Execution Environment for Integrated Real-Time Systems based on Software-Defined Networking

DISSERTATION

zur Erlangung des Grades
eines Doktors der Ingenieurwissenschaften (Dr.-Ing.)

vorgelegt von
M.Sc. Hongjie FANG
geb. am 26.11.1990 in Guangdong, China

eingereicht bei der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen
Siegen 2019

Gedruckt auf alterungsbeständigem holz- und säurefreiem Papier.

Betreuer und erster Gutachter

Prof. Dr. Roman Obermaisser
Universität Siegen

Zweiter Gutachter

Prof. Dr. Raimund Kirner
University of Hertfordshire

Tag der mündlichen Prüfung
13.05.2020

This dissertation is dedicated to my beloved wife and my parents.

Acknowledgements

I would like to express my sincere appreciation and heartfelt thanks to my advisor Mr. Prof. Dr. Roman Obermaisser for his invaluable guidance during the process of my PhD research. Mr. Prof. Dr. Roman Obermaisser provided me constructive suggestions and generous assistance throughout our cooperation in the last years. Moreover, I appreciate the professional support and advice provided by the members of my examination committee, which essentially helped me to improve my dissertation.

I would also like to acknowledge the unconditional love and support from my wife, which inspires and motivates me to overcome the difficulties during this phase of my life. My sincere thanks to my parents for their infinite support in my academic career, and their beliefs in me to reach my goals in my life.

Finally, many thanks to all my friends and colleagues for your encouragement. It is a precious memory in my life to know you and work with you.

Abstract

Today there exists a wide range of industrial systems that are based on federated architectures, which means that the each computing node in the system is exclusively assigned to one function. Due to the increasing computing capability of a single processor and the increasing amount of computing processors on a single platform, extensive research on integrating multiple functions with different criticality levels on a shared platform was carried out. For example, in the avionic domain, the development trend has moved from federated to integrated architectures. The ARINC 653 standard was released, which defines the execution environment for hosting several avionic software functions within a single computing node. ARINC 653 was successfully implemented (e.g., Airbus A380) and achieved its primary goals (cost and weight reduction, enabling modular certification).

However, the existing execution environments based on an integrated architecture support only static system configurations. In specific domains like the railway industry, dynamic system adaptation is required during runtime, which affects both the application execution environment and the data communication mechanisms. In this dissertation, our focus is on an execution environment based on an integrated architecture, which guarantees the safe integration of mixed-criticality applications and also addresses the system reconfiguration problem.

In order to close the research gap, we introduce an execution environment for integrated real-time applications by leveraging the Software-Defined Networking (SDN) paradigm. We extend the temporal and spatial isolation mechanisms from the application layer to the execution environment, so that the integrated applications share the computing node without interference. For the data communication of the integrated applications, we propose a virtual switch supporting temporal and spatial isolation between data flows and leverage the SDN paradigm to address the reconfiguration requirements of data flows. Besides, we also address the controlled import and export of messages between data flows in the proposed virtual switch. For the deterministic communication requirements of hard real-time applications, we propose a virtual switch that is IEEE 802.1Qbv and IEEE 802.1Qci capable according to the Time Sensitive Networking (TSN) standard, in order to close the research gap of virtual switching guaranteeing bounded delay with low jitter in an integrated architecture.

In the proof-of-concept implementations, we demonstrate the non-interference between applications in the execution environment by fault injection. In our virtual switch demonstrators, we evaluate the fundamental isolation mechanisms and determinism of message switching, while measuring the caused overhead for message transmission as well as controlled data exchange, where the measured overhead in the proposed virtual switch is less than 10 μs.

Kurzfassung

Heute gibt es eine breite Palette von Industriesystemen, die auf föderierten Architekturen basieren, was bedeutet, dass jeder Rechenknoten im System ausschließlich einer Funktion zugeordnet ist. Aufgrund der zunehmenden Rechenleistung eines einzelnen Prozessors und der zunehmenden Anzahl von Rechenprozessoren auf einer einzigen Plattform wurde umfangreiche Forschung zur Integration mehrerer Funktionen mit unterschiedlichen Kritikalitätsstufen auf einer gemeinsamen Plattform durchgeführt. So hat sich beispielsweise im Bereich der Avionik der Entwicklungstrend von föderierten zu integrierten Architekturen verlagert. Der ARINC 653 Standard wurde veröffentlicht, der die Ausführungsumgebung für das Hosting mehrerer Avionik-Softwarefunktionen in einem einzigen Rechenknoten definiert. ARINC 653 wurde erfolgreich implementiert (z.B. Airbus A380) und erreichte seine primären Ziele (Kosten- und Gewichtsreduzierung, modulare Zertifizierung möglich).

Die bestehenden Ausführungsumgebungen auf Basis einer integrierten Architektur unterstützen jedoch nur statische Systemkonfigurationen. In bestimmten Bereichen wie der Bahnindustrie ist eine dynamische Systemanpassung zur Laufzeit erforderlich, die sowohl die Anwendungsausführungsumgebung als auch die Datenkommunikationsmechanismen betrifft. In dieser Dissertation liegt unser Fokus auf einer Ausführungsumgebung, die auf einer integrierten Architektur basiert, die die sichere Integration von mixed-criticality-Anwendungen garantiert und auch das Problem der Systemrekonfiguration angeht.

Um die Forschungslücke zu schließen, stellen wir eine Ausführungsumgebung für integrierte Echtzeitanwendungen vor, indem wir das Paradigma des Software-Defined Networking (SDN) nutzen. Wir erweitern die zeitlichen und räumlichen Isolations-mechanismen von der Anwendungsschicht auf die Ausführungsumgebung, so dass sich die integrierten Anwendungen den Rechenknoten störungsfrei teilen. Für die Datenkommunikation der integrierten Anwendungen schlagen wir einen virtuellen Switch vor, der die zeitliche und räumliche Isolation zwischen den Datenflüssen unterstützt und das SDN-Paradigma nutzt, um die Rekonfigurationsanforderungen der Datenflüsse zu erfüllen. Darüber hinaus befassen wir uns auch mit dem kontrollierten Import und Export von Nachrichten zwischen Datenflüssen im vorgeschlagenen virtuellen Switch. Für die deterministischen Kommunikationsanforderungen von harten Echtzeitanwendungen schlagen wir einen virtuellen Switch vor, der IEEE 802.1Qbv und IEEE 802.1Qci nach dem Time Sensitive Networking (TSN)-Standard ist, um die Forschungslücke des virtuellen Switchings zu schließen, das eine begrenzte Verzögerung mit geringem Jitter in einer integrierten Architektur garantiert.

In den Proof-of-Concept-Implementierungen zeigen wir die Nicht-Interferenz zwischen Anwendungen in der Ausführungsumgebung durch Fehlerinjektion. In unseren Virtual-Switch-Demonstratoren bewerten wir die grundlegenden Isolationsmechanismen und den Determinismus des Message-Switching, während wir den verursachten Overhead für die Nachrichtenübertragung sowie den kontrollierten Datenaustausch messen, wobei der gemessene Overhead im vorgeschlagenen Virtual-Switch weniger als 10 μs beträgt.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the roadmaps of the semiconductor industry, according to the Pollack's Rule [104], a new generation of micro processors with a doubled number of transistors leads to about 40% performance increase. The Moore's law [112] states that the number of transistors on a microprocessor doubles every two years. That means, a new generation of micro processors with a doubled number of transistors every two years leads only to about 40% performance increase. However, the Moore's law is coming to the end since the traditional shrinking of transistors is reaching its limits [78]. If the added transistors are used for multiple cores instead of a single core, then the performance and computation capability are doubled by doubling the transistors, which means that the penalty of Pollack's rule is avoided. As a consequence, the computing platforms are using multi-core processors instead of single-core processors, in order to gain more computation capability and energy efficiency with reduced deployment cost and complexity [32]. Due to the increasing computing power of modern integrated platforms based on multi-core processors, extensive research on integrating multiple applications with different criticality levels on a shared platform was carried out in different domains such as avionics and automotive systems. Corresponding industry standards like ARINC 653 [23] and AUTOSAR [41] were released. Integrating applications on a shared computing node leads to a mixed-criticality system, since safety-critical applications can be combined with the non-critical ones [97]. In order to address the integration of mixed-critical applications with reconfiguration requirements, this dissertation introduces an execution environment for an integrated architecture, which leverages the SDN paradigm to support dynamic time-triggered communication. The proposed execution environment guarantees the safe integration of mixed-criticality applications in an integrated system and also addresses the specific requirements of system reconfiguration.

Today there exists a wide range of industry systems adopting federated architectures, which means that the computing nodes are function-specific, i.e., every computing node in the system hosts exactly one function. For example, in today's railway domain, the Train Control and Monitoring System (TCMS) is a train-borne distributed control system which provides a single point of monitoring and controlling of all applications within a train. The existing TCMS execution environments are based on a federated architecture because of the specific industry constraints and the

long development life-cycles. The railway transportation systems have suffered from a limited adoption of novel technological advancements in electronic hardware and software, communication networks and embedded computing.

As analysed in [98], a federated system provides better fault containment than an integrated architecture due to the nature of "one function - one ECU". Either a hardware fault or a development fault can be limited to affect a single application. The development of applications based on federated architectures is independent from each other thanks to the restricted interactions between applications via gateways. Since each application within the federated architecture owns its dedicated computing node, the unintended interference between applications is ruled out, which consequently reduces the system complexity. However, sharing an integrated system among several applications contributes to hardware cost reduction and simplifies wiring of a system, which can increase the system dependability, because more than 30% of electrical system failures are caused by wiring malfunction [118]. Furthermore, a safety-critical system should be fault-tolerant to keep the safety-critical applications functioning properly despite the occurrence of faults. Computing and communication resources in an integrated platform can serve as a generic backup for multiple applications, which leads to the reduction of replicated hardware in comparison to a federated system, where computing nodes are application specific.

The development based on integrated architectures in automotive and avionic industries led to significant industrial successes (e.g., Boeing 787, Airbus 380, etc.), while the adoption of an integrated architecture in other domains offers research questions due to the respective domain-specific requirements.

1.1 Problem Statement

The research problem we address in this dissertation is how to leverage the SDN paradigm to build up an execution environment for integrated real-time systems, which support hard real-time and system reconfiguration during runtime. This problem is divided into three major parts stated as follows.

- **Research problem 1: Dependable execution environment that enables integration of safety-critical and non-critical applications and supports system reconfiguration.**

  In the avionic and automotive industry, research on integrating applications with different criticality levels on the same computing node was carried out. The foundations for this integration are mechanisms for temporal and spatial partitioning, which are designed to prevent fault propagation between partitions and guarantee the reserved computing resources for each partition, so that safety critical applications cannot be affected by applications of lower criticality. Previous systems such as automotive and avionic systems were static, whereas other domains such as railway require more dynamic system structures to cope

with inauguration problem during runtime, which is the open challenge to be addressed. More specifically, train inauguration consists of operations executed in case of train coupling to give all computing nodes their addresses, orientation and other necessary information.

- **Research problem 2: Virtual switching supporting time-space partitioning and dynamic configuration for integrated systems.**

  In the proposed execution environments based on integrated architectures in the automotive and avionic domains, the data communication is statically configured through channels between communication ports. When the fundamental architecture of the execution environment evolves from a federated architecture to an integrated one, and the data communication requires for online dynamic reconfiguration, the online rescheduling is a major challenge which requires the execution environment to dynamically adjust to changes of applications.

- **Research problem 3: Property transformation and encapsulation between data flows of different applications in an integrated system.**

  Since industry applications are certified to have different safety critical levels, encapsulation between data flows is necessary to prevent fault propagation from non-safety critical applications to safety critical ones. The messages of the data flows could have different properties (e.g., syntax, semantic, etc.), which requires the proposed virtual switch to be able to achieve property transformation when conveying messages between different data flows. In this dissertation, we propose a virtual gateway within the proposed virtual switch to address the controlled import and export of messages between different data flows.

1.2 Contributions

In this dissertation, based on the existing execution environments in automotive and avionic domains, we extend and adjust the temporal and spatial partitioning mechanisms to meet other requirements like the dynamic characteristics in the railway industry, and meanwhile preserve the timing determinism and non-interference in the spatial and temporal aspects.

We address the data communication by leveraging the SDN paradigm to propose a virtual switch residing in the execution environment of integrated systems. Our proposed virtual switch enables hard real-time communication of applications on a single computing node.

The challenge of property transformation and encapsulation between data flows of different applications in the proposed virtual switch is also tackled in this work. We propose the gateway services in the virtual switch and evaluate the caused overhead to show that the determinism of the data communication is not affected.

1.3 Document Structure

The structure of the dissertation is as follows:

In Chapter 2, the basic concepts and terms that are used throughout this dissertation are discussed. It starts with dependability concepts ranging from the attributes to faults that affect the system dependability, and the necessary means to protect system dependability. The notion of a system architecture and real-time system are discussed, before the fault hypothesis concept is discussed. Followed by the partitioning and fault & error containment concepts, the communication mechanisms are discussed. At the end of this chapter, the system reconfiguration concept is explained.

Chapter 3 gives an overview of the related work in the area of execution environments in integrated real-time systems. The state-of-the-art analysis of Real-Time Operating Systems (RTOSs) discusses the RTOS capabilities and the representative products, which are either commercial or open-source. The ARINC 653 and AUTOSAR standards are analysed with respect to system architecture and real-time capacity. The other related research fields are virtual networking and SDN.

Chapter 4 introduces the requirements for the execution environment of integrated real-time systems, based on which the functional distribution framework concept is proposed including the logical and physical components. This chapter ends up with the proof-of-concept implementation and the corresponding results.

Chapter 5 addresses the data communication in the execution environment of integrated real-time systems. This chapter presents the virtual switching of data flows on an integrated platform, which enables real-time communication on a single computing node, so that federated computing nodes are enabled to be integrated on a single node while preserving the deterministic real-time communication.

Chapter 6 concludes the research work in this dissertation and looks into the future work in the development of integrated real-time systems.

Chapter 2

Basic Concepts and Terms

In this chapter, the basic concepts and terms used in this dissertation are presented and explained, in order to provide the background knowledge for later chapters.

2.1 Dependability

In [66] dependability is defined as the property of a computer system that reliance can be justifiably placed on its service. A system behaves as it is perceived by the user(s) in delivering the service, where a user is another system (human or physical) which interacts with the serving system. The concept of dependability can be organised in a tree structure as shown in Figure 2.1. In this systematic illustration of dependability, one can distinguish the concerned attributes (e.g., availability, safety, etc.) that compose the general dependability for critical systems. A fault is the fundamental cause of an error, which can lead to a system failure. Consequently, the presented measures to preserve the system dependability range from fault forecasting to fault prevention before a faulty situation happens, and fault tolerance/removal in the faulty scenarios to keep the system dependable.

2.1.1 Attributes

In the following sections, the attributes that characterise the system dependability are discussed in detail.

2.1.1.1 Availability

Availability states the accessibility of a system to the system users. According to [45], the traditional definition of availability emphasises the capabilities of a system regarding the failure and repair characteristics. In the traditional consideration, neither the readiness of a system to carry out an operation nor the accessibility via networks are taken into account, which results in the overestimation of availability. The proposed definition of availability is treated as a scalar measure that is based on the recovery-enabled reliability as mention in chapter 2.1.1.2. It reflects the instantaneous availability [125] of a system that is capable to recovery after failures. Other proposed

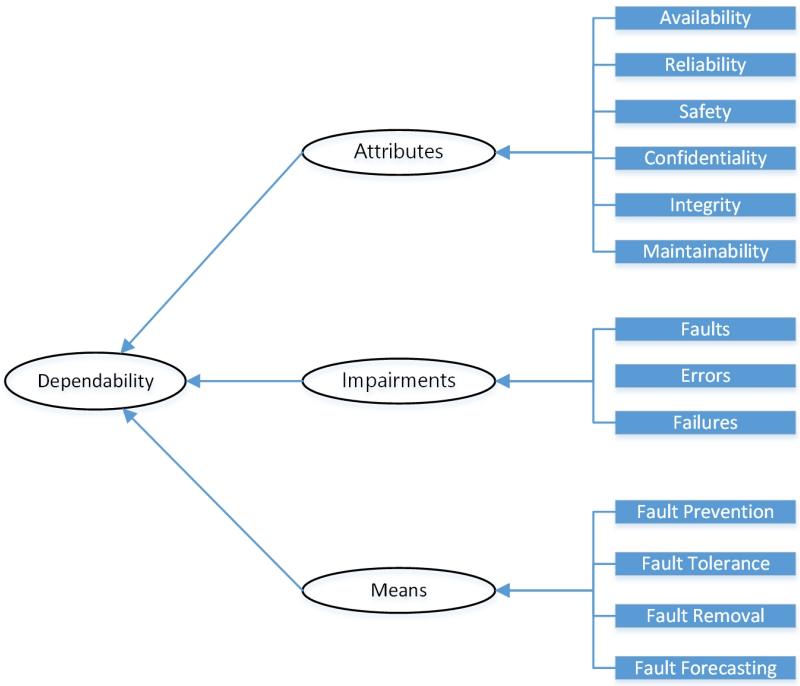

FIGURE 2.1: Dependability Tree

availability measures[21, 83] take more complex system characteristics (e.g., system size, composition) into account.

2.1.1.2 Reliability

As discussed in [45], the system reliability captures the ability of the system to operate continuously without interruption. The authors in [45] refined the existing reliability definition with the extension of casual points in time to measure the system reliability, and renamed it as system reliability. Apart from the system reliability, the authors also define the service reliability, which states the ability of the system to complete a service successfully, even in the presence of failures, given that the system accepted the service request in the first place. The prerequisite of this definition is that the system is capable to detect, repair and recover from failures and design errors. The definition of service reliability acknowledges the fact that system failures can happen and the system capability to recover from failures affects the system reliability positively.

2.1.1.3 Safety

As discussed in chapter 2.1.1.1 and chapter 2.1.1.2, availability and reliability concentrate on the system capability to avoid failures, while safety emphasises the avoidance of catastrophic system failures, which can lead to severe human injury and environmental damage [4]. As illustrated in [102], safety represents the probability of a system to stay in safe states even when encountering failures of system components.

For example, safety is of great importance to the transportation industry, because passengers on-board vehicles should be ensured to stay safe in both normal and system malfunction scenarios. The safety requirements for the execution environment proposed in this dissertation is discussed in section 4.1.2.1.

2.1.1.4 Confidentiality

Confidentiality is one of the attributes that are grouped into the scope of system security [65]. The confidentiality of a system ensures that only the authorised end users can access their assigned resources. As an example, in the security engineering for vehicle communication [56], the confidentiality is treated as one of the security goals to guide the analysis of the security requirements, so that the content of communicated messages is only allowed to be accessed by the authorised sender and receiver.

2.1.1.5 Integrity

According to [67], integrity excludes the occurrence of improper alteration of information in a computer system, and it is also one of the prerequisites for availability, reliability and safety. However, guaranteeing integrity does not necessarily ensure confidentiality of a system, because a message that has not been tampered with can still be attacked via passive listening, which results in information leakage.

2.1.1.6 Maintainability

As stated in [66], maintainability is defined as the ability of a system to undergo repairs and evolution. Maintainability consists of not only corrective maintenance, but also adaptive and predictive maintainence. From the viewpoint of a computer system, the system should be able to preserve the ability of delivering mandatory services with respect to the service agreement. Another relevant capability is that a system should be maintainable in the face of environmental changes (e.g., adoption of new operating systems or new system databases).

2.1.2 Impairments

According to the "fault-error-failure" chain discussed in [67], a fault within a system could lead to a system error, which states that the system deviates from the predefined state, and consequently the system's services cannot be fulfilled. The failure of a service can subsequently represent another fault, ultimately leading to a so-called system failure.

As analysed above, the essential reason of an observable system failure is a system fault. In order to maintain the dependability of a system, we analyse different kinds of system faults and the corresponding means to maintain dependability in the following sections.

2.1.3 Fault Classes

Since a system consists of different building blocks such as hardware infrastructures, software entities and end users, system faults can be correspondingly classified into hardware faults, software faults and interaction faults. System faults can also come from the environment (e.g., EMI, radiation), which are not emphasised in this dissertation.

2.1.3.1 Hardware Faults

As discussed in [93], the faults that hit the system's physical resources are called hardware faults. The hardware faults either result from manufacturing faults during the production process, or are caused by environmental threats during operation. Assumed that hardware diversity is leveraged to avoid common mode failures and the computing nodes are well protected against the environmental threats (e.g., shielding against EMI, separate grounding), then each computing node forms a Fault-Containment Region (FCR).

2.1.3.2 Software Faults

Since this dissertation concentrates on an integrated architecture, according to [93], the software in the whole system can be classified into system software and application software. The same system software deployed on multiple computing nodes could be affected by the same software faults (e.g., design fault), which prompts the computing nodes to form a common FCR. From this aspect, system software should be designed to be simple enough for thorough validation (e.g., micro-kernel of PikeOS [39]). Regarding the application software, a group of replicated instances must be treated as the FCR.

2.1.3.3 Interaction Faults

As defined in [65], interaction faults represent the faults caused by human operators due to their faulty inputs provided to the computer system. This kind of faulty operations can originate from poor system design of man-machine interfaces or lack of system assistance for the operators to make the correct decisions during the interaction. Both intentional or unintentional interaction faults should be tolerated by the system to avoid system failures.

2.1.4 Means of Dependability

2.1.4.1 Fault Prevention

Fault prevention includes the means to prevent the occurrence or introduction of faults [5]. Fault prevention is traditionally treated as part of the engineering process, i.e., during the development of hardware and software entities. Compared to a

development methodology that passively records the faults in the products, fault prevention leverages developed mechanisms to actively reduce the introduced faults in those products.

2.1.4.2 Fault Tolerance

As stated in [67], fault tolerance consists of error processing and fault treatment. In case a system fault happens, the error caused by the fault should be dealt with and a failure should be prevented, in order to achieve the fault tolerance.

Error processing includes error detection, error diagnosis and error recovery that try to identify an erroneous system state, to assess the damages caused by propagated or detected errors, and to recover from an erroneous state, respectively.

The mentioned fault treatment firstly diagnoses an existing fault to determine the causes, so that the fault could be passivated by preventing faulty components from execution. Thereafter reconfiguration can happen in case that the whole system is not able to provide the required service after isolation. This kind of reconfiguration is often acceptable even at the price of degraded services.

2.1.4.3 Fault Removal

During system development, fault removal [5] comprises steps like verification, diagnosis and correction. The verification step aims to test the system either statically or dynamically to find out whether the system conforms to its specification and possibly to reveal both hardware and software faults. After testing, the system is diagnosed to spot the system faults and then these identified faults should be corrected.

During the usage phase of a system, fault removal [5] emphasises the corrective or preventive maintenance. Corrective maintenance aims to remove faults that already cause errors in the system, while preventive maintenance takes the potential hardware and software faults into account.

2.1.4.4 Fault Forecasting

Fault forecasting [65] aims to estimate the current number of faults and the future incidence of faults as well as the possible consequences resulting from the faults. The system faults can be identified and classified in a qualitative way and the corresponding mechanisms to deal with the identified faults can also be identified. In a quantitative way, the satisfaction probabilities of the aforementioned attributes of dependability (cf. section 2.1.1) can be evaluated. Tools exist to carry out the qualitative and quantitative evaluations separately or in a coordinated way [66].

2.2 System Architecture

From the logical point of view, a system consists of various applications (e.g., driving assistance, entertainment in car), which can be composed by distributed functions. From the physical point of view, a system is built up with physically networked computing nodes. Based on the mapping between functions and computing nodes, a system can be classified into a federated architecture or an integrated architecture. From the functional criticality point of view, both types of system architectures can be mixed-criticality architectures.

2.2.1 Federated Architecture

In a system based on a federated architecture, one distributed function owns its dedicated computing nodes and an application is built up with the distributed functions that are connected via physical communication channels [61, 99]. We depict an example federated architecture in Figure 2.2. In this example, each computing node hosts one function and the functions belonging to the same application (i.e., functions in the same colour) communicate with each other via the network.

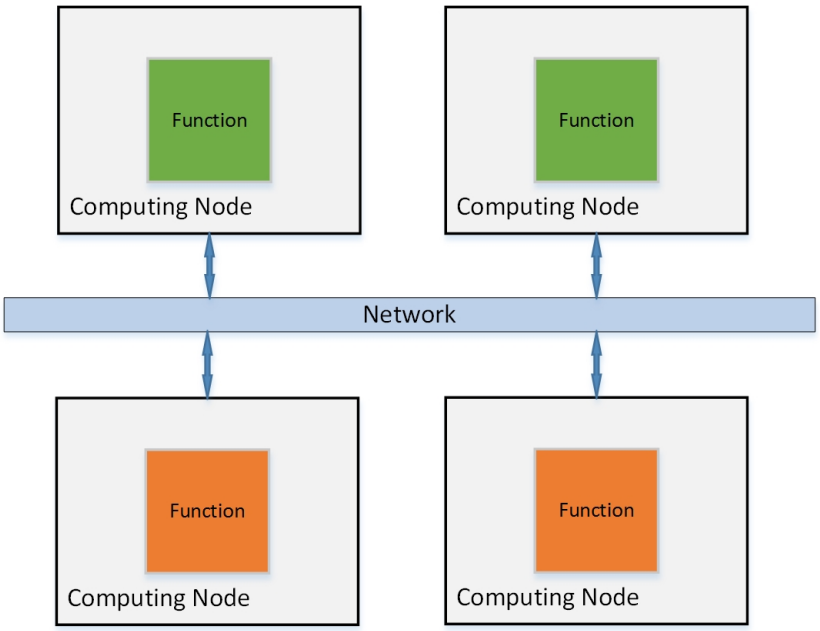

FIGURE 2.2: Example of a Federated Architecture

In the case of an ultra-dependable application, the federated architecture has so far been preferred due to the natural advantages of the architecture [96]. Thanks to the clear boundaries between functions and the heterogeneity of hardware platforms, neither a hardware fault nor a software fault can affect the other connected nodes. Errors within one function can be handled within one node before propagating to other ones. The interaction between applications can be resolved by gateway services, so that the development of applications can be done independently, which also enables the separation of concerns between applications.

Based on the system paradigm of a federated architecture, the amount of computing nodes in a system increases linearly with respect to the number of functions. Consequently, the lack of multiplexing at the hardware level leads to high cost of computing nodes and wiring effort, and implicitly results in large dimensions and high energy consumption.

2.2.2 Integrated Architecture

As mentioned in [98], one can introduce an integrated architecture if one computing node is shared among different functions. As depicted in Figure 2.3, each computing node in the system hosts more than one function in comparison to the above mentioned federated architecture. The consequential advantage of an integrated architecture is the reduction of the used computing nodes as well as the wiring for the communication network. Less wiring and the reduced amount of connectors contributes to the improvement of the system dependability. Furthermore, a safety-critical system should be fault-tolerant to maintain the functionalities of safety-critical functions also during faulty situations. If an integrated architecture contains general-purpose computing nodes, then they can serve as a backup resource for multiple functions, which leads to a reduction of the required replicated hardware compared to a federated system, in which the computing nodes are function specific. Integrating different functions on the same computing node also eases the coordinated communication between these functions, since the functions access the same timing resource that is essential for time-driven execution.

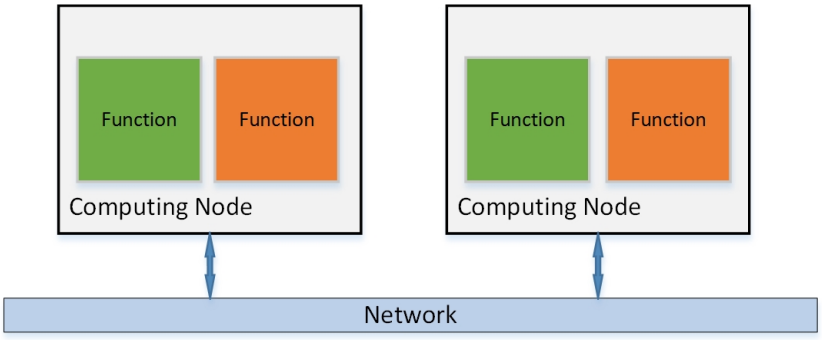

FIGURE 2.3: Example of an Integrated Architecture

From the system integration point of view, the integrated architecture increases the complexity of the computing nodes, due to the potential interference between functions hosted on the same computing node. This challenge is solved in the state-of-the-art and the corresponding measures like time-space isolation and virtual communication links were proposed [96].

2.2.3 Mixed-Criticality Architecture

As introduced in [15], a system is treated as a mixed criticality system, when the functions of the system are of distinct criticality levels (as defined in standards such

as IEC 61508 [49], DO-178C [31], DO-254 [109] and ISO 26262 [50]).

The example mixed-criticality system in Figure 2.4 shows that an integrated system can host several functions of different Safety Integrity Levels (SIL) (e.g. SIL 2 and SIL 4 according to IEC 61508). Another depicted aspect in this example is that the partitioning mechanism as discussed in section 2.5 is a prerequisite to segregate the functions in a modular way. A mixed-criticality system enables different applications to be certified against different criticality levels by leveraging the time-space isolation paradigm, so that the overall system need not be certified to the highest criticality level.

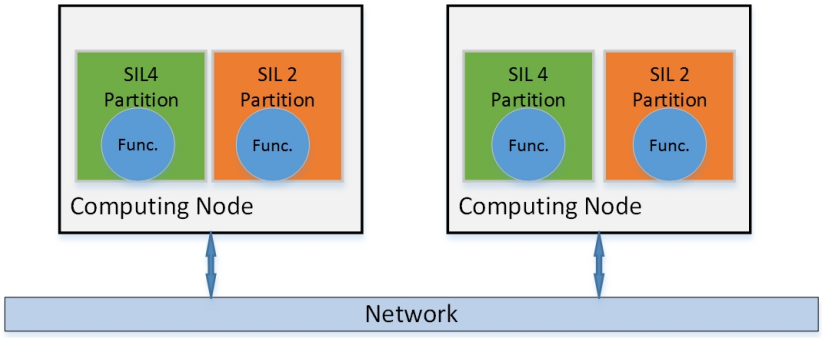

FIGURE 2.4: Example of Mixed-Criticality Architecture

2.2.4 Execution Environment

An execution environment within a computing node consists of the computational resources (e.g., memory) and architectural services (e.g., clock service) to execute the hosted applications of the computing node. Within an integrated computing node, the execution environment can establish isolated computational resources for different applications, in order to prevent faults from propagating between applications.

2.3 Real-Time Systems

As defined in [62], a real-time system provides correct computational results in both the logical and temporal domains. In other words, a real-time system should finish each operation correctly without exceeding the corresponding deadline.

From the system point of view, a real-time system is either permanently ready to process random inputs from the physical environment, or it is scheduled to be ready for processing specific inputs that appear at predetermined times. The required system outputs are generated before predetermined deadlines. From the viewpoint of the system, if a real-time system consists of multiple dependent functions, the intermediate results of the functions should meet the derivative requirements in the time and value domains.

A real-time system is generally classified based on the consequences when the system misses predetermined deadlines. A real-time system can be classified into hard real-time or soft real-time as discussed in the following sections.

2.3.1 Hard Real-Time System

In a hard real-time system, applications should be guaranteed to meet the specified deadlines without any exceptions, and any violated time constraints can lead to a severe system failure.

An example of a hard real-time system is the Anti-lock Braking System (ABS) in vehicles. Missing a response deadline of the ABS can result in a vehicle being out of control, which endangers the safety of the passengers.

2.3.2 Soft Real-Time System

A soft real-time system has timing constraints that are not requested to be strictly satisfied. In other words, missing a deadline in a soft real-time system can affect the system behaviour without causing a critical system failure.

One example of a soft real-time system are the multimedia services in networked environments. The frames transmitted over a network can be delayed or even dropped alongside the transmission channels, which results in stagnation or frame skipping of the media.

2.4 Fault Hypothesis

As stated in [93], in the scope of integrated architectures, a fault hypothesis aims to identify the FCRs, and it also specifies the failure modes and provides realistic failure rate assumptions. Based on the fault hypothesis, the implementation and validation of fault-tolerant strategies is enabled. In the following sections, we discuss the constituting parameters of a fault hypothesis in detail.

2.4.1 Fault-Containment Regions

According to [64], a FCR is a collection of components that operates correctly regardless of any arbitrary logical or electrical fault outside the region. One can classify the faults into either a FCR fault or a system fault, where a system consists normally of different FCRs. According to the fault-error-failure chain introduced by Laprie [4], a fault within one FCR A can lead to the failure of FCR A and this failure can cause a fault in FCR B if it propagates to FCR B through the communication network between FCR A and FCR B. Redundancy-based mechanisms are widely implemented to achieve failure masking for the FCRs.

2.4.2 Failure Mode Assumptions

From the view point of the user, one FCR could have different kinds of failure modes that determine the necessary redundancy mechanisms to achieve error containment of the FCR [93]. In this dissertation, we address the critical failure modes that are relevant in an integrated architecture for safety-critical environments. According to [60], the failure modes are as follows:

- Babbling idiot failures: A FCR suffers from babbling idiot failures when it sends out messages during unassigned time slots in a time driven communication network. This kind of failure can be detected and handled by guardian systems within the communication system.
- Masquerading failures: A FCR can masquerade other FCRs if it is able to assume the identity of other FCRs. A masquerading failure could be a serious threat to the whole system with respect to the widely applied fault-containment mechanism via replication and voting on replicated messages, because a single faulty application within one FCR can even masquerade multiple replicated applications and send out masqueraded messages to overwrite the correct ones. Consequently, if the messages contain naming or addressing information for the underlying routing infrastructure, extra detection mechanisms within the computing node should be carried out to protect the whole system from masquerading failures.
- Slightly-off-Specification (SoS) failures: Different FCRs in a system can interpret the same received faulty message as valid or invalid based on their own specification of this message. The faulty message could have slight differences with respect to time, value or coding aspects.
- Crash/Omission (CO) failures: The CO failure mode states that a computing node operates correctly or crashes, and the communication system transports messages correctly or is unable to transport any messages. In the communication system, acknowledgement or membership services help to detect omission failures. Within computing nodes, detection of FCR crashes should be carried out at the architecture level, because a FCR crash results in not produced outputs that can be detected by other FCRs without guarantees.
- Massive transient disturbances: Phenomena like Electro-Magnetic Interference (EMI) can disturb communication systems and even lead to temporary loss of communication between FCRs. During such a period, computing nodes need to detect the communication failure and carry out default safe actions until the communication system finishes recovery. Proper quality engineering of the communication medium also contributes to avoiding such failures.

2.4.3 Failure Rate Assumptions

As discussed in [93], the failure rate depends on the kinds of failure modes. In addition, the differentiation of failure rates is affected by the failure persistence (e.g., transient failures, permanent failures).

2.4.4 Recovery Interval of an FCR

The FCR recovery interval is the maximum interval of time between the point in time of a FCR failure happens and the FCR provides the correct service again [93].

2.4.5 Maximum Number of Failures

As defined in [93], the parameter specifies the maximum number of FCR failures that a system should be capable to handle. The maximum number of failures is determined by the aforementioned failure rate and recovery interval. Today the prevalent assumption in safety-critical systems is a single fault hypothesis, i.e., a system suffers at one point in time a maximum of one FCR failure.

2.5 Partitioning

Partitioning is one of the central concepts for a computing node of an integrated architecture to host applications of different critical levels. Partitioning represents the isolation mechanism that can be applied at different levels (e.g., processor, memory, I/O) within an integrated system, in order to provide independent execution environments for different applications. In this sense, each application within one partition forms a Fault Containment Region (FCR) that delimits the immediate effect of a fault [97]. As stated in [110], a partitioned system should provide fault containment equivalent to a federated system in which each partition is allocated an independent processor and associated peripherals and all inter-partition communications are carried on dedicated lines.

The partitioning paradigm is classified into temporal and spatial partitioning that will be discussed in detail in the following sections.

2.5.1 Temporal Partitioning

Temporal partitioning guarantees that applications residing within a computing node of an integrated architecture are segregated with respect to the execution time. According to [97], temporal partitioning ensures that the FCRs have no effect on each other on the ability to access shared resources (e.g., common network, shared processor). From the view point of a partition, the temporal properties (e.g., latency, jitter, duration of availability) of the services provided by the shared resources during the scheduled accessing time interval are not affected by the other ones [110].

At the processor level, the partitions within the same computing module are normally scheduled on a fixed and cyclic basis. In order to implement this scheduling mechanism, the CPU time is divided into Major Time Frames (MTF) with preconfigured duration. The hosted partitions are activated within at least one time window of every MTF and only one partition can be activated during one time window. The system integrator defines the order of the activated partitions, taking the partition attributes like period and Worst Case Execution Time (WCET) into account.

At the communication network level, there are different protocols (e.g., TTEthernet [52], ARINC 664 [22]) guaranteeing timely delivery of messages. TTEthernet leverages Time-Division Multiplexing (TDM) to create dedicated virtual channels between communicating entities, so that the other unauthorised entities cannot access or interfere with the communication in their unassigned time intervals. Temporal partitioning in ARINC 664 is realized through Virtual Links (VLs) that are characterised by Bandwidth Allocation Gaps (BAGs) and jitter. The BAG specifies the minimum time interval between two messages of the same VL, and messages passing through the scheduler can be dispatched in a bounded time interval that represents the maximum admissible jitter. The VL represents a logical unidirectional connection from one source computing node to at least one destination computing node and each VL is timely isolated from the others.

2.5.2 Spatial Partitioning

According to the definition in [110], spatial partitioning must ensure that software in one partition cannot change the software or private data of another partition (either in memory or in transit) nor command the private devices or actuators of other partitions.

At the memory level, partitions in an integrated architecture should have their allocated memory areas. The allocation of the system memory is based on the partition requirements and it is achieved at the partition granularity. In many COTS products (e.g., PikeOS, VxWorks), a Memory Management Unit (MMU) is used to implement the spatial isolation of memory resources at the hardware level [97].

At the processor level, since the processors could be timely shared between partitions (i.e., temporal partitioning in chapter 2.5.1), there could be processor context (e.g., register values) left from the last active partition. In order to avoid a partition being affected by the last executed partition on the same processor, the processor context should be carefully dealt with, so that the behavior of a partition is not implicitly affected by the last one. For more detailed background on spatial partitioning, we point to [110].

At the communication network level, since the physical communication infrastructure is timely multiplexed among the applications, the major point to be concerned with in spacial partitioning is the possibility that one application could manipulate the messages of other applications via unintended access (e.g., altering of messages in message buffers not owned by the application) [93]. From the viewpoint of the

execution environment on a computing node, it is mandatory to isolate the system resources for the communication of different applications.

2.6 Fault and Error Containment

When we talk about safety-critical applications integrated with other non safety-critical applications on the same computing platform, the protection against failures affecting safety-critical applications must be addressed.

2.6.1 Fault Containment

In the domain of integrated architecture, different applications integrated on the same computing node share the computing hardware, power supply, timing source etc., which can fail due to a hardware fault and result in the failure of the whole computing node. According to the definition of FCR (see section 2.4.1), one computing node hosting multiple applications should be treated as a single FCR. In a single FCR, a fault cannot immediately affect another FCR, however, a faulty application could send out faulty messages to the other applications residing in different FCRs, which results in the required error containment in the following section.

2.6.2 Error Containment

2.6.2.1 Within computing node

When a software fault happens within one application that is integrated with other applications and hosted on the same computing node (i.e., the same FCR), there is the need to prevent the error resulting from the fault in the application from propagating to other applications on the same platform. According to [96], the temporal and spatial isolation between applications contributes to avoiding timely interference of the CPU and other resources (e.g., memory, I/O) for each application. This isolation applies also to the node-internal communication mechanisms.

Messages carrying erroneous values or delivered untimely could propagate from a faulty FCR to another FCR and lead to another failure in it. In order to prevent error propagation between FCRs, communication guardians at the architectural level can be leveraged to avoid timing error propagation and voting at the application level contributes to the value failure handling [60].

2.7 Communication Mechanisms

In order to fulfil the communication requirements between different applications running in parallel, message-based communication and shared-memory based communication are the two mainly used technologies [68].

2.7.1 Shared-memory based Communication

In case of communication based on a shared memory, the applications are authorized to access identical memory areas to achieve the inter-process communication between the applications. If a computing platform is integrated with homogeneous computing resources, the shared memory communication is favorable, and the message-passing mechanism can be the preferred option on heterogeneous computing resources.

In the Multi-Processor Systems-on-Chip (MPSoC) domain, the communication based on shared memory is competitive due to the low cost of on-chip memory accesses [103].

2.7.2 Message-based Communication

In the message-based communication, the communicating entities manage the local resources and exchange information via messages. In an integrated computing environment, the message-based communication provides an effective abstraction between computing nodes and the underlying network, as well as between applications on the same platform.

As discussed in [95], message-based communication contributes to complexity control for integrated architectures in the way that the well-defined messages represent the properties of the application in a clear fashion. The message-based communication is leveraged in ARINC 653 and AUTOSAR standards that are discussed in the following chapter.

2.8 System Reconfiguration

In an integrated system, system changes caused by intended or unintended reasons result in systematic reconfiguration. An example of an intended system change is the train inauguration process in the railway domain. The transportation vehicles in the railway domain are enabled to couple and decouple with each other during daily operation, in order to achieve the dynamic composition purpose. One representative unintended reason causing system reconfiguration is an unpredicted module fault in an integrated system that affects safety-critical applications, which can be moved from a faulty module to a redundant one [10].

In this dissertation, system reconfiguration means the dynamic adaptation of the execution environment during system runtime, which affect both the configuration of the applications and the communication infrastructures. As discussed in [55], the following requirements should be fulfilled to achieve the predictable timing and continuity of service during the system reconfiguration.

- **Assured Reconfiguration [117].** This requires the effects of system reconfiguration to be known beforehand. If a system reconfiguration affects a safety-critical application, the caused consequences should be predictable.

- **Bounded Reconfiguration Time.** The system reconfiguration should be finished within bounded time interval that is available in the executing system.
- **Continuity of Service.** If a system reconfiguration affects subset of the applications, the other subset of applications should be satisfied with required resources to provide continuous service.
- **Consistent Configuration.** From the viewpoint of the applications, the system configuration should be consistent during runtime. System reconfiguration should not cause intermediate configurations that result in various system configurations for different applications.
- **Robust Reconfiguration Mechanisms.** The system reconfiguration process should avoid to cause failures, which can potentially cause a system failure.
- **State Preservation.** As discussed in [13], relevant system states that are valid after system reconfiguration should be retained, in order to avoid causing system failure.

Chapter 3

Related Work

This chapter analyses the related work in the field of execution environments for integrated real-time systems.

3.1 Real-Time Operating System

A system hosting real-time applications requires an RTOS to provide a deterministic framework, so that logical and temporal correctness of the real-time applications can be guaranteed. In this section, the memory management and scheduling mechanisms of an RTOS are discussed, followed by the discussion of selected RTOSs (PikeOS and Linux RTAI).

3.1.1 RTOS Capability

This section addresses the memory management and scheduling issues of an RTOS.

3.1.1.1 Memory Management

As discussed in [86], the memory management mechanism within a RTOS can be classified into static and dynamic memory management.

- Static memory management. The free memory in a system is allocated statically to the tasks, which hold and free the shared memory during execution. Specific memory areas can be reserved for critical tasks while meeting the worst-case demand of the tasks.
- Dynamic memory management. As a dynamic strategy, one single memory space can be multiplexed by a task. In another word, a task can run on a memory space that is smaller than required by overlaying the same memory area during execution.

Most of the COTS RTOSs leverage a MMU (e.g., LynxOS [80], PikeOS [54] and Xtratum [24]) or a Memory Protection Unit (MPU) (e.g., VxWorks [80]) from the underlying hardware platform to protect application specific memory from undesired accessing.

3.1.1.2 Scheduling

For the scheduling in hard/soft real-time systems, applications can be scheduled either statically or dynamically.

- Static scheduling. In static scheduling algorithms, the information of the scheduled entities is completely known before execution and the target schedule is statically computed [58], which can be in a preemptive or non-preemptive fashion. A static preemptive schedule can work in clock-driven way like the temporal partitioning mechanism in the ARINC 653 standard [23] that will be discussed later. In case the allocated execution time of the partitions is longer than the worst-case execution time of the partitions, no preemption is necessary during partition switching, which results in the static non-preemptive schedule.
- Dynamic scheduling. Similar to the static scheduling, dynamic scheduling can be preemptive or non-preemptive. One representative of dynamic preemptive scheduling is the Earliest Deadline First (EDF) scheduling mechanism [72], where the task with the earliest deadline is assigned the highest priority and enabled to preempt the running task. When the priory assignment mechanism stays unchanged and tasks in execution cannot be preempted, the EDF scheduling algorithm works in a non-preemptive fashion.

3.1.2 Related Work on RTOS

In this dissertation, the involved RTOSs are the PikeOS and the real-time Linux variant Linux RTAI. The PikeOS is an RTOS certified to the highest criticality levels (SIL4 according to IEC61508, Class A according to DO178C), and the Linux RTAI is an open source RTOS. These RTOSs will be discussed in detail in this section.

3.1.2.1 PikeOS

PikeOS is a type 1 hypervisor that supports hardware virtualisation and para-virtualization functionalities. The CPU time is divided into time windows called time partitions and the isolated time partitions are grouped to form a scheduling schema, which is executed cyclically. PikeOS can switch between different scheduling schemes during run-time. Processes within one time partitions are scheduled in a priority-based preemptive fashion. The extra time partition 0 is defined as a background partition to run critical or non-critical processes, so that processes with low latency constraints can be executed timely, meanwhile the idle CPU time can be utilized by the non-critical processes. Partitions are independent execution environments to host other OSs with the capability to configure the affinity of processors.

In PikeOS, the physical memory can contain multiple physical memory regions, which are used to serve the per-partition memory allocation (e.g., thread/task descriptors). This kind of physical memory partitioning ensures that one partition is served by one memory controller, and specific physically tagged cache entries are

not evicted, therefore predictability in the per-partition memory access is preserved. As above mentioned, PikeOS leverages a hardware MMU to perform its memory isolation. Each partition and every task within a partition has its own virtual memory space. The memory assignment is done statically during the system integration process.

3.1.2.2 Linux RTAI/LXRT

The RTAI/LXRT Linux extension was proposed in [82] to add a middleware between the hardware infrastructure and the regular Linux kernel, in order to provide real-time services for the hosted applications.

- Scheduling Strategy. The patched RTAI Linux kernel can block or redirect hardware interrupts to avoid delays of real-time applications. Static and dynamic priority-based scheduling are both enabled in this kernel, where the normal Linux kernel is configured with lowest priority and runs only when the real-time kernel is idle.
- Memory Management. Linux RTAI/LXRT leverages a MMU from the underlying computing platform to protect the memory space of real-time applications from unauthorized accessing.

As researched in [92], the temporal isolation capability of RTAI Linux can be preserved by introducing a time-triggered scheduling task with the highest priority in the real-time kernel. Meanwhile the RTAI APIs are restricted to rule out potential interference between partitions. The naming services in RTAI are suggested to be extended, so that applications are only enabled to access their own resources.

3.2 ARINC 653

In the avionic domain, the development trend of the computing platforms is evolving from federated to integrated architectures, which are called Integrated Modular Avionics (IMA). Following this trend, research on integrating functions with different criticality levels on the same computing node was carried out and corresponding industry standards such as ARINC 653 were released. The ARINC 653 standard defines the execution environment to host multiple avionic functions on a single computing node using strict temporal and spatial partitioning. From the view point of industrial adoption, ARINC 653 was successfully implemented (e.g., Airbus A380) and achieved its primary goal that aims at cost and weight reduction and enabling modular certification. We will discuss in detail the execution environment defined by ARINC 653 in the following sections.

3.2.1 System Architecture

The architecture of a standard ARINC 653 system is illustrated in Figure 3.1. In this architecture, the user partitions are logically separated from the underlying computing platform via the APplication EXecutive (APEX), which provides the interfaces for the user partitions to access the architectural services.

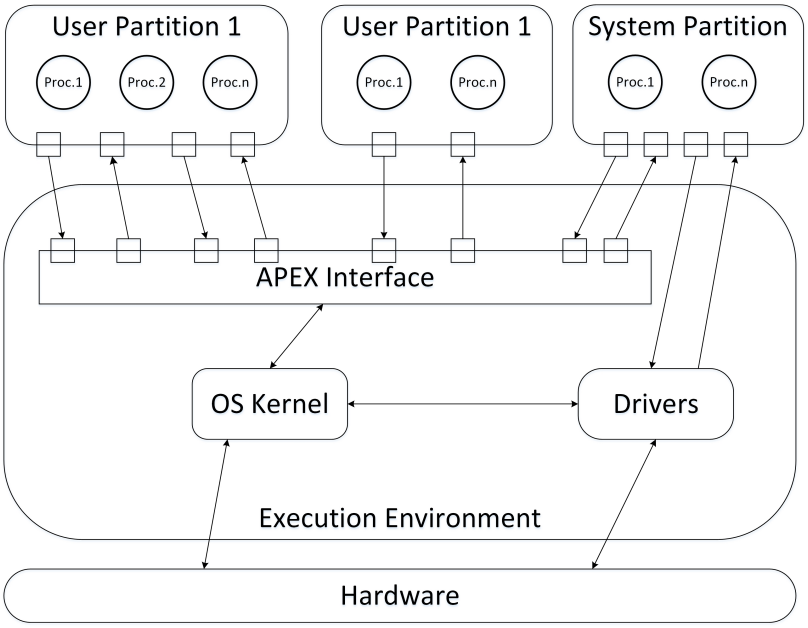

FIGURE 3.1: ARINC 653 System Architecture

Every partition contains multiple processes and a partition (except the system partitions) leverages the services provided by the APEX interface to accomplish its functionalities. The OS kernel provides services (e.g., process management, time service) to the APEX layer, in order to hide the platform specific implementation from the hosted user partitions.

3.2.2 Partitioning

As mentioned in the ARINC 653 standard [23], the central mechanism in the ARINC 653 philosophy is the partitioning concept. The applications residing within an integrated computing platform are partitioned with respect to the execution time and computing resources, which is called temporal and spatial partitioning, correspondingly.

A partition can consist of several processes, which can be executed concurrently to achieve the target system functionalities. In this case, either preemptive or non-preemptive scheduling could be adopted for the process scheduling within a partition.

The partitions defined in ARINC 653 are statically scheduled according to the configuration generated during system integration phase. For the resource allocation and the scheduling perspective, extensive research ([2, 29, 115, 121, 122]) has been done by leveraging Mixed Integer Linear Programming (MILP). For example, the

authors in [29] concentrated on the non-periodic task scheduling problem, while the other mentioned papers aimed at resolving the scheduling problem of periodic tasks. Among the above mentioned related work, only the authors in [2] proposed a solution for the partition allocation and scheduling problem, while considering strict periodicity and system constraints. In [70], Lhachemi, Hugo, et al. developed a model for simultaneous partition allocation and schedule design, so that the distribution of applications over partitions and the scheduling patterns could be automatically adjusted to optimise the system performance. The authors in [108] proposed an online partition rescheduling algorithm to switch between offline verified schedules. As discussed in [38], inspired by the software-based synchronisation used in the space shuttle, Roscoe Ferguso proposed the partition synchronisation mechanism in an ARINC 653 system, where partition switches on the computing nodes are aligned.

Regarding the timing determinism in a temporally partitioned execution environment, as discussed in [130], I/O interrupts can be handled by providing dynamically decreasing time slots in between the fixed time windows within each MTF, in order to guarantee timing determinism at the ARINC 653 MTF level.

In ARINC 653, partitions own the dedicated system resources that are statically allocated before the system starts up. The allocation of the system resources is based on the partition requirements and it is achieved at the partition level. The processes within the same partition have access to the allocated resources of the partition. The concurrent execution of processes within the same partition causes potential conflicts like race conditions during memory access. Therefore, semaphores and mutexes are supposed to avoid conflicts with concurrent or exclusive access to the partition resources. In order to detect race conditions of resource accesses within a partition, the authors in [19] presented the AR653 tool to monitor the synchronised operations and shared resources at the process level. The race conditions can be detected after collecting the monitoring information.

3.2.3 Communication Mechanism

In the ARINC 653 standard, the communication scenarios are categorised into inter-partition and intra-partition communication.

3.2.3.1 Inter-partition Communication

The inter-partition communication includes both the communication between partitions residing on the same computing node and on different ones. The communication is message based, which enables the communicating user partitions to abstract from the underlying infrastructure and reduces system complexity. In another word, the application developers only take care of the exchanged messages to understand the system inter-operation.

As depicted in Figure 3.2, the communicating partitions exchange messages either through queuing ports or through sampling ports, which are connected by the

underlying channels. A channel is the logical link between partitions and statically configured during system integration.

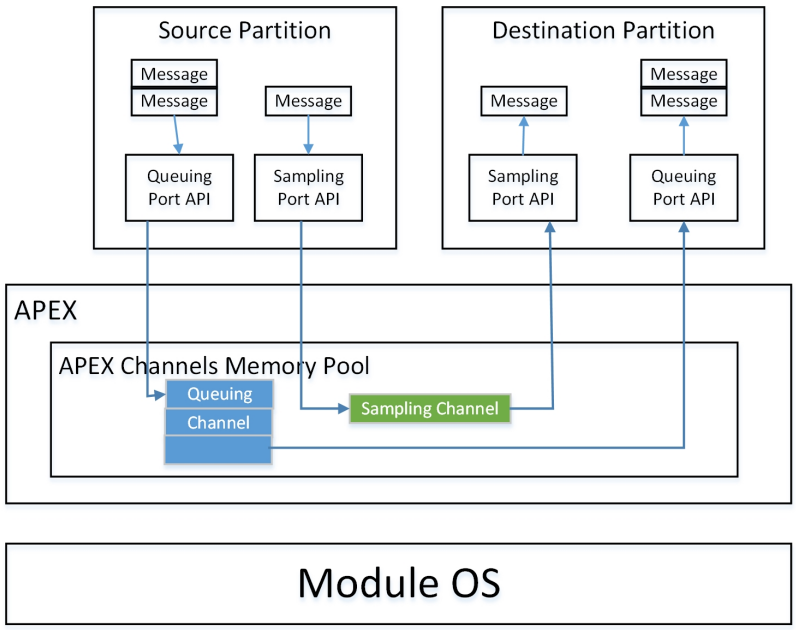

FIGURE 3.2: ARINC 653 Inter-partition Channel

In order to ensure the portability, the implementation of inter-partition communication channels is platform specific. From the view point of a partition, the sent and received messages via the channels contain no explicit location information. The message routing is done based on the offline configuration provided by the system integrator [23]. The authors in [69] proposed a configurable and extensible schema of inter-partition communication defined in ARINC 653. They implemented a network manager as a system partition to abstract diverse underlying network protocols, which also works as a gateway entity to convey messages between different networks.

3.2.3.2 Intra-partition Communication

Regarding the communication between processes within a user partition, buffers and blackboards are defined for the messages that are queued or updated in place, correspondingly. Semaphores and mutexes are the legacy mechanisms for the synchronised access of the buffers and blackboards. A process can also trigger an event to activate the execution of another process.

The intra-partition communication mechanisms are limited to the memory resources assigned to a user partition during system initialization. In another word, as long as there are enough memory resources, the interactions between processes within an application partition could be changed, without affecting the other partitions on the same computing platform.

3.2.4 System Reconfiguration

In the avionic domain, the reconfiguration of an IMA platform was researched in projects like DIANA and SCARLETT. In the SCARLETT project, Bieber et al. [10] proposed a preliminary design to enable an IMA platform to be reconfigurable. The authors revealed in this work the reconfiguration scenarios and proposed the corresponding processes to switch between pre-defined configurations under the identified safety constraints [11] in a centralised way. In the DIANA project, Engel, Christian, et al. in [33] leveraged a byzantine agreement algorithm to propose a configuration selection mechanism, in order to activate one of the pre-qualified configurations during system run-time in a distributed way. Both projects aim to improve the system reliability during run time by dealing with exceptions (e.g. substitution of one computing node by another node).

The authors in [75] proposed a framework to meet the fault tolerance requirements in avionic systems based on the ARINC 653 standard. Safety-critical partitions are duplicated and the communication mechanisms are extended, so that the backup partitions could be reconfigured to replace detected faulty partitions.

3.2.5 Research Gap

To the best of our knowledge, the communication mechanisms within the integrated architectures in the avionic domain are managed by switching between pre-defined configurations in a safety process, without addressing the system reconfiguration during runtime.

3.3 AUTOSAR

The AUTomotive Open System ARchitecture (AUTOSAR) is a development partnership that aims at establishing an open industry standard for the software architecture in the automotive domain [41]. The target software architecture helps to isolate the automotive software from the underlying hardware to enable the independent development. AUTOSAR also enables the integration of legacy automotive software on a single computing node without specifications from the vendors.

In the following sections, we explore the AUTOSAR from the basic architectural design to the detailed communication mechanisms.

3.3.1 System Architecture

The system architecture specified in the AUTOSAR standard is depicted in Figure 3.3. The layered architecture leverages the RunTime Environment (RTE) to enable the independent development of the application software and the basic software [14]. The defined unified AUTOSAR interface ensures the portability and reusability of the application software by abstracting from the infrastructure related components. From the view point of the hardware suppliers, the RTE offers an independent

development environment, so that the hardware specific software components can be evolved independently.

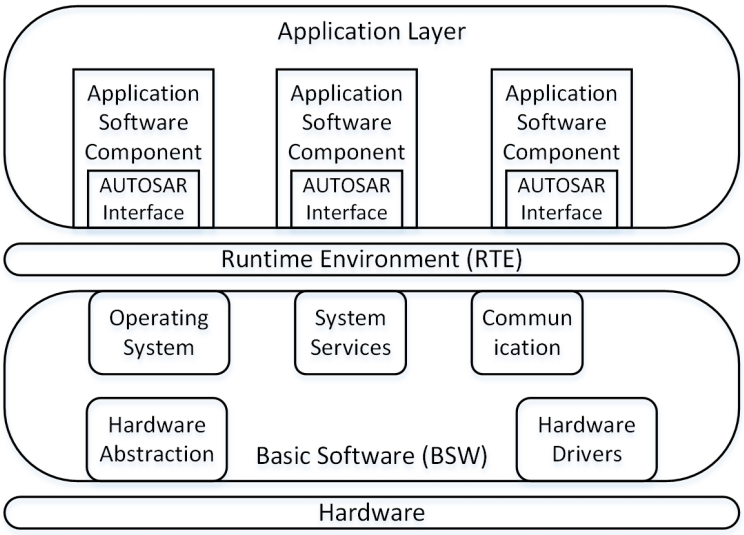

FIGURE 3.3: AUTOSAR System Architecture

According to the system architecture in Figure 3.3, multiple application software components are integrated on a single computing node, which calls for both spatial and temporal partitioning mechanisms to prevent fault propagation between software components. As discussed in [142], AUTOSAR groups different executable tasks into system applications and leverages the OSEK's application protection facilities to avoid spatial interference between system applications. The activation and execution of AUTOSAR executable entities is basically done in an event-triggered manner and optional monitoring mechanisms are applied to detect errors in the configured timing behaviours.

The grouping of executable tasks residing in different software components requires communication mechanisms that will be discussed in the following section.

3.3.2 Communication Mechanisms

In the AUTOSAR standard, the Virtual Functional Bus (VFB) is defined as the conceptual communication mechanism to support message exchange between software components, meanwhile the VFB also provides separation between software components and the underlying infrastructures. The VFB interconnects different software components via well-defined ports. The communication mechanism in AUTOSAR is classified into client-server and sender-receiver paradigms [25, 17].

3.3.2.1 Client/Server Paradigm

The Client/Server (C/S) paradigm provides the possibility for an application software component to invoke services that are provided by other components. The authors in [17] discussed the specific requirements (e.g., self-adaptation, timing monitoring)

of C/S communication in AUTOSAR and proposed a template based programming method to develop a C/S module in AUTOSAR, in order to enable the migration from legacy federated systems to integrated ones based on AUTOSAR.

3.3.2.2 Sender/Receiver Paradigm

In the sender/receiver paradigm, a sender can distribute data frames to multiple receivers, or multiple senders can send messages to the same receiver. Regarding the message semantics in this communication mechanism, state data and queued data are supported to transfer "last updated" and "buffered" messages. Data filtering can be done by leveraging the OS capabilities like OSEK-COM data filters.

According to the analysis in [25], the major characteristics of the discussed communication paradigms in AUTOSAR are compared and listed in TABLE 3.1.

TABLE 3.1: Comparison of Communication Paradigms

	Client/Server	Sender/Receiver
Data Content	operation-oriented	data-oriented
Multiplicity	n clients: 1 server (n>=0)	depends on data semantics
Concurrency and Ordering	non-guaranteed ordering	guaranteed ordering

3.3.3 System Timing

According to the authors in [105], the timing aspect in an AUTOSAR system is not addressed due to the primary objective of easing system integration of system components from different suppliers. The authors also reveal model mismatches that potentially lead to unpredictable timing behaviours of applications. In order to optimise the communication timing within local computing nodes, Long, Rongshen, et al. [74] propose extended mapping rules of AUTOSAR runnable entities in the execution environment. The major idea is to refine the mapping of runnable entities to the underlying OS tasks with the goal to reduce the context switch overhead and consequently the communication delay.

3.3.4 System Reconfiguration

In the AUTOSAR standard, a system is configured statically when the whole setup is integrated. More specifically, the available system resources (e.g., memory, I/O) within an AUTOSAR system are allocated statically. Extensive research has been done with respect to the dynamic reconfiguration within an AUTOSAR system during runtime.

Approaches based on enumerating possible system configurations and switching between different schemes were researched in work like [8] and [43]. These mechanisms are beneficial for system verification and certification. However, they are

essentially based on static system configuration and can lead to unavoidable system overhead (e.g., memory resources, re-certification cost).

In [140], Marc Zeller et al. identified the challenges with respect to extending the AUTOSAR architecture with run-time adaptation capability. Thereafter they proposed the necessary extensions as basic software components or application software components to achieve run-time monitoring and application activation/deactivation, which is treated as a run-time adaptation mechanism in AUTOSAR. In [126], the authors proposed a middleware based on the AUTOSAR architecture with the extension of self-configuration and self-healing capabilities. The essential mechanism of the proposed extension is that the networked computing nodes are enabled to negotiate the distribution of applications in a distributed way according to the proposed metric of Quality of Service (QoS). The proposed self-adaptation is application independent and the timing aspect of the self-configuration and self-healing processes is not addressed in this work.

With respect to timing requirements, the authors in [9] proposed a hierarchical architecture to realise application dependent reconfiguration within AUTOSAR systems. In the proposed mechanism, the application behavior is separated from the reconfiguration behavior and the scheduling period of the application is extended to run the reconfiguration behavior, in order to meet hard real-time requirements. The timing correctness of the proposed model is also verified with timed automata and model checking tools.

The above discussed approaches are mainly concerned about the adaptation at the application layer. In contrast, the authors in [135] introduced different approaches to integrate the Service-Oriented Driver Assistance (SODA) framework [134] into AUTOSAR at the architectural level. In another word, the authors concentrated on extending the basic software components of AUTOSAR in this work. Firstly, the SODA framework can be integrated into the AUTOSAR system as an independent complex driver. Secondly, the SODA framework can replace the existing Universal Measurement and Calibration Protocol (XCP) in AUTOSAR. Last but not least, one can adapt and enhance the transport protocols in AUTOSAR to achieve the required integration. Refer to [135] for more details about these approaches.

