It is change, continuing change, inevitable change, that is the dominant factor in society today.
No sensible decision can be made any longer without taking into account not only the world as it is, but the world as it will be...

Isaac Asimov

Acknowledgements

A big thank you to all....

Abstract

Automation solutions are omnipresent in modern society as a part of the infrastructure that provides utility services such as water and power. At the core of these systems is the controller, a specialized computer designed to operate in harsh environments where unplanned downtime can be costly. High-quality hardware, software, and spatial redundancy (i.e., hardware multiplication) are commonly employed to mitigate disruptions.

Industrial control systems are evolving into more interconnected and interoperable architectures, marking a shift toward network-centric designs where the network, rather than the controller, becomes the central part of the system. Concepts traditionally associated with information technology, such as edge and cloud computing, containerization, and orchestrators, are entering the operational technology domain. New standards, such as OPC UA, with its information model and communication protocols, are gaining traction to facilitate interoperability.

This evolution presents redundancy challenges, such as adapting failure detection and state transfer mechanisms needed by standby redundancy to a network context, and opportunities, such as utilizing systems previously confined to the information technology domain. This shift toward a network-centric control system architecture is the overarching motivation for this thesis's revisit of spatial redundancy.

Specifically, this thesis investigates orchestrator-aided failure recovery as a complement to traditional redundancy. It also proposes a failure detection mechanism that maintains consistent control during network partitioning between redundant controllers. The thesis also examines the behavior of OPC UA PubSub in a standby redundancy context. It introduces a method for processing priority based on information embedded in incoming network frames. Additionally, the thesis proposes an architecture that enables the distribution of redundancy-related state data. It also investigates checkpointing solutions and communication protocols to identify a suitable mechanism for transferring state data between redundant controllers.

Sammanfattning

Vårt moderna samhälle är beroende av automationssystem för samhällskritiska tjänster som vatten och el. En nyckelkomponent i dessa system är den industriella kontrollern, en specialiserad dator för krävande miljöer där driftstopp kan vara kostsamma. Risken för driftstopp reduceras typiskt med högkvalitativ hårdvara, mjukvara och redundanslösningar.

Teknikutvecklingen driver automationssystem mot mer sammankopplade och nätverksorienterad arkitekturer, där nätverket, snarare än kontrollern, är systemets mittpunkt. Med denna förändring gör koncept som traditionellt förknippas med informationsteknik, såsom molnbaserad tjänster och orkestreringssystem, intåg styrsystemssammanhang. I fotspåren av detta skifte följer också nya standarder, som OPC UA med dess informationsmodell och kommunikationsprotokoll, vars syfte är att underlätta informationsutbyte mellan styrsystemsprodukter från olika tillverkare.

Ovannämnda utveckling medför redundansrelaterade utmaningar, som att anpassa standby-redundans relaterade funktioner till en nätverksbaserad kontext. Samtidigt uppstår nya möjligheter, till exempel att utnyttja teknik som tidigare varit begränsade till IT-system i automationssammahang. Skiftet mot en nätverkscentrerad arkitektur för styrsystem utgör den övergripande anledningen för avhandlingens nyintresse av redundanslösningar för industriella kontrollrar.

Specifikt presenterar avhandlingen en utvärdering av automatisk reparation med stöd av orkestreringssystem som komplement till traditionell redundans. Den föreslår även en mekanism för felupptäckt som upprätthåller konsistent styrning vid nätverkspartitionering mellan redundanta kontrollrar. Vidare analyserar avhandlingen OPC UA PubSub i standby-redundanssammanhang och introducerar en metod för att hantera exekveringsprioritet baserat på information inbäddad i inkommande nätverkspaket. Avhandlingen föreslår också en arkitektur som möjliggör distribution av tillståndsdata för redundans, samt undersöker lösningar för checkpointing och kommunikationsprotokoll i syfte att identifiera och presentera en lämplig mekanism för överföring av tillståndsdata mellan redundanta kontrollrar.

List of Publications

Papers included in this thesis[1]

Paper A: *Kubernetes Orchestration of High Availability Distributed Control Systems*
Bjarne Johansson, Mats Rågberger, Alessandro V. Papadopoulos, and Thomas Nolte.
In 23rd IEEE International Conference on Industrial Technology (ICIT), 2022.

Paper B: *Consistency Before Availability: Network Reference Point based Failure Detection for Controller Redundancy*
Bjarne Johansson, Mats Rågberger, Alessandro V. Papadopoulos, and Thomas Nolte.
In 28th International Conference on Emerging Technologies and Factory Automation (ETFA), 2023.

Paper C: *OPC UA PubSub and Industrial Controller Redundancy*
Bjarne Johansson, Olof Holmgren, Martin Dahl, Håkan Forsberg, Alessandro V. Papadopoulos, and Thomas Nolte.
In 29th International Conference on Emerging Technologies and Factory Automation (ETFA), 2024.

Paper D: *Priority Based Ethernet Handling in Real-Time End System with Ethernet Controller Filtering*
Bjarne Johansson, Mats Rågberger, Alessandro V. Papadopoulos, and Thomas Nolte.
In 48th Annual Conference of the Industrial Electronics Society (IECON), 2022.

[1]The included papers have been reformatted to comply with the thesis layout. Additionally, minor inconsistencies have been corrected, such as parameter naming and spelling errors.

Paper E: *Partible State Replication for Industrial Controller Redundancy*
Bjarne Johansson, Olof Holmgren, Alessandro V. Papadopoulos, and Thomas Nolte.
In 25th IEEE International Conference on Industrial Technology (ICIT), 2024.

Paper F: *Checkpointing and State Transfer for Industrial Controller Redundancy*
Bjarne Johansson, Björn Leander, Olof Holmgren, Alessandro V. Papadopoulos, and Thomas Nolte.
Submitted to IEEE Open Journal of the Industrial Electronics Society (OJIES), June 2025.

Related publications (not included in this thesis)

Paper X1: *Actors for Timing Analysis of Distributed Redundant Controllers*
Marjan Sirjani, Edward A. Lee, Zahra Moezkarimi, Bahman Pourvatan, **Bjarne Johansson**, Stefan Marksteiner, and Alessandro V. Papadopoulos.
In Gul Agha's Festschrift (GulFest), 2025.

Paper X2: *A Methodology to Map Industrial Automation Traffic to TSN Traffic Classes*
Kasra Ekrad, Inés Alvarez Vadillo, **Bjarne Johansson**, Saad Mubeen, and Mohammad Ashjaei.
In 30th International Conference on Emerging Technologies and Factory Automation (ETFA), 2025.

Paper X3: *Redundancy Link Security Analysis: An Automation Industry Perspective*
Björn Leander, **Bjarne Johansson**, Saad Mubeen, Mohammad Ashjaei, Tomas Lindström.
In 29th International Conference on Emerging Technologies and Factory Automation (ETFA), 2024.

Paper X4: *Work-In-Progress: Towards High-Integrity Redundancy Role Leasing*
Bjarne Johansson, Olof Holmgren, Håkan Forsberg, Thomas Nolte and Alessandro V. Papadopoulos.
In 29th International Conference on Emerging Technologies and Factory Automation (ETFA), 2024.

Paper X5: *Work-In-Progress: Real-Time Fault Diagnosis of Node and Link Failures for Industrial Controller Redundancy*
Kasra Ekrad, Sebastian Leclerc, **Bjarne Johansson**, Inés Alvarez Vadillo, Mohammad Ashjaei, Saad Mubeen.
In 29th International Conference on Emerging Technologies and Factory Automation (ETFA), 2024.

Paper X6: *Work-In-Progress: Towards Developing a Supervisory Agent for Adapting the QoS Network Configurations*
Sebastian Leclerc, Kasra Ekrad, **Bjarne Johansson**, Inés Alvarez Vadillo, Mohammad Ashjaei, Saad Mubeen.
In 29th International Conference on Emerging Technologies and Factory Automation (ETFA), 2024.

Paper X7: *The Computing Continuum: From IoT to the Cloud*
Auday Al-Dulaimya, Matthijs Jansen, **Bjarne Johansson**, Animesh Trivedi, Alexandru Iosup, Mohammad Ashjaei, Antonino Gallettad, Dragi Kimovski, Radu Prodan, Konstantinos Tserpes, George Kousiouris, Chris Giannakos, Ivona Brandic, Nawfal Ali, André B. Bondi, Alessandro V. Papadopoulos.
In Internet of Things (IoT) - Elsevier, 2024.

Paper X8: *Formal Verification of Consistency for Systems with Redundant Controllers*
Bjarne Johansson, Bahman Pourvatan, Zahra Moezkarimi, Alessandro V. Papadopoulos, Marjan Sirjani.
In Models for Formal Analysis of Real Systems (MARS), 2024.

Paper X9: *Dependability and Security Aspects of Network-Centric Control*
Björn Leander, **Bjarne Johansson**, Tomas Lindström, Olof Holmgren, Thomas Nolte, Alessandro V. Papadopoulos.
In 28th International Conference on Emerging Technologies and Factory Automation (ETFA), 2023.

Paper X10: *Centralised Architecture for the Automatic Self-Configuration of Industrial Networks*
Inés Alvarez Vadillo, Daniel Bujosa Mateu, **Bjarne Johansson**, Mohammad Ashjaei, Saad Mubeen.
In 28th International Conference on Emerging Technologies and Factory Automation (ETFA), 2023.

Paper X11: *Heartbeat Bully: Failure Detection and Redundancy Role Selection for Network-Centric Controller*
Bjarne Johansson, Mats Rågberger, Alessandro V. Papadopoulos, Thomas Nolte.
In 46th Annual Conference of the Industrial Electronics Society (IECON), 2020.

Paper X12: *Concurrency Defect Localization in Embedded Systems using Static Code Analysis: An Evaluation*
Bjarne Johansson, Alessandro V. Papadopoulos, Thomas Nolte.
In 30th IEEE International Symposium on Software Reliability Engineering (ISSRE), 2019.

Paper X13: *Work-In-Progress: Classification of PROFINET I/O Configurations Utilizing Neural Networks*
Bjarne Johansson, Björn Leander, Aida Čaušević, Alessandro V. Papadopoulos, and Thomas Nolte.
In 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2019.

Granted Patent 1: *Method for Failure Detection and Role Selection in a Network of Redundant Processes*
Bjarne Johansson, Mats Rågberger, Anders Rune.
EP20183508A 2020-07-01, US11748217B2 2023-09-05

Granted Patent 2: *Methods and Means for Failure Handling in Process Control Systems*
Bjarne Johansson, Mats Rågberger.
EP21204217A 2021-10-22, US12224679B2 2025-02-11

Patent Application 1: *Safe Failover Between Redundant Controllers*
Bjarne Johansson, Olof Holmgren, Mats Rågberger.
EP23164720A 2023-03-28, EP23196499A 2023-09-11

Patent Application 2: *Managing Introduction of Updates in a Process Control Function*
Bjarne Johansson, Olof Holmgren, Stefan Sällberg.
EP24182353 2024-06-14

Patent Application 3: *State Transfer Scheduling of Internal States in an Industrial Environment*
Bjarne Johansson, Björn Leander, Olof Holmgren.
EP25180047 2025-05-31

Licentiate Thesis: *Dependable Distributed Control System: Redundancy and Concurrency Defects*
Bjarne Johansson
Mälardalen University, Licentiate Thesis no. 330, 2022.

Contents

I

Thesis

Chapter 1

Introduction

This thesis revisits spatial redundancy as a method for increasing fault tolerance in industrial control systems. The following sections introduce the domain of industrial controllers and the principles of fault-tolerant design through redundancy. Readers who are already familiar with industrial controllers and redundancy and primarily interested in this thesis contributions may skip directly to Section 1.4.

1.1 Industrial Controllers and Distributed Control Systems

Industrial controllers are typically rugged, specialized computers designed for long-term, continuous operation in potentially harsh environments [1]. As the name implies, controllers monitor and control various operations, such as machines, production lines, or chemical processes. They are generic and capable of supporting a wide range of automation tasks. This flexibility is achieved by allowing the controller to execute applications specifically developed for the intended automation solution. These applications are typically created by end users or organizations with specialized domain knowledge. The execution of an application on a controller generally follows three phases: (i) *Copy-in*, (ii) *Execute*, and (iii) *Copy-out*, as shown in Figure 1.1 [2, 3].

The *Copy-in* phase involves sampling the current state of the controlled process, typically by reading values from Input/Output (I/O) devices connected to sensors sensing the controlled operation. In the *Execute* phase, the application processes these values to determine appropriate actions, including calculating output values for the next phase, the *Copy-out* phase. During the *Copy-out* phase, the controller sends these output values to I/O devices, such

as actuators, which influence the controlled process. Figure 1.1 illustrates this execution cycle.

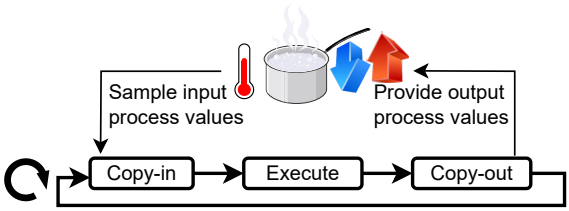


Figure 1.1: A simplified example of the typical, repetitive execution cycle of an application running in a controller. The *Copy-in* phase sample process values; the *Execute* phase uses these values in the control loop logic to deduce the needed control actions. The actions are output values communicated in the *Copy-out* phase.

This thesis focuses on challenges and opportunities primarily derived from the Distributed Control Systems (DCSs) context, which are large-scale automation systems typically composed of multiple interconnected controllers that exchange information. DCSs are used to automate entire sites, in contrast to Programmable Logic Controllers (PLCs), which are generally used to automate individual machines or parts of machines. In this context, controllers are commonly referred to as Distributed Control Nodes (DCNs), a term introduced in the Open Process Automation™ Standard (O-PAS) [4].

The prevailing DCS architecture today is hierarchical, as illustrated in Figure 1.2. At the bottom of this hierarchy are the I/O and field devices that interact with the physical world. The field devices communicate with the DCN through I/O channels on the DCN itself or over fieldbuses. As the name implies, a fieldbus allows the DCN to exchange data with devices located in the "field," which can be geographically distant from the DCN. There exists a wide variety of different fieldbuses based on varying underlying technologies. However, recent trends show increasing adoption of Ethernet-based fieldbuses, such as PROFINET, over traditional non-Ethernet-based alternatives [5, 6].

The trend with increased use of Ethernet-based fieldbuses is part of a broader shift in the automation industry toward more network-oriented systems that offer increased flexibility, interconnectivity, and interoperability [7, 8]. O-PAS envisions a network-centric architecture built on a unified network backbone called the O-PAS Connectivity Framework (OCF). Within this framework, Operational Technology (OT) and Information Technology (IT) coexist

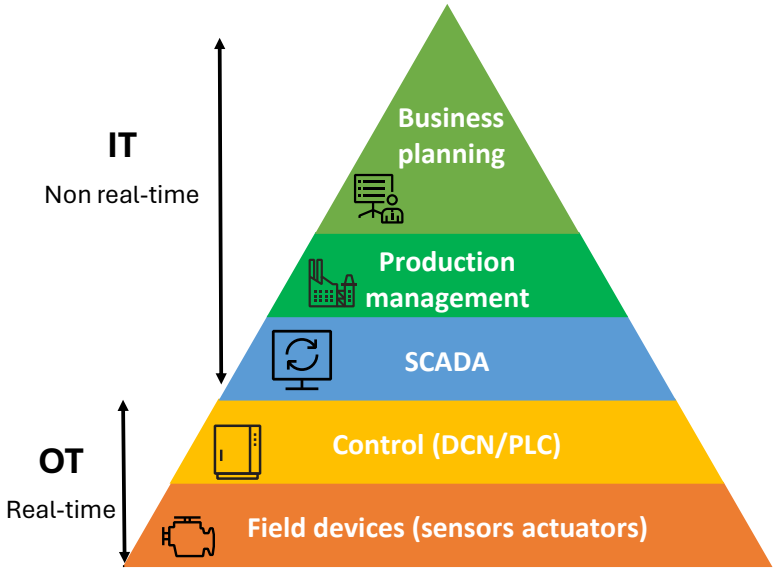


Figure 1.2: The automation pyramid.

on a shared Ethernet-based infrastructure, as illustrated in Figure 1.3. O-PAS prescribes implementing OCF using Ethernet as the communication technology and OPC UA (Open Platform Communications Unified Architecture) as both the communication protocol and information model [4]. O-PAS also introduces the Advanced Computing Platform (ACP) as a computationally competent device that can run multiple Virtualized DCN (VDCN) instances, actualizing the interest in virtualization technologies in the OT context.

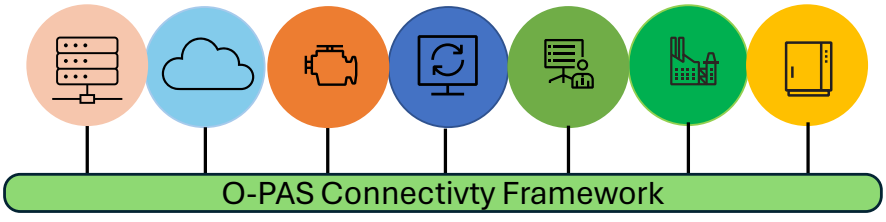
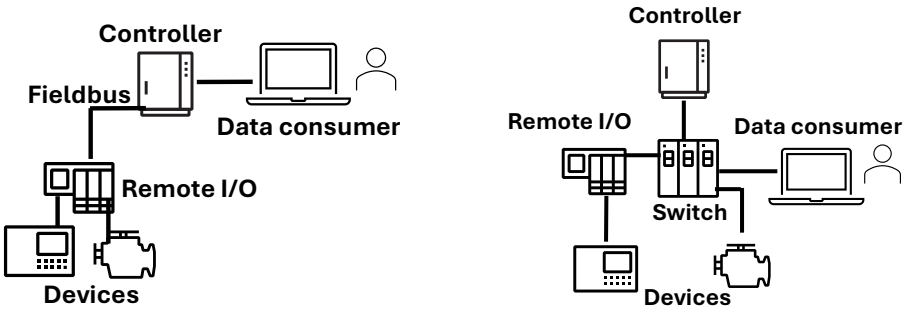


Figure 1.3: Collapsed automation pyramid illustrating a unified connectivity backbone that integrates both OT and IT components.

This shift implies that the DCS domain transforms from a controller-centric architecture to a network-centric architecture [8]. Specifically, and somewhat simplified, this means that the network replaces the controller as the system center, as shown in Figure 1.4. Time-Sensitive Networking



(a) Controller-centric. (b) Network-centric.

Figure 1.4: Simplified overview of controller-centric and network-centric control systems.

(TSN) is a set of standard extensions to the IEEE 802.1Q Ethernet networking standard. In combination with OPC UA, TSN is seen as an enabler of converged networks and network-centric architectures, as TSN can provide bounded low-latency communication [7].

The transformation from the hierarchical automation pyramid, shown in Figure 1.2, to the converged architecture depicted in Figure 1.3 provides the domain-induced motivation for this thesis. Motivated by this change, this thesis revisits fault tolerance with a focus on spatial DCN redundancy.

1.2 Fault Tolerance

DCSs are commonly used in domains where production stops are highly undesirable, including unplanned stops due to failures in the automation system. Hence, control system dependability is crucial. Dependability is a broad term comprising five attributes: (i) Reliability, (ii) Availability, (iii) Maintainability, (iv) Integrity, and (v) Safety [9].

Reliability and availability are aspects of the likelihood that the system is operational. Reliability refers to the probability of continuous operation, whereas availability denotes the proportion of time the system is ready to perform as intended. Maintainability addresses the ability to maintain the system and repair faults. Integrity focuses on reducing the probability of invalid system states leading to faulty outputs. Finally, safety concerns lessen the likelihood that a system failure causes harm when operating in a potentially hazardous context.

This thesis primarily addresses fault tolerance to increase the likelihood

of continuous operation, that is, reliability. A specific threat to reliability is faults. Faults can lead to errors that cause the system to fail and become non-operational. Hence, fault tolerance is a means to improve reliability. A fault-tolerant system is a system that can operate in the presence of faults. Thus, fault tolerance refers to a system's ability to continue operating and performing its designated functions even in the presence of faults [10]. Several fault-tolerance techniques exist, including error correction codes, temporal message replication, and the method addressed by this thesis, hardware replication via spatial redundancy [11].

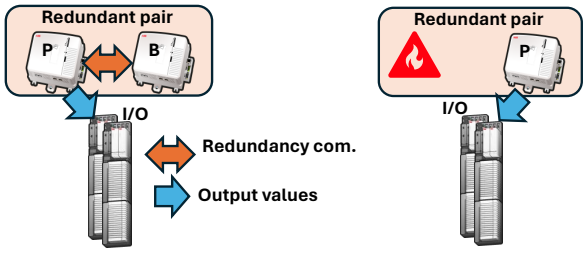
1.2.1 Spatial Controller Redundancy

As mentioned, spatial redundancy is one form of redundancy among many. Another form is information redundancy, i.e., the addition of information for fault detection and correction purposes [11]. Temporal redundancy is yet another, where, for example, a message is sent multiple times at different times, increasing the probability of tolerance to transient faults.

In the context of control systems, spatial redundancy is typically realized through hardware replication in a standby redundancy approach [12, 13]. Standby redundancy means that one controller acts as the active primary, providing output, while the secondary backup is ready to take over if the primary fails. Figure 1.5 provides an overview of standby controller redundancy. The primary controller provides output values to the controlled process, as shown in Figure 1.5a. The primary continuously synchronizes the backup with the latest state through checkpoint-aided state transfer; at the same time, it also sends keep-alive signals, illustrated by the "Redundancy com." arrow in Figure 1.5a. The continuous keep-alive and state transfer allow the backup to detect failures in the primary and resume operations with the latest state of the primary, thereby continuing to drive the process, as shown in Figure 1.5b. The transition from backup to the primary role due to a failure in the original primary is commonly referred to as a *failover*.

1.3 Industrial Controller Redundancy in a Network-Centric Context

A controller-centric architecture tightly couples I/Os and devices with a specific controller. Therefore, in a redundant controller-centric controller pair, both controllers must be able to access the I/O, which constrains the redundant pair's distribution, deployment, and connectivity [8].



(a) Both operational. (b) After failover.

Figure 1.5: Simplified standby controller redundancy overview. The primary controller (P) is the active controller, providing output to control the process. In the event of a primary failure, the backup (B) assumes the primary role and begins to provide output.

As mentioned, the shift from a controller-centric architecture to a network-centric architecture involves moving towards a flatter network structure, where controllers, as well as the I/O and devices, are connected to the same network backbone, as illustrated in Figure 1.3 and Figure 1.4b. The network-centric architecture does not limit access to I/O and devices to a single controller, like the controller-centric architecture shown in Figure 1.4a.

In a network-centric architecture, the I/O connects to the network instead of directly to the controller, enabling more flexible controller deployment alternatives. Ultimately, any node on the network that can meet the required real-time requirements and execute a control application can act as a controller.

The transition from controller-centric to network-centric systems implies that traditional fault tolerance methods, such as spatial standby redundancy, need to adapt to the flexibility and opportunities of network-centric control systems, as any two nodes on the network capable of running the control application can form a redundant controller pair. However, this also means that redundancy functionality cannot depend on specialized hardware, which would limit the use of redundant controllers to nodes with such hardware. Instead, redundancy functionality should rely only on standard Ethernet networking functions to be hardware-agnostic.

The essence of this thesis is to revisit fault tolerance, with a specific focus on spatial controller redundancy. This revisit is motivated by the aforementioned shift toward network-centric architectures, aiming to leverage the benefits while addressing the challenges introduced by this architectural transition.

1.4 Contribution and Outline

The overall motivation of this thesis is to explore opportunities and address challenges associated with fault tolerance and spatial controