN},20}, \mathcal{C}^2_{\mathtt{FN},20}\} \\
\mathcal{V}^0_{\mathtt{FC},20} &= \{\mathcal{L}^1_{\mathtt{FC},20}, \mathcal{L}^2_{\mathtt{FC},20}\}; \mathcal{V}^1_{\mathtt{FC},20} = \{\mathcal{L}^0_{\mathtt{FC},20}, \mathcal{L}^2_{\mathtt{FC},20}\}; \mathcal{V}^2_{\mathtt{FC},20} = \{\mathcal{L}^0_{\mathtt{FC},20}, \mathcal{L}^1_{\mathtt{FC},20}\} \\
\mathcal{R}^0_{\mathtt{FC},20} &= \mathcal{R}^1_{\mathtt{FC},20} = \mathcal{R}^2_{\mathtt{FC},20} \\
\mathcal{R}^0_{\mathtt{FC},20}(r) &= \begin{cases} 21.28\text{Kbit/s} & \text{if } r = \text{bandwidth} \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

Within the contract, the bandwidth requirements also account for the UDP packet headers and scheduling is specified according to the period synchronization algorithm's behaviour. Also, the required sizes of disk and memory are included in the contract, as well as the necessity that the FS nodes are executed on different IE nodes.

The next section describes example timing measurements of scheduling operations with this prototype.

12.2 Example Timing Measurements of Scheduling Operations

With the Linux ftrace interface [104] and the synchronization test setups presented below, as well as a ping-pong application, the times required for the scheduling operations specified in Section 6.4 on page 38 were measured and the results listed in Table 12.2. The architecture column indicates whether the connection between the IE nodes was direct (classic) or via a switch (dynamic). The architectural type is omitted for IE node local scheduling decisions, since no networking hardware is involved in this case. During the test for $\delta^{\mathtt{FS}}$ no parallel FS node schedule occurred, but the cache was empty. The initial setup of the shadow page table

Scheduling Operations	Value	Architecture
$\delta^{\texttt{drv}} + \delta^{\texttt{DMA}} + \delta^{\texttt{net}} + \delta^{\texttt{DMA}} + \delta^{\texttt{drv}} + \delta^{\texttt{task}}$	148μs	classic
$\delta^{\texttt{drv}} + \delta^{\texttt{DMA}} + \delta^{\texttt{net}} + \delta^{\texttt{DMA}} + \delta^{\texttt{drv}} + \delta^{\texttt{task}}$	152μs	dynamic
$\delta^{\texttt{vdrv}}$	171μs	-
$\delta^{\texttt{drv}} + \delta^{\texttt{task}}$ (in FS node)	316μs	-
$\delta^{\texttt{FS}}$	984μs	-

Table 12.2: Example timing measurements of scheduling operations.

creation for the FS node was accounted to $\delta^{\texttt{FS}}$, where the page faults occurred with less than 10μs interruption in-between. The first interruption, which was longer than those 10μs, had a duration of 2ms. As can be seen, $\delta^{\texttt{FS}}$ is by far the most time consuming operation, taking almost 1ms. The second most time consuming is $\delta^{\texttt{drv}} + \delta^{\texttt{task}}$ within the FS, which is about a third of $\delta^{\texttt{FS}}$.

12.3 Static-Cyclic Scheduling

Since the Linux kernel does not provide static-cyclic scheduling, simulation was used to evaluate the synchronization mechanisms' behaviour under this scheduling scheme. The loading duration of a partition is assumed as 1ms, in accordance with the measured $\delta^{\texttt{FS}}$, which is simply subtracted from the scheduling slice in the simulation. The transmission times of communication links are simulated using normal distributed random numbers with a mean of $\mu = 0.6$ms and a standard deviation of $\sigma = 0.06$ms. The mean transmission time was also selected according to the measurements presented in the section above and taking into account that $\delta^{\texttt{task}}$ on IE node level is included in $\delta^{\texttt{vdrv}}$. Zero computation time is accounted for the execution of the algorithms themselves. The time granularity of all simulations is 10μs and each simulation run covered $5s$. All local clocks are perfectly synchronous and the scheduling pattern only differs in the (initial) offsets of the scheduling slices. In all simulation runs the scheduling period was set to 50ms and the offset of FS node 0 to 0ms. The offsets of the other two FS nodes covered all permutations in the full range from 0ms to 49ms with steps of 1ms. Since the remaining time for computation can be deduced from the time required by the synchronization mechanism, the workloads themselves were not simulated.

Periodic Synchronization

Figure 12.2 shows the simulation results for the maximum synchronization phase durations. The lower bounds derived with s_{lmax} (defined in Section 9.1) are indicated on the y-axis. The scenarios with the scheduling slices S_1 to S_4 cover the ratios between scheduling slice and scheduling period differentiated by s_{lmax}. As seen in the figure, the maximum synchronization phase duration is the sum of the network transmission time and the lower bound for the worst case derived with s_{lmax} for S_2 and S_4. Recalling Figure 9.2 on page 59, these two scenarios are those in which the sending of the last message is delayed by the scheduler on the sending node. The

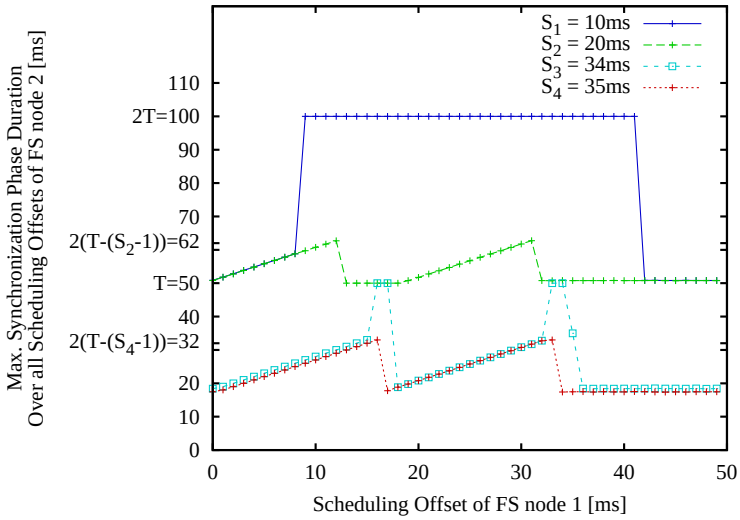


Figure 12.2: Maximum synchronization phase duration of periodic synchronization with direct links and static-cyclic scheduling slices of 10ms, 20ms, 34ms and 35ms.

delay of the corresponding reception occurs only indirectly. In the other two cases, the last message reception is a direct result of the late scheduling of the scheduler on the receiving node and identical with the value obtained using s_{lmax}. Therefore, for periodic synchronization with a redundant network and static-cyclic scheduling, the lower bound for the maximum synchronization phase duration s_{lmax} (defined in Section 9.1) is a good approximation for the actual maximum synchronization phase duration.

Figure 12.3 shows the maximum synchronization phase duration of periodic synchronization in a redundant network in correlation to the relative offsets between the FS node schedulers. These simulation results are the same as the values for the estimated lower bounds s_{lmax} defined in Section 10.1. The late message receptions in the worst case are always a direct result of a delayed scheduling by the local scheduler. Consequently, the message transmission time has no influence on the worst-case synchronization phase duration.

TSM

The lower bounds for the maximum token round trip times r_{lmax} (Section 10.2) are a good approximation for the worst-case token round trip time in the fault-free case, as can be seen with the simulation results in Figure 12.4. The simulated maximum round trip times for S_1 and S_2 are the same as those obtained with r_{lmax}, which are indicated on the y-axis. For S_3, as in the case of some scenarios of periodic synchronization found above, the worst case for the simulation is caused by the delayed sending of the last message. Consequently, the resulting worst-case round trip time is r_{lmax} plus the message transmission time. The three scenarios cover all three different cases of r_{lmax}.

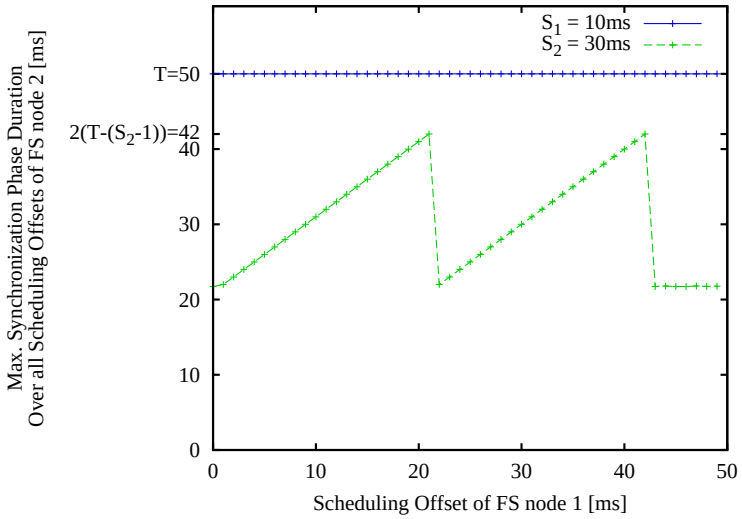


Figure 12.3: Maximum synchronization phase duration of periodic synchronization with a redundant network and static-cyclic scheduling slices of 10ms and 30ms.