3.3.5 Research Gap

As discussed, the AUTOSAR executable entities are basically scheduled in an event-triggered fashion, which results in the lack of determinism of the system timing. The discussed mechanisms addressing system reconfiguration concentrate on the approaches at the application layer in stead of the architectural level.

3.4 Virtual Networking

3.4.1 History of Network Virtualization

As Thomas Anderson et al. discussed in [3], the network development was initially based on an ossified architecture, since legitimate needs based on the network were satisfied by incremental modifications that could not improve the network architecture. Network virtualization was proposed as a potential mechanism to support network architecture development. Thereafter, network virtualization was extensively researched in different domains.

As classified in [20], network virtualization can be grouped into Virtual Private Networks (VPN), programmable networks and overlay networks. The programmable network and the overlay network are conceptually similar to this work. The authors in [16] summarised the programmable network model, in which the message transportation entities (e.g., router, switch) are equipped with extended computation facilities to deploy dynamic network services. Regarding the overlay network, different communicating nodes can be connected via the VLs created on top of the physical network, where diverse network requirements (e.g., QoS, availability, security) could be met. Thomas Anderson et al. [3] argued that the overlay network lacks consideration of the interaction between different virtual networks and it is mostly accomplished at the application layer based on IP networks, therefore an overlay networks does not essentially drive the network architecture innovation.

3.4.2 Network Virtualization Deployment

As Raj Jain and Subharthi Paul discussed in [51], from the perspective of the network composition, the network virtualization includes Network Interface Card (NIC) virtualization, switch virtualization and Local Area Network (LAN) virtualization, which are deployed in data centers, cloud computing and network function virtualization environments.

3.4.2.1 Data Center and Cloud Computing

Date centers and clouds are networked computing infrastructures that are at the premises of the enterprises or remotely located, respectively. The physical computing resources are virtualized in these setups, where different Virtual Machines (VMs) can be integrated on the same computing platform. The VMs belonging to identical tenants/users should be coupled with the same virtual network (e.g., virtual L2/L3 networks), in order to provide location transparency of VMs and inter virtual network segregation [90].

The Internet Engineering Task Force (IETF) working group proposed the Virtual eXtensible Local Area Network (VXLAN) [79] and the Network Virtualization using Generic Routing Encapsulation (NVGRE) [42] to correspondingly address the network virtualization at Ethernet and Internet level. Other related work like the Stateless

Transport Tunneling (STT) [30] protocol and the TRansparent Interconnection of Lots of Links (TRILL) protocol [90] tackle the encapsulation of virtual networks when overlay networks are deployed to achieve the network virtualization.

3.4.2.2 Network Function Virtualization

The Network Function Virtualization (NFV) [18] allows network functions to abstract from proprietary hardware platforms and enables them to run on standard industry hardware, which was proposed in the telecommunication industry intended to reduce the OPerating EXpenses (OPEX) and CAPital EXpenses (CAPEX) and meanwhile increase the deployment agility of new services [88].

According to European Telecommunications Standards Institute (ETSI), the NFV architecture is depicted in Figure 3.4. The building blocks of the whole system are the NFV infrastructure (including physical and virtualized infrastructure), the virtualized network functions and the management & orchestration component [36].

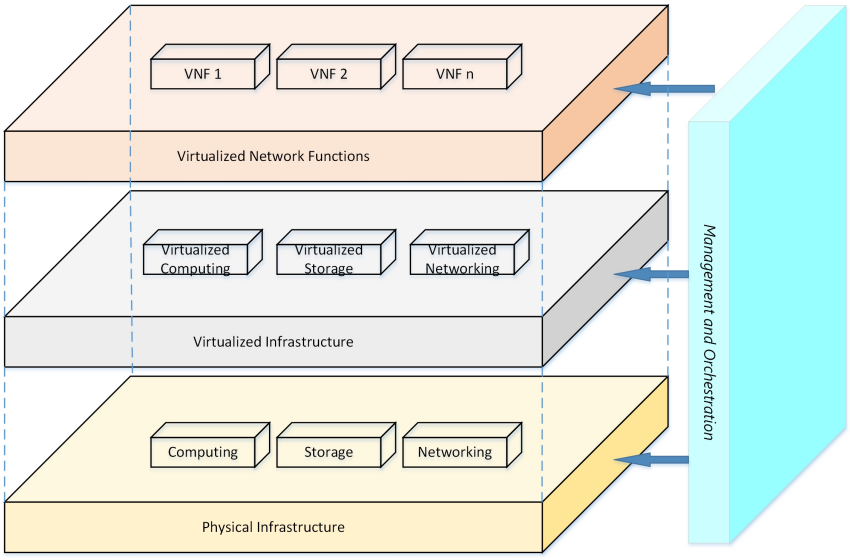

FIGURE 3.4: Network Function Virtualization Architecture

The infrastructure consists of hardware resources and software platforms for the VNFs. The virtualization infrastructure created by software (e.g., hypervisor) enables the underlying hardware to abstract from the hosted VNFs to enable the independent development of network functions and hardware platforms in the telecommunication industry. As discussed in [88], the network functions (e.g., DHCP servers, firewalls) in home networks can be ported from the proprietary devices to virtual execution environments like VMs, and the network functions can be chained to create specific services by leveraging the virtualized network facilities provided by the underlying infrastructure. The NFV MANagement and Orchestration (NFV MANO) component contributes to the configuration and management of both the infrastructure and the VNFs with the help of the included databases that provide necessary system properties. The ETSI proposed the MANO framework in [36] also considered the

inter-MANO cooperation and the capability to enable the coordination between MANO components and the existing network management systems, in order to support the evolution of NFV in the telecommunication industry.

3.4.3 Network Virtualization in an Integrated Architecture

In the research field of virtual networking based on an integrated architecture, the authors in [95] proposed the virtual network paradigm for both safety-critical and non safety-critical applications based on a physical time-triggered network. The proposed virtual networks leverage temporal and spatial partitioning between applications to ensure encapsulation of different communicating entities. The proposed architecture has a layered structure to enable the computing platform at the computing node level to abstract from the functions of an application. The proposed virtual network was implemented in a DECOS node computer [94], where the bandwidth resources of the underlying time-triggered network are subdivided into time slots for applications. Control transformation functionalities are necessary when the event-triggered messages are mapped to the time-triggered network. The controlled message exchange between virtual networks is achieved by the proposed virtual gateway [91]. The proposed virtual networks contributes to both infrastructure cost saving and reliability improvement by means of wiring and connection reduction.

3.4.4 Virtualized Network Management

In a virtualized networking environment, management should be addressed to improve the effectiveness and reliability of virtual networks [35]. In this section, we analyse the state-of-the-art network management protocols Simple Network Management Protocol (SNMP) and NETwork Configuration Protocol (NETCONF).

3.4.4.1 SNMP

The IETF defined the SNMP in the RFC 1213 protocol [84] for the management of networked entities. The managed networked entities leverage the Management Information Base (MIB) to record the management related information, which can be retrieved by the SNMP server running on the managed entities to reply to queries that are issued by the SNMP clients residing on the managing entities.

Kiran Voderhobli discussed in [132] the adoption of SNMP as the communication mechanism to collect management-related system information in a virtualized execution environment. SNMP was used in [131] to communicate between analytical entities and VMs to study the network traffic characteristics in virtualized networks. Apart from leveraging SNMP as the communication mechanism, Ya-Shiang Peng and Yen-Cheng Chen [101] customised the MIB for VMs and proposed a hierarchical architecture to monitor both virtual and physical entities. The authors in [46] leveraged the SNMP MIB to control the operations in VMs, which demonstrated the control capability of SNMP.

Asai et al. extended in [127] the MIB to be a Virtual Machine Management MIB (VMM-MIB), which is unsuitable for controlling virtual routers in a virtualized environment due to the read-only property of VMM-MIB [111]. Daitix et al. [26] extended the Virtual Router (VR-MIB) [116] to define a virtual network management interface that is capable to create and remove virtual routers, and even dynamically bind virtual network interfaces to a physical network interface.

3.4.4.2 NETCONF

Since the above discussed SNMP is originally not optimal for configuration management, the IETF released the NETCONF protocol that addresses the configuration management of networked entities. Along with NETCONF comes the Yet Another Next Generation (YANG) data modelling language to abstract from the properties of networked entities in a unified way. According to the RFC 6241 protocol [34], the layered NETCONF protocol is summarized in TABLE 3.2.

TABLE 3.2: NETCONF Protocol Layers

Layer	Example
Content	Configuration/State/Notification Data
Operations	<get-config>/<edit-config>
Messages	<rpc>, <rpc-reply>, <notification>
Secure Transport	SSH, TLS, BEEP/TLS, SOAP/HTTP/TLS, ...

As shown in TABLE 3.2, the NETCONF protocol consists of 4 layers. The content layer contains the service- and device-related parameters that are modelled in data models (e.g., YANG [12]). The operation layer provides an abstraction in the interactions between management clients and servers based on Remote Procedure Calls (RPCs) and the encoded parameters are in XML-form. The message layer under the operation layer defines an independent framing mechanism to encode the RPCs and notifications, so that the message layer can be deployed on any secure transport mechanisms that meets the NETCONF specific requirements (e.g., connection-oriented, SSH support). In another word, security is supported and implemented at the transport layer in the NETCONF protocol.

After the IETF released the NETCONF protocol, there existed several NETCONF implementations. To name a few examples, YENCA [141] was the first NETCONF capable implementation, which combined an agent and a manager that were implemented in the C language and Java, correspondingly. Thereafter, YENCA was improved by the LORIA-INRIA laboratories to include a python implemented agent and a web-based manager, which were connected via SSH. The open-source implementation Netopeer [71] implemented the RFC 4741 (predecessor of RFC 6241), in which the manager was both web- and command-line-based. Another notable implementation is called Yuma. It was RFC 6241 compliant and the management

servers on the managed entities can be extended in a plugin fashion, which freed the servers from recompiling during runtime. The lately implemented libnetconf library [63] targeted the RFC 6241 compliance in the GNU/Linux environment. The major advancement of libnetconf is that the defined NETCONF capabilities are provided as functions. More specifically, the developer can leverage the NETCONF functions from the libnetconf library to build up customised NETCONF servers and clients. The Yuma and libnetconf implementations are tested with respect to the interoperability of the NETCONF protocol in [6]. As discussed in [113], for the perspective of industrial deployment, major manufactures (e.g., Ericsson, Cisco) already shipped their devices with NETCONF software, which is enabled by the NETCONF development kits provided by Netconf Central and SNMP Reasearch.

3.4.4.3 Comparison between NETCONF and SNMP

In this section, we summarise the major differences between SNMP and NETCONF in TABLE 3.3 according to the discussion in [139].

TABLE 3.3: Comparison between NETCONF and SNMP

	NETCONF	SNMP
Approach	Document-oriented (XML)	Variable-oriented
Protocol layers	Content, operation, message, transport	Application layer
Management capabilities	Configuration management	Fault/performance management
Architecture	Manager-agent	Manager-agent
Data structure	YANG model	MIB
Transaction capability	Entire configuration	Single item

As summarised above, the NETCONF protocol defines the communication approach to be document-oriented, which means that the configuration files of the managed entities are exchanged between management servers and clients in well-structured documents. In contrast, a SNMP manager communicates with managed agents via simple variables without specific structures that may lead to the lack of an overall view of the device configuration. The essential differences in the communication approach are resulting from the defined target management capabilities. The NETCONF protocol aims to cover the configuration management of networked devices, which is not optimally addressed by SNMP. The management-related items on a SNMP agent are organised in a MIB that is implemented in a data-oriented language derived from Abstract Syntax Notion 1 (ASN.1) [139]. This makes the development based on MIB a complex and time-consuming task, while the defined YANG data modelling language for the NETCONF protocol is human-readable and well-structured to represent the device configuration. As experimentally shown in [139],

the NETCONF protocol outperforms the SNMP for the perspectives of efficiency, effectiveness and security.

3.4.5 Research Gap

To the best of our knowledge, the research on network virtualization address either dynamic virtual networking without hard real-time support, or static virtual networking with hard real-time support. In this dissertation, we address the virtualized networking that supports both hard real-time and dynamic system configuration.

3.5 Software-Defined Networking

According to the definition provided by the Open Networking Foundation (ONF), SDN means the physical separation of the network control plane from the forwarding plane, where a control plane controls several devices. Except for the separation of the data plane and control plane, the control plane is defined to be programmable. In this section, we discuss the SDN paradigm in detail.

3.5.1 SDN Architecture

The reference SDN architecture proposed by ONF is shown in Figure 3.5. The overall architecture consists of three layers, namely the infrastructure layer, the control layer and the application layer. The physical networking devices build up the infrastructure layer, where the devices are capable to communicate status data to the controllers and carry out the instructions received from the controllers. The control layer represents the service access point (e.g., Application Programming Interfaces) for the applications at the application layer. In a large-scale networking scenario, the controllers should be able to synchronise infrastructural information with each other and execute in a coordinated fashion. The SDN applications are obligated to deploy the network management tasks by leveraging the programmability of the control layer.

In the following sub-sections, we will discuss the related work in these SDN layers.

3.5.1.1 Infrastructure Layer

The SDN infrastructure mainly consists of switching devices and the interconnection mechanisms between the switching devices.

As discussed in [137], a SDN-enabled switching device can be logically decomposed into two parts: a data plane and a control plane. The logical data plane is normally mapped to the processor that is configured to switch packets according to the switching rules stored in the local memory, which constitutes the physical mapping entities of the logical control plane. In comparison to conventional switching devices that run complex routing protocols, a SDN-enabled switching device

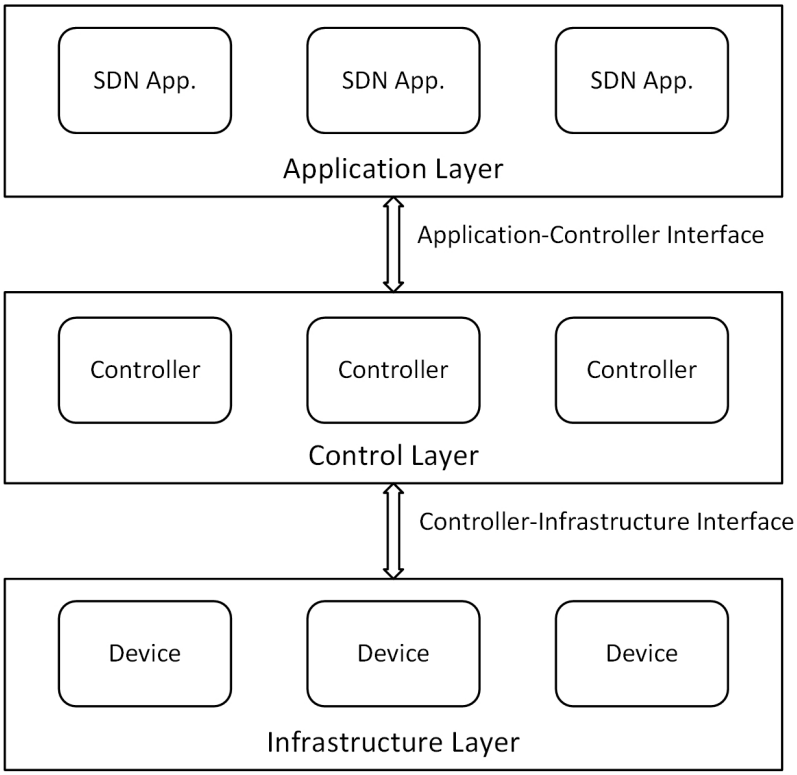

FIGURE 3.5: SDN Architecture

is simpler in terms of computing capacities. Due to the essential changes of the architectural paradigm, the conventional switching devices should be redesigned to be SDN-enabled.

As discussed before, the control plane is physically mapped to the local memory, therefore the key challenge is the way to use the local memory in specific scenarios. For example, in a dependable system (e.g., X-by-wire), the network scale is predictable, which means that the amount of message switching rules stored in the local memory of a device can be known during system design. In such dependable systems, one of the prerequisites is a deterministic networking facility for safety-critical data flows. That means the memory management techniques in the switching devices should be developed in a timely deterministic fashion. Traditional memory management services in the network switching entities leverage aggregation [57] to reduce the amount of route entries in memory. Other researched mechanism like Adaptive Least Frequently Evicted (ALFE) [100] aim at preserving cache entries for data flows of specific priorities, in order to guarantee the cache hit rate of these flows. To the best of our knowledge, there exist no memory management systems that address timely deterministic data flows in SDN-enabled switching devices.

The data plane in a SDN switching device ought to forward messages according to the rules stored in the control plane based on flexible message identification and classification mechanisms. Hardware accelerated classification is researched based on both commodity hardware [119, 120] and customised hardware [77], which is proven to reduce the network packet processing delay and consequently increases the

network throughput. However, for the safety critical data communication in real-time systems, the bounded latency with low jitter between communication entities should be addressed [97]. Guohan Lu et al. proposed in [76] an offloading mechanism for an Application-Specific Integrated Circuit (ASIC) to avoid exhausting memory and to prevent interference between specific data flows. As for the message switching for safety-critical data flows, similar mechanisms can be developed to meet the real-time networking requirements.

Apart from the switching devices, the other important component of the infrastructure layer is the transmission media. The currently existing transmission media includes wired, wireless and optical media. Wired packet switching is the de facto mechanism in the above discussion. In addition, wireless and optical media will be discussed as follows.

- As Manu Bansal et al. discussed in [7], modern wireless systems normally have common processing blocks at the physical layer, and different configurations with various characteristics. Based on that, OpenRadio [7] and Dyson [89] were proposed to decouple wireless protocols from hardware and extend hardware to provide statistic information, correspondingly. This work attempted to evolve the wireless transmission environment towards Software-Defined Radio (SDR) [129] that can be integrated by SDN systems.

- Regarding optical fibers, the Generalized Multi-Protocol Label Switching (GMPLS) [81] defined the control plane for optical networks, which enables unified management of switching domains (e.g., wired, optical). To date, there are various attempts proposing a unified control plane for mixed transmission environments. Extensions of optical switches to support switching rules of higher layers potentially lead to a unified managed data plane while causing the economic difficulty of upgrading existing switching infrastructures [27, 28]. Virtualized switching [73] was proposed to establish proxy messages between optical interfaces and software switch interfaces at the cost of increased communication latency.

3.5.1.2 Control Layer

As Wenfeng Xia et al. proposed in [137], a controller is logically composed of a language translator, a rules updating process, a status collection process and a status synchronization process. The detailed functionalities of the building blocks and their interactions will be discussed as follows.

- Language translator: The language translator translates the commands from the application layer into packet switching rules that are finally carried out by the infrastructure layer. Existing communication protocol like the Command Line Interface (CLI) for Cisco Internetwork Operating System (IOS) can be adopted as the configuration language. However, the common configuration

languages provide basic low level hardware abstractions that result in error-prone development processes. Therefore, a high level language is a better option for the language translator. To date, there are well-developed high level language like the Flow-based Management Language (FML) [47], the Frenetic language [40] and the Nettle language [133], which can be potential options for the language translator.

- Rule updating process: The rule updating process generates the message switching rules based on the input from the language translator and communicates the generated rules to the infrastructure layer. During the process of rule updating, consistent rules should be guaranteed to provide deterministic network behaviours. Different levels of consistency are proposed for rule updating. More specifically, the strict consistency and eventual consistency are defined. When strict consistency is required, a packet or a data flow in the switching devices is forwarded according to a single rule set (i.e., either an old one or an updated one). In contrast, packets of the same data flow can be switched according to both original rules and updated ones to keep them eventually consistent. Mark Reitblatt et al. [106] and Rick McGeer [85] provided the proof-of-concept implementation for the discussed consistency constraints.

- Status collection process: The status collection process is responsible for querying network status information from the switching devices, thereafter the collected information is provided to the network applications. The OpenTM introduced in [124] targets at estimating a network Traffic Matrix (TM) in an intelligent fashion, in order to avoid increasing load on switching devices. Except for the network traffic statistics (e.g., transmission duration, packet number, etc.), the data flows are analyzed with respect to the message volume. Data flows are classified into "elephant" and "mice" flows, where an "elephant" flow is more important to be identified for collecting network status information. Similar to the mixed-criticality networking environment, the status information of critical data flows should have the highest priority to be collected and provided to the network applications for decision making.

- Status synchronization process: The status synchronization process is defined to synchronise network status information between controllers, which ensures the unique global view of the involved network. The authors proposed in [123] the HyperFlow that leverages the publish/subscribe mechanism to synchronise collected network status information between controllers. A similar attempt done by H Yin et al. [138] used communication among controllers to both share network status information and coordinate behaviours of controllers.

As discussed above, a controller deals with two data streams. One is the deployment of policies and rules from network applications to the switching devices; the other one is the collection and transmission of network status information from

switching devices to network applications, meanwhile guaranteeing the synchronised network views in the control plane.

3.5.1.3 Application Layer

At the application layer, the network applications are able to analyze the latest network status information to manage the underlying infrastructures through the control layer. To date, there are network applications ranging from switching control to network security. Selected related work is discussed in the following sections.

Message routing and switching are the main functionalities of a network facility. The SDN paradigm enables the message switching process to be centralised in a dynamically controlled fashion. The initial effort addressing load balancing with message forwarding rules in SDN was done in [136], where the authors proposed to compute a message path dynamically based on the network traffic status. In [59], Koerner et al. addressed independent traffic specific balancing strategies in a SDN system, which essentially matches the requirements of mixed-criticality systems, where data flows of different criticalities need to be treated specifically. Regarding the network QoS, application-dedicated requirements can be addressed with resources reservations via a SDN controller [53, 37], which demonstrates the cross-layer cooperation of a SDN system.

Network virtualization technologies (e.g., VLAN, tunnel) are widely used in common practice to facilitate the coexistence of different networks on shared physical infrastructures. However, these conventional virtualization technologies rely on complex distributed infrastructural configuration that causes extra network overhead for the switching devices to agree on the network view and converge to target configurations. The authors proposed in [128] the libNetVirt to function as a centralised network controller for network virtualization management, which tackles the above discussed disadvantage. Another example for network resource slicing in a SDN-enabled system is the FlowVisor [114], which selectively transmits control messages between controllers and switching entities, so that each controller is able to manage its own network entities.

In the cloud computing field, virtual switching is one of the key aspects of cloud computing, and the Open vSwitch is a famous SDN-capable OpenFlow switch addressing the networking problem between applications. In [87], Mian et al. validated that Open vSwitch is secure at the cost of increased round trip times in comparison to non-virtualized execution environments for cloud computing. He and Liang [44] evaluated the security, QoS and network performance of Open vSwitch, which showed positive results. In order to guarantee QoS, Akella and Xiong [1] proposed to allocate bandwidth for satisfying QoS requirements of the priority cloud users based on Open vSwitch. In this research, the proposed QoS approach uses a bandwidth and path length based metric and queuing techniques for different users. Other existing software switches (e.g.,VALE switch [107], mSwitch [48]) are mainly concerned about throughput and packet rates.

In this dissertation, the mainly concerned aspects of the application layer are the configurable routing, the switching capability, and the virtualized networking technologies on a single physical infrastructure or networked environment, which are closely related to the work that is presented in the following chapters.

3.5.2 Research Gap

For the deterministic data communication in hard real-time systems, the bounded latency with low jitter between communication entities is an important requirement to be addressed, which is out of the scope of the existing SDN research.

Chapter 4

SDN-based Execution Environment for Integrated Architecture

This chapter discusses the functional distribution framework concept that builds up the execution environment for integrated systems and the proof-of-concept implementation in this dissertation.

4.1 Requirements

In this section, the requirements for the execution environment of mixed-criticality applications based on an integrated architecture are discussed.

4.1.1 Technical Requirements

The technical requirements consist mainly of the functional characteristics that are provided by the framework, so that the hosted applications are able to fulfil the technical specifications defined by the developers.

4.1.1.1 Systematic Adaptation

Existing execution environments (e.g., TCMS in the railway domain) are based on a federated architecture which is different to an integrated architecture. Every computing node in a federated system hosts one function, therefore, the execution environment on a computing node is function-specific. In integrated systems, one computing node can host multiple applications due to the increasing computing capabilities of modern platforms. Another prerequisite is that the execution environment and the underlying platform provide the capability to spatially and temporally separate different functions, so that a non-critical application cannot interfere with a critical application.

Today the execution environments for integrated architectures are typically static. For example, in the avionic domain, the configuration of the execution environment is done by the system integrator during design time. Also the configurations of the communication mechanisms are statically done during system development. Reconfiguration of the execution environment is often reduced to switching between

predefined configuration modes, which is mainly due to safety concerns, because pre-configured scheduling schemes can be statically analyzed and proven to be safe.

In order to support also more dynamic application domains with a changing system structure and adaptive behaviours, the integrated computing node should provide a reconfigurable execution environment for mixed-criticality applications, the systematic adaptation during runtime should be addressed under the precondition that further system requirements are not violated.

4.1.1.2 Temporal and Spatial Partitioning

In the scope of this dissertation, applications can have different safety criticality levels (e.g., SIL1 to SIL4 according to IEC 61508) when they are integrated on a computing node. The execution environment based on an integrated architecture provides shared computing resources for different applications. The existing execution environments based on a federated architecture achieve partitioning due to the physical separation of nodes in federated systems. In order to prevent interference between applications with different criticality levels and guarantee sufficient hardware resources for applications, the target execution environment should ensure spatial partitioning for the hosted applications. One possible implementation of spatial partitioning is hardware MMU based.

For hard real time control applications missing the deadline can cause serious consequences. For example, when a vehicle brake control function misses its deadline and fails to stimulate the actor, the vehicle does not brake and this can result in passenger injury or even death. In order to guarantee the real-time properties of applications on an integrated architecture, the execution environment should ensure strict system-level time partitioning. Similar to ARINC 653, partitions should be scheduled on a cyclic basis, which enables the execution environment to allocate computing resources to each function for meeting the following timing requirements.

4.1.1.3 Timing Determinism

The framework aims at providing an infrastructure for real-time applications, therefore timing determinism is a prerequisite.

At the application level, a hard real-time application can be scheduled to execute in a time-triggered fashion to meet the required timing constraints. In this sense, the target framework needs to ensure the timely dispatching and termination of applications.

At the data communication level, data flows between critical applications need to meet their timing requirements specified by the application developers. To date, data flows in a federated system are assigned to dedicated physical channels that naturally guarantee the exclusive access to communication facilities. In an integrated system, different data flows are multiplexed on one physical communication channel, where messages are dispatched in a time-triggered way or under rate constraints.

Regarding the local communication within one computing node, the target framework should provide temporal and spatial isolation at the data flow level. In addition to the partitioning mechanism, the data communication should be dynamically adaptable according to the discussed requirement in section 4.1.1.1.

4.1.2 Non-technical Requirements

Except from the technical requirements, there are non-technical requirements (e.g., safety, security) that need to be addressed in the target framework.

4.1.2.1 Safety

In the human involved field, safety is one of the most important aspects to be considered. From the perspective of the framework, the safety requirement applies to the software components of the execution environment. From the conceptual design process to the final deployment, the software components should be developed according to target safety integrity levels (e.g. from SIL1 to SIL4 according to EN ISO/IEC 61508). As discussed in Section 4.1.1.1, static configuration and switching between offline defined configurations for safety critical systems can be proven to be safe. However, the required systematic adaptation during runtime raises further safety requirement in this work, which will be addressed in the later sections.

4.1.2.2 Security

In one aspect, the safety-related framework is sensitive against systematic faults that are introduced during the system development process as well as random operational faults. In another aspect, such a framework is also sensitive to security threats (e.g., intentional human attack). The ISA/IEC 62443 standard series addresses the security of industrial systems and provides guidance from defining security levels to countermeasures for different security levels. Security is not the major requirement to be tackled in this dissertation, but the target framework is designed to handle specific security issues and be extendable for further security matters.

4.2 Functional Distribution Framework Concept

In the avionic and automotive industry, research on integrating applications with different criticality levels on an integrated platform have lead to domain specific standards like ARINC 653 and AUTOSAR. In this section, the fundamental mechanisms of these standards are extended to define a framework that is capable of hosting dynamically changing applications, which is an open research problem. Since the discussed applications consist of distributed functions, we define the synonym "Functional Distribution Framework" for the target framework.

4.2.1 Logical Structure

The logical structure of the framework is depicted in Figure 4.1.

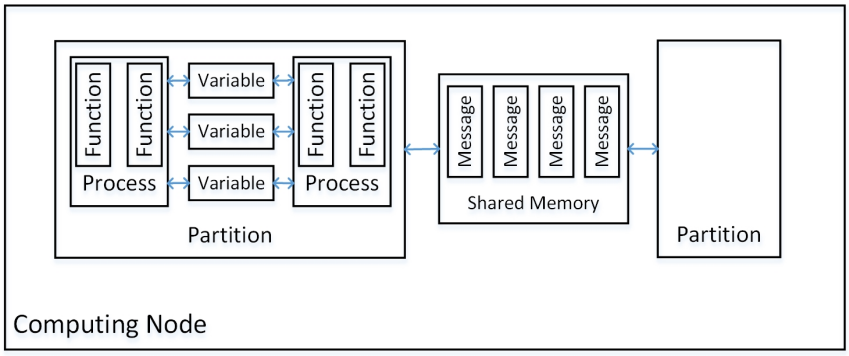

FIGURE 4.1: Logical Structure of Functional Distribution Framework

The resources of a computing node are partitioned into isolated sets that are assigned to applications. In another word, a partition is allocated dedicated hardware resources and a computing node is able to host different partitions simultaneously. Different processes consisting of functions in a partition are scheduled to share the hardware resources of the partition. The communication between processes within a partition is done by variables, and the messages stored in shared memory are leveraged to convey exchanged information between partitions, which host different applications. The above mentioned logical elements will be discussed in detail in the following sections.

4.2.2 Conceptual Building Blocks

The proposed framework is built up with the conceptual elements, which define the basic interactive components for the framework.

4.2.2.1 Variable and Message

A variable represents a data structure that encloses necessary information to enable the communication between processes of an application. The variables accessed by multiple processes are protected against concurrent writing operations, in order to assure a consistent system view for different processes. Similar to the buffers and blackboards defined in ARINC 653, the semaphore and mutex are the common mechanisms for controlled concurrent data accessing at the process level. The buffers and blackboards are defined to convey messages of different semantics (i.e., queued and non-queued messages), and a variable is also treated with respect to the defined semantics.

As shown in Figure 4.1, the partitions within one computing node communicate with each other via the defined shared memory, where the well-defined messages are stored. A message is essentially an unit of variables, which are configured to be exchanged between partitions. According to the specific system configuration,

variables are merged into a single message to reduce the load of information exchange between partitions. Since the shared memory areas are accessed by various partitions, there should be protection mechanisms to rule out concurrent operations that can result in an inconsistent system status.

4.2.2.2 Function and Process

A function is the basic scheduled unit in the system, which can be part of an application. An example functionality of a function is to process dedicated inputs and generate the outputs within a specified deadline. The resource access of a function is protected in the scope of the process. The functions within a process can leverage either implicit or explicit synchronization mechanisms to achieve the required scheduled execution.

A process consists of multiple functions to implement specific functionality. Processes can be concurrently executed either using a static schedule or in dynamic priority-based fashion, which is configured based on system wide functional and non-functional requirements. A process accessing mutually-exclusive resources is protected from being preempted, in order to ensure the safe execution of a critical process. Except from the executable units (i.e., functions), a process is comprised of data, priority, timing attributes and necessary counters/pointers. At the process level, the framework provides management capabilities to be leveraged by the processes, which will be discussed in the later section.

4.2.2.3 Partition

The partition indicates the combination of a resource partition and a time partition. A resource partition encloses the system resources (e.g., memory) that are shared by the configured processes, similarly, a time partition includes the processes, which can be activated in the same time slot. The time partitions are exclusively scheduled in a cyclic MTF, and the resource partitions are assigned to the scheduled time partitions. This schema is illustrated by an example in Figure 4.2.

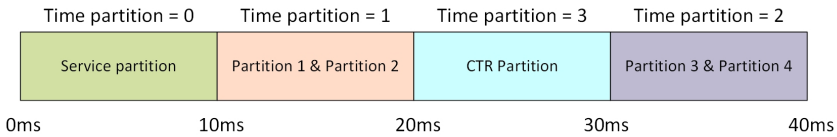

FIGURE 4.2: Example of Partition

In this example, different time partitions (e.g., 0,1,2 and 3) are scheduled exclusively, while the resource partitions (e.g., 3 and 4) can be scheduled simultaneously in the same time partition.

The partitions have direct access to the computing platform via well-defined services that will be discussed. In many COTS products (e.g., PikeOS, VxWorks,

etc.), a MMU is used to implement the spatial isolation of memory resources at the hardware level.

Besides the resource partition and the time partition, a partition can be classified into service partition or user partition. A service partition contains the processes that provide architectural services (e.g., I/O driver), while a user partition consists of processes implementing user-defined logic to realise user-specified functionalities.

4.2.2.4 Shared Memory

In the proposed logical structure, a shared memory is designed to store the communicated messages between partitions. In another word, the shared memory is accessed at the partition level. Due to the concurrent accessing, the shared memory region is protected to maintain the system consistency. Similar to the partition, the memory is statically allocated during system design, which creates a maximum limitation of the dynamic memory allocation for the messages from the shared memory.

At the system level, there is an individually designed management service for the shared memory accessing. The user partitions at the application level access the shared memory in a transparent synchronized way through the interfaces provided by the framework.

4.2.3 Execution Environment Services

The functional distribution framework is aimed to provide an execution environment for the applications, where the provided services of the framework are proposed for the mixed-criticality applications integrated on a single computing node. In this section, we firstly define the minimum service set for the target execution environment, thereafter discussing the details of the defined services, from the view point of both structural and operational behaviours.

4.2.3.1 Basic architectural services

In this work, the target framework aims at providing basic architectural services that fulfil the requirements discussed in section 4.1. These basic architectural services are leveraged by application designers to develop customised applications with specific critical levels.

- Timely deterministic communication service. This service guarantees the deterministic and timely transport of exchanged information between different communicating entities, which is the prerequisite of the execution environment for distributed critical functions.
- Reconfiguration service. The framework provides the capability to adapt to changing system structure during runtime while preserving the determinism in the execution environment.

- Error containment service. Another important service provided by the execution environment is the error containment capability. The execution environment provides FCRs for different applications to achieve error containment between applications, meanwhile fault tolerance based on redundancy is leveraged to achieve error masking within a single application.

- Synchronization service. Distributed critical applications require coordinated execution of processes, which results in the need for a synchronization service of the execution environment.

4.2.3.2 Software Components

In this section, the designed software components (including framework manager, configuration manager, function manager, variable manager, message manager, network manager and synchronization manager) are discussed in detail.

The first defined component is the framework management entity that exposes all the APIs provided for the applications. The detailed functionalities exposed by the framework manager are illustrated below.

- Configuration. The framework manager receives the user configuration and manages the configuration in a centralized fashion. The user configuration is managed in the defined configuration manager.

- Initialization. The framework manager initializes the configured software components according to the initial configuration provided by the system integrator.

- Registration. User specified entities are initialized by the framework manager and registered in the corresponding managers (e.g., variable manager, function manager).

- Execution. The execution of hosted applications proceeds in a timely exclusive way, so that processes belonging to different applications are free of competition with respect to computing resources.

- Get variable. From the view point of the framework manager, the managed variables are specific memory segments. The applications are allowed to get their assigned variables through this interface.

As mentioned above, the framework manager initializes and calls the configuration manager to load and organise the configuration items provided by the user applications. In another word, the configuration manager manages the configuration information for an instantiation of the framework. The major interfaced component of the configuration manager is the allocated memory for storing the configuration

information. Since the allocated memory is implicitly shared by various user applications, it should be protected against concurrent accesses that can result in inconsistent system states.

As mentioned in section 4.2.2.2, the functions are the basic scheduled entities in a system, therefore, the function manager is a mandatory component in the framework. The function manager is responsible for managing the registered functions and the accompanying attributes. To be more specific, all the functions are executed after successful conditions (e.g., correct input, correct timing, etc) check, and the function outputs are managed after successful execution.

The variable manager is responsible for internal variable access (i.e., read, write operations). As discussed in section 4.2.3.1, fault tolerance based on redundancy is a measure to address error containment, where a safety-critical variable can be replicated and stored in different variable stores, so that erroneous variables can be identified and corrected. For concurrent access protection, the system supports mutexes and semaphores in this manager.

Similar to the variable manager, the message manager provides access services for the defined messages. Except for these services, the message manager also manages the composing and decomposing of the messages that are built up with various variables. The attributes (e.g., timestamp) belonging to messages are in the management spectrum of this managing entity. Since messages are stored in a shared memory, access synchronization mechanisms like mutexes are deployed to achieve controlled concurrent access.

When the communication happens between computing nodes, the network manager is involved to manage the message exchange process through physical networks. From the view point of the network manager, all the transported messages are treated as datagrams without knowledge of message content.

The synchronization between computing nodes is managed in the framework, where the local time is synchronized to a global time with a defined precision. If the underlying hardware platforms provide the capability to synchronize each other, the synchronization manager is not a mandatory component in the framework.

4.2.3.3 Behavioural View

After a system starts up, the framework processes configure the software components. Thereafter, the software components are initialized based on the prepared configuration. In the end, the system is executed when the framework finishes initializing the software components. In this section, the interoperation between the defined software components are discussed.

The configuration procedure is depicted in Figure 4.3. The framework manager and the configuration manager are mainly involved in this procedure, where the framework manager triggers the loading process of the configuration manager, and the configuration items are loaded into the allocated memory region by the configuration manager. In case of configuration items for safety-critical components, the

configuration manager duplicates the items and stores them in mirrored memory regions.

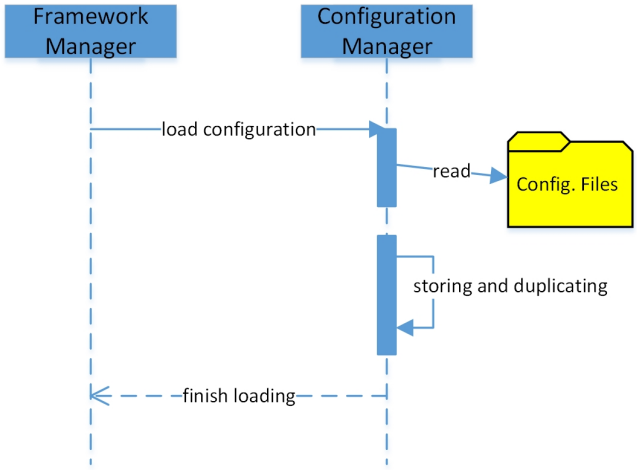

FIGURE 4.3: Configuration Procedure

Regarding the initialization procedure, the main initialized components are the variables, messages and the functions realizing the framework services.

- Variable/message initialization. As shown in Figure 4.4 and Figure 4.5, the framework triggers the variable/message manager to initialize a variable/message entity according to the retrieved configuration from the configuration manager. The variable/message manager opens and maps the shared memory for the variable/message before initializing them. The initialized variables and messages are protected with mutexes against undesired concurrent access.

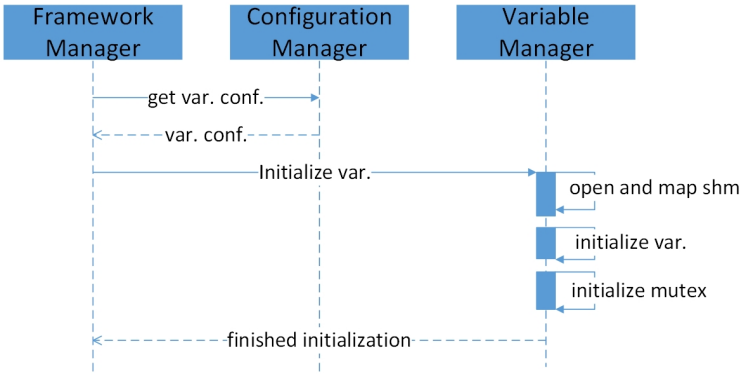

FIGURE 4.4: Variable Initialization Procedure

- Function initialization. The function initialization procedure shown in Figure 4.6 is designed for all functions. The initialization of a function entity is triggered by the framework manager and conducted by the specific component managers (e.g., network manager). The initialized functions are generally managed by the function manager, which records all function-related information.

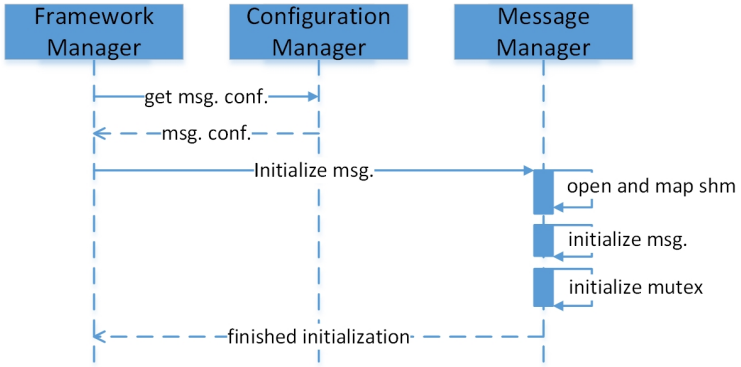

FIGURE 4.5: Message Initialization Procedure

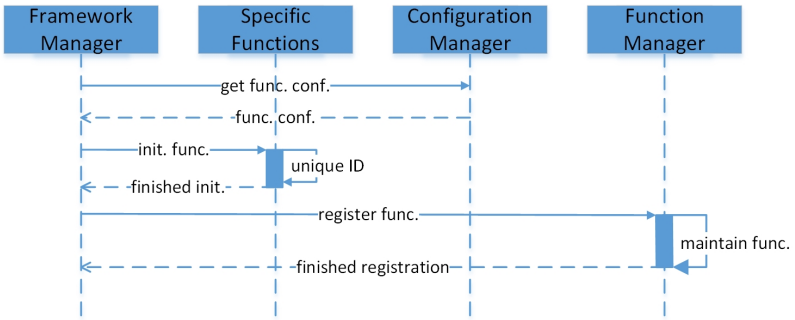

FIGURE 4.6: Function Initialization Procedure

After the framework finishes initializing the configured components, the execution procedure is activated. The general execution procedure is shown in Figure 4.7. A user application triggers its execution using the framework manager, which activates the function manager to dispatch the corresponding function. The function specific components (e.g., variables, synchronization mechanisms) are managed during the procedure.

The execution of specific user applications is derived from the general process. The differences between various applications depend on the functional logic and the timing requirements of the user applications.

4.3 Proof-of-concept Implementation

In this section, the proof-of-concept implementation of the above proposed framework is discussed in detail. At the end of this section, the experimental results are shown and discussed.

4.3.1 Design Instantiation Based on PikeOS

In this dissertation, the proof-of-concept implementation of the framework is based on PikeOS, which provides the real-time execution environment for the framework.

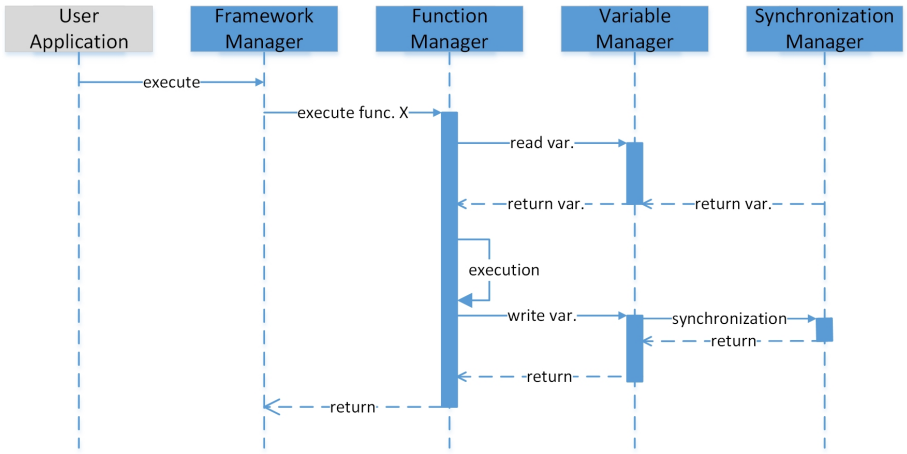

FIGURE 4.7: Function Execution Procedure

According to the architectural design of PikeOS, the defined framework can be implemented in different ways. One possibility is to implement it as a middleware between the hosted applications and the PikeOS System SoftWare (PSSW). Alternatively, the framework could also be implemented as an application within a resource partition. In this proof-of-concept implementation, the framework is designed to be a middleware providing the defined services.

4.3.1.1 Framework Manager

As depicted in Figure 4.8, the framework manager is designed as a C++ class that provides the defined interfaces and manages the necessary system information that is stored in a statically allocated system memory. The service functions are function objects inherited from the basic function class. This inheritance mechanism also applies to the application processes, i.e., the application processes are registered as user functions in the framework. The execution of both the user and the services functions is managed by the framework manager in line with the pre-defined configuration.

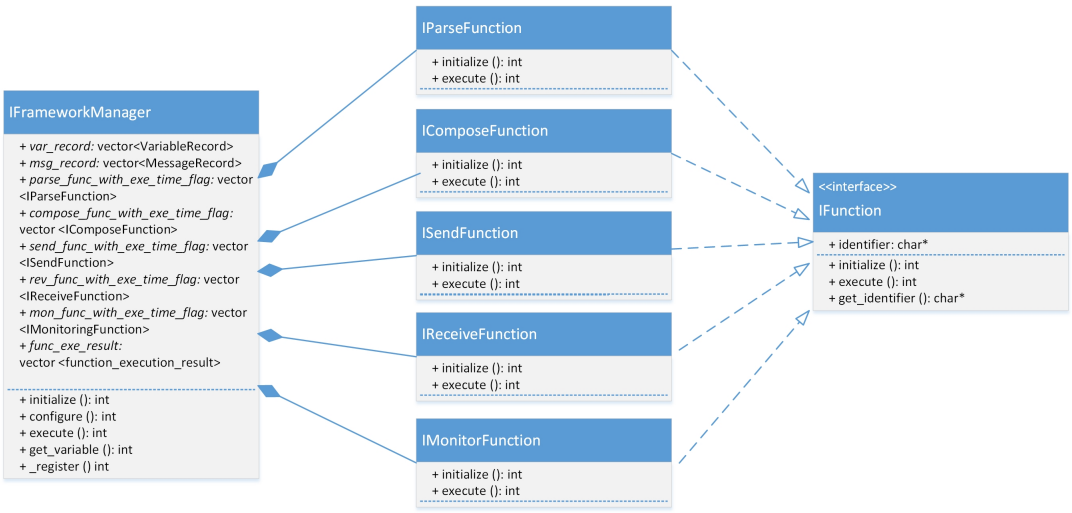

FIGURE 4.8: Structural Design of Framework Manager

4.3.1.2 Configuration Manager

The configuration manager is in charge of importing configurations into the framework before the system starts up. In the implementation that is depicted in Figure 4.9, the configuration manager is responsible for managing the imported configuration items which are represented as variables in the framework.

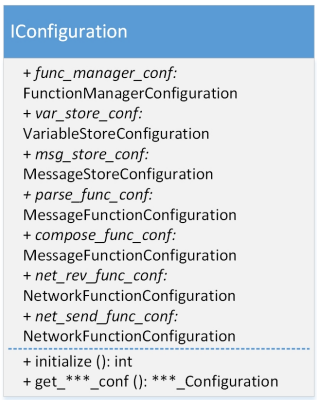

FIGURE 4.9: Structural Design of Configuration Manager

An initialization and a set of getter functions are exposed by the configuration manager, so that the framework manager is able to both trigger the importation of configuration items and query the imported items during system run-time.

4.3.1.3 Function Manager

The functions of the software components in the framework are registered and managed in the function manager depicted in Figure 4.10. The function manager dispatches the registered functions according to the schedule provided by the framework manager.

In this implementation, the mapping between emulated function types and function names is leveraged in order to enable switching between different function types. The vector called func_schedule_entry is designed to maintain the execution parameter of each functions. All the registered functions are managed as the objects of the basic function class.

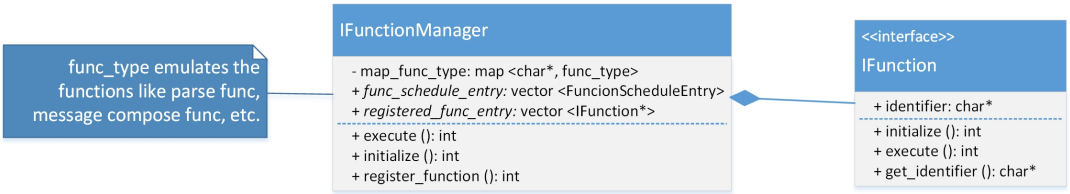

FIGURE 4.10: Structural Design of Function Manager

4.3.1.4 Variable Manager

The variable manager provides the access point for the stored variables in the framework. In the implementation shown in Figure 4.11, it is implemented as the parent class, from which different kinds of variables are inherited.

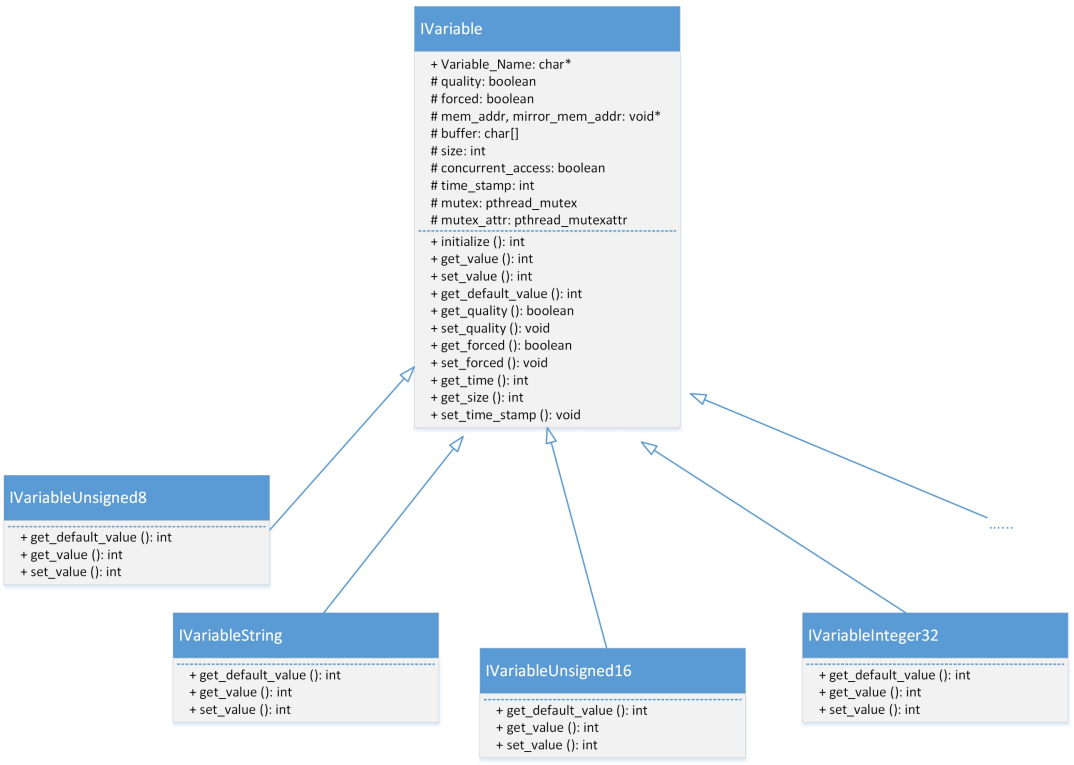

FIGURE 4.11: Structural Design of Variable Manager

Accessing a variable includes setting and getting operations. For each setting operation of the variables, the mutex provided by PikeOS is leveraged to manage concurrent access to the objects. The getting operations differ from the setting operations, where the variable values need to be judged to be correct, when the variables are mirrored based on the required SIL level.

The variables in the framework are stored in a shared memory space and the corresponding mirrored memory in case of critical variables. The shared memory in PikeOS is designed for the interaction between different resource partitions, which are configured in the PikeOS system integration project. Therefore, the shared memory is also accessible for the processes within the same resource partition.

4.3.1.5 Message Manager

As shown in Figure 4.12, the message manager consists mainly of the parsing and composing interfaces, which are inherited from the basic function class. Apart from the general function attributes, the composing function and the parsing function need to manage either the target messages or the source messages and the involved variables of the managed messages.

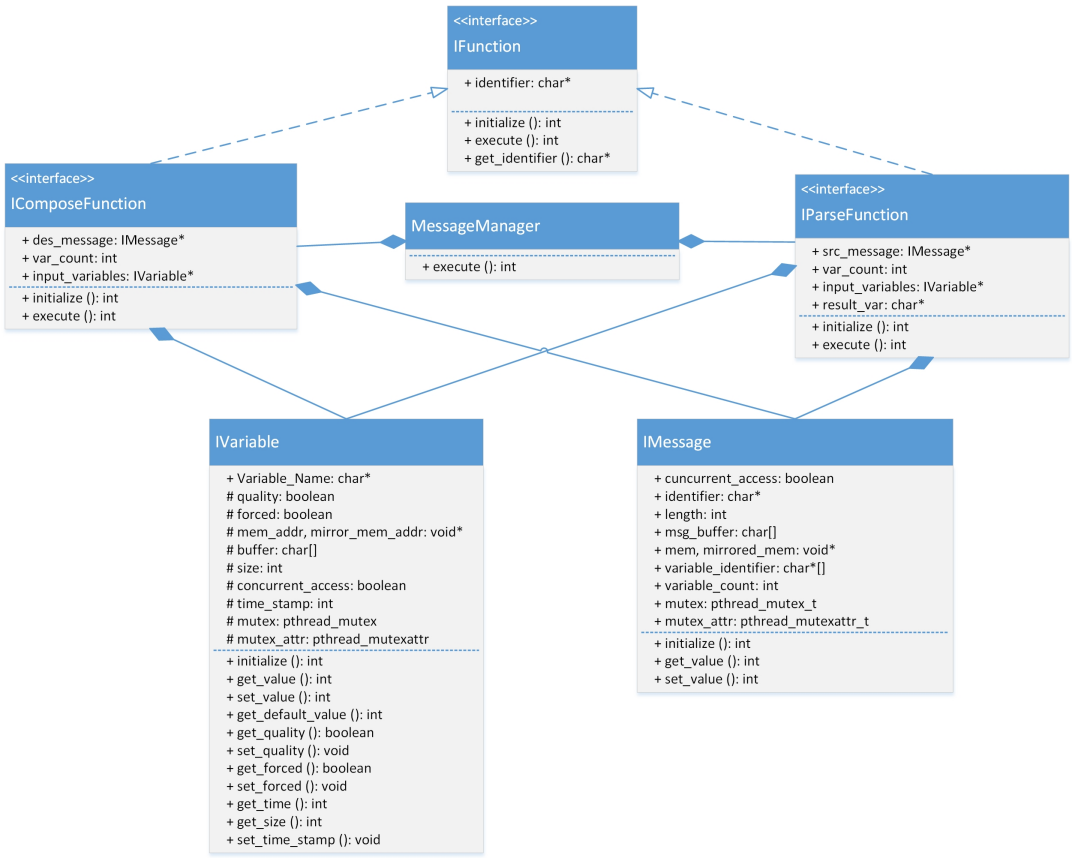

FIGURE 4.12: Structural Design of Message Manager

4.3.1.6 Network Manager

In a similar way as the message manager, the network manager (see Figure 4.13) is in charge of sending and receiving messages that form the two major building blocks of this manager.

The receive function inherits the basic function and it can receive messages from specific sockets. After a receiving operation, the message manager as above presented is activated to parse the received messages and produce the variables. The send function works in the opposite fashion and it is worth mentioning that the difference is that a send function should manage the length of the sent messages. The reason for the difference in the implementation is that a receiving process is able to receive messages when there is enough buffer space without errors during run-time. However, when a sending process sends out messages with an unexpected length, it can cause potential unknown system failures.

4.3.2 Experimental Results

In this section, the experimental results of the proof-of-concept implementation are presented. The major evaluated aspect of the proposed framework is the capability to guarantee non-interference between applications of different safety-critical levels.

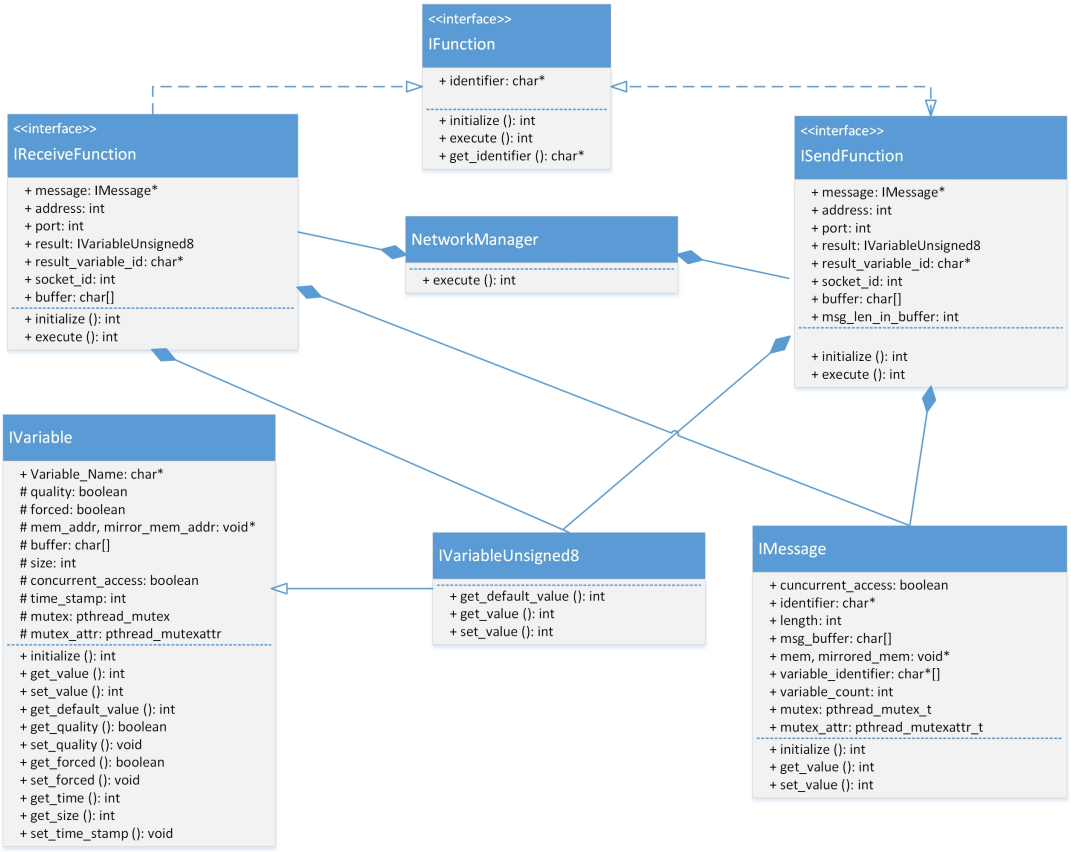

FIGURE 4.13: Structural Design of Network Manager

4.3.2.1 Use Case

In order to demonstrate the capability of the framework, the user application "Bogie Monitoring System" application (BMS) is developed and hosted by the framework. Another application named "Rogue Application" (RA) is developed to cause interference to the BMS.

The BMS application is defined to monitor bogie temperatures that are measured by the temperature sensors and sent to the BMS application. The BMS application decides on the output alarm levels based on configured temperature thresholds. More specifically, the generated output can indicate a normal or abnormal bogie temperature. The BMS application is used to simulate a safety-critical application in the experiment.

As illustrated in Figure 4.14, the BMS and the RA are integrated and hosted by the framework on the same computing node. The BMS receives sensor inputs through the framework and sends out the computed output to the framework, which is responsible for data exchange between partitions. The BMS is instantiated as a user application residing in one partition, where the functionality is implemented as specific functions inherited from the basic function class. The RA residing in another partition intends to behave in a manner that is out of the scope of the system configuration. In this use case, the RA tries to access unauthorized variables and

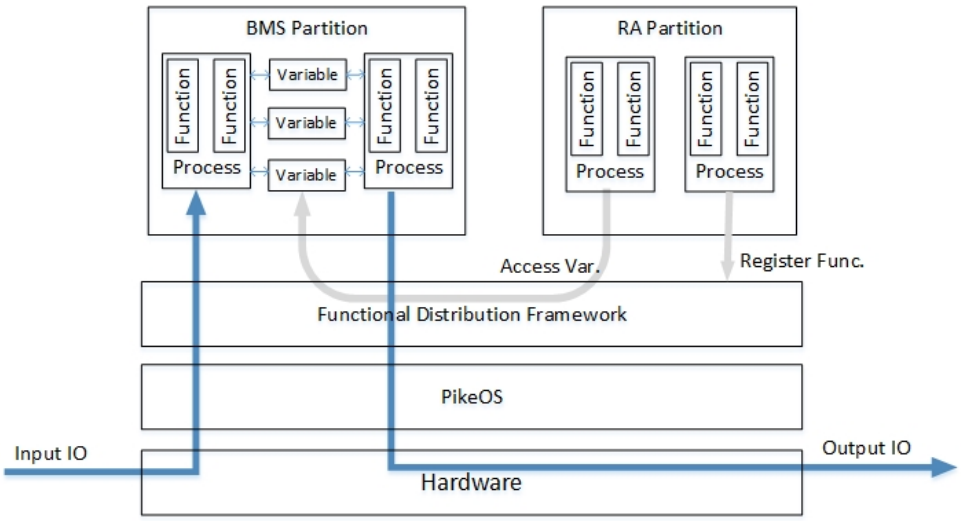

FIGURE 4.14: Experimental Use Case

register functions that are not configured, in order to inject systematic fault into the framework.

4.3.2.2 Results

The experimental setup in this demonstration is shown in Figure 4.15, where the BMS partition and the RA partition are integrated on an X86 computing node. We generate the input messages for the BMS partition and receive the output messages on the IO simulator. The RA controller and monitor is used to generate the control messages to activate the fault injection of the RA partition and receive the corresponding results of the triggered actions.

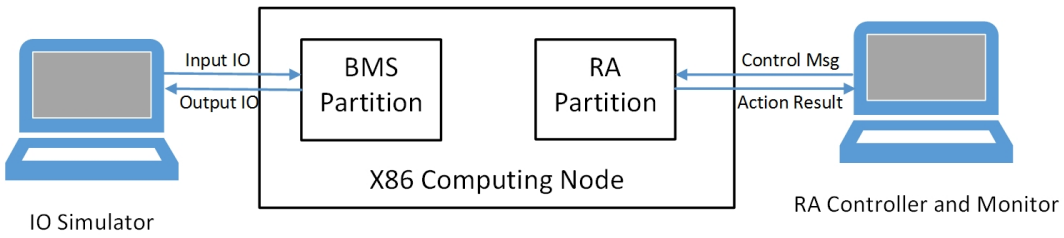

FIGURE 4.15: Experimental Setup

The schedule of the both partitions on the computing node is depicted in Figure 4.16. These two partitions are scheduled in a period of 50 ms, in which the BMS partition is allocated with 40 ms CPU time and the left 10 ms is assigned to the RA partition.

In the BMS application, there exist 6 configured variables (see Table 4.1) to store the variables contained in the input messages and the computing results of the functions. The InputTemperature and InputState record the measured value and system status of the temperature sensors, the AlarmState represents the calculated

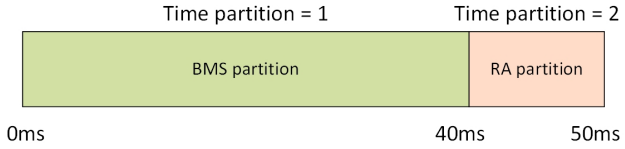

FIGURE 4.16: Schedule of Use Case

alarm levels based on the input values and the BMSState indicates the status of the BMS application.

TABLE 4.1: Configuration of BMS Variables

Identifier	Inter Process	Access Mode	Type	Default	Range
InputTemperature1	true	concurrent	int	0	20 - 30
InputState1	true	concurrent	bool	0	0 or 1
InputTemperature2	true	concurrent	int	0	20 - 30
InputState2	true	concurrent	bool	0	0 or 1
AlarmState	true	concurrent	bool	0	0 or 1
BMSState	true	concurrent	bool	0	0 or 1

The configured variables in the RA are listed in Table 4.2. The ActionID indicates the triggered action in the RA, and the corresponding action result is stored by the ActionResult variable. If the triggered fault is successfully injected in the framework, then the value of ActionResult equals to 1.

TABLE 4.2: Configuration of RA Variables

Identifier	Inter Process	Access Mode	Type	Default	Range
ActionID	true	concurrent	string	0	-
ActionResult	true	concurrent	bool	0	0 or 1

The detail structures of the input and output messages for the BMS application are shown in Figure 4.17. Based on the data structures, one can tell that the BMS application majorly computes the state information and adds some extra identification information (i.e., XXID) to the output messages.

The RA controller and monitor in Figure 4.15 communicates with the RA partition by the unified message structure shown in Figure 4.18. When the RA partition is triggered to inject specific systematic faults that are identified by ActionID, it carries out the action and sends the results (i.e., success or fail) to the RA controller and monitor.

We run the experiment for a duration of several hours and triggered the RA at unspecific points in time. The exchanged messages between the IO simulator and the BMS partition, as well as the ones between the RA partition and the RA controller and monitor are captured via Wireshark. The example input and output message

```
struct BMS_Input_Msg{
   P4_sint16_t ComID;
   char AxleboxID[3];
   bool AlarmState;
   P4_sint16_t Temp1;
   P4_uint64_t TimeStamp1;
   bool Sensor1State;
   P4_sint16_t Temp2;
   P4_uint64_t TimeStamp2;
   bool Sensor2State;
}__attribute__((packed));
```

```
struct BMS_Output_Msg{
   P4_sint16_t ComID;
   char BMSID[3];
   bool BMSState;
   char VehicleID[3];
   char BogieID[3];
   char AxleboxID[3];
   bool AlarmState;
   P4_sint16_t Temp1;
   P4_uint64_t TimeStamp1;
   bool Sensor1State;
   P4_sint16_t Temp2;
   P4_uint64_t TimeStamp2;
   bool Sensor2State;
}__attribute__((packed));
```

FIGURE 4.17: Structures of BMS Input and Output Messages

```
struct ROGUE_MON_msg{
   char RogueAppID[3];
   char ActionID[3];
   bool ActionResult;
}__attribute__((packed));
```

FIGURE 4.18: Structure of RA Messages

of the BMS application are shown in Figure 4.19 and Figure 4.20, correspondingly, where the payload in the messages are marked in blue.

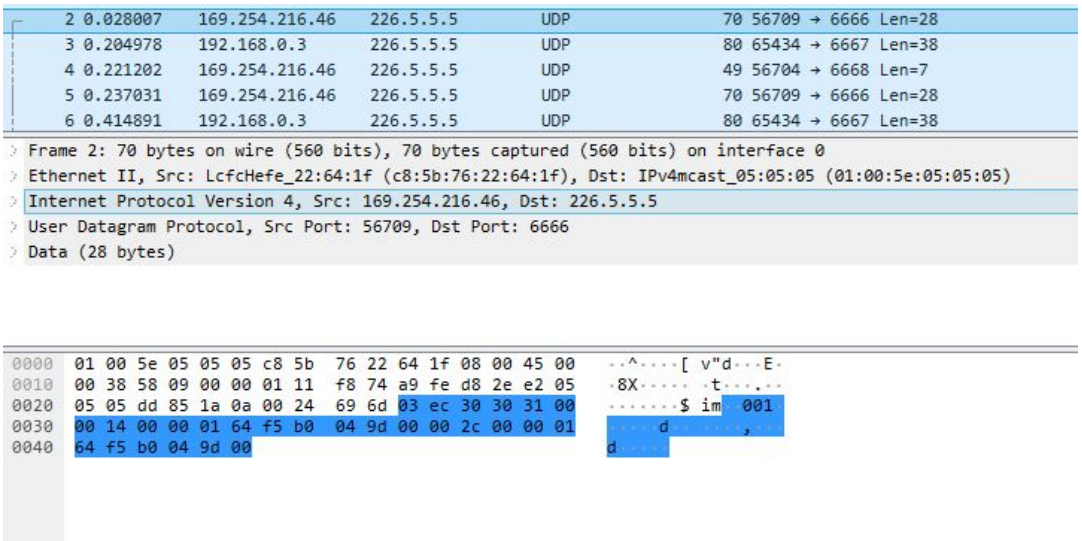

FIGURE 4.19: BMS Input Message

According to the presented data structure of the input messages, the value of Temp1 and Temp2 are 20 and 44, and the values of the sensor states are both 0, which means no error of the sensors in this simulation. In the captured output message, the 6th and 16th bytes record the calculated states of the BMS partition and the alarm state. The 6th byte is 0, which indicates that the BMS partition is functioning without errors. The 16th byte is set to 1 to indicate that the measured temperatures are out of

the reasonable range (i.e., [20, 30]).

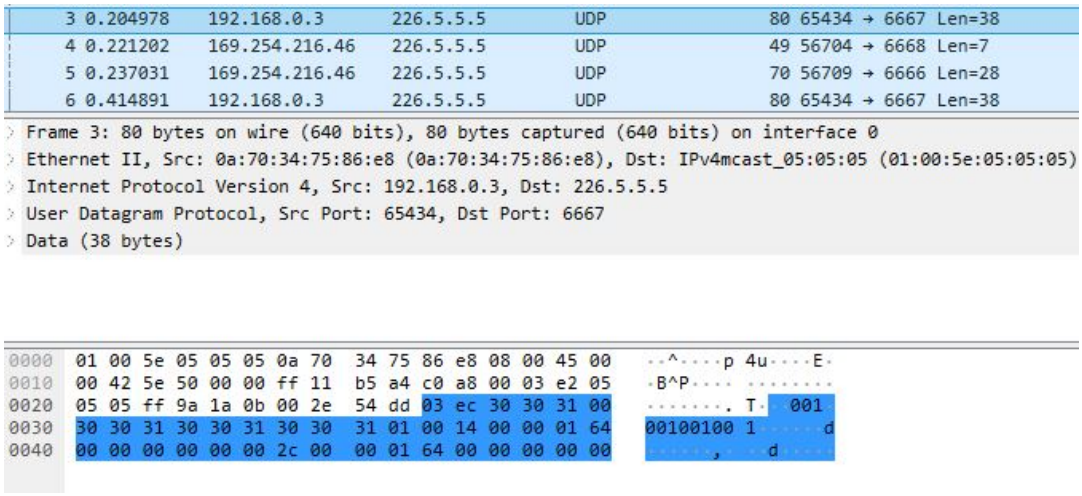

FIGURE 4.20: BMS Output Message

As presented, the systematic fault injection actions carried by the RA include accessing unauthorized variables and registering functions that are not configured. One example of the exchanged messages between the RA partition and the RA controller and monitor are shown in Figure 4.21 and Figure 4.22.

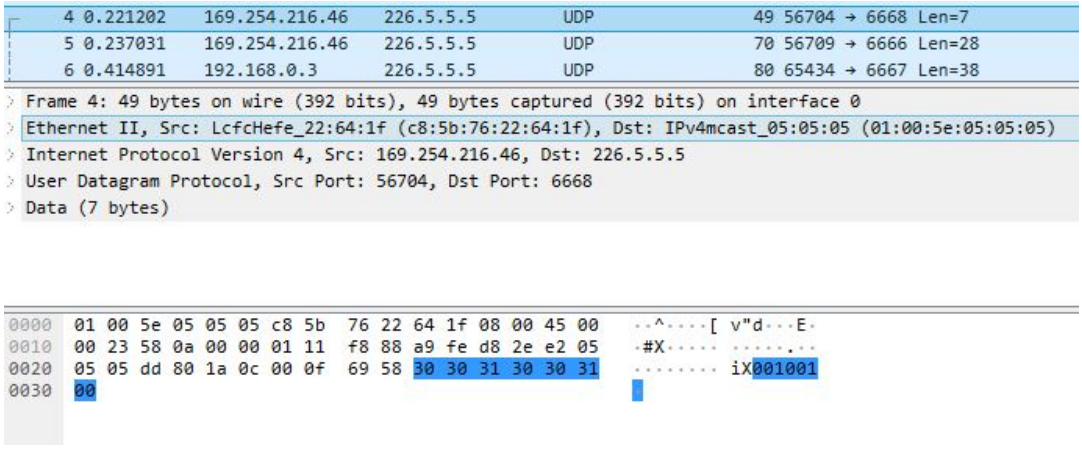

FIGURE 4.21: RA Input Message

One can read that the value of the action result (i.e., the 7th byte) in the output message stays unchanged in compare to the input message, which means that the triggered injection action is failed.

This demonstration mainly proves the non-interference between applications, which are timely and spatially isolated on the same computing node. Consequently, the timing determinism of the system is maintained during run-time. For simplification, the configuration parameters of the application are statically initialized in the configuration manager. Other ways to access the configuration file through a volume provider or a ROM file system are also feasible, therefore, dynamic system adaptation can be tackled. Since the system reconfiguration mainly affects the data

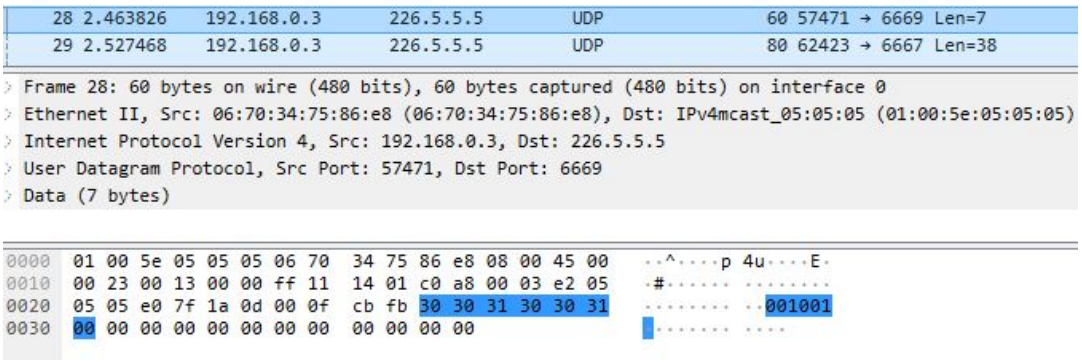

FIGURE 4.22: RA Output Message

communication, this issue is left open in this chapter and will be addressed in the next chapter, where the data communication is addressed.

4.3.3 Conclusion

In this chapter, the requirements for the execution environment based on an integrated architecture are analysed, based on which the functional distribution framework is conceptually proposed and demonstrated with an instantiation based on PikeOS. Regarding the software FCR, the proposed temporal and spatial partitioning between different applications rules out the error propagation between applications in the logical and temporal domains. In the proposed execution environment, redundancy mechanism (e.g., replicated variables) is leveraged to tackle the fault scenario like undesired variable modification.

The capability of the framework to exclude interference between applications of different criticalities is demonstrated in this chapter, and dynamically configurable data communication will be addressed in the next chapter.

Chapter 5

Virtual Data Communication for Integrated Real-Time Systems based on SDN

This chapter addresses the data communication within the execution environment for integrated real-time systems presented in Chapter 4.

5.1 Virtual Switch Supporting Time-Space Partitioning and Dynamic Configuration

In this section, we propose a virtual switch that ensures temporal and spatial partitioning between data flows of the integrated applications hosted on the same computing node. The switch leverages the SDN paradigm to be reconfigurable to address the dynamic adaptation requirement.

5.1.1 System requirements

In this section, we discuss the major requirements for the virtual switch in the execution environment for integrated systems.

5.1.1.1 Temporal and Spatial Partitioning

The target virtual switch extends the temporal and spatial isolation concept from the partition level to the data communication level, in order to guarantee the absence of interference between data flows.

The partitions connected to the virtual switch within one computing node should be isolated with respect to the execution time, in the way that the partitions are scheduled on a fixed and cyclic basis, so that the non safety-critical applications are not able to affect computing resources of the safety critical applications. The data transportation for each ingress port within the virtual switch should also be temporally separated.

With respect to spatial isolation, the ports of the virtual switch belonging to different partitions should have their own dedicated memory resources, in order to

prevent fault propagation between data flows, where a data flow can overwrite the memory area of another data flow.

5.1.1.2 System Reconfiguration

Existing data communication in integrated architectures (e.g., ARINC 653) comprises of statically configured channels between ports of different partitions, since there are no requirements for system reconfiguration of the computing nodes during runtime. However, in order to tackle the system reconfiguration requirement of further systems (e.g., the railway domain), the target virtual switch should be capable to adapt to the changes of the integrated system.

5.1.2 System Architecture

In this section, we propose our architecture design to address the identified partitioning and reconfiguration requirements.

5.1.2.1 General Architecture

As depicted in Figure 5.1, we define the general architecture model of the virtual switch. The virtual switch consists of the data plane and the address server, which are responsible for the message switching and address resolution, respectively. The executions in the virtual switch are triggered by the control application that is spatially and temporally isolated to the user applications.

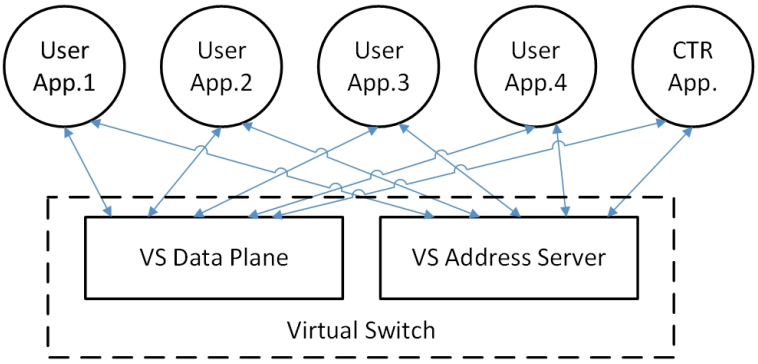

FIGURE 5.1: General Architecture Model

The detailed architecture model of the data plane is shown in Figure 5.2. The virtual ports provide the interface for the messages (i.e. the blue boxes) exchange between the hosted applications and the data plane. A virtual port is initialized with its specific configuration (e.g., buffer), and the message transmission in the data plane is triggered based on the configured schedule. The link configurations in the data plane specify the interconnections between the virtual ports, which are dynamically configured during runtime.

The architecture model of the address server is depicted in Figure 5.3. The central building block of the address server is the address database, which stores

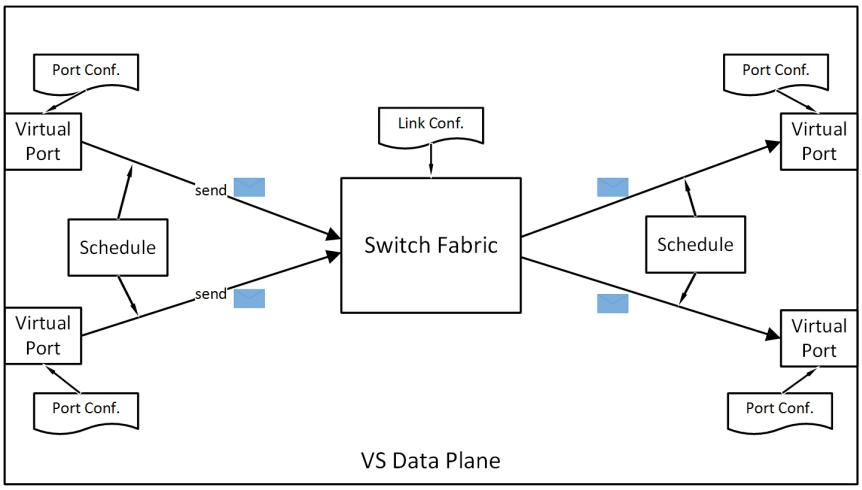

FIGURE 5.2: Architecture Model of VS Data Plane

the mapping information for the address resolution. An user application queries a message's destination virtual ports in the data plane, while a control application (e.g., CTR App. in Figure 5.1) updates the stored mapping information in case of system reconfiguration. Since the address database is accessed by both the user applications and the control application, concurrent access should be addressed to avoid system inconsistency. In an integrated computing node, the control application is assigned to time slots that are exclusively allocated, so that the discussed system inconsistency is ruled out by design.

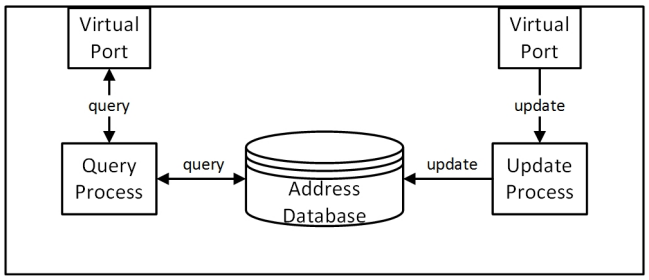

FIGURE 5.3: Architecture Model of VS Address Server

5.1.2.2 Communication Architecture

The communication mechanism of the virtual switch is inspired by the SDN paradigm, where a switching entity is logically separated as the data plane and the control plane. According to the SDN paradigm, we define the VS Data Plane as the data plane of the virtual switch and the CTR application as the control component.

- Data plane. The address server in Figure 5.1 maps a message URI to the destination ports. Before sending a message, the user application queries the address server with the destination message URI for the destination ports within the VS Data Plane. Thereafter it configures the path within the VS Data Plane, and then sends out the messages. The VS Data Plane is capable for buffering

the received messages in a dedicated buffer of each ingress port, which ensures the spatial separation between data flows.

- Control plane. The message transport within the data plane is triggered by the CTR application during the assigned time windows. The messages are not moved to the destination ports right after reception in the ingress ports, so that the destination ports are not exposed to a data race situation, in which a non-safety critical message can delay a safety critical one. The order of ports, from which the VS Data Plane moves the messages to the destination ports, is configured by the CTR application.

The general process of the data communication is illustrated in Figure 5.4.

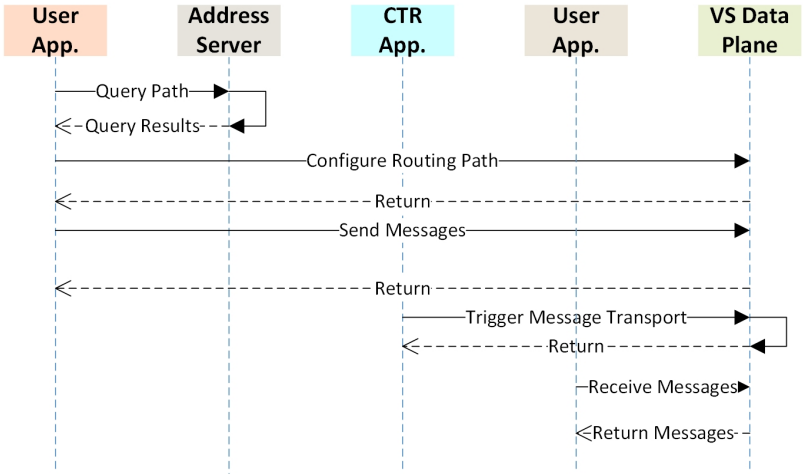

FIGURE 5.4: Data Communication Process

5.1.2.3 Resource Partition and Time Partition

The resource partition and time partition concepts are introduced to guarantee non spatial or temporal interference between different applications.

A resource partition represents an application and is assigned to a time partition that owns statically allocated time windows within the fixed cyclic time frame. An RTOS scheduler can dispatch the time partitions in a time-driven manner to meet the timing requirements of real-time applications. In contrast, the threads within resource partitions assigned to the same time partition can be scheduled in priority-based preemptive fashion.

As long as the system resources (e.g., memory, I/O, etc.) are statically assigned to a partition, the MMU is leveraged to enforce that each partition has exclusive access to the assigned resources. This kind of spatial isolation contributes to preventing faults of one resource partition propagating to another one.

5.1.3 Proof-of-Concept Implementation

In this section, we discuss the proof-of-concept implementation and present the experimental results of the prototype. In order to support hard real time communication, all the related components should have bounded WCET. We discuss the relevant data structure and the scheduling in the implementation, in order to prove the deterministic behavior of the proposed virtual switch.

5.1.3.1 System Setup

In this implementation, we use the QEMU emulator to emulate a generic X86 platform for the generated PikeOS ROM image, which integrates the PikeOS kernel, kernel drivers, system extensions and the user applications. Our emulation runs on the PC hardware with two cores of 2.3 GHz and 24 GB memory.

5.1.3.2 Implementation

The general architecture of the implemented virtual switch is depicted in Figure 5.5. The virtual switch consists of the data plane and the local DNS server, which are implemented as kernel drivers in the PikeOS system. The DNS server is a mock up instance and extended to map a message URI to the destination ports. In this implementation, there are four resource partitions (Partition X) communicating through the virtual switch and one control partition (CTR Partition) managing the virtual switch.

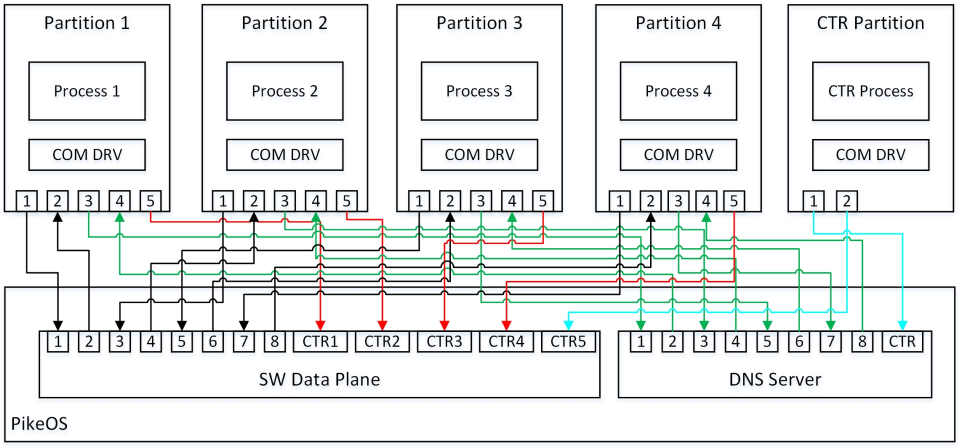

FIGURE 5.5: General Architecture Design

The processes within partition 1 to 4 communicate with each other via the Communication Driver (COM DRV), which encapsulates the whole message sending and receiving process and provides well defined interfaces to the applications. The COM DRV exchanges messages with the DNS server and the SW Data Plane through statically configured ports. The shown ports are of different functionalities. For example, the ports No. 1 to 8 of the SW Data Plane and the DNS Server are defined as the data ports, while the ports named with "CTR" are supposed to be the configuration entry

points of these components. The connections between ports are coloured differently to identify different data flows and partitions.

5.1.3.3 Relevant data structures

The implemented components of the virtual switch are the SW Data Plane and the DNS Server. The related data structures are discussed in detail in this section.

- SW Data Plane. As shown in Figure 5.6, the major data structure in the SW Data Plane consists of three parts. The first part is the array port_bitmap, which is designed for each ingress port to record the bitmap of the destination ports. A sending partition needs to configure this bitmap before dispatching a message. The second part is the buffer_array for each ingress port to buffer the received messages before being conveyed to the destination ports. When the SW Data Plane is triggered to transport the messages from the ingress ports to the egress ports, the bitmap port_control_list is used to regulate the priorities of the ingress ports.

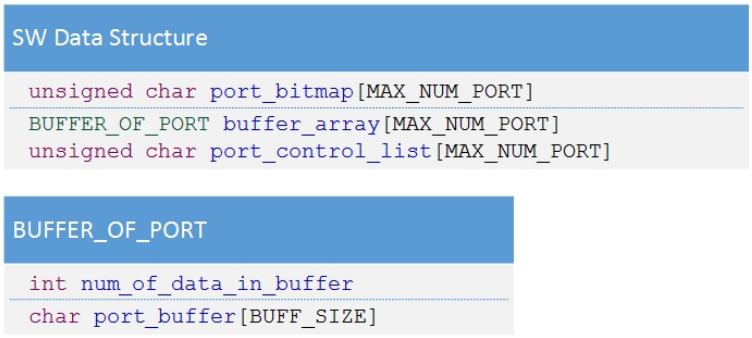

FIGURE 5.6: The Major Data Structures in SW Data Plane

- DNS Server. As depicted in Figure 5.7, the DNS Server stores the mapping of the application URIs, the IP addresses and the corresponding port cookies in the uri_ip_port_mapping_array to resolve the queries from the partitions. The port_buffer array is defined for each port to record the query results. Every port within the DNS Server and the SW Data Plane is assigned a unique cookie. The query results can contain more than one destination port cookie, in case one partition needs to send out a multi-cast message.

5.1.3.4 Scheduling

As shown in Figure 5.5, there are four application partitions and one control partition hosted on the platform. In this implementation, Partition 1 and 2 are defined as the message sender and Partition 3 and 4 as the receiver. In order to preserve the temporal isolation between message senders and receivers, the scheduling scheme in the implementation is designed as depicted in Figure 5.8.

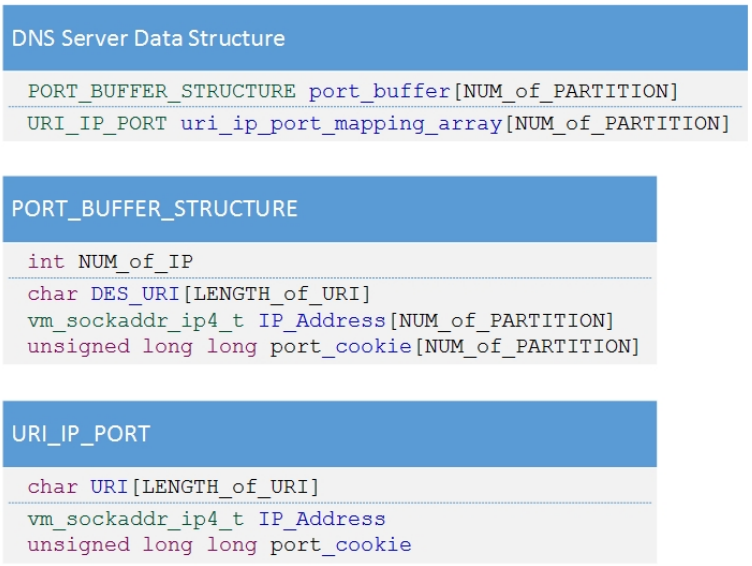

FIGURE 5.7: The Major Data Structure in DNS Server

PikeOS also introduces a background time partition, which is active during runtime and contains threads of both high priority (e.g., error handler) and low priority. This background time partition is designed for the safety critical threads to preempt other threads and for the non-safety critical threads to consume the idle CPU time of all the time partitions.

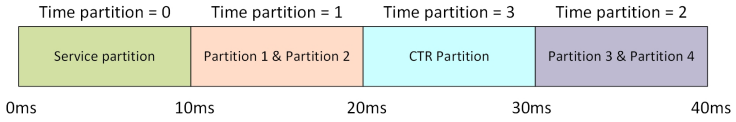

FIGURE 5.8: Scheduling Scheme

In this scheduling scheme, a cyclic time frame of 40ms is defined, within which each of the time windows is set to 10ms, so that despite of system overhead, each sending or receiving process can be finished within one time window. The Service partition is the background partition that is beforehand discussed. The Partition 1 and 2 are scheduled to run in parallel within the second time window. After the senders finish sending messages to the ingress ports, the CTR Partition is scheduled to manage the SW Data Plane and DNS Server, as well as to trigger the SW Data Plane to finish moving message from ingress ports to the configured egress ports. As the receivers, the Partition 3 and 4 run after the CTR Partition to receive the expected messages.

5.1.3.5 System reconfiguration

In the example system architecture in Figure 5.5, we assume that the network topology related information is updated periodically and the DNS Server keeps the up-to-date mapping of the application URIs, IP addresses and the assigned ports within the SW Data Plane. The reconfiguration of the SW Data Plane is done by each COM DRV

before sending out a data message. In more detail, if one process requests to send out a data message with a destination URI, the COM DRV will query the DNS Server for the destination port of this message, and then configure the SW Data Plane through the connected CTR port.

5.1.4 Experimental Results

In this proof-of-concept implementation, we measured the consumed time for a partition to send and receive a message. The emphasis is placed on the non-interference between partitions and the deterministic behaviour of the configuration procedure. If one sending partition sends more than one message within one time window, there will be no context switching for the sending thread and the sending overhead will be dramatically decreased. In order to make the test scenario as close as possible to the worst case scenario and based on the thread model of PikeOS, we ensure that each sending partition sends at most once within one time window.

5.1.4.1 Sending and Configuration Overhead

In Figure 5.9, the results consist of the consumed time for Partition 1 and 2 to send out messages. The measured sending process starts from sending out the query to the DNS Server and ends at the successful return from sending out a message (see Figure 5.4). This figure also depicts the results of measuring the delay for both partitions to configure the SW Data Plane, before the data messages are sent out. On average, the consumed time for configuring the SW Data Plane takes more than 50% of the sending delay.

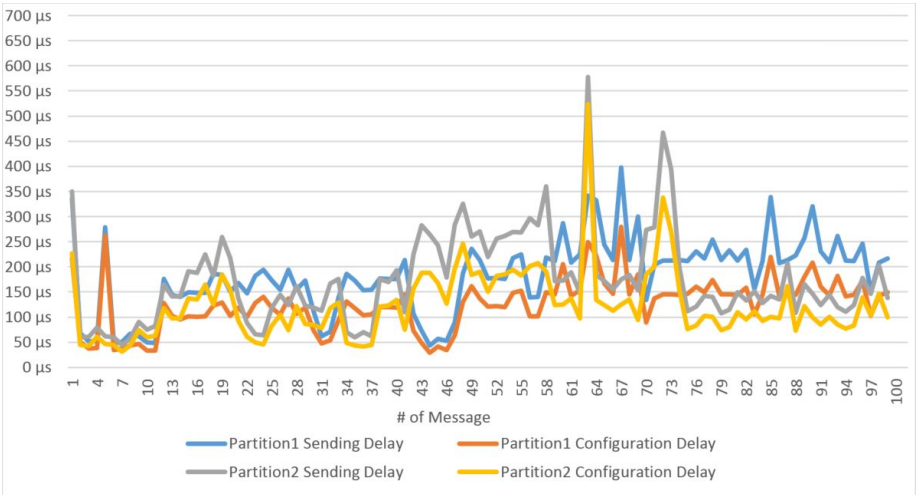

FIGURE 5.9: Sending and Configuration Delay

The delay of sending a message to its ingress port in the SW Data Plane is the difference between the consumed time of the whole sending process and the consumed time of configuring the SW Data Plane (about 50-100 μs in Figure 5.9).

5.1.4.2 Receiving Overhead

Similar to Figure 5.9, the results in Figure 5.10 show the receiving delay of Partition 3 and 4. The receiving process covers two steps: sending out the receive command and receiving the expected message. On average, receiving a message takes about 120 μs that is less than the sending overhead, because in the sending procedure, the sender partition needs to configure the SW Data Plane before sending out the messages, which causes the observed increased delay.

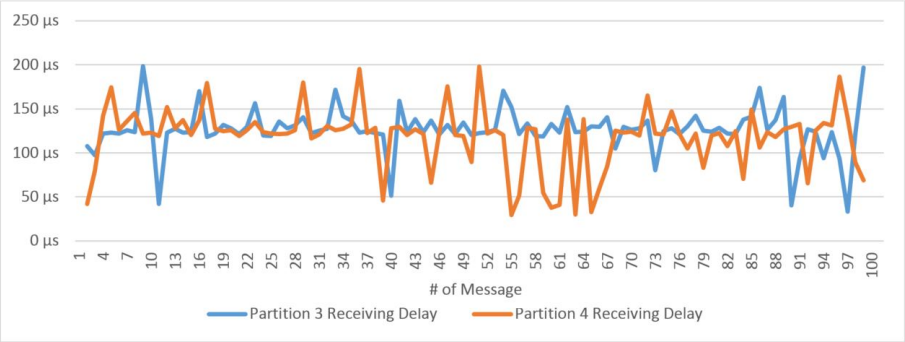

FIGURE 5.10: Receiving Delay

The observed overhead for sending a message to the ingress port (see Figure 5.9) is on average significantly less than the receiving overhead. The reason is that there is no context switching after configuring the SW Data Plane and the sending thread can send out the message in the same thread context.

5.1.4.3 Relaying Overhead

Except for the overhead during message sending and receiving, the overhead for the SW Data Plane to move two messages from the ingress ports to the egress ports is shown in Figure 5.11. The most measured overhead vary between 50 μs and 150 μs.

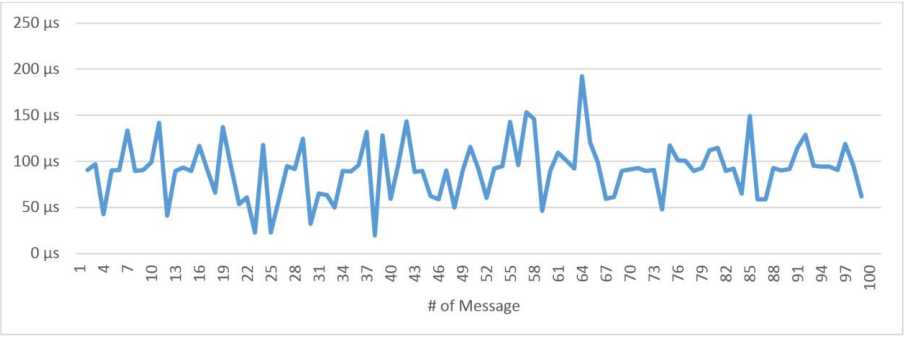

FIGURE 5.11: Local Message Transmission Overhead

Since the PikeOS kernel has to deal with high priority interrupts (e.g., clock interrupt), the jitter of the measured delay during runtime in Figure 5.9, 5.10 and5.11 is unavoidable.

5.1.4.4 Temporal Isolation

In order to verify the temporal isolation of different time partitions, we use the trace server in the PikeOS kernel to trace the time partition switching events in the kernel. Since the time partition switching events happening on different CPUs are similar, we chose the traced results on CPU #1 that are shown in Figure 5.12. As verified in Figure 5.12, there is only one active time partition on CPU #1 at any point of time. The results also confirm the static cyclic schedule scheme in Figure 5.8.

Activity	CPU	Type	Name	Subevent	Timestamp [ns]	Delta [ns]	Attributes
PikeOS ukernel (1000, 0x00000000)	1	Timepartition	Interrupt 511	Time partition switch	5.959.818	0	oldtp=2; newtp=0
PikeOS ukernel (1000, 0x00000000)	1	Timepartition	Interrupt 511	Time partition switch	18.782.923	12.823.105	oldtp=0; newtp=1
PikeOS ukernel (1000, 0x00000000)	1	Timepartition	Interrupt 511	Time partition switch	28.270.509	9.487.586	oldtp=1; newtp=3
PikeOS ukernel (1000, 0x00000000)	1	Timepartition	Interrupt 511	Time partition switch	35.980.948	7.710.439	oldtp=3; newtp=2
PikeOS ukernel (1000, 0x00000000)	1	Timepartition	Interrupt 511	Time partition switch	47.825.780	11.844.832	oldtp=2; newtp=0

FIGURE 5.12: Temporal Isolation

5.1.5 Conclusion

The work in this section leverages the SDN paradigm to define a reconfigurable virtual switch within the execution environment for integrated real-time systems. The proposed virtual switch guarantees spatial and temporal isolation between data flows to meet the requirement of mixed-criticality data communication. By leveraging the isolation mechanisms, a non-critical data flow can neither occupy the allocated time interval, nor access the transmitted information of a critical data flow. The proposed reconfiguration mechanism at the communication level addresses the dynamic structural changes both in intended or unintended system reconfiguration. More specifically, a faulty partition treated as a software FCR can be replaced by a redundant partition that results in the adaptation of the virtual likes at the communication level. The virtual switch is capable to leverage the up-to-date structural information to achieve the information transmission.

5.2 Virtual Gateway

In the previous section, a virtual switch supporting time-space isolation and dynamic configuration was proposed to address the local communication within an integrated computing node. However, controlled information import and export between different data flows in the virtual switching environment is an open research problem.

In this section, we propose a virtual gateway residing in the virtual switching environment to resolve property mismatches between different data flows and prevent fault propagation between applications of different safety levels. As shown in Figure 5.13, the virtual gateway is involved in the message transmission procedure within the data plane of the virtual switch, when the input and the output messages are of different specifications.

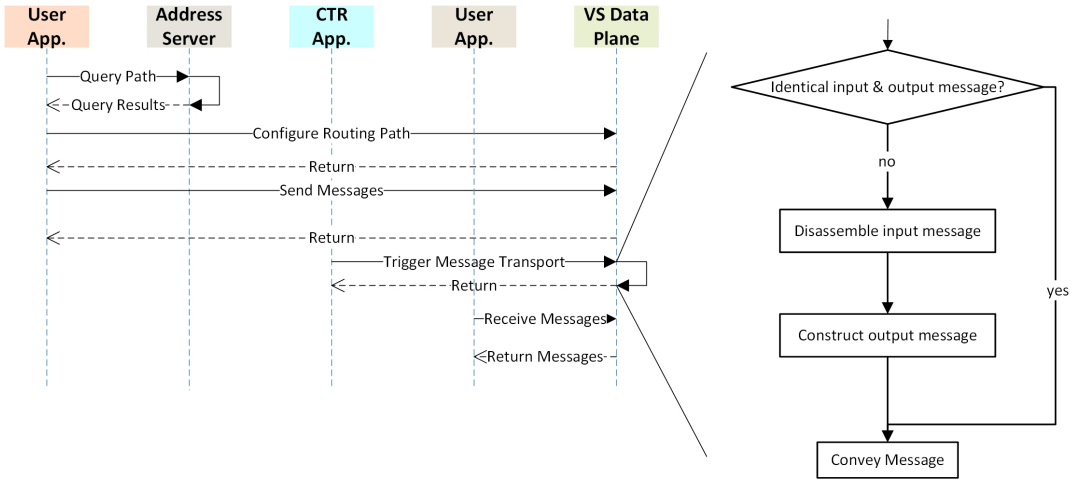

FIGURE 5.13: Generic Description of the Virtual Gateway

5.2.1 Overall Description

In the proposed virtual switch, the spatial and temporal isolation is extended from the application layer to the data switching layer, i.e., the virtual switch provides dedicated system resources (e.g., memory) to buffer both incoming and outgoing messages for each port, and the data transportation of the ports is guaranteed not to interfere with each other. A port in this context defines the interface between an application and the data plane of the virtual switch. Spatial isolation between ports that are assigned to different data flows also contributes to security such as handling masquerading failures between applications. In the SDN paradigm, the network forwarding control plane is separated from the data forwarding hardware, which enables the control logic and state to be reconfigured during runtime. The aforementioned benefit of SDN is leveraged by the proposed virtual switch to address the reconfiguration of the communication network.

As depicted in Figure 5.14, from the viewpoint of the logical structure, the virtual gateway resides in the proposed virtual switch to enable the virtual switch to control the import and export of messages between different data flows. The virtual gateway leverages the core services provided by the underlying RTOS to implement the required services.

5.2.2 Requirements of the Virtual Gateway

In the proposed virtual switch, the applications are connected via VLs. Since data redirection between different VLs contributes to reducing amount of redundant applications, the virtual gateway is required to support the controlled coupling of VLs. Due to the system reconfiguration requirement, configurable routing of a message with deterministic overhead should also be addressed by the virtual gateway.

From the view point of the virtual switch, the specification of the ports that link the applications with the data plane of the virtual switch needs to be extended to capture operational properties, semantic and addressing information. This extension

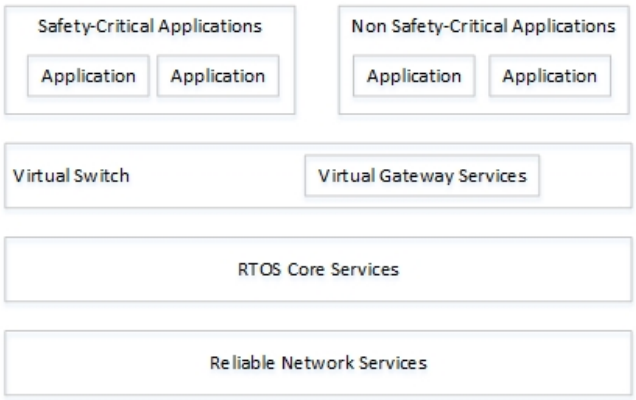

FIGURE 5.14: Logical Structure of the Virtual Gateway

contributes to enable the virtual gateway to either manipulate the passing messages at a finer granularity than the proposed virtual switch or achieve the controlled data transmission. Since the applications of mixed-criticalities can reside on the same integrated computing node, the virtual gateway needs to encapsulate the data flows to avoid error propagating from non safety-critical applications to safety-critical ones.

5.2.3 Role of the Virtual Gateway

As shown in Figure 5.15, the virtual gateway is supposed to convey messages between different VLs. From the perspective of an application, the port (i.e., interface between an Application and its VL) specifications differ from each other, which results in the property mismatch between VLs. Resulting from this property mismatch, one of the major functionalities of the virtual gateway is to achieve the property transformation. In the execution environments based on an integrated architecture, the applications of different safety criticalities could be virtually coupled via the virtual gateway. Consequently, the other functionality of the virtual gateway is to encapsulate information exchanged between data flows.

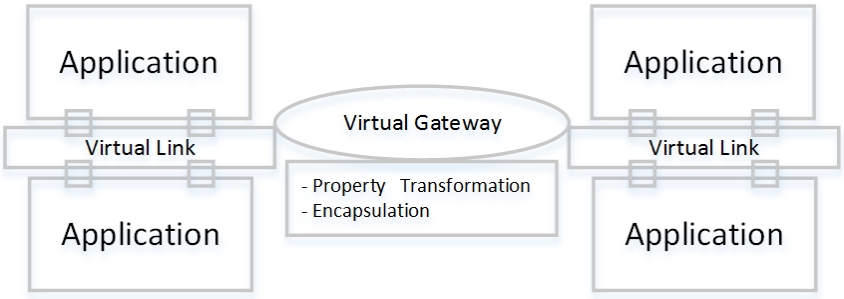

FIGURE 5.15: Role of the Virtual Gateway

5.2.3.1 Property Transformation

Since the work in this dissertation focuses on the message exchange between data flows within the same computing node, the property mismatch occurs at semantic

and operational level. An important source of semantic incoherence comes from different application specific rules (e.g, naming rules). Since the proposed virtual gateway resides underlying the application level, the major addressed property transformation takes place at the operational level.

Each port connecting an application and the data plane is configured with the specific characteristics (e.g., message syntax, temporal specification). This port description enables the virtual gateway to transform messages into others. The central mechanism within the virtual gateway for the syntax transformation is an inner repository to store the convertible elements that build up the messages according to the configuration of each port. The virtual gateway can resolve temporal differences (e.g., different message periods) of the transported messages by leveraging the inner repository to buffer the transformed messages.

5.2.3.2 Encapsulation

Since an integrated platform is open to host applications of different safety criticalities, different data flows are encapsulated from each other by the virtual gateway via selective redirection of messages. The virtual gateway is in charge of making decision on the authorisation of the passed messages through the gateway according to the configured rules (e.g., max/min values, periods). This implies that the virtual gateway also applies the configured error detection mechanisms during the process of selective redirection, i.e., messages carrying erroneous contents are forbidden to crossover the FCR in the system.

5.2.4 Architecture of the Virtual Gateway

As discussed, the major functionalities of the virtual gateway are property transformation and encapsulation. The general architecture of the virtual gateway is shown in Figure 5.16, where the role of the virtual gateway is demonstrated with the help of three communicating partitions. Assume that partition 1 and partition 2 belong to the same application and partition 3 belongs to another application, the data flows between partition 1 and partition 2 are conveyed by the virtual switch and the virtual gateway is not activated during the switching process. In the case that partition 3 requires to receive messages from partition 1, the virtual gateway should be involved in the transmission process. The central building block of the virtual gateway is the database to store the convertible elements of the messages. The messages passing through each port are predefined and configured in each port. The virtual gateway dismantles the messages with respect to the configured syntax of the input messages and assembles the required output messages according to the configuration of the output port. The building blocks of the output messages are stored in the database. Another functionality of the database is to buffer the messages to resolve temporal inconsistency of the transported messages.

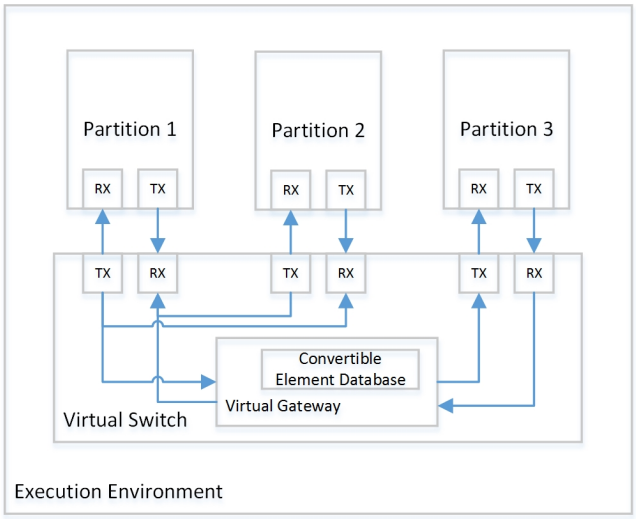

FIGURE 5.16: General Architecture of the Virtual Gateway

From the operation point of view, the detail architecture of the virtual gateway is depicted in Figure 5.17. The sending and receiving processes are scheduled by aligning with the scheduling of the virtual switch. According to the configuration of the communicating ports, the incoming messages are decomposed into convertible elements that can be used to compose the outgoing messages. In this sense, the virtual gateway can adapt to dynamic configurations of communicating ports.

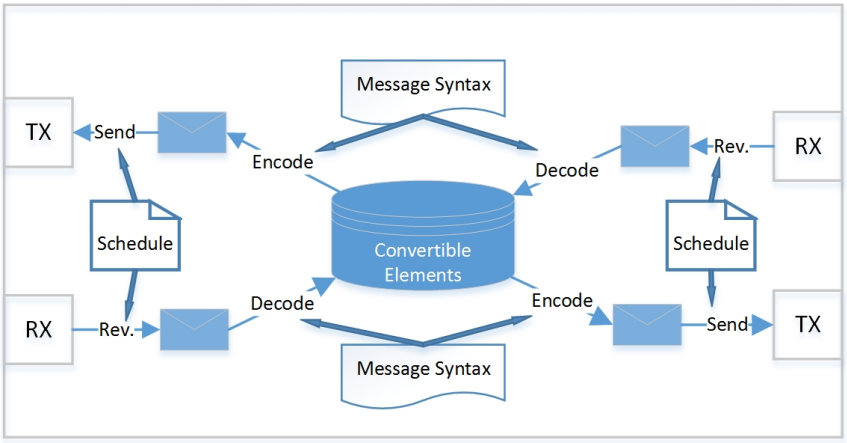

FIGURE 5.17: Operational Structure of the Virtual Gateway

With respect to the defined encapsulation capability, the virtual gateway should apply controlled filtering mechanisms between the input and the output ports. More specifically, the content and temporal domains of the transported messages should be within the pre-configured ranges.

5.2.5 Proof-of-concept Implementation

This proof-of-concept implementation extends the proposed virtual switch with the proposed gateway functionalities. The emphasis is placed on the extended

functionalities of the virtual gateway.

5.2.5.1 Data Communication

In the previous work, we leveraged the SDN paradigm to implement our data communication architecture as a virtual switch residing in the execution environment based on an integrated architecture, in order to address the dynamic system reconfiguration requirement. Taking the gateway defined functionalities into account, this implementation follows the communication architecture depicted in Figure 5.18.

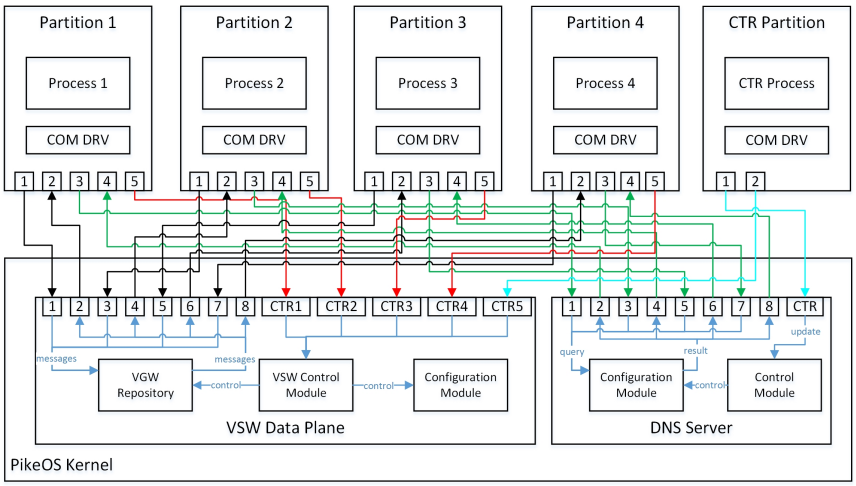

FIGURE 5.18: Architecture of Data Communication

In this data communication architecture, the partitions are spatially isolated and the processes within partition 1 to 4 communicate with each other through the Communication Driver (COM DRV), which encapsulates the message sending and receiving processes and provides well defined interfaces to the applications. The COM DRV exchanges messages with the DNS server and the virtual switch data plane implemented as PikeOS Kernel Level Driver in the kernel via statically configured gates. The gates in this implementation are (i.e., mapped entities of ports) assigned to transport either data messages or control messages.

The virtual gateway is implemented as modules within the PikeOS kernel. The developed system callbacks of a kernel module run in the context of the calling partition, which results in less context switching and better timing behaviour comparing to a software partition implementing the gateway functionalities. The virtual gateway repository is implemented as a database to buffer the convertible elements of the input messages, in order to enable the message transformation of different VLs. More details about this repository will be discussed later. The configuration modules in the data plane and the DNS server store the specifications of gates and the specifications of VLs, respectively. Both the repository and the configuration modules are controlled by the control module, i.e., the message transport and configuration update are triggered by the control module, which deals with the incoming requests from both user applications and control partition.

5.2.5.2 Component Specification

In the previous work, the virtual switch was implemented as a PikeOS kernel driver. This implementation extends the developed kernel driver with the defined gateway functionalities that require the extended specifications for the kernel driver.

- Gate Specification. From the viewpoint of the applications, the gates defined in the PikeOS system provide the access points for the user applications to communicate with the kernel modules residing in the PikeOS kernel. In the previous work, the messages between applications are exchanged without syntactic nor semantic specifications. In order to address the proposed property transformation functionality, the gates connecting the applications and the kernel module are extended to be configurable.
- Virtual Link Specification. A VL in the implementation represents a connection between applications residing in different partitions. In order to selectively redirect a message from an input gate to the output gate(s) and transform the properties of the transported messages, different VLs require a prior specifications that contain the following aspects:
  - Message Syntax: the message syntax of each VL is pre-configured, so that the virtual gateway can disassemble an incoming message into convertible elements and store them in the gateway repository to be assembled for the link specified outgoing messages.
  - Message Semantic: Semantic like event and state carried by the messages need to be considered when the messages are transported between different VLs.
  - Message Addressing: from the viewpoint of an application, the connected entities of a VL are addressed by system-wide identifiers. In this way, the applications are addressed based on the functionalities and reconfiguration of the underlying communication mechanisms is transparent to the applications.

5.2.5.3 Convertible element database

The virtual gateway is responsible for conveying messages between different VLs. Consequently, messages of different syntax need to be decomposed and composed by leveraging the database developed within the kernel module. The structure of the database is depicted in Figure 5.19. This gateway repository provides for each gate a separated buffer to store the convertible elements of the messages passing through the gates. This implementation aims at maintaining the spacial isolation for different data flows. And based on this structural design, accessing the database requires always deterministic overhead.

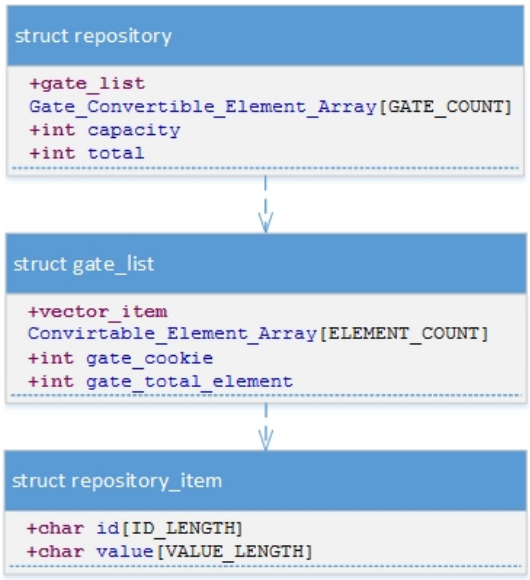

FIGURE 5.19: Data Structure of the Virtual Gateway Repository

5.2.5.4 System Workflow

The work flow of the data plane and the DNS server are depicted in Figure 5.20 and Figure 5.21, respectively. Both the data plane and the DNS server are implemented as PikeOS kernel driver, which provides system callbacks for the user applications to access the kernel modules.

As shown in Figure 5.20, during the initialization phase, the data plane allocates the necessary memory for all modules. Due to the systematic design, reallocation of memory after the initialization phase is not possible. Thereafter the data plane loads the pre-defined configuration (e.g., gate specification) into the configuration module.

After initialization, the data plane accepts both data messages and control messages from the user applications. Data messages are copied between user memory space and the data plane in PikeOS kernel, in order to protect the kernel space from unauthorised access of the user applications. In this framework, the control messages can be message switching trigger or reconfiguration instructions. The message switching is guarded by the Gate Control List (GCL), which defines the order of VLs to be activated. This mechanism aims to avoid the temporal interference between VLs within the data plane. The data messages could be conveyed directly from the input gate to the destination gates, when the input and output gate configurations specify the identical message structure. Otherwise the input messages should be disassembled and stored into the gateway repository, in order to provide input for the assembling process of the output messages.

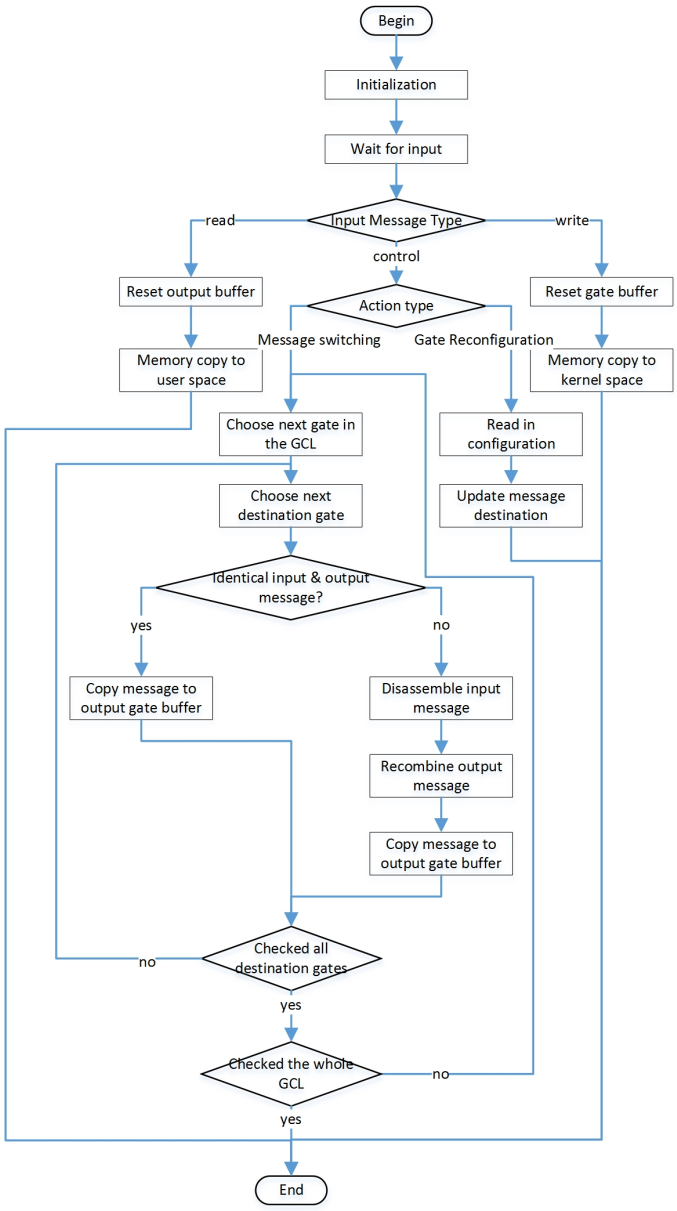

FIGURE 5.20: Workflow of the Data Plane

The DNS server in Figure 5.21 is responsible to resolve the address of the target entities to the output gates within the data plane. In order to simplify the whole system, we assume that the DNS server is always up to date with respect to the topology changes. The DNS server receives queries from the user applications and return the mapped output gates for the queried VL. The DNS server could also be triggered to update the IP addresses of target applications to simulate the system reconfiguration scenario. In order to provide location transparency of the user applications, the mapping between the application address and the output gate can be changed during run time.

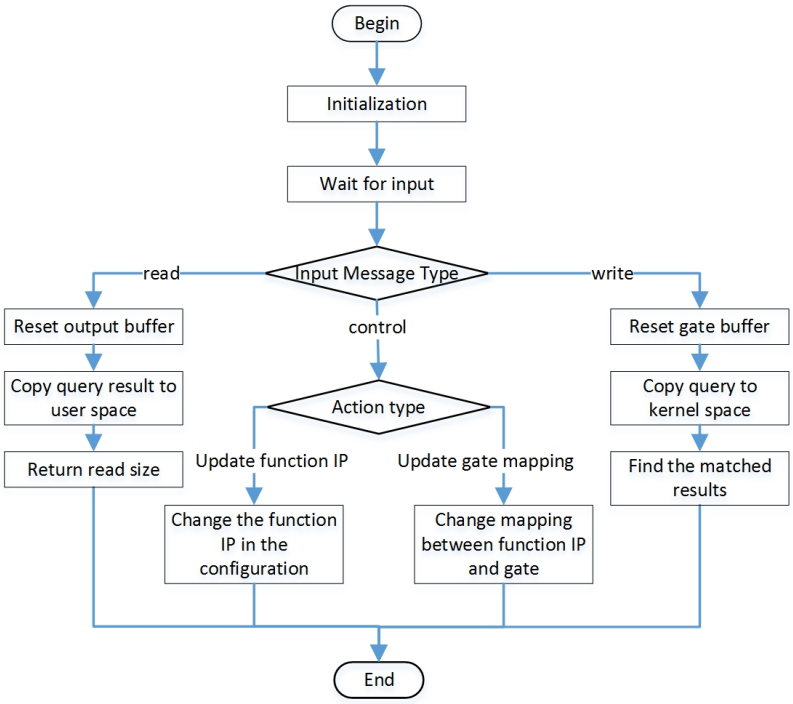

FIGURE 5.21: Workflow of the DNS Server

5.2.6 Use Case and Experimental Results

In this work, we demonstrate the functionalities of the virtual gateway in the specified use case. The emphasis is placed on the gateway functionalities after a simulated system reconfiguration.

5.2.6.1 Use Case

In this work, from the viewpoint of the applications, the use case in Figure 5.22 describes the scenario when applications P1 and P3 communicate via pre-defined VL, meanwhile P2 communicates with P4 via another VL. The defined message types and structures are also summarised in the figure. The addressing mechanism is defined at the gate level. Existing addressing mechanisms (e.g., in railway domain) are based on a federated system architecture, where one function is hosted by one device that forms the nature spatial isolation between applications, and since our proposed execution environment is based on an integrated architecture and functions are integrated on the same computing platform, the partitions containing applications build up the isolated execution environments at the system level.

We assume that the sent and received messages of a pre-defined VL are of the same structure, i.e., the gates linked to the VL have the same gate specification (e.g., gate 1 and gate 3 in Figure 5.22). In this case, the transported messages are buffered without further manipulation. Another link between gate 1 and 5 is activated by the DNS server during run time (i.e., to simulate a system reconfiguration), which results in the VL between P1 and P4 with different gate specifications of input and output

gates. The virtual gateway functionalities are activated in this case. According to the configured message type, P4 receives the RemoteTemp message (through gate 5) that contains a subset of information in the RemoteIO message, which requires the gateway to perform the message transformation.

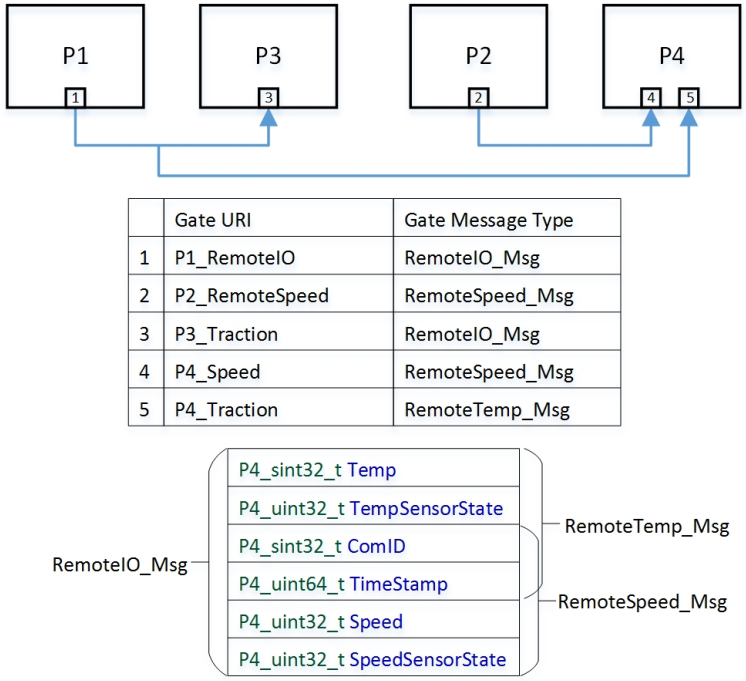

FIGURE 5.22: Use Case in the Proof-of-Concept

In this use case, the scheduling of the applications is shown in Figure 5.23. This schedule runs cyclically, and within each cycle, P1 and P2 send out messages before the control application triggers the message transmission in the kernel space. After the sending and switching phases, P3 and P4 receive the specified messages respectively.

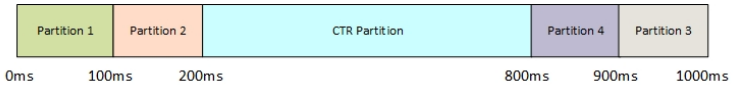

FIGURE 5.23: Example Schedule of the Applications

5.2.6.2 Experimental Results

The experiment is run on a generic X86 platform that is emulated by the QEMU emulator, and the emulation runs on the PC hardware with two cores of 2.3 GHz and 24 GB memory. In this experiment, we investigate the gateway functionalities and the side effects caused by the virtual gateway.

Figure 5.24 depicts the latency caused by the data communication mechanism through measuring the consumed time for a message to pass through the data plane. The sending applications are configured to send out 100 messages, and the control partition enables the VL between P1 and P4 after the 50th message. This represents the reconfiguration scenario in a simplified way.

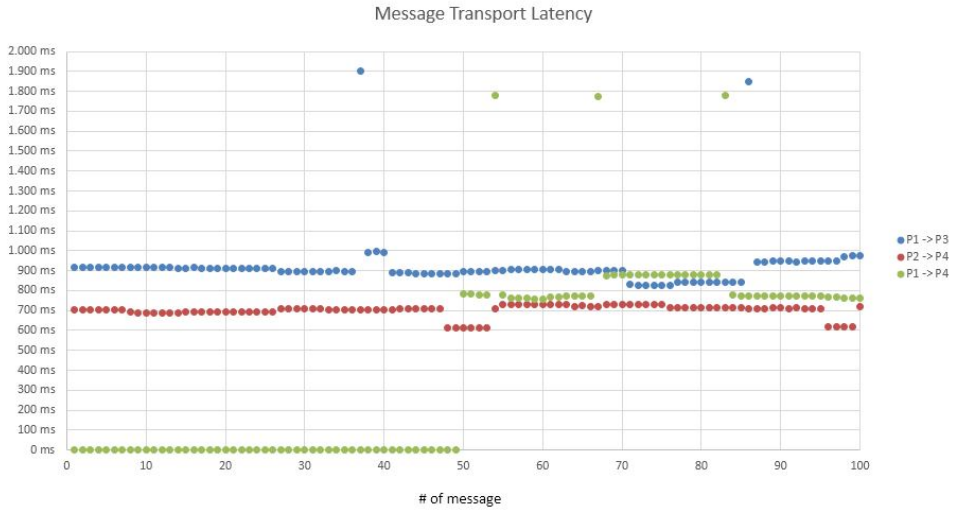

FIGURE 5.24: Message Transport Delay

As shown in the results, the measured latency for the three VLs are within the range from 600ms to 1000ms, with some exceptions to be observed. These exceptions are caused by the timing restriction of an emulated platform. The scheduling mechanism determines that the minimum delay is 600ms (e.g., between P2 and P4), because the CTR partition is assigned with a time window of 600ms. The green points before the 50th message are set to 0, just to show that there is no data transportation in this VL during this time slot.

Due to the defined message types and structures in the use case, the gateway is able to selectively redirect the messages while transforming the message properties like syntax. The transformation of temporal properties is not directly addressed in the implementation. However, the repository is implemented for the message buffering and extendable to deal with the temporal misalignment. Since we implement the error detection mechanism with respect to the time domain of the transported messages in the data plane, which majorly causes the exceptions of higher latency than 1000ms in Figure 5.24. The target PikeOS image is run on an emulated platform and there could be system interrupts delaying the message transportation. As shown in Figure 5.18, if a message with earlier time stamp is delayed before sent into the data plane, then this message is recognised as faulty in the time domain when a message with later time stamp arrives beforehand. In this situation, the faulty message is abandoned and the receiver reads the last message again, which results in the observed higher latency.

There is no observed correlation between the measured latency of the three data flows. The reason is that the temporal and spatial partitioning mechanism is extended from the user application level to the data communication level. All the VLs are specified with their own memory area and the data transportation of different data flows are free of temporal interference between each other in the proposed architecture.

5.2.7 Conclusion

The proposed virtual gateway in the execution environment based on an integrated architecture permits selective information exchange between different VLs, while preserving encapsulation of the linked data flows. More specifically, a faulty message (e.g., in the time domain) can be identified and treated according to the configured rules at the data switching level, so that an erroneous message cannot propagate to other VLs. The experimental results demonstrate the functionalities of the proposed virtual gateway, and meanwhile show the strength of the partitioning mechanism. This work extends our previous research of the system execution environment on integrated architectures and tackles the system reconfiguration problem.

5.3 Virtual Switch supporting IEEE 802.1 Qci and Qbv

In the previous section, a virtual switch supporting time-space isolation and dynamic configuration has been proposed. In this section, we propose a virtual switch that is IEEE 802.1Qbv and IEEE 802.1Qci capable according to the TSN standard, in order to close the research gap of virtual switching guaranteeing bounded delay with low jitter.

5.3.1 System Requirements

The obligatory functionality of a virtual switch is to provide message switching services. In this section, we analyze the requirements of the virtual switch that targets at dependable data transmission in an integrated architecture.

5.3.1.1 Timing Determinism

The virtual switch aims at providing a communication infrastructure for real-time applications, therefore timing determinism is one of the most important prerequisites. Along the data transmission path, the latency caused by the switching entities should be bounded with low jitter. From the viewpoint of a switching entity, the timing requirement should apply both for ingress and egress points.

5.3.1.2 Spatial Isolation

For integrated applications within a computing node, in order to exclude the spatial interference between applications of different safety-criticalities, the virtual switch should provide each application with dedicated resources to rule out unintended resource interference.

5.3.2 General Design

Inspired by the development of TSN, the virtual switch leverages the mechanisms defined in the TSN Qci and Qbv protocols to achieve the timing policing. We assume

that Time-Triggered (TT) traffic is selected for the transmission of safety-critical messages and each TT message arrives at the ingress port of the switch at a specific point in time according to the schedule. The virtual switch checks the incoming time of TT messages based on a pre-defined time-based Access Control List (ACL) and relays the temporally correct messages to the queues of their egress ports, which are determined based on the routing tables. Regarding the egress policing, the GCL defined in the Qbv protocol is specified to control the dispatching process of each egress port. Only the GCL-enabled queues are able to dispatch the buffered messages, so that no messages of different critical data flows can interleave with each other. The ACLs and GCLs within a virtual switch are aligned with each other by taking the relay overhead into account.

5.3.3 System Model

In general, we model the system from the viewpoints of the system architecture and the applications.

5.3.3.1 Architectural Model

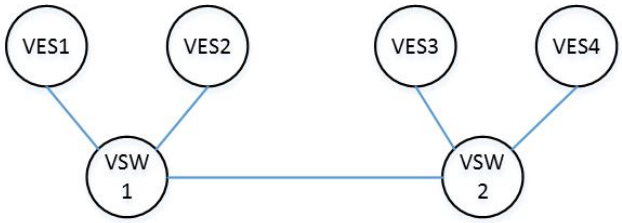

FIGURE 5.25: Architecture Model

As depicted in Figure 5.25, we define an example architecture model with multiple virtual switches (i.e., VSW1 and VSW2 in Figure 5.25) residing on an integrated computing node. In order to demonstrate the functionalities of the virtual switches, multiple virtual end systems (i.e. VESn in Figure 5.25) are defined. The virtual end systems communicate with each other via the virtual switches, which result in two communication scenarios:

- VES $\rightarrow$ VSW $\rightarrow$ VES: this scenario represents the case that the communicating virtual end systems are connected to the same virtual switch. In other words, the transmitted messages between the VESs pass through only one VSW.
- VES $\rightarrow$ VSW $\rightarrow$ VSW $\rightarrow$ VES: this scenario represents the case that the communicating virtual end systems are connected to different virtual switches on the same node, and the virtual switches are connected with each other.

5.3.3.2 Application Model

As depicted in Figure 5.26, the example application model is shown on the left side, and the task scheduling on the right side.

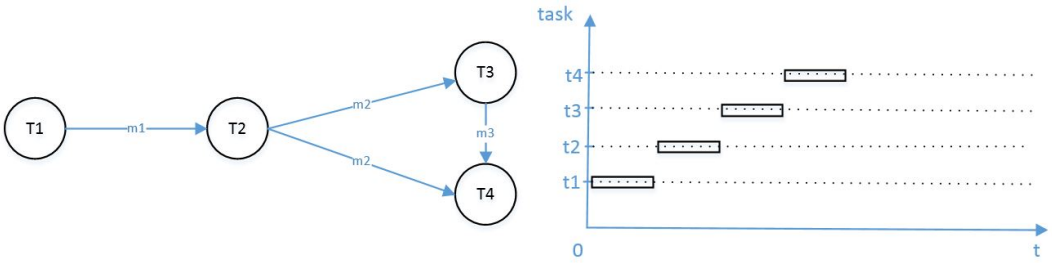

FIGURE 5.26: Application Model and Task Scheduling

In this model, the messages (i.e., m1, m2 and m3) between the tasks (i.e., T1, T2, T3 and T4) build up the task dependencies that result in the shown task scheduling. This task scheduling shows the general execution order of the tasks, while task activation and execution duration are implementation specific.

5.3.3.3 Schedule Model for Multiple Virtual Switches

From the viewpoint of the integrated platform, it is logically necessary to map the applications to the architecture, before discussing the overall schedule problem. An example mapping from the application model to the architecture model is shown in Table 5.1.

TABLE 5.1: Example Task Mapping

Task	Mapped Entity
T1	VES1
T2	VES2
T3	VES3
T4	VES4

Based on the task scheduling and the defined mapping, the schedule model containing the tasks and virtual switches is depicted in Figure 5.27. In this schedule model, the tasks and virtual switches are activated periodically and the schedule in the gantt chart represents the system schedule within one period.

The messages m2 and m3 in Figure 5.27 are dispatched simultaneously from VSW2 to t4, which result in the racing situation at the egress port connected to t4. In order to tackle this problem, we define the target dispatching algorithm for the virtual switches.

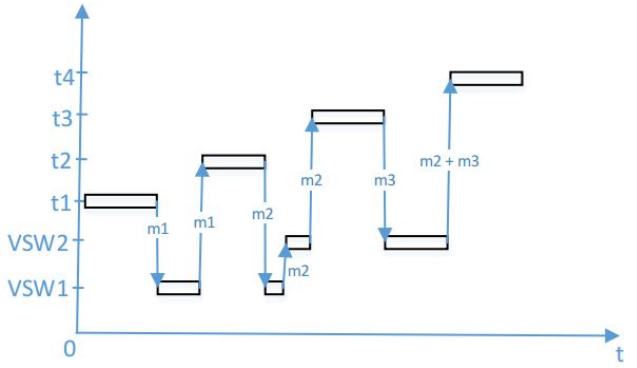

FIGURE 5.27: System Schedule Model

5.3.4 Dispatching Algorithm

Based on the schedule model, we propose the dispatching algorithm in Figure 5.28. The ingressing process is shown on the top left side, and the dispatching process is placed on the right side. The GCL execution process is placed on the bottom.

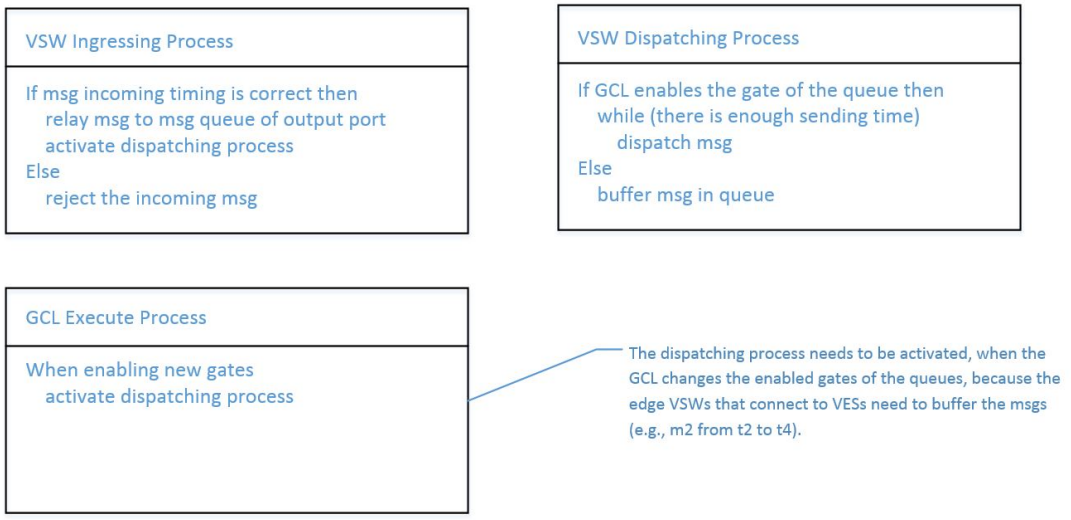

FIGURE 5.28: Ingressing and Egressing Process

If an incoming message is safety-critical, the incoming timing is checked against the configured ACL. Only the messages arriving in their assigned time slots are relayed to the egress ports. Otherwise the received messages are treated as untimely and abandoned. Right after enqueuing the messages at the egress ports, the dispatching process should be activated.

According to proposed dispatching algorithm, when the dispatching process is called, if the GCL enables the selected queue of a egress port to dispatch messages, then the selected queue is authorised for data transmission. The other prerequisite for message transmission is that there should be enough time left for the selected queue to dispatch the enqueued messages. Otherwise the messages should be buffered.

The GCL execute process is activated in time-triggered fashion, so that the dispatching process of each egress port is executed at the pre-defined points in time.

5.3.5 Virtual Switch Workflow

The workflow of the message switching within a virtual switch is depicted in Figure 5.29. An ingressing port waits for the semaphore that is assigned to the attached real-time fifo, which connects different communicating entities. Except for the timely synchronised message switching, the semaphore is also defined as the synchronisation mechanism between communicating entities. The incoming time of a message is checked against the ACL after classification and matching the egress ports. The successfully ingressed critical messages are enqueued in the egress ports and dispatched if the GCL enables the enqueued queue for data transmission, otherwise the messages are buffered.

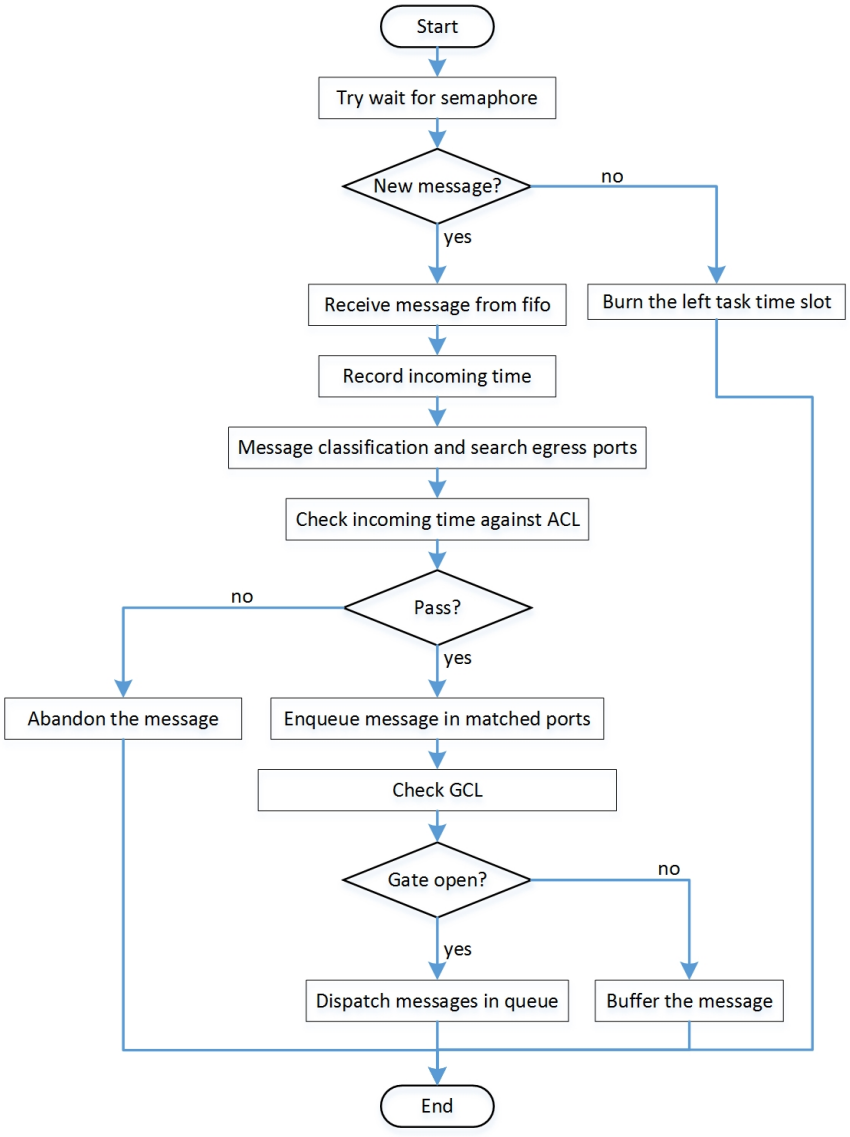

FIGURE 5.29: Workflow of Virtual Switch

5.3.6 Proof-of-Concept Implementation

In this section we describe the proof-of-concept implementation of the virtual switch, which includes the employed platform and the detailed realisation of the virtual switch. The emphasis is placed on the deterministic message switching by leveraging the TSN mechanisms.

5.3.6.1 Implementation Platform

The implementation runs on the PC platform with 16GB RAM and 6 physical cores of 3.2GHz. We deploy on this platform the real-time Linux LXRT/RTAI as the execution environment of the developed virtual switches. In order to provide dedicated computing resources for each virtual switch, we reserve two of the physical cores on the platform and run the virtual switches on the reserved cores in a one to one mapping. Therefore the context switches on the dedicated physical cores are avoided, which may cause unpredictable behavior during runtime.

5.3.6.2 Realisation of Virtual Switch

We implement a virtual switch as a kernel module, which can access the RTAI real-time scheduler and services that are implemented as kernel modules. As discussed above, one virtual switch is exclusively assigned to one physical core.

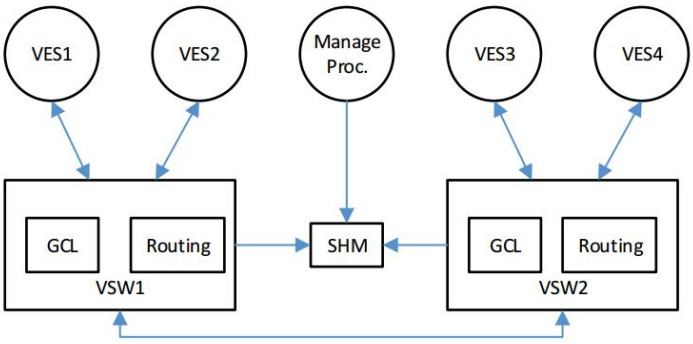

FIGURE 5.30: General Realisation

As depicted in Figure 5.30, we leverage the real-time fifos provided by Linux RTAI to implement the communication channels between virtual end systems and virtual switches, as well as the communication channels between virtual switches. The shared memory (i.e., SHM in Figure 5.30) is used to store the up-to-date configuration parameters (e.g., routing information, GCL) that are managed by the manager process and consumed by the virtual switches. From the viewpoint of the computing platform, this way contributes to the unified configuration for multiple virtual switches that reside on the same platform, and the configuration parameters are managed in a logically centralised fashion.

5.3.6.3 Related Data Structures

In this section, we discuss the major data structures of the implemented virtual switch.

From the viewpoint of a virtual switch, each egress port within the virtual switch should have its own schedule, more specifically, the GCL. In order to guarantee low jitter from design, we implement the whole set of GCLs as a static array with pre-defined maximum count of egress ports, instead of dynamically linked list. For the schedule of each egress port as shown in Listing 5.1, the GCL_duration defines the period of the cyclically enabled queues within this egress port. And this period is subdivided into time intervals (i.e., GCL_item), which are assigned with different queue masks. The queues of critical data flows are exclusively enabled to transmit messages, so that one critical data flow is timely isolated from other flows.

```
static struct egress_port_schedule
{
    struct schedule egress_port_GCL[count_egr_port];
};

static struct schedule
{
    unsigned int GCL_duration;
    struct GCL_item GCL[GCL_len];
};

static struct GCL_item
{
    int offset;
    int duration;
    uint8_t queue_mask;
};
```

LISTING 5.1: Data Structure for Schedule of Egress Port

In addition to the data structures for egress port schedule, we also define the data structures for the message buffers of the egress ports in Listing 5.2.

As discussed, the real-time fifos provided by Linux RTAI are leveraged to implement the communication channels between virtual switches, as well as channels between virtual end systems and virtual switches, we implement the egress port to be one-to-one mapped to the fifos. In other words, the queues of an egress port share the fifo attached to this port. Each queue records the number of the stored messages with a predefined maximum count. The way in our implementation to rule out messages of different critical data flows interleaving through the fifo is that each critical data flow owns one dedicated queue.

```
static struct egress_port_buf
{
    int output_fifos_id;
    struct queue all_queues[queue_count];
}egress_port;

static struct queue
{
    int item_count;
    int head_index;
    int tail_index;
    struct queue_item queue[queue_len];
}queue;

static struct queue_item
{
    int msg_len;
    char item[max_msg_len];
}queue_item;
```

LISTING 5.2: Data Structure for Buffer of Egress Port

Similarly, each ingressing message is stored in the defined vsw_ingress_buf in Listing 5.3. Each ingressed message is classified based on the switching rules to find the corresponding egress ports, which are recorded in the matched_egress_port before activating the relay process.

```
static struct vsw_ingress_buf
{
    int msg_len;
    uint8_t vsw_rv_msg[max_msg_len];
}vsw_ingress_buf;

static struct matched_egress_port
{
    int count;
    int port_cookie[count_egr_port];
}matched_egress_port;
```

LISTING 5.3: Data Structure for Buffer of Ingress Port

5.3.6.4 Temporal and Spatial Partitioning

For each virtual switch that owns a dedicated physical core, we implement a dedicated process to manage each ingressing port of the connected entities. For example, in VSW1 in Figure 5.30, we implement two processes to manage the ingressed messages from VES1 and VES2, correspondingly. As shown in Figure 5.31, each process is run exclusively for a duration of 5 ms and in a 10 ms period, which ensures the temporal isolation between processes in a virtual switch and consequently eliminates race conditions of the processes.

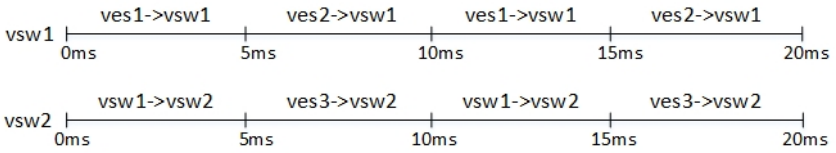

FIGURE 5.31: Task Schedule in Virtual Switch

In this implementation, assume that each egress port owns 8 queues and two of them are assigned to critical data flows, we use the example GCL (see Table 5.2) for all egress ports. For simplification purpose, we assume the frame processing time is neglectable, so that the ACL is identical to the GCL.

From the FCR point of view, since the setup is based on a single computing node that forms the single hardware FCR, the addressed aspects by the temporal and spatial partitioning mechanisms are related to software FCR, where error propagation between different VESs are ruled out in the time and value aspects.

TABLE 5.2: Implemented Gate Control List

start time	duration	queue mask	remark
0μs	30μs	10000000	time-triggered frames
30μs	30μs	01000000	time-triggered frames
60μs	40μs	00111111	other frames

5.3.6.5 Realisation of Application

In this implementation, we implement the tasks as a simple linux task sending message periodically and waiting for the incoming messages. For the delay measurement purpose, we insert a time stamp of u_int64_t type between the default Ethernet header and the data as shown in Figure 5.32. Since T1 in Figure 5.26 provides the source message of the whole setup, we configure the T1 to send the initial messages in a 5 ms period, and the other tasks work in busy waiting fashion.

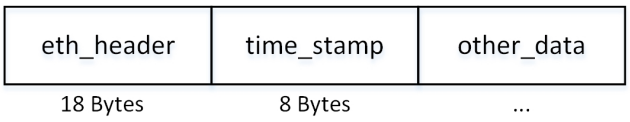

FIGURE 5.32: Extension of Data Frame

5.3.7 Experimental Results

In this experiment, we investigate the determinism of the message switching within the virtual switches. As discussed, there are two identified communication scenarios in the system setup, which happen either within single virtual switch boundary or involve multiple virtual switches.

We measure the message switching overhead caused by the virtual switches, more specifically, the consumed time from the ingress ports to the egress ports that are connected to the source and sink virtual end systems. In the experiment, the source virtual end system is configured to send out 1000 messages. The measured results in Figure 5.33 and Figure 5.34 show the overhead of the local switching within a virtual switch.

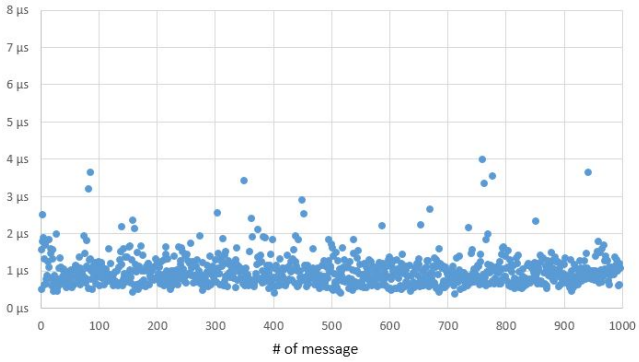

FIGURE 5.33: Overhead for Data Flow between VES1 and VES2

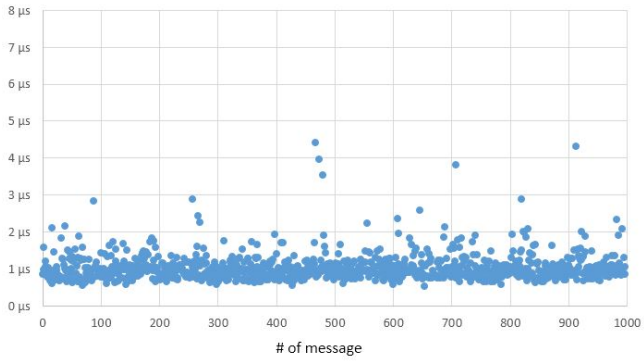

FIGURE 5.34: Overhead for Data Flow between VES3 and VES4

As shown in the results, the overall latency caused by the virtual switches for the local message transportation is in the range from 0.5 μs to 2 μs. Several exceptions up to 5 μs could be observed in Figure 5.33 and Figure 5.34. In this implementation, we isolate one physical core to run a virtual switch, as discussed in section 5.3.6.1. Since timing resource is necessary to enable the process switching, one aspect needs to be mentioned is that hardware timer interrupts are redirected to the physical cores that are not isolated in this case, which results in the inter-core communication for measuring the overhead that can cause the observable jitters. Another aspect is that the jitter caused by the RTAI scheduler can also accumulate to the significant jitters in the measured results.

Another measured overhead is of the message transmission through two virtual switches. The overhead for data flow between VES2 and VES3/VES4 in Figure 5.30 is measured one after another within VSW2, therefore the results in Figure 5.35 and Figure 5.36 are in the same distribution. The difference of the measured overhead

(depicted in Figure 5.37) for the same message in these two data flows indicates the overhead for enqueuing and dispatching a message, since a message from VES2 is relayed to VES3 and VES4 in order.

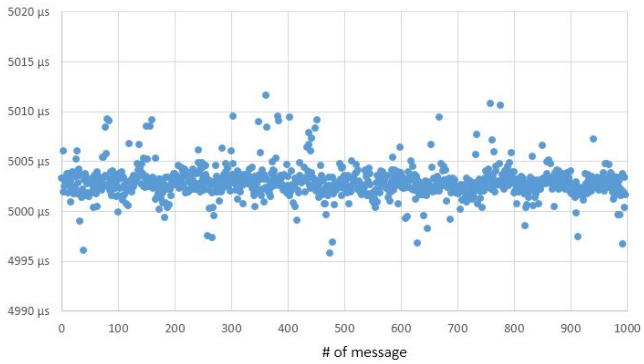

FIGURE 5.35: Overhead for Data Flow between VES2 and VES3

As discussed in section 5.3.6.4, the ingressing processes of the virtual switches are run in a 10 ms period and each with 5 ms runtime, and the relay/dispatching actions are also finished in the ingressing process context. In our configuration, the process in VSW2 for receiving messages from VSW1 is scheduled to run after the dispatching process in VSW1. In another word, the VSW2 receives messages from VSW1 in about 5 ms, after VSW1 dispatches messages to VSW2. This logical analysis is also confirmed by the results in Figure 5.35 and Figure 5.36, which show the switching overhead for a data flow passing through two virtual switches.

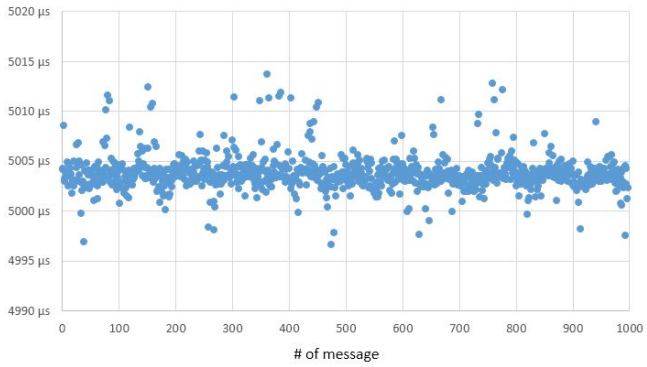

FIGURE 5.36: Overhead for Data Flow between VES2 and VES4

Similar to the measured local delay caused by virtual switches that are depicted in Figure 5.33 and Figure 5.34, the majority of the overhead in Figure 5.35 range from 5000 μs to 5005 μs. Despite the delay caused by schedule (i.e., 5 ms), the left overhead (i.e., 0-5 μs) is implementation related that need to be discussed. As aforementioned, the major reasons for this observable overhead are inter-core communication due to timer on other physical core and the jitter caused by RTAI scheduler. For this inter virtual switches communication scenario, jitters could accumulate along the message switching path, which result in the wider range of overhead (5 μs) than the intra virtual switch communication (2 μs).

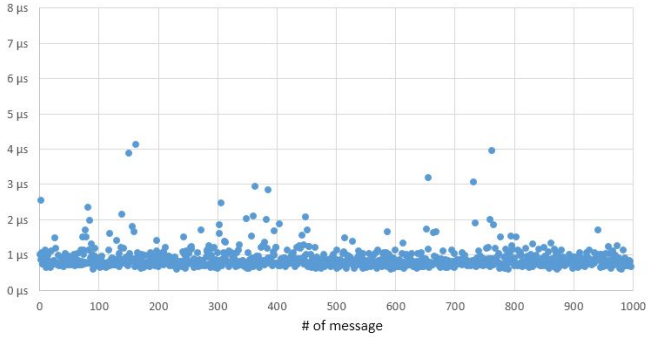

FIGURE 5.37: Overhead for Enqueuing and Dispatching of one Message

5.3.8 Conclusion

In this section, we propose the IEEE 802.1Qbv and IEEE 802.1Qci enabled virtual switch for integrated real-time systems residing on single platform and define the model of communication infrastructure with multiple virtual switches. The temporal and spatial isolation mechanisms at the data switching level rule out the error propagation between different data flows. In the realisation of the virtual switch, the configuration of the setup is managed in a centralised fashion, which enables the logically centralised control of the switching entities that can be extended in a physically networked environment to tackle the unified system reconfiguration problem. The corresponding schedule model is defined as the base for the dispatching algorithm in a time-triggered way. The proof-of-concept implementation is done in a resource dedicated way by leveraging the Linux RTAI patch. The experimental results demonstrate the capability of the virtual switch to switch messages in a timely deterministic way.

Chapter 6

Conclusion

Integrating applications on a shared computing node leads to a mixed-criticality system, since safety-critical applications can be combined with the non-critical ones. Integration of mixed-critical applications with reconfiguration requirements is desirable, since domain-specific applications (e.g., in the railway domain) require system reconfiguration during runtime. Existing execution environments based on an integrated architecture support only static system configurations. This dissertation proposed an execution environment for an integrated architecture, which leverages the SDN paradigm to support dynamic time-triggered communication. The proposed execution environment guarantees the safe integration of mixed-criticality applications in an integrated system and also addresses the system reconfiguration requirement.

Furthermore, the virtual data communication for integrated real-time applications was also addressed in this dissertation. Beyond the static communication mechanisms in the state-of-the-art (cf. Section 3.2, Section 3.3), a virtual switch that ensures temporal and spatial isolation between data flows of the integrated applications hosted on the same computing node was proposed. The proposed virtual switch leverages the SDN paradigm to support dynamic data communication, and the controlled data exchange between different data flows is supported by the proposed virtual gateway. In the proof-of-concept implementation, the virtual switch was implemented as kernel modules of PikeOS, which is an RTOS certified to the highest criticality levels. The fundamental isolation mechanisms and determinism of message switching were demonstrated, while the caused overhead for message transmission and controlled data exchange were also evaluated.

In order to close the research gap of virtual switching guaranteeing bounded delay with low jitter, the virtual switch supporting IEEE 802.1 Qci and Qbv was proposed. The model of the communication infrastructure with multiple virtual switches was defined, and the corresponding schedule model was proposed as the base for the dispatching algorithm that functions in a time-triggered way. The proof-of-concept implementation was done in a time-triggered way by leveraging the Linux RTAI patch. The experimental results demonstrate the capability of the virtual switch to switch messages in a timely deterministic way, where the measured overhead is less than 10 μs.

In this dissertation, we concentrated on the data communication on a computing node. One further research direction is to extend this work in physically networked environments, where the data communication involving virtual switches and physical switches is still an open research problem. The future work will be the hybrid switching environment with physical and virtual switches to enable a physically networked system, which also brings the challenges in the consistent dynamic configuration of the whole system. The SDN-based execution environment enables the logically centralized control of physically distributed switching entities within physically connected computing nodes, which will address the future research challenge.

Bibliography

[1] Anand V Akella and Kaiqi Xiong. "Quality of service (QoS)-guaranteed network resource allocation via software defined networking (SDN)". In: *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*. IEEE. 2014, pp. 7–13.

[2] Ahmad Al Sheikh, Olivier Brun, and Pierre-Emmanuel Hladik. "Partition scheduling on an IMA platform with strict periodicity and communication delays". In: *18th international conference on real-time and network systems*. 2010, pp. 179–188.

[3] Thomas Anderson et al. "Overcoming the Internet impasse through virtualization". In: *Computer* 4 (2005), pp. 34–41.

[4] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. "Fundamental concepts of computer system dependability". In: *Workshop on Robot Dependability: Technological Challenge of Dependable Robots in Human Environments*. 2001, pp. 1–16.

[5] Algirdas Avizienis et al. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33.

[6] Vaibhav Bajpai and Jürgen Schönwälder. "NETCONF Interoperability Lab". In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE. 2014, pp. 1–2.

[7] Manu Bansal et al. "Openradio: a programmable wireless dataplane". In: *Proceedings of the first workshop on Hot topics in software defined networks*. ACM. 2012, pp. 109–114.

[8] Basil Becker et al. "Model-based extension of autosar for architectural online reconfiguration". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2009, pp. 83–97.

[9] Christian Berger and Matthias Tichy. "Towards transactional self-adaptation for AUTOSAR on the example of a collision detection system". In: *INFORMATIK 2012* (2012).

[10] Pierre Bieber et al. "Preliminary design of future reconfigurable IMA platforms". In: *ACM Sigbed Review* 6.3 (2009), p. 7.

[11] Pierre Bieber et al. "Preliminary design of future reconfigurable IMA platforms-safety assessment". In: *27th Congress International Council of the Aeronautical Sciences (ICAS 2010)*. 2010.

[12] Martin Bjorklund. "YANG-a data modeling language for the network configuration protocol (NETCONF)". In: (2010).

[13] Uwe Brinkschulte, Etienne Schneider, and Florentin Picioroaga. "Dynamic real-time reconfiguration in distributed systems: timing issues and solutions". In: *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*. IEEE. 2005, pp. 174–181.

[14] Stefan Bunzel. "Autosar–the standardized software architecture". In: *Informatik-Spektrum* 34.1 (2011), pp. 79–83.

[15] Alan Burns and Robert Davis. "Mixed criticality systems-a review". In: *Department of Computer Science, University of York, Tech. Rep* (2013), pp. 1–69.

[16] Andrew T Campbell et al. "A survey of programmable networks". In: *ACM SIGCOMM Computer Communication Review* 29.2 (1999), pp. 7–23.

[17] Hao Chen et al. "Research on Client/Server Communication Mechanism in AUTOSAR System". In: *2013 IEEE 11th International Conference on Dependable, Autonomic and Secure Computing*. IEEE. 2013, pp. 220–226.

[18] Margaret Chiosi et al. "Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action". In: *SDN and OpenFlow World Congress*. Vol. 48. sn. 2012.

[19] Eu-Teum Choi et al. "Detecting Atomicity Races in ARINC 653 Applications". In: *2015 8th International Conference on Grid and Distributed Computing (GDC)*. IEEE. 2015, pp. 24–27.

[20] NM Mosharaf Kabir Chowdhury and Raouf Boutaba. "A survey of network virtualization". In: *Computer Networks* 54.5 (2010), pp. 862–876.

[21] Brian A Coan, Brian M Oki, and Elliot K Kolodner. "Limitations on database availability when networks partition". In: *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*. ACM. 1986, pp. 187–194.

[22] AEE Committee et al. "Aircraft Data Network Part 7, Avionics Full Duplex Switched Ethernet (AFDX) Network, ARINC Specification 664". In: *Annapolis, Maryland: Aeronautical Radio* (2002).

[23] Airlines Electronic Engineering Committee. "Avionics application software standard interface part 1-required services". In: *ARINC Document ARINC Specification 653P1-4, Aeronautical Radio, Inc., Maryland* (2015).

[24] Alfons Crespo, Ismael Ripoll, and Miguel Masmano. "Partitioned embedded architecture based on hypervisor: The xtratum approach". In: *2010 European Dependable Computing Conference*. IEEE. 2010, pp. 67–72.

[25] Wang Dafang et al. "Communication mechanisms on the virtual functional bus of AUTOSAR". In: *2010 International Conference on Intelligent Computation Technology and Automation*. Vol. 1. IEEE. 2010, pp. 982–985.

[26] Fabio Fabian Daitx, Rafael Pereira Esteves, and Lisandro Zambenedetti Granville. "On the use of SNMP as a management interface for virtual networks". In: *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*. IEEE. 2011, pp. 177–184.

[27] Saurav Das, Guru Parulkar, and Nick McKeown. "Simple unified control for packet and circuit networks". In: *2009 IEEE/LEOS Summer Topical Meeting*. IEEE. 2009, pp. 147–148.

[28] Saurav Das, Guru Parulkar, and Nick McKeown. "Unifying packet and circuit switched networks". In: *2009 IEEE Globecom Workshops*. IEEE. 2009, pp. 1–6.

[29] Tatjana Davidovi et al. *Mathematical programming-based approach to scheduling of communicating tasks*. GERAD, HEC Montréal, 2004.

[30] Bruce Davie and Jesse Gross. "A stateless transport tunneling protocol for network virtualization (STT)". In: *Mar* 5 (2012), pp. 1–19.

[31] RTCA DO. "178C, Software Considerations in Airborne Systems and Equipment Certification. RTCA". In: *Inc., Washington, DC, USA (December 1992)* (2011).

[32] Christian El Salloum et al. "The ACROSS MPSoC–A new generation of multi-core processors designed for safety–critical embedded systems". In: *Microprocessors and Microsystems* 37.8 (2013), pp. 1020–1032.

[33] Christian Engel et al. "Enhanced dispatchability of aircrafts using multi-static configurations". In: *Embedded Real Time Software and Systems Congress (ERTS 2010), Toulouse, France*. 2010.

[34] Rob Enns, Martin Bjorklund, and Juergen Schoenwaelder. "Network configuration protocol (NETCONF)". In: *Network* (2011).

[35] Rafael Pereira Esteves, Lisandro Zambenedetti Granville, and Raouf Boutaba. "On the management of virtual networks". In: *IEEE Communications Magazine* 51.7 (2013), pp. 80–88.

[36] NFVGS ETSI. "Network functions virtualisation (nfv); management and orchestration". In: *NFV-MAN* 1 (2014), p. v0.

[37] Andrew D Ferguson et al. "Participatory networking: An API for application control of SDNs". In: *ACM SIGCOMM computer communication review*. Vol. 43. 4. ACM. 2013, pp. 327–338.

[38] Roscoe Ferguson. "Middleware Technique to Synchronize Applications Across Modules Using an ARINC 653 RTOS". In: *AIAA SPACE 2007 Conference & Exposition*. 2007, p. 6187.

[39] Stuart Fisher. "Certifying applications in a multi-core environment: The world's first multi-core certification to sil 4". In: *SYSGO white paper* (2013).

[40] Nate Foster et al. "Frenetic: a high-level language for OpenFlow networks". In: *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*. ACM. 2010, p. 6.

[41] Simon Fürst et al. "AUTOSAR–A Worldwide Standard is on the Road". In: *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*. Vol. 62. 2009, p. 5.

[42] Pankaj Garg and Y Wang. *NVGRE: Network virtualization using generic routing encapsulation*. Tech. rep. 2015.

[43] Arne Haber et al. "Hierarchical variability modeling for software architectures". In: *2011 15th International Software Product Line Conference*. IEEE. 2011, pp. 150–159.

[44] Zongjian He and Guanqing Liang. "Research and evaluation of network virtualization in cloud computing environment". In: *2012 Third International Conference on Networking and Distributed Computing*. IEEE. 2012, pp. 40–44.

[45] Abdelsalam Heddaya, Abdelsalam Helal, et al. *Reliability, availability, dependability and performability: A user-centered view*. Tech. rep. Boston University Computer Science Department, 1997.

[46] Ricardo Hillbrecht and Luis Carlos E de Bona. "A SNMP-based virtual machines management interface". In: *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*. IEEE Computer Society. 2012, pp. 279–286.

[47] Timothy L Hinrichs et al. "Practical declarative network management". In: *Proceedings of the 1st ACM workshop on Research on enterprise networking*. ACM. 2009, pp. 1–10.

[48] Michio Honda et al. "mSwitch: a highly-scalable, modular software switch". In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. ACM. 2015, p. 1.

[49] IEC61508 IEC et al. "Functional safety of electrical/electronic/programmable electronic safety-related systems". In: *BS IEC* 61508 (1998).

[50] ISO26262 ISO. "26262: Road vehicles-Functional safety". In: *International Standard ISO/FDIS* 26262 (2011).

[51] Raj Jain and Subharthi Paul. "Network virtualization and software defined networking for cloud computing: a survey". In: *IEEE Communications Magazine* 51.11 (2013), pp. 24–31.

[52] M Jakovljevic. "Deterministic Ethernet: SAE AS6802" Time-Triggered Ethernet". In: *SAE International, Nov* (2011).

[53] Kwangtae Jeong, Jinwook Kim, and Young-Tak Kim. "QoS-aware network operating system for software defined networking with generalized OpenFlows". In: *2012 IEEE Network Operations and Management Symposium*. IEEE. 2012, pp. 1167–1174.

[54] Robert Kaiser. "Combining partitioning and virtualization for safety-critical systems". In: *White Paper WP CPV* 10 (2007), A4.

[55] Roland Kammerer, Roman Obermaisser, and Bernhard Fromel. "Dynamic configuration of a time-triggered router for controller area network". In: *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*. IEEE. 2012, pp. 1–10.

[56] Frank Kargl, Zhendong Ma, and Elmar Schoch. "Security engineering for vanets". In: *4th Workshop on Embedded Security in Cars (escar 2006)*. Citeseer. 2006.

[57] Elliott Karpilovsky et al. "Practical network-wide compression of IP routing tables". In: *IEEE Transactions on Network and Service Management* 9.4 (2012), pp. 446–458.

[58] Krishna Kavi, Robert Akl, and Ali Hurson. "Real-Time Systems: An Introduction and the State-of-the-Art". In: *Wiley Encyclopedia of Computer Science and Engineering* (2007), pp. 2369–2377.

[59] Marc Koerner and Odej Kao. "Multiple service load-balancing with OpenFlow". In: *2012 IEEE 13th International Conference on High Performance Switching and Routing*. IEEE. 2012, pp. 210–214.

[60] Hermann Kopetz. "Fault containment and error detection in the time-triggered architecture". In: *Autonomous Decentralized Systems, 2003. ISADS 2003. The Sixth International Symposium on*. IEEE. 2003, pp. 139–146.

[61] Hermann Kopetz. *From a federated to an integrated architecture for dependable embedded systems*. Tech. rep. TECHNISCHE UNIV VIENNA (AUSTRIA), 2004.

[62] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[63] Radek Krejci. "Building NETCONF-enabled network management systems with libnetconf". In: *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. IEEE. 2013, pp. 756–759.

[64] Jaynarayan H Lala and Richard E Harper. "Architectural principles for safety-critical real-time applications". In: *Proceedings of the IEEE* 82.1 (1994), pp. 25–40.

[65] J-C Laprie. "Dependability of computer systems: concepts, limits, improvements". In: *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*. IEEE. 1995, pp. 2–11.

[66] Jean-Claude Laprie. "Dependability: Basic concepts and terminology". In: *Dependability: Basic Concepts and Terminology*. Springer, 1992, pp. 3–245.

[67] Jean-Claude Laprie. "Dependable computing: Concepts, limits, challenges". In: *Special Issue of the 25th International Symposium On Fault-Tolerant Computing*. 1995, pp. 42–54.

[68] Thomas J LeBlanc and Evangelos P Markatos. "Shared memory vs. message passing in shared-memory multiprocessors". In: *[1992] Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*. IEEE. 1992, pp. 254–263.

[69] Sang-Hun Lee, Sanghyun Han, and Hyun-Wook Jin. "A Configurable, Extensible Implementation of Inter-Partition Communication for Integrated Modular Avionics". In: *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE. 2012, pp. 453–458.

[70] Hugo Lhachemi et al. "Partition modeling and optimization of ARINC 653 operating systems in the context of IMA". In: *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. IEEE. 2016, pp. 1–8.

[71] Ladislav Lhotka and Jiri Novotny. *Netopeer*. Tech. rep. Technical report, Cesnet. Available from: http://www. liberouter. org/netopeer, 2002.

[72] Chung Laung Liu and James W Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment". In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61.

[73] Lei Liu et al. "OpenFlow-based wavelength path control in transparent optical networks: A proof-of-concept demonstration". In: *2011 37th European Conference and Exhibition on Optical Communication*. IEEE. 2011, pp. 1–3.

[74] Rongshen Long et al. "An approach to optimize intra-ecu communication based on mapping of autosar runnable entities". In: *2009 International Conference on Embedded Software and Systems*. IEEE. 2009, pp. 138–143.

[75] Víctor López-Jaquero et al. "Supporting ARINC 653-based dynamic reconfiguration". In: *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. IEEE. 2012, pp. 11–20.

[76] Guohan Lu et al. "Using CPU as a traffic co-processing unit in commodity switches". In: *Proceedings of the first workshop on Hot topics in software defined networks*. ACM. 2012, pp. 31–36.

[77] Yan Luo et al. "Accelerating OpenFlow switching with network processors". In: *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM. 2009, pp. 70–71.

[78] Chris A Mack. "Fifty years of Moore's law". In: *IEEE Transactions on semiconductor manufacturing* 24.2 (2011), pp. 202–207.

[79] Mallik Mahalingam et al. *Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks*. Tech. rep. 2014.

[80] Reihaneh Torkzadeh Mahani and Negin Mahani. "VxWorks vs. LynxOS Real-Time Operating Systems for Embedded Systems". In: (2015).

[81] Eric Mannie. "Generalized multi-protocol label switching (GMPLS) architecture". In: (2004).

[82] Paolo Mantegazza et al. "RTAI: Real-time application interface". In: (2000).

[83] Giancarlo Martella, B Ronchetti, and Fabio A Schreiber. "Availability evaluation in distributed database systems". In: *Performance Evaluation* 1.4 (1981), pp. 201–211.

[84] Keith McCloghrie and Marshall Rose. "Management Information Base for network management of TCP/IP-based internets: MIB-II". In: (1991).

[85] Rick McGeer. "A safe, efficient update protocol for OpenFlow networks". In: *Proceedings of the first workshop on Hot topics in software defined networks*. ACM. 2012, pp. 61–66.

[86] S Baskiyar N Meghanathan. "A survey of contemporary real-time operating systems". In: *Informatica* 29.2 (2005).

[87] Adnan Noor Mian et al. "Effects of virtualization on network and processor performance using open vswitch and xen server". In: *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE. 2014, pp. 762–767.

[88] Rashid Mijumbi et al. "Network function virtualization: State-of-the-art and research challenges". In: *IEEE Communications surveys & tutorials* 18.1 (2015), pp. 236–262.

[89] Rohan Murty et al. "Dyson: An Architecture for Extensible Wireless LANs." In: *Usenix annual technical conference*. 2010.

[90] T Narten et al. *Problem statement: Overlays for network virtualization*. Tech. rep. 2014.

[91] Roman Obermaisser. "Formal specification of gateways in integrated architectures". In: *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer. 2008, pp. 34–45.

[92] Roman Obermaisser and Bernhard Leiner. "Temporal and spatial partitioning of a time-triggered operating system based on real-time Linux". In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. IEEE. 2008, pp. 429–435.

[93] Roman Obermaisser and Philipp Peti. "A fault hypothesis for integrated architectures". In: *Intelligent Solutions in Embedded Systems, 2006 International Workshop on*. IEEE. 2006, pp. 1–18.

[94] Roman Obermaisser and Philipp Peti. "Realization of virtual networks in the DECOS integrated architecture". In: *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE. 2006, 8–pp.

[95] Roman Obermaisser, Philipp Peti, and Hermann Kopetz. "Virtual gateways in the DECOS integrated architecture". In: *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. IEEE. 2005, 8–pp.

[96] Roman Obermaisser, Philipp Peti, and Fulvio Tagliabo. "An integrated architecture for future car generations". In: *Real-Time Systems* 36.1-2 (2007), pp. 101–133.

[97] Roman Obermaisser and Donatus Weber. "Architectures for mixed-criticality systems based on networked multi-core chips". In: *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*. IEEE. 2014, pp. 1–10.

[98] Roman Obermaisser et al. "DECOS: an integrated time-triggered architecture". In: *e & i Elektrotechnik und Informationstechnik* 123.3 (2006), pp. 83–95.

[99] Roman Obermaisser et al. "From a federated to an integrated automotive architecture". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.7 (2009), pp. 956–965.

[100] Tian Pan et al. "ALFE: A replacement policy to cache elephant flows in the presence of mice flooding". In: *2012 IEEE International Conference on Communications (ICC)*. IEEE. 2012, pp. 2961–2965.

[101] Ya-Shiang Peng and Yen-Cheng Chen. "SNMP-based monitoring of heterogeneous virtual infrastructure in clouds". In: *2011 13th Asia-Pacific Network Operations and Management Symposium*. IEEE. 2011, pp. 1–6.

[102] Juan R Pimentel. "Safety-reliability of distributed embedded system fault tolerant units". In: *Industrial Electronics Society, 2003. IECON'03. The 29th Annual Conference of the IEEE*. Vol. 1. IEEE. 2003, pp. 945–950.

[103] Francesco Poletti et al. "Energy-efficient multiprocessor systems-on-chip for embedded computing: Exploring programming models and their architectural support". In: *IEEE Transactions on Computers* 56.5 (2007), pp. 606–621.

[104] Fred J Pollack. "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies". In: *micro*. IEEE. 1999, p. 2.

[105] Razvan Racu et al. "Automotive software integration". In: *Proceedings of the 44th annual Design Automation Conference*. ACM. 2007, pp. 545–550.

[106] Mark Reitblatt et al. "Consistent updates for software-defined networks: Change you can believe in!" In: *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM. 2011, p. 7.

[107] Luigi Rizzo and Giuseppe Lettieri. "Vale, a switched ethernet for virtual machines". In: *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM. 2012, pp. 61–72.

[108] Joaquim Rosa, Joao Craveiro, and José Rufino. "Safe online reconfiguration of time-and space-partitioned systems". In: *2011 9th IEEE International Conference on Industrial Informatics*. IEEE. 2011, pp. 510–515.

[109] Inc RTCA. *Design Assurance Guidance for Airborne Electronic Hardware*. RTCA, Incorporated, 2000.

[110] John Rushby. *Partitioning in avionics architectures: Requirements, mechanisms, and assurance*. Tech. rep. SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB, 2000.

[111] Paulo Roberto da Paz Ferraz Santos, Rafael Pereira Esteves, and Lisandro Zambenedetti Granville. "Evaluating SNMP, NETCONF, and RESTful web services for router virtualization management". In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE. 2015, pp. 122–130.

[112] Robert R Schaller. "Moore's law: past, present and future". In: *IEEE spectrum* 34.6 (1997), pp. 52–59.

[113] Jurgen Schonwalder, Martin Bjorklund, and Phil Shafer. "Network configuration management using NETCONF and YANG". In: *IEEE communications magazine* 48.9 (2010), pp. 166–173.

[114] Rob Sherwood et al. "Flowvisor: A network virtualization layer". In: *OpenFlow Switch Consortium, Tech. Rep* 1 (2009), p. 132.

[115] Thanikesavan Sivanthi and Ulrich Killat. "A satisficing momip framework for reliable real-time application scheduling". In: *2006 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing*. IEEE. 2006, pp. 187–194.

[116] E Stelzer et al. "Virtual router management information base using smiv2". In: *draft-ietf-ppvpn-vr-mib-05* (2003), p. l9.

[117] Elisabeth A Strunk and John C Knight. "Dependability through assured reconfiguration in embedded system software". In: *IEEE Transactions on Dependable and Secure Computing* 3.3 (2006), pp. 172–187.

[118] Johnathan Swingler, John W McBride, and Christian Maul. "Degradation of road tested automotive connectors". In: *IEEE Transactions on Components and Packaging Technologies* 23.1 (2000), pp. 157–164.

[119] Voravit Tanyingyong, Markus Hidell, and Peter Sjödin. "Improving pc-based openflow switching performance". In: *2010 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2010, pp. 1–2.

[120] Voravit Tanyingyong, Markus Hidell, and Peter Sjödin. "Using hardware classification to improve pc-based openflow switching". In: *2011 IEEE 12th International Conference on High Performance Switching and Routing*. IEEE. 2011, pp. 215–221.

[121] Sivanthi Thanikesavan and Ulrich Killat. "Global scheduling of periodic tasks in a decentralized real-time control system". In: *IEEE International Workshop on Factory Communication Systems, 2004. Proceedings.* IEEE. 2004, pp. 307–310.

[122] Sivanthi Thanikesavan and Ulrich Killat. "Static scheduling of periodic tasks in a decentralized real-time control system using an ilp". In: *11th International Conference on Parallel and Distributed Systems (ICPADS'05)*. Vol. 2. IEEE. 2005, pp. 639–643.

[123] Amin Tootoonchian and Yashar Ganjali. "Hyperflow: A distributed control plane for openflow". In: *Proceedings of the 2010 internet network management conference on Research on enterprise networking*. Vol. 3. 2010.

[124] Amin Tootoonchian, Monia Ghobadi, and Yashar Ganjali. "OpenTM: traffic matrix estimator for OpenFlow networks". In: *International Conference on Passive and Active Network Measurement*. Springer. 2010, pp. 201–210.

[125] Kishor S Trivedi. *Probability & statistics with reliability, queuing and computer science applications*. John Wiley & Sons, 2008.

[126] Wolfgang Trumler et al. *Self-configuration and Self-healing in AUTOSAR*. Tech. rep. SAE Technical Paper, 2007.

[127] Tina Tsou et al. "Management Information Base for Virtual Machines Controlled by a Hypervisor". In: *Management* (2015).

[128] Daniel Turull, Markus Hidell, and Peter Sjödin. "Using libNetVirt to control the virtual network". In: *2012 IEEE 1st International Conference on Cloud Networking (CLOUDNET)*. IEEE. 2012, pp. 148–152.

[129] Tore Ulversoy. "Software defined radio: Challenges and opportunities". In: *IEEE Communications Surveys & Tutorials* 12.4 (2010), pp. 531–550.

[130] Steven H VanderLeest. "Taming interrupts: deterministic asynchronicity in an ARINC 653 environment". In: *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*. IEEE. 2014, 8A3–1.

[131] K Voderhobli. *Achieving the green theme through the use of traffic characteristics in data centers, green information technology—A sustainable approach*. 2015.

[132] Kiran Voderhobli. "Adoption of a Legacy Network Management Protocol for Virtualisation". In: *Strategic Engineering for Cloud Computing and Big Data Analytics*. Springer, 2017, pp. 141–153.

[133] Andreas Voellmy and Paul Hudak. "Nettle: A language for configuring routing networks". In: *IFIP Working Conference on Domain-Specific Languages*. Springer. 2009, pp. 211–235.

[134] Marco Wagner, Dieter Zöbel, and Ansgar Meroth. "SODA: Service-oriented architecture for runtime adaptive driver assistance systems". In: *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE. 2014, pp. 150–157.

[135] Marco Wagner, Dieter Zobel, and Ansgar Meroth. "Towards runtime adaptation in AUTOSAR: adding Service-orientation to automotive software architecture". In: *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. IEEE. 2014, pp. 1–7.

[136] Richard Wang, Dana Butnariu, Jennifer Rexford, et al. "OpenFlow-Based Server Load Balancing Gone Wild." In: *Hot-ICE* 11 (2011), pp. 12–12.

[137] Wenfeng Xia et al. "A survey on software-defined networking". In: *IEEE Communications Surveys & Tutorials* 17.1 (2014), pp. 27–51.

[138] H Yin et al. "Sdni: A message exchange protocol for software defined networks (sdns) across multiple domains". In: *IETF draft, work in progress* (2012).

[139] James Yu and Imad Al Ajarmeh. "An empirical study of the NETCONF protocol". In: *2010 Sixth International Conference on Networking and Services*. IEEE. 2010, pp. 253–258.

[140] Marc Zeller et al. "Towards runtime adaptation in AUTOSAR". In: *ACM SIGBED Review* 10.4 (2013), pp. 17–20.

[141] B Zores. *R. State, and O. Festor," YENCA", https://sourceforge.net/projects/yenca/, 2019-07-15*.

[142] Alexander Zuepke, Marc Bommert, and Daniel Lohmann. "AUTOBEST: a united AUTOSAR-OS and ARINC 653 kernel". In: *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2015, pp. 133–144.

List of Abbreviations

ABS	**A**nti-lock **B**raking **S**ystem
ACL	**A**ccess **C**ontrol **L**ist
ALFE	**A**daptive **L**east **F**requently **E**victed
APEX	**AP**plication **EX**ecutive
ARINC	**A**eronautical **R**adio, **INC**orporated
ASIC	**A**pplication-**S**pecific **I**ntegrated **C**ircuit
AUTOSAR	**AUT**omotive **O**pen **S**ystem **AR**chitecture
ASN	**A**bstract **S**yntax **N**otion
BAG	**B**andwidth **A**llocation **G**ap
BMA	**B**ogie **M**onitoring **A**pplication
CAPEX	**CAP**ital **EX**penses
CLI	**C**ommand **L**ine **I**nterface
CO	**C**rash/**O**mission
COM DRV	**COM**munication **DR**i**V**er
COTS	**C**ommercial **O**ff-**T**he-**S**helf
C/S	**C**lient/**S**erver
DECOS	**D**ependable **E**mbedded **CO**mponents and **S**ystems
DHCP	**D**ynamic **H**ost **C**onfiguration **P**rotocol
DIANA	**D**istributed, equipment **I**ndependent environment for **A**dvanced avio**N**ics **A**pplications
ECU	**E**lectronic **C**ontrol **U**nit
EDF	**E**arliest **D**eadline **F**irst
EMI	**E**lectro-**M**agnetic **I**nterference
ETSI	**E**uropean **T**elecommunications **S**tandards **I**nstitute
FCR	**F**ault-**C**ontainment **R**egion
FML	**F**low-based **M**anagement **L**anguage
GCL	**G**ate **C**ontrol **L**ist
GMPLS	**G**eneralized **M**ulti-**P**rotocol **L**abel **S**witching
IETF	**I**nternet **E**ngineering **T**ask **F**orce
IMA	**I**ntegrated **M**odular **A**vionics
IOS	**I**nternetwork **O**perating **S**ystem
LAN	**L**ocal **A**rea **N**etwork
MANO	**MAN**agement add **O**rchestration
MIB	**M**anagement **I**nformation **B**ase
MILP	**M**ixed **I**nteger **L**inear **P**rogramming

MMU	**M**emory **M**anagement **U**nit
MPSoC	**M**ulti-**P**rocessor **S**ystems-on-**C**hip
MPU	**M**emory **P**rotection **U**nit
MTF	**M**ajor **T**ime **F**rame
NETCONF	**NET**work **C**onfiguration **P**rotocol
NFV	**N**etwork **F**unction **V**irtualization
NIC	**N**etwork **I**nterface **C**ard
NVGRE	**N**etwork **V**irtualization using **G**eneric **R**outing **E**ncapsulation
ONF	**O**pen **N**etworking **F**oundation
OPEX	**OP**erating **EX**penses
PSSW	**P**ikeOS **S**ystem **S**oft**W**are
QoS	**Q**uality of **S**ervice
RA	**R**ogue **A**pplication
RPC	**R**emote **P**rocedure **C**all
RTAI	**R**eal**T**ime **A**pplication **I**nterface
RTE	**R**un**T**ime **E**nvironment
RTOS	**R**eal **T**ime **O**perating **S**ystem
SCARLETT	**SCA**lable & **R**ecofigurab**L**e **E**lectronics pla**T**forms and **T**ools
SDN	**S**oftware-**D**efined **N**etworking
SHM	**SH**ared **M**emory
SIL	**S**afety **I**ntegrity **L**evel
SNMP	**S**imple **N**etwork **M**anagement **P**rotocol
STT	**S**tateless **T**ransport **T**unneling
SODA	**S**ervice-**O**riented **D**river **A**ssistance
SoS	**S**lightly-off-**S**pecification
TCMS	**T**rain **C**ontrol and **M**onitoring **S**ystem
TDM	**T**ime-**D**ivision **M**ultiplexing
TM	**T**raffic **M**atrix
TRILL	**TR**ansparent **I**nterconnection of **L**ots of **L**inks
TSN	**T**ime **S**ensitive **N**etworking
TT	**T**ime-**T**riggered
URI	**U**niform **R**eource **I**dentifier
VES	**V**irtual **E**nd **S**ystem
VFB	**V**irtual **F**unctional **B**us
VL	**V**irtual **L**ink
VM	**V**irtual **M**achine
VMM	**V**irtual **M**achine **M**anagement
VPN	**V**irtual **P**rivate **N**etwork
VSW	**V**irtual **SW**itch
VXLAN	**V**irtual e**X**tensible **L**ocal **A**rea **N**etwork
WCET	**W**orst **C**ase **E**xecution **T**ime
YANG	**Y**et **A**nother **N**ext **G**eneration

List of Publication

Hongjie Fang and Roman Obermaisser. “Execution Environment for Mixed-Criticality Train Applications Based on an Integrated Architecture”. In: *2017 International Conference on Promising Electronic Technologies (ICPET)*. IEEE. 2017, pp. 1–7.

Hongjie Fang and Roman Obermaisser. “Virtual Gateway in TCMS Execution Environment based on an Integrated Architecture”. In: *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*. IEEE. 2019, pp. 524–531.

Hongjie Fang and Roman Obermaisser. “Virtual Switch for Integrated Real-Time Systems based on SDN”. In: *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE. 2019.

Hongjie Fang and Roman Obermaisser. “Virtual Switch Supporting Time-Space Partitioning and Dynamic Configuration for Integrated Train Control and Management Systems”. In: *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE. 2018, pp. 735–739.