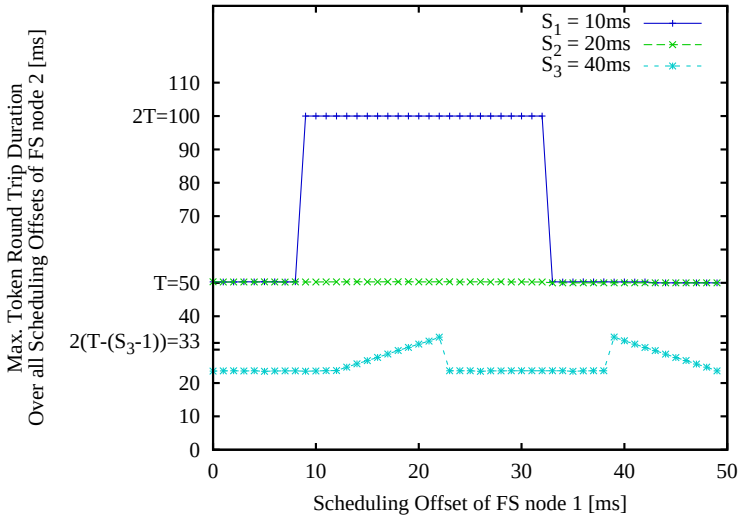


Figure 12.4: Maximum token round trip duration with static-cyclic scheduling slices of 10ms, 20ms and 40ms.

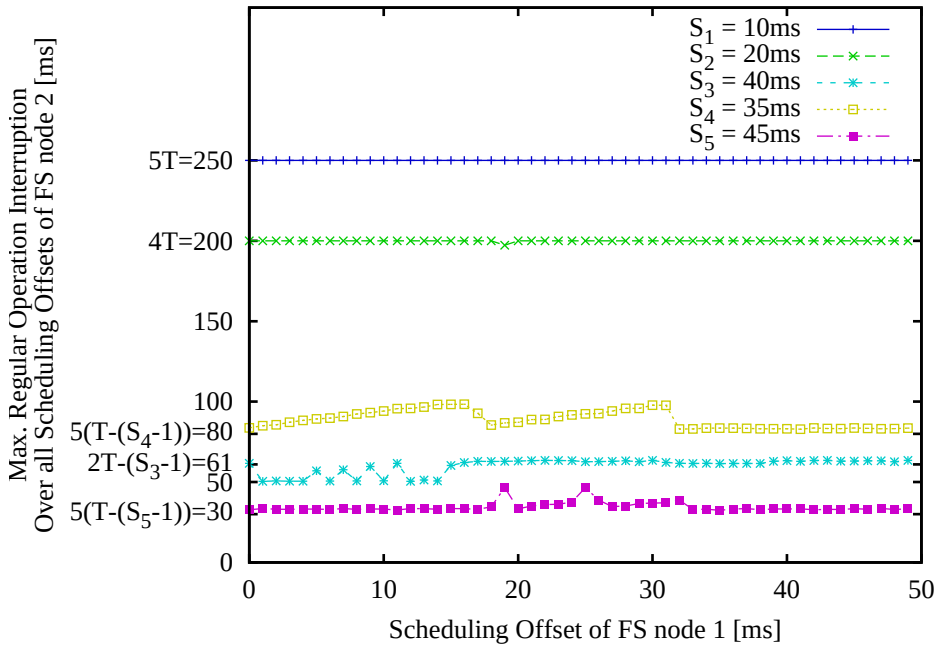


Figure 12.5: Maximum TSM regular operation interruption when an FS node crashes with static-cyclic scheduling slices of 10ms, 20ms and 40ms.

The regular operation interruption time when an FS node has crashed is another important characteristic of TSM, which the application must be able to tolerate, as described in Section 7.3. Figure 12.5 shows the maximum interruption times obtained by using the same simulation slices for S_1, S_2 and S_3 as for the token round trip time before and setting the timeouts to their smallest possible value. Two additional scenarios were added, to cover all cases of g_{lmax}. S_4 with a slice duration of 35ms and S_5 with a duration of 45ms. The crash behaviour of the FS node was modeled as fail silent. The interruption times for S_1 and S_2 correspond to the lower bound for the regular operation interruption time g_{lmax} as specified in Section 10.2 and indicated on the y-axis. The interruption times for S_3 to S_5, i.e. the scenarios where the slice to period ratio is $T_i < \frac{3}{2}S_i$, are underestimated substantially. While there is the delay of two transmission times missing for S_3, the membership timeouts for S_4 and S_5 were set to low in the definition of $g_{lmax,i}$. This originates in the assumption of zero message transmission and computation times during analysis, which is suitable for the slice to period ratio of S_1 and S_2. Consequently, a different formula should be used for determining the regular operation interruption time when integrating FS nodes with long scheduling slices.

These simulations demonstrate that the example scenarios presented during the analysis for integrating $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$ provide realistic values for the CPU usage. They also show that, with the exception of $g_{lmax,i}$ for scheduling period to slice ratios of $T_i < \frac{3}{2}S_i$, these lower bounds together with a detailed timing analysis of the scheduling operations on the IE node provide a reliable basis for the feasibility evaluation method defined in the IE contract.

12.4 Preemptive Fixed-Priority Scheduling

As established in the previous chapters, two composable solutions exist for static IEs providing fixed-priority scheduling: to reserve a single CPU core per FS node or provide synchronization as an IE service. For the dynamic IE, the options are reduced to the dedicated core solution[1].

Synchronization within FS Nodes

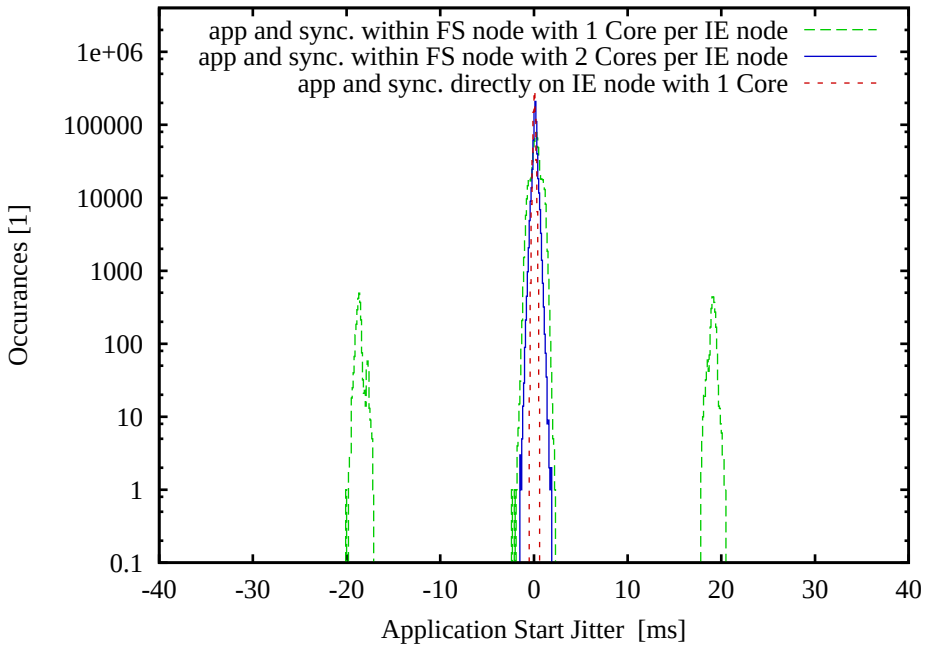


Figure 12.6: Application start jitter when executing the periodic synchronization mechanism in three different scenarios.

To illustrate the influence of the composability layer and number of available CPU cores, the periodic synchronization mechanism has been executed together with one $\mathcal{F}_{20}$ type of application in three different scenarios. In the first one, the application and synchronization mechanism were executed directly on the IE node, i.e. without a virtualization layer, and with one core enabled on each IE node. For the second scenario, they were executed in the FS node with one CPU core available on each IE node. Here, the requirement for a dedicated CPU core is violated. In the last scenario, both were executed within the FS node and two CPU cores were provided on each IE node. At most one FS node was executed per IE node in this test. Figure 12.6 shows the different application jitter measurements when executing these scenarios. Here, the additional overhead of virtualization can be seen with the increase of the jitter from the direct execution

[1] As described above, this is realized in the prototype by providing the defined number of cores, but without pinning of FS nodes to cores.

on the IE node to the execution within the FS node (virtual machine) having two cores enabled. Another visible aspect is the very large increase in jitter for the case where the synchronization mechanism and application are executed within the FS nodes but have only one CPU core enabled on each IE node. This is caused by scheduling the application within the FS node, before the last message has been sent on the IE node. This message is sent, as soon as the application finishes execution. When providing one CPU core for FS node execution and another one for the IE node's services, the jitter is only increased by the additional scheduling operation $\delta^{\mathtt{FS}}$, as the sending task can be executed in parallel.

In another scenario, where the FS nodes for $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$ are executed also on one CPU core together with the rest of the IE node's processes, $\mathcal{F}_{20}$ lost synchronization within a few seconds after the start. Although it was assigned a higher priority than $\mathcal{F}_{10}$, it could not tolerate the jitter of more than 25ms.

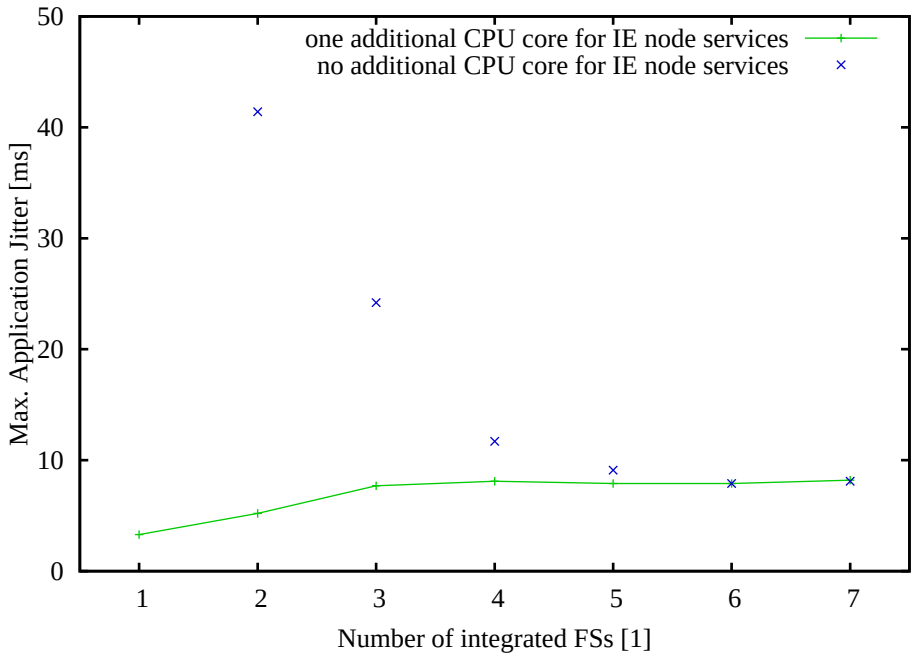


Figure 12.7: Maximum application jitter in dependency of the number of integrated FSs with one CPU core per FS node. Lines drawn to show the trend only.

The strategy for providing one CPU core per FS node and one additional for the IE node's services was successful as can be seen with the application jitter for "required number of CPU cores provided" in Figure 12.7. For the measured tests the jitter increases to 8.1ms with four integrated FSs and then remains stable. Taking the formula of the lower bound for the synchronization phase duration $s_{lmax} = 5\delta^{\mathtt{FS}}_{\max}$ (defined in Section 9.2) and assuming 1ms for $\delta^{\mathtt{FS}}_{\max}$, then the lower bound for the maximum jitter in this setup is just below 10ms, depending on the minimum synchronization phase duration. Consequently, the lower bound s_{lmax} is a good starting point for defining a feasibility evaluation method for an IE with this architecture.

When providing only one CPU core for each FS node and no separate one for the IE nodes' tasks, i.e. violating the IE contract defined below, the measured jitter rises, as depicted the Figure 12.7. A high jitter was measured for the scenarios with two and three FSs, but not for the others. Here, we see the influence between the FS nodes for the first two cases, where it is certain that predictability is no longer given. The worst-cases for the test scenarios only occur when the execution of the FS nodes are aligned. This becomes less likely over the number of FSs, which is why the observed maximum application jitter decreases over the number of integrated FSs.

These test results show that if FSs provide the synchronization mechanism themselves, the IE must provide at least one dedicated CPU core on the IE nodes per integrated FS node. This must be enforced within the feasibility evaluation method of the IE contract.

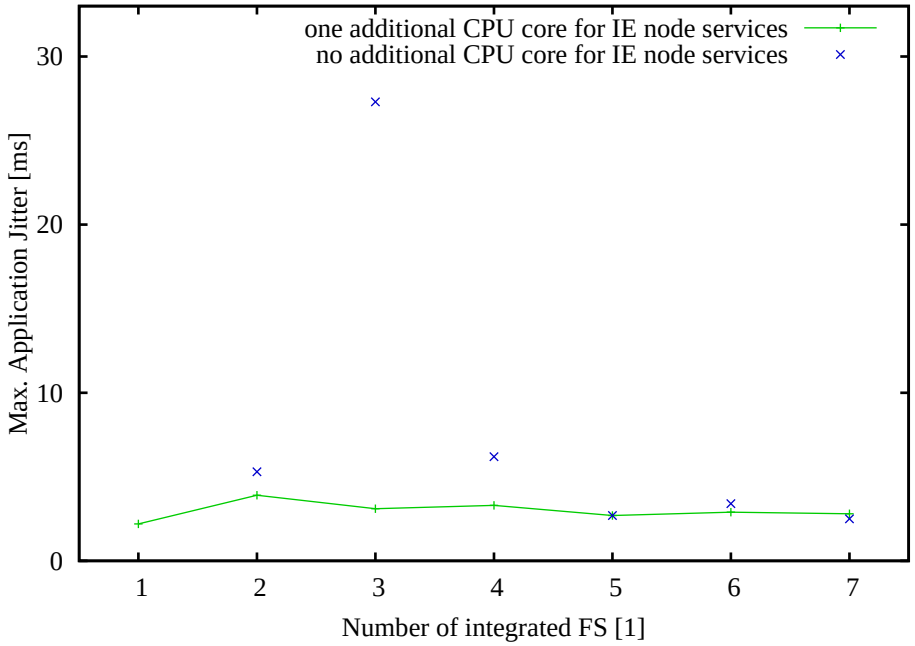


Figure 12.8: Maximum application jitter in dependency of the number of integrated FS, with one CPU core per FS node and periodic synchronization in a redundant network. Lines drawn to show the trend only.

Similar results are achieved with the periodic synchronization mechanisms for redundant networks in the dynamic IE. Figure 12.8 shows a reduction of the synchronization jitter in this setting, when compared with the results above for periodic synchronization with directly connected nodes. Here, the formula for the lower bound for synchronization duration is $s_{lmax} = 3\delta^{\mathrm{FS}}_{\max}$ (defined in Section 10.1), which predicts a jitter of just below 6ms, under the same assumptions as for periodic scheduling above. This is also a good starting point for the feasibility evaluation method.

The measurements concerning a violation of the CPU requirements, showed a high jitter when integrating three FSs with exactly three CPU cores enabled. Also, here, the worst-cases

occur with aligned FS executions, which was only observed for the scenario with three integrated FSs. Similar to Figure 12.7, the measured values can be seen as lower bound of the maximum application jitter.

Using the TSM algorithm for synchronization within the FS nodes results in a completely different behaviour of the overall system. In the configuration with only one FS on top of the IE, the token round trip time stays under 100μs, which causes a significant network load. Here, the initial assumption, that FSs will have a mostly CPU- and not network-intensive profile, does not apply any more. Consequently, within our prototype, the lower priority FSs lose synchronization even when providing one CPU core per FS node. Thus, TSM without a restriction on the token speed is not suitable for constructing a composable environment with a preemptive fixed-priority IE.

Contracts for the IE Scenario with Synchronization within the FS node

For illustration, the IE contract for the prototype IE $\mathcal{E}_0$ with synchronization service and classic TMR architecture is:

$$
\begin{aligned}
\mathcal{C}_{\mathtt{E},0} &= \{\mathcal{C}_{\mathtt{EN},0}, \mathcal{C}_{\mathtt{EC},0}, f_0, \emptyset\} \\
\mathcal{C}_{\mathtt{EN},0} &= \{\mathcal{C}^0_{\mathtt{EN},0}, \mathcal{C}^1_{\mathtt{EN},0}, \mathcal{C}^2_{\mathtt{EN},0}\} \\
\mathcal{P}^0_{\mathtt{EN},0} &= \mathcal{P}^1_{\mathtt{EN},0} = \mathcal{P}^2_{\mathtt{EN},0} = \{\text{mmu}, \text{different\_ie\_nodes}\} \\
\mathcal{R}^0_{\mathtt{EN},0} &= \mathcal{R}^1_{\mathtt{EN},0} = \mathcal{R}^2_{\mathtt{EN},0} \\
\mathcal{R}^0_{\mathtt{EN},0}(r) &= \begin{cases} 20\text{GiB} & \text{if } r = \text{disk} \\ 4\text{GiB} & \text{if } r = \text{memory} \\ 8 & \text{if } r = \text{cpu} \\ \text{prio} & \text{if } r = \text{sched} \\ \perp & \text{otherwise} \end{cases} \\
\mathcal{C}_{\mathtt{EC},0} &= \{\mathcal{C}^0_{\mathtt{EC},0}, \mathcal{C}^1_{\mathtt{EC},0}, \mathcal{C}^2_{\mathtt{EC},0}\} \\
\mathcal{L}^0_{\mathtt{EC},0} &= \{\mathcal{C}^0_{\mathtt{EN},0}, \mathcal{C}^1_{\mathtt{EN},0}\}; \mathcal{L}^1_{\mathtt{EC},0} = \{\mathcal{C}^1_{\mathtt{EN},0}, \mathcal{C}^2_{\mathtt{EN},0}\}; \mathcal{L}^2_{\mathtt{EC},0} = \{\mathcal{C}^0_{\mathtt{EN},0}, \mathcal{C}^2_{\mathtt{EN},0}\} \\
\mathcal{R}^0_{\mathtt{EC},0} &= \mathcal{R}^1_{\mathtt{EC},0} = \mathcal{R}^2_{\mathtt{EC},0} \\
\mathcal{R}^0_{\mathtt{EC},0}(r) &= \begin{cases} 0.5\text{Gbit/s} & \text{if } r = \text{bandwidth} \\ \perp & \text{otherwise} \end{cases}
\end{aligned}
$$

Here, the feasibility evaluation method f_0 is specified analog to the example f_j presented in Section 11.2 and only extended by concrete definitions of the functions for determining the feasibility of node resources fnr_0 and feasibility on interconnect fi_0. For these additional definitions, the function for determining the integrated FS nodes of an IE node $integ\_fn(\mathcal{C}_{\mathtt{IN},k}, \mathcal{C}^s_{\mathtt{EN},j})$ is reused. Checking resource feasibility on the node level is performed for each type of resource in a separate function:

$$
\begin{aligned}
fnr_0(\mathcal{C}_{\texttt{IN},k}) &= \bigwedge_{\mathcal{C}^s_{\texttt{EN},0} \in \mathcal{C}_{\texttt{EN},0}} fnrn_0(\mathcal{C}_{\texttt{IN},k}, \mathcal{C}^s_{\texttt{EN},0}) \\
fnrn(\mathcal{C}_{\texttt{IN},k}, \mathcal{C}^s_{\texttt{EN},0}) &= \bigwedge_{r \in \mathbf{R}_{\texttt{EN}}} fnrn^r(\mathcal{C}_{\texttt{IN},k}, \mathcal{C}^s_{\texttt{EN},0}) \\
fnrn^{\text{disk}}(\mathcal{C}_{\texttt{IN},k}, \mathcal{C}^s_{\texttt{EN},0}) &= \mathcal{R}^s_{\texttt{EN},0}(\text{disk}) \geq 100\text{MiB} + \\
&\qquad \sum_{\mathcal{C}^r_{\texttt{FN},i} \in integ\_fn(\mathcal{C}_{\texttt{IN},k}, \mathcal{C}^s_{\texttt{EN},0})} \mathcal{R}^r_{\texttt{FN},i}(\text{disk}) + 0.5\text{MiB} \\
fnrn^{\text{memory}}(\mathcal{C}_{\texttt{IN},k}, \mathcal{C}^s_{\texttt{EN},0}) &= \mathcal{R}^s_{\texttt{EN},0}(\text{memory}) \geq 200\text{MiB} + \\
&\qquad \sum_{\mathcal{C}^r_{\texttt{FN},i} \in integ\_fn(\mathcal{C}_{\texttt{IN},k}, \mathcal{C}^s_{\texttt{EN},0})} \mathcal{R}^r_{\texttt{FN},i}(\text{memory}) + 1\text{MiB} \\
fnrn^{\text{cpu}}(\mathcal{C}_{\texttt{IN},k}, \mathcal{C}^s_{\texttt{EN},0}) &= 8 \geq 1 + \sum_{\mathcal{C}^r_{\texttt{FN},i} \in integ\_fn(\mathcal{C}_{\texttt{IN},k}, \mathcal{C}^s_{\texttt{EN},0})} 1 \\
fnrn^{\text{sched}}(\mathcal{C}_{\texttt{IN},k}, \mathcal{C}^s_{\texttt{EN},0}) &= \forall \mathcal{C}^r_{\texttt{FN},i} \Big(\mathcal{C}^r_{\texttt{FN},i} \in integ\_fn(\mathcal{C}_{\texttt{IN},k}, \mathcal{C}^s_{\texttt{EN},0}) \implies \\
&\qquad \mathcal{R}^{i,r}_{IC,k}(\text{sched\_prio}) \neq \bot \wedge 75 < \mathcal{R}^{i,r}_{IC,k}(\text{sched\_prio}) < 90 \wedge \\
&\qquad \forall \mathcal{C}^b_{\texttt{FN},a} \big(\mathcal{C}^b_{\texttt{FN},a} \in integ\_fn(\mathcal{C}_{\texttt{IN},k}, \mathcal{C}^s_{\texttt{EN},0}) \wedge \\
&\qquad \mathcal{R}^{i,r}_{IC,k}(\text{sched\_prio}) = \mathcal{R}^{a,b}_{IC,k}(\text{sched\_prio}) \implies a = i \wedge r = b \big) \Big)
\end{aligned}
$$

This definition also ensures the availability of the additional resources, which the IE needs for managing the FS execution. The function $fnrn^{\text{sched}}$ tests whether the FS nodes' priorities are assigned in the correct scheduling band and that the FS nodes integrated on the same IE node have different priorities, as we required in Section 9.2.

Let $integ\_fc(\mathcal{C}_{\texttt{IC},k}, \mathcal{C}^q_{\texttt{EC},j}) = \{e | e \mapsto (\mathcal{C}^q_{\texttt{EC},j}, b) \in \mathcal{C}_{\texttt{IC},k}\}$ be the set of FS interconnections integrated on the IE interconnection $\mathcal{C}^q_{\texttt{EC},j}$. The feasibility evaluation method for the interconnect (fi) checks the availability of the bandwidth (fir) and communication partners (fil), as well as whether the FSs' restrictions on sharing physical links are satisfied by the integration contract (fiv):

$$
\begin{aligned}
fi_0(\mathcal{C}_{\texttt{IC},k}) &= \bigwedge_{\mathcal{C}^q_{\texttt{EC},0} \in \mathcal{C}_{\texttt{EC},0}} fir(\mathcal{C}_{\texttt{IC},k}, \mathcal{C}^q_{\texttt{EC},0}) \wedge fil(\mathcal{C}_{\texttt{IC},k}, \mathcal{C}^q_{\texttt{EC},0}) \wedge fiv(\mathcal{C}_{\texttt{IC},k}, \mathcal{C}^q_{\texttt{EC},0}) \\
fir(\mathcal{C}_{\texttt{IC},k}, \mathcal{C}^q_{\texttt{EC},0}) &= \mathcal{R}^q_{\texttt{EC},0}(\text{bandwidth}) \geq \sum_{\mathcal{C}^p_{\texttt{FC},i} \in integ\_fc(\mathcal{C}_{\texttt{IC},k}, \mathcal{C}^q_{\texttt{EC},0})} \mathcal{R}^p_{\texttt{FC},0}(\text{bandwidth})
\end{aligned}
$$

$$fil(\mathcal{C}_{\mathtt{IC},k}, \mathcal{C}^q_{\mathtt{EC},0}) = \bigwedge_{\mathcal{C}^p_{\mathtt{FC},i} \in integ\_fc(\mathcal{C}_{\mathtt{IC},k}, \mathcal{C}^q_{\mathtt{EC},0})} \mathcal{L}^p_{\mathtt{FC},i} \subseteq \mathcal{L}^q_{\mathtt{EC},0}$$

$$fiv(\mathcal{C}_{\mathtt{IC},k}, \mathcal{C}^q_{\mathtt{EC},0}) = \bigwedge_{\mathcal{C}^p_{\mathtt{FC},i} \in integ\_fc(\mathcal{C}_{\mathtt{IC},k}, \mathcal{C}^q_{\mathtt{EC},0})} \mathcal{V}^p_{\mathtt{FC},i} \cap \mathcal{L}^q_{\mathtt{EC},0} = \emptyset$$

In this case, the maximum bandwidth with which the IE can guarantee fast message transmission times has already been defined in the IE interconnection contracts. Consequently, the feasibility evaluation method does not need to enforce an even lower limit.

Finally, the integrated system $\mathcal{I}_2$ with two FSs, $\mathcal{F}_{10}$ and $\mathcal{F}_{20}$, has the example integration contract $\mathcal{C}_{\mathtt{I},2}$:

$$\mathcal{C}_{\mathtt{IN},2} = \{\mathcal{C}^0_{\mathtt{FN},0} \mapsto (\mathcal{C}^0_{\mathtt{EN},0}, \mathcal{R}^{0,0}_{\mathtt{IN},2}), \mathcal{C}^1_{\mathtt{FN},0} \mapsto (\mathcal{C}^1_{\mathtt{EN},0}, \mathcal{R}^{0,1}_{\mathtt{IN},2}), \mathcal{C}^2_{\mathtt{FN},0} \mapsto (\mathcal{C}^2_{\mathtt{EN},0}, \mathcal{R}^{0,2}_{\mathtt{IN},2})\}$$

$$\mathcal{R}^{10,0}_{\mathtt{IC},2} = \mathcal{R}^{10,1}_{\mathtt{IC},2} = \mathcal{R}^{10,2}_{\mathtt{IC},2} = \begin{cases} 79 & \text{if } r = \text{sched\_prio} \\ \bot & \text{otherwise} \end{cases}$$

$$\mathcal{R}^{20,0}_{\mathtt{IC},2} = \mathcal{R}^{20,1}_{\mathtt{IC},2} = \mathcal{R}^{20,2}_{\mathtt{IC},2} = \begin{cases} 80 & \text{if } r = \text{sched\_prio} \\ \bot & \text{otherwise} \end{cases}$$

$$\mathcal{C}_{\mathtt{IC},2} = \{\mathcal{C}^0_{\mathtt{FC},0} \mapsto (\mathcal{C}^0_{\mathtt{EC},0}, \mathcal{R}^{0,0}_{\mathtt{IC},2}), \mathcal{C}^1_{\mathtt{FC},0} \mapsto (\mathcal{C}^1_{\mathtt{EC},0}, \mathcal{R}^{0,1}_{\mathtt{IC},2}), \mathcal{C}^2_{\mathtt{FC},0} \mapsto (\mathcal{C}^2_{\mathtt{EC},0}, \mathcal{R}^{0,2}_{\mathtt{IC},2})\}$$

$$\mathcal{R}^{10,0}_{\mathtt{IC},2} = \mathcal{R}^{10,1}_{\mathtt{IC},2} = \mathcal{R}^{10,2}_{\mathtt{IC},2} = \mathcal{R}^{20,0}_{\mathtt{IC},2} = \mathcal{R}^{20,1}_{\mathtt{IC},2} = \mathcal{R}^{20,2}_{\mathtt{IC},2} = \begin{cases} 3 \cdot 10^4 & \text{if } r = \text{bandwidth} \\ \bot & \text{otherwise} \end{cases}$$

Under the assumption that the definition of the FS contract $\mathcal{C}_{\mathtt{F},10}$ for the FS $\mathcal{F}_{10}$ has similar requirements as $\mathcal{F}_{20}$, the feasibility evaluation method f_0 maps this integration contract to $\top$.

Synchronization as IE service

In the prototype, the periodic synchronization as IE service is implemented with a synchronization server. This server receives the FS nodes' results as UDP messages and executes the synchronization mechanism with the smallest synchronization period required by the integrated FS nodes. This is 50ms according to our example FSs types $\mathcal{F}_{20}$ and $\mathcal{F}_{10}$.

Figure 12.9 shows a scheduling example of the FS nodes and processes on IE node 0, when executing periodic synchronization as an IE service. First, the synchronization server starts the synchronization process. As soon as it has finished, it sends the first synchronized message to $\mathcal{F}^0_0$ executing a $\mathcal{F}_{20}$ type of workload. After receiving the FS node's result, it proceeds by sending the second synchronized message to $\mathcal{F}^0_1$, which executes a workload of type $\mathcal{F}_{10}$, as does $\mathcal{F}^0_2$. As soon as the synchronization server has received the last synchronization message, the CPU is occupied with executing the FS nodes and the synchronization server without any break until the workload is completed. This closes the semantic gap between the virtual machines of the same IE node.

Figure 12.10 shows the maximum application and synchronization jitter in correspondence to the number of integrated FSs. Here, the first FS executed a $\mathcal{F}_{20}$ type of workload, while each

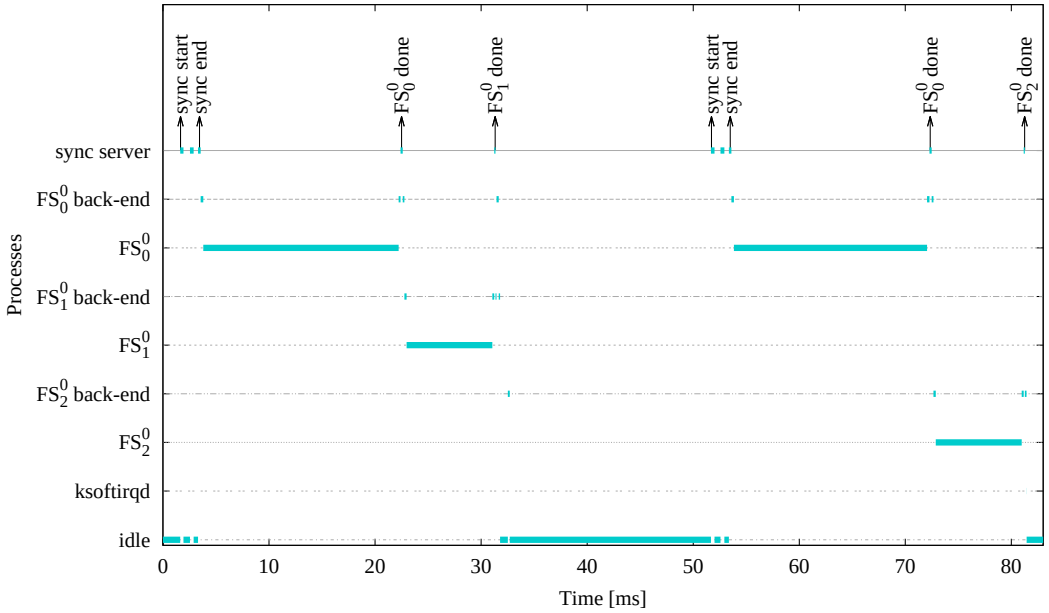


Figure 12.9: FS and process scheduling example on IE node 0 for periodic synchronization as IE service with one $\mathcal{F}_{20}$ and two $\mathcal{F}_{10}$ workloads.

additional one performed a $\mathcal{F}_{10}$ type of workload. The synchronization jitter remains stable, while the application jitter increases with the number of integrated FSs. This is caused by the sequential workload execution and sending of UDP messages in-between. The fact that the application jitter does not increase in-between the scenarios from two to three and four to five FSs is a result of the alternating allocation of the additionally integrated FSs in the synchronization period of the synchronization server. FS nodes of FS 1 and FS 3 are scheduled every odd synchronization round, while the FS node of FS 2 and FS 4 compute in even rounds. The FS nodes of FS 0 operate on 50ms and must be served each round.

12.5 Preemptive EDF Scheduling

Seeing that the properties of preemptive EDF and according schedule ability tests have already been extensively studied, this section concentrates on the evaluation of the practical implementation with virtualization.

The composability strategy with EDF-based scheduling has strict constraints regarding the wakeup of FS nodes, i.e. only once per synchronization period. Regarding our prototype, an EDF scheduler is available since version 3.14 of the Linux kernel mainline. The first version of this EDF scheduler has been presented in by Faggioli et al. in 2009 [42].

Figure 12.11 shows the wakeup delays within a Linux-based virtual machine on top of our prototype IE when executing a high priority process. As soon as this process is scheduled, it goes back to sleep again. It does so with a period of 10ms. The ftrace interface was used to record the process switches and a sampling time of 16.4s fit in the trace buffer. As can be seen, the virtual

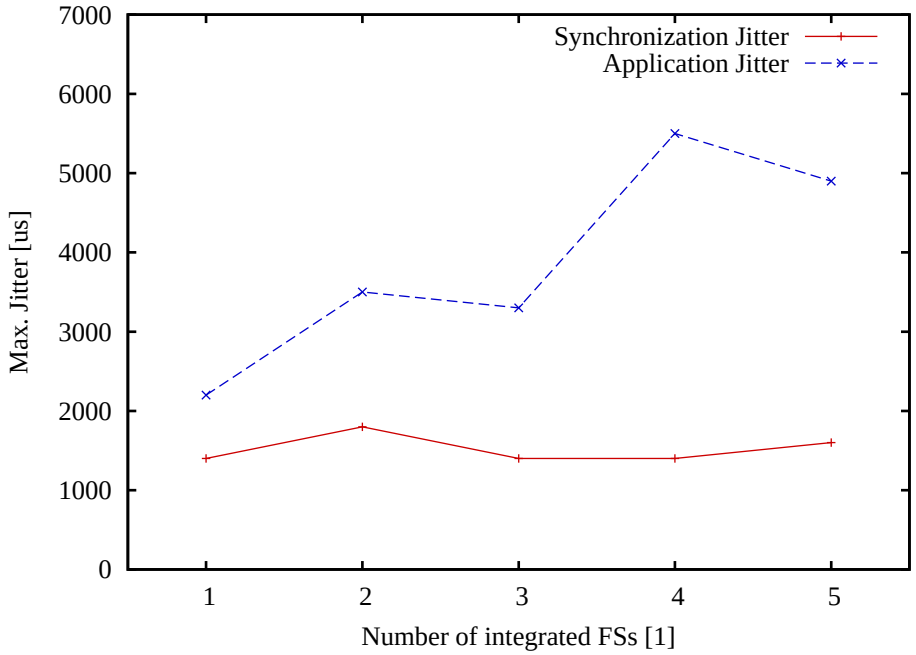


Figure 12.10: Maximum application and synchronization jitter in dependency of the number of integrated FSs with synchronization as an IE service and only one CPU core enabled. Lines drawn to show the trend only.

machine woke up most of the time due to the high priority process. There were only a few other wake-ups within the recorded time frame. Most of these were caused by receiving network packets, but not exclusively, e.g. a timeout expired within the init process. From this we can draw two consequences: first, Linux is not suitable for constructing a composable system with EDF-based scheduling. Second, other means than network interfaces must be used to exchange the data between the IE node and FS nodes in such an EDF-based system, since otherwise any network packet can break the schedule. In the case of an embedded hypervisor, a possible solution would be to use a message-passing interface. Consequently, virtualization is not a good foundation for EDF-based composability, since in such a scenario the reuse of legacy applications and available services within the virtual machine are very limited.

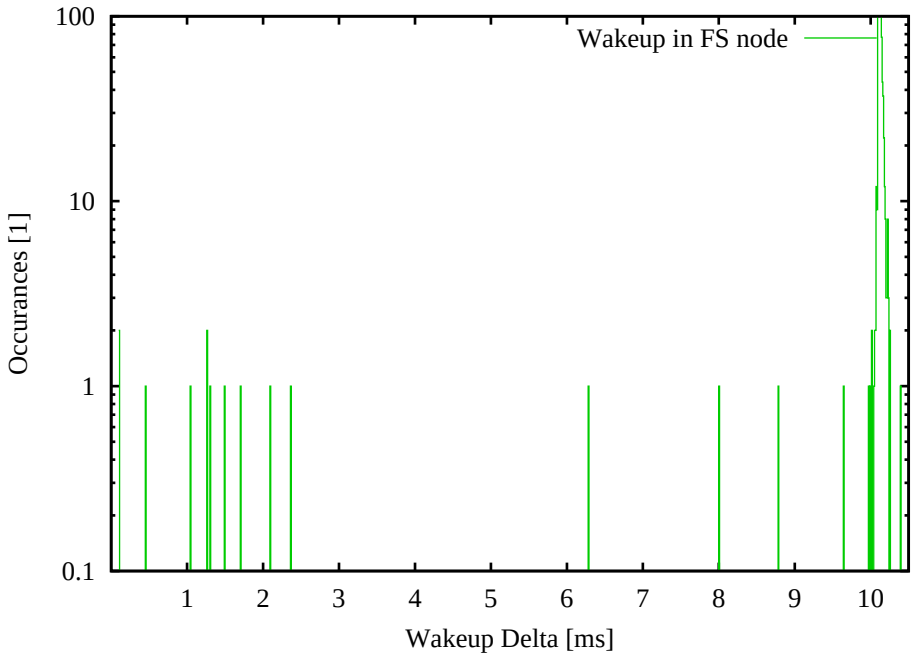


Figure 12.11: Wakeup delays in-between processes scheduling operations on a Linux virtual machine, when a high priority process tries to get the CPU every 10ms.

CHAPTER 13

Conclusion

The aim of this thesis was the integration of existing fail-safe TMR applications using the methods of composability and mixed-criticality. These applications are subject to certification. The benefits of this approach are the independent certification of the applications, software reuse and a better utilization of hardware resources. These goals contradict the strong requirements of periodically synchronized applications concerning the properties of the communication links between individual application instances, their reactivity and the existing strategy for certification. Consequently, a suitable composable solution must provide a balance between the objectives of independent certification (predictability and separation), the application reaction time requirements (performance) and the hardware utilization. The technical and conceptual foundations were established in this work.

Two additional constraints were identified after studying the practical aspects of the targeted TMR applications. The first of these is to keep the applications unchanged, which is in line with the goal for software reuse. The second constraint is to provide the synchronization mechanism within the application's fault containment regions. This ensures strict fault containment and enables the reuse of the existing applications' safety strategy.

The initial analysis has shown that the concepts of TMR and composability are orthogonal in that composability requires failure confinement regions, while TMR is based on fault containment regions. There are multiple ways of combining these two types of regions. The existing TMR architectures were aligned within this solution space and their benefits and needs were systematically identified when augmented with a composability layer. Additionally, software-based TMR schemes that specifically leverage the existence of composability were proposed, which use the available hardware resources more efficiently.

The scheduling of the CPU has been identified as the most critical resource regarding predictability and utilization for the targeted software-based TMR applications. The periodically synchronized applications are strongly affected by the scheduling strategy of the integration environment, especially if the schedulers between the nodes are not aligned. There are two main reasons for this. The first is that the integration environment cannot determine the state of the encapsulated application instances. The second reason is that scheduling of such application

instances is costly in terms of CPU time.

It was shown that static-cyclic and preemptive EDF scheduling without architectural and application-specific measures are suitable for applications with relaxed timing requirements, since this scheduling strategy causes very long execution times of the TMR synchronization mechanism. Changing the interconnection between the nodes of the integration environment and adapting the synchronization mechanism to require fewer scheduling operations reduces the execution time of the TMR synchronization mechanism. The applications' reaction times are shortened as well, but still remain substantial. Additionally, a token-ring-based synchronization mechanism has been proposed, which in this case, is also suitable for applications with long reaction times. A simulation environment was set up providing static-cyclic scheduling, normal distributed message transmission times and zero time computation. It was used to validate the results regarding behaviour of the synchronization mechanisms under static-cyclic scheduling.

Of all analyzed scheduling strategies only preemptive fixed-priority scheduling provides an out-of-the-box solution. This solution requires that the number of integrated safety-critical application instances are at most the available number of CPU cores per node. One possible solution to overcome this limitation requires a small adaptation of the synchronization mechanism. Specifically, to provide the synchronization mechanism as a service of the integration environment, which avoids aligning the schedulers of the nodes themselves. However, it weakens the separation of application instances and they must operate based on the same synchronization period. This method also demands that the integration environment have a static assignment of application instances to nodes and a limited number of these nodes. Measurements with a Linux and KVM-based prototype confirmed these results.

A different mechanism was suggested for preemptive EDF scheduling, where small buffers with a timeout determine the release time of application instances. This enables the use of EDF scheduling tests for integration of applications, while allowing fast reaction times and keeping the synchronization independent. Here, the adaptations of the synchronization mechanism and of the applications themselves are mandatory. The prototype was used to determine whether Linux-based virtual machines are suitable for EDF-based scheduling, which they are not since one cannot restrict when they wake up.

Based on the architectures of the solutions and the analyses for determining the application reaction times, a (formal) contract concept for independent certification has been proposed. The applications are independently certified together with a contract stating their requirements. Also, the composable environment is certified with its own contract, which defines its provided resources. This integration environment contract must additionally specify a feasibility evaluation method for checking whether specific configurations of applications can be integrated without violating their requirements. For an integrated system, an integration contract must then specify the set of integrated applications and the integration environment. This integrated system is then certified using only the provided contracts, by checking the feasibility of the integration contract with the corresponding feasibility evaluation method.

Ultimately, the profiles of the integrated applications and the overall targets for integration determine which of the presented solutions for composability and mixed-criticality fits best.

CHAPTER 14

Future Work

The solutions and analyses in this thesis were focused on TMR applications which only need to exchange small messages. Extending this approach to data-intensive applications must also incorporate the effects of the network infrastructure. This would have two major benefits. First, it would enable the integration of such applications which must (timely) exchange a large amount of data. Second, it would also cover the impact of the accumulated network load in very large integration environments.

An extension regarding the preemptive EDF-based solution would be to incorporate the EDF-based mixed-criticality scheduling scheme. This strategy guarantees the availability of the requested computation time to the application in correlation with its criticality. Applications with a higher criticality have a higher level of assurance to receive their CPU share. This would allow for a tighter integration of applications and subsequently, an even better utilization of the CPU.

As presented, a possible solution to overcome some performance limitations of static-cyclic scheduling is to align the periods of the schedulers. Here, an in-depth analysis regarding the effects of such an alignment can provide a basis for identifying possible new failure modes in otherwise loosely coupled application instances.

The ongoing research regarding the interference between applications on multi-core CPUs promises a profound understanding of such effects. This can be beneficial for the solution based on preemptive fixed-priority scheduling with one CPU core per application instance. It could provide a solid basis for calculating tight bounds for the synchronization mechanisms' execution times.

Bibliography

[1] T. Abdelzaher, A Shaikh, F. Jahanian, and Kang Shin. RTCAST: lightweight multicast for real-time process groups. In *Real-Time Technology and Applications Symposium, 1996. Proceedings., 1996 IEEE*, pages 250–259, Jun 1996.

[2] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 4–13, Dec 1998.

[3] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 2–13. ACM, 2006.

[4] B. Akesson, A. Molnos, A. Hansson, J.A. Angelo, and K. Goossens. Composability and predictability for independent application development, verification, and execution. In *Multiprocessor System-on-Chip*, pages 25–56. Springer New York, 2011.

[5] M. Aldea, G. Bernat, I. Broster, A. Burns, R. Dobrin, J. M. Drake, Gerhard Fohler, P. Gai, M. Gonzalez Harbour, G. Guidi, J.J. Gutierrez, T. Lennvall, G. Lipari, J.M. Martinez, J.L. Medina, J.C. Palencia, and M. Trimarchi. Fsf: A real-time scheduling architecture framework. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, San Jose, California, USA, April 2006.

[6] J. Alves-Foss, P.W. Oman, C. Taylor, and W.S. Harrison. The MILS architecture for high-assurance embedded systems. *International Journal of Embedded Systems*, 2(3):239–247, 2006.

[7] J. Alves-Foss, C. Taylor, and P. Oman. A multi-layered approach to security in high assurance systems. In *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, pages 10–pp. IEEE, 2004.

[8] Yair Amir, Louise E Moser, Peter M Melliar-Smith, Deborah A Agarwal, and Paul Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems (TOCS)*, 13(4):311–342, 1995.

[9] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, pages 193–202, Dec 2001.

[10] ARINC Inc. ARINC Specification 653: Part 3, Avionics Application Software Standard Interface, Conformity Test Specification. 2006.

[11] ARINC Inc. ARINC Specification 653: Part 2, Avionics Application Software Standard Interface, Extended Services. 2008.

[12] ARINC Inc. ARINC Specification 653: Part 1, Avionics Application Software Standard Interface, Required Services. 3, 2010.

[13] AUTOSAR. Automotive Open Systems Architecture. `http://www.autosar.org/`.

[14] Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. *Volume II: Technical concepts of component-based software engineering*. Carnegie Mellon University, Software Engineering Institute, 2000.

[15] Andrea Baldovin, Enrico Mezzetti, and Tullio Vardanega. A Time-composable Operating System. In Tullio Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 69–80, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[16] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[17] Günther Bauer and Hermann Kopetz. Transparent redundancy in the time-triggered architecture. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 5–13. IEEE, 2000.

[18] Albert Benveniste, Benoit Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas Henzinger, and Kim G. Larsen. Contracts for System Design. Research report RR-8147, INRIA, November 2012.

[19] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple viewpoint contract-based specification and design. In FrankS. de Boer, MarcelloM. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 5382 of *Lecture Notes in Computer Science*, pages 200–225. Springer Berlin Heidelberg, 2008.

[20] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. Nonstop® advanced architecture. In *International Conference on Dependable Systems and Networks, 2005. DSN 2005. Proceedings.*, pages 12–21. IEEE, 2005.

[21] Marko Bertogna and Sanjoy Baruah. Tests for global EDF schedulability analysis. *Journal of Systems Architecture*, 57(5):487 – 497, 2011. Special Issue on Multiprocessor Real-time Scheduling.

[22] Enrico Bini, Giorgio Buttazzo, Johan Eker, Stefan Schorr, Raphael Guerra, Gerhard Fohler, Karl-Erik Årzén, Vanessa Romero, and Claudio Scordino. Resource management on multicore systems: The actors approach. *IEEE Micro*, 31(3):72–81, 2011.

[23] P. Binns. A robust high-performance time partitioning algorithm: the digital engine operating system (deos) approach. In *Digital Avionics Systems, 2001. DASC. 20th Conference*, volume 1, pages 1B6/1–1B6/12 vol.1, Oct 2001.

[24] Kenneth P. Birman. Isis: A system for fault-tolerant distributed computing. Technical report, Ithaca, NY, USA, 1986.

[25] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, January 1987.

[26] T.C. Bressoud and F.B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996.

[27] Stefan Bunzel. AUTOSAR – the Standardized Software Architecture. *Informatik-Spektrum*, 34(1):79–83, 2011.

[28] CENELEC, E.N. 50126-Railway Applications: The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS). *European Committee for Electrotechnical Standardization*, 1999.

[29] CENELEC, E.N. 50129-Railway Applications: Communication, signalling and processing systems - Safety related electronic systems for signalling. *European Committee for Electrotechnical Standardization*, 2003.

[30] CENELEC, E.N. 50128-Railway Applications: Software for Railway Control and Protection Systems. *European Committee for Electrotechnical Standardization*, 2011.

[31] Liming Chen and Algirdas 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)*, pages 3–9, 1978.

[32] P.M. Chen and B.D. Noble. When virtual is better than real [operating system relocation to virtual machines]. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 133–138, May 2001.

[33] International Electrotechnical Commission. IEC 61508-1: Functional safety of electrical/electronic/programmable electronic safety-related systems part 1: General requirements. 2010.

[34] International Electrotechnical Commission. IEC 61508-2: Functional safety of electrical/electronic/programmable electronic safety-related systems part 2: Requirements for electrical / electronic / programmable electronic safety-related systems. 2010.

[35] International Electrotechnical Commission. IEC 61508-3: Functional safety of electrical/electronic/programmable electronic safety-related systems part 3: Software requirements. 2010.

[36] Philippa Conmy, Mark Nicholson, and John McDermid. Safety assurance contracts for integrated modular avionics. In *Proceedings of the 8th Australian workshop on Safety critical systems and software-Volume 33*, pages 69–78. Australian Computer Society, Inc., 2003.

[37] D. Czajkowski and M. McCartha. Ultra low-power space computer leveraging embedded SEU mitigation. In *Proc. IEEE Aerospace Conf*, volume 5, pages 2315–2328, 2003.

[38] Steven C Dake, Christine Caulfield, and Andrew Beekhof. The corosync cluster engine. In *Linux Symposium*, volume 85, pages 61–68, July 2008.

[39] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

[40] K. Elphinstone. Future directions in the evolution of the l4 microkernel. In *NICTA Formal Methods Program Workshop on Operating Systems Verification*, page 19, 2004.

[41] H. Espinoza, A. Ruiz, M. Sabetzadeh, P. Panaroni, et al. Challenges for an open and evolutionary approach to safety assurance and certification of safety-critical systems. In *Software Certification (WoSoCER), 2011 First International Workshop on*, pages 1–6. IEEE, 2011.

[42] Dario Faggioli, Fabio Checconi, Michael Trimarchi, and Claudio Scordino. An EDF scheduling class for the Linux kernel. In *11th Real-Time Linux Workshop*, September 2009.

[43] Jane Fenn, Richard Hawkins, Phil Williams, and Tim Kelly. Safety case composition using contracts - refinements based on feedback from an industrial case study. In Felix Redmill and Tom Anderson, editors, *The Safety of Systems*, pages 133–146. Springer London, 2007.

[44] Mikel Fernández, Roberto Gioiosa, Eduardo Quiñones, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. Assessing the suitability of the ngmp multi-core processor in the space domain. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 175–184, New York, NY, USA, 2012. ACM.

[45] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange. Autosar–a worldwide standard is on the road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, 2009.

[46] A. Gerstinger. *Diversity for complex safety-critical computer systems*. PhD thesis, Vienna University of Technology, 2008.

[47] A. Gerstinger. Runtime diversity against quasirandom faults. In *Systems, 2009. ICONS '09. Fourth International Conference on*, pages 145–148, March.

[48] Andreas Gerstinger, Heinz Kantz, and Christoph Scherrer. TAS control platform: A platform for safety-critical railway applications. *ERCIM News*, 2008(75), 2008.

[49] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning: theory & practice*. Elsevier, 2004.

[50] T.M. Ghazalie and T.P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9(1):31–67, 1995.

[51] Men Mikro Elektronik GmbH. D602 - Triple-Redundant 6U CompactPCI© PowerPC© SBC. `https://www.men.de/products/pdfdatasheet,products,02D602-DS,1.html`, March 2011.

[52] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Eli: Bare-metal performance for i/o virtualization. *SIGPLAN Not.*, 47(4):411–422, March 2012.

[53] Stefan Groesbrink, Simon Oberthür, and Daniel Baldin. Architecture for adaptive resource assignment to virtualized mixed-criticality real-time systems. *SIGBED Rev.*, 10(1):18–23, February 2013.

[54] Goal Structuring Notation Working Group. GSN Community Standard Version 1. *Origin Consulting (York) Limited*, November 2011.

[55] Z. Gu and Q. Zhao. A state-of-the-art survey on real-time issues in embedded systems virtualization. *Journal of SW Engineering and Applications*, 5(4):277–290, 2012.

[56] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in xen. In Maarten van Steen and Michi Henning, editors, *Middleware 2006*, volume 4290 of *Lecture Notes in Computer Science*, pages 342–362. Springer Berlin Heidelberg, 2006.

[57] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. 1994.

[58] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are virtual machine monitors microkernels done right. In *10th HotOS*, pages 1–6, 2005.

[59] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):2:1–2:24, January 2009.

[60] G. Heiser and B. Leslie. The okl4 microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, pages 19–24. ACM, 2010.

[61] G. Heiser, V. Uhlig, and J. LeVasseur. Are virtual-machine monitors microkernels done right? *ACM SIGOPS Operating Systems Review*, 40(1):95–99, 2006.

[62] T.A Henzinger and S. Matic. An interface algebra for real-time components. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 253–266, April 2006.

[63] R. Hillman, O. Swift, P. Layton, M. Conrad, C. Thibodeau, and F. Irom. Space processor radiation mitigation and validation techniques for an 1,800 mips processor board. In *Radiation and Its Effects on Components and Systems, 2003. RADECS 2003. Proceedings of the 7th European Conference on*, pages 347–352, Sept 2003.

[64] IBM Corporation. Best practices for KVM. White Paper `http://pic.dhe.ibm.com/infocenter/lnxinfo/v3r0m0/topic/liaat/liaatbestpractices_pdf.pdf`, 2012.

[65] Joachim Janle and Günter Poppe. Mehrabschnitts-Achszähler Az LM - die neue Achszähler-Generation. *Signal+Draht*, (11), 2000.

[66] X. Jean, D. Faura, M. Gatti, L. Pautet, and T. Robert. Ensuring robust partitioning in multicore platforms for ima systems. In *Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st*, pages 7A4–1–7A4–9, Oct 2012.

[67] Lei Ju, S. Chakraborty, and A. Roychoudhury. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1–6, April 2007.

[68] R. Kaiser and S. Wagner. Evolution of the PikeOS microkernel. In *First International Workshop on Microkernels for Embedded Systems*, page 50, 2007.

[69] H. Kantz and C. Koza. The ELEKTRA railway signalling system: field experience with an actively replicated system with diversity. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 453–458, June 1995.

[70] T.P. Kelly. *Arguing Safety - A systematic approach to managing safety cases*. PhD thesis, University of York, 1999.

[71] Rob Kitchin and Martin Dodge. *Code/space: Software and everyday life*. MIT Press, 2011.

[72] N König and H Kantz. TAS control platform: A vital computer platform for railway applications. *Revista de telecomunicaciones de Alcatel*, (2):272–276, 2004.

[73] H. Kopetz. TTA supported service availability. In Miroslaw Malek, Edgar Nett, and Neeraj Suri, editors, *Service Availability*, volume 3694 of *Lecture Notes in Computer Science*, pages 1–14. Springer Berlin Heidelberg, 2005.

[74] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, Jan 2003.

[75] H. Kopetz, C. El Salloum, B. Huber, R. Obermaisser, and C. Paukovits. Composability in the time-triggered system-on-chip architecture. In *SOC Conference, 2008 IEEE International*, pages 87–90. IEEE, 2008.

[76] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *Computers, IEEE Transactions on*, C-36(8):933–940, Aug 1987.

[77] Ondrej Kotaba, Jan Nowotsch, Michael Paulitsch, Stefan M Petters, and Henrik Theiling. Multicore in real-time systems–temporal isolation challenges due to shared resources. In *Proc. of Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems (at DATE Conf.)*, 2013.

[78] Adam Lackorzyński, Alexander Warg, Marcus Völp, and Hermann Härtig. Flattening hierarchical scheduling. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 93–102, New York, NY, USA, 2012. ACM.

[79] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *J. ACM*, 32(1):52–78, January 1985.

[80] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[81] N.G. Leveson. Safety as a system property. *Communications of the ACM*, 38(11):146–, 1995.

[82] Giuseppe Lipari and Sanjoy Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)*, RTAS '01, pages 26–, Washington, DC, USA, 2001. IEEE Computer Society.

[83] Giuseppe Lipari and Gerhard Fohler. A framework for composing real-time schedulers. In *Proceedings of the International Workshop on Test and Analysis of Component Based Systems*, Warsaw, Poland, April 2003.

[84] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, January 1973.

[85] C.Douglass Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–53, 1992.

[86] M. Masmano, I. Ripoll, A. Crespo, and J.J. Metge. Xtratum: A hypervisor for safety critical embedded systems. In *Proceedings of the 11th Real-Time Linux Workshop. Dresden. Germany*, 2009.

[87] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[88] S.P. Miller, D.D. Cofer, Lui Sha, J. Meseguer, and A. Al-Nayeem. Implementing logical synchrony in integrated modular avionics. In *Digital Avionics Systems Conference, 2009. DASC '09. IEEE/AIAA 28th*, pages 1.A.3–1–1.A.3–12, Oct 2009.

[89] A. Molnos, A. Milutinovic, D. She, and K. Goossens. Composable processor virtualization for embedded systems. In *Proceedings of the Workshop on Computer Architecture and Operating System Co-Design (CAOS)*. Springer, 2010.

[90] Network Time Foundation. NTP: The Network Time Protocol. `http://www.ntp.org/`.

[91] M. Neukirchner, S. Stein, H. Schrom, and R. Ernst. A software update service with self-protection capabilities. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 903–908, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[92] M. Neukirchner, S. Stein, H. Schrom, J. Schlatow, and R. Ernst. Contract-based dynamic task management for mixed-criticality systems. In *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*, pages 18–27, June 2011.

[93] J. Nowotsch and M. Paulitsch. Leveraging Multi-core Computing Architectures in Avionics. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 132–143, May 2012.

[94] Brendan Anthony O'Connell. Achieving fault tolerance via robust partitioning and n-modular redundancy. Master's thesis, Massachusetts Institute of Technology, 2007.

[95] Jon Perez, David Gonzalez, Salvador Trujillo, Ton Trapman, and Jose Miguel Garate. A safety concept for a wind power mixed-criticality embedded system based on multicore partitioning. In *Proc. WMC, RTSS*, pages 25–30, 2013.

[96] Ronald Perez, Leendert van Doorn, and Reiner Sailer. Virtualization and hardware-based security. *IEEE Security & Privacy*, 6(5):24–31, 2008.

[97] S. Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6(3):289–316, 1994.

[98] David Powell, Jean Arlat, Ljerka Beus-Dukic, Andrea Bondavalli, Paolo Coppola, Alessandro Fantechi, Eric Jenn, Christophe Rabéjac, and Andy Wellings. GUARDS: A generic upgradable architecture for real-time dependable systems. *Parallel and Distributed Systems, IEEE Transactions on*, 10(6):580–599, 1999.

[99] P.J. Prisaznuk. ARINC 653 role in integrated modular avionics (IMA). In *IEEE/AIAA 27th Digital Avionics Systems Conference (DASC 2008).*, pages 1–E. IEEE, 2008.

[100] Himanshu Raj and Karsten Schwan. High performance and scalable i/o virtualization via self-virtualized devices. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, HPDC '07, pages 179–188, New York, NY, USA, 2007. ACM.

[101] B Rajagopalan and Philip K McKinley. A token-based protocol for reliable, ordered multicast communication. In *Reliable Distributed Systems, 1989., Proceedings of the Eighth Symposium on*, pages 84–93. IEEE, 1989.

[102] Ragunathan Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: a resource-centric approach to real-time and multimedia systems. volume 3310, pages 150–164, 1997.

[103] Stefan Resch, Andreas Steininger, and Christoph Scherrer. Software Composability and Mixed Criticality for Triple Modular Redundant Architectures. In *Proceedings of the 2013 SASSUR Workshop*, pages 91–102, 2013. talk: SASSUR Workshop 2013, Toulouse; 2013-09-24.

[104] S. Rostedt. Finding origins of latencies using ftrace. In *Proceedings of the Eleventh Real-Time Linux Workshop*, pages 117–130. OSADL - Real-Time Linux Foundation Working Group, 2009.

[105] RTCA. DO-297, Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations. Technical report, Radio Technical Commission for Aeronautics, Inc. (RTCA), Washington, DC, 2005.

[106] J.M. Rushby. Design and verification of secure systems. In *ACM SIGOPS Operating Systems Review*, volume 15, pages 12–21. ACM, 1981.

[107] John Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, DTIC Document, 2000.

[108] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, August 1983.

[109] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[110] IEC Standard. IEC 60812: Analysis techniques for system reliability - Procedure for failure mode and effects analysis (FMEA). *Bureau Central de la Commission Electrotechnique Internationale, Genève, Suisse*, 2.0, 2006.

[111] H. Theiling. PikeOS and Time-Triggering. White Paper `http://www.sysgo.com/products/document-center/whitepapers/pikeos-and-time-triggering/`, 2013.

[112] M. Tiwari, J.K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F.T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 189–200. ACM, 2011.

[113] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Commun. ACM*, 39(4):76–83, April 1996.

[114] S.H. VanderLeest. ARINC 653 Hypervisor. In *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*, pages 5–E. IEEE, 2010.

[115] W.M. Vanfleet, R.W. Beckwith, B. Calloni, J.A. Luke, C. Taylor, and G. Uchenick. MILS: Architecture for high-assurance embedded computing. *CrossTalk*, 18(8):12–16, 2005.

[116] S. Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 239–243, Dec 2007.

[117] Marcus Völp, Adam Lackorzynski, and Hermann Härtig. On the Expressiveness of Fixed-Priority Scheduling Contexts for Mixed-Criticality Scheduling. In *Proc. WMC, RTSS*, pages 13–18, 2013.

[118] John Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies*, 34:43–98, 1956.

[119] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2002.

[120] J.H. Wensley. SIFT: Software Implemented Fault Tolerance. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part I*, pages 243–253. ACM, 1972.

[121] Inc. Wind River Systems. Wind River VxWorks 653 Platform 2.3. `http://www.windriver.com/products/product-notes/PN_VE_653_Platform2_3_0410.pdf`, 2010.

[122] B. Witwer. Systems integration of the 777 airplane information management system (aims): a honeywell perspective. In *Digital Avionics Systems Conference, 1995., 14th DASC*, pages 389–393. IEEE, 1995.

[123] Wenji Wu, Matt Crawford, and Mark Bowden. The performance analysis of linux networking packet receiving. *Computer Communications*, 30(5):1044 – 1057, 2007. Advances in Computer Communications Networks.

Curriculum Vitae
Stefan Resch

Personal Details

Date of birth: 17.01.1985
Citizenship: Austria

Education

Vienna University of Technology	Vienna, Austria
Computer Engineering, PhD	*Feb. 2012 - Dec. 2014*
Vienna University of Technology	Vienna, Austria
Computer Engineering, MSc	*Sept. 2008 - Dec. 2011*
Vienna University of Technology	Vienna, Austria
Computer Engineering, BSc	*Sept. 2004 - Sept. 2008*
Höhere Technische Bundeslehranstalt Wien Ottakring	Vienna, Austria
Secondary School	*Sept 1999 - June 2004*

Experience

Thales Austria GmbH	Vienna, Austria
PhD Student	*Jan. 2012 to date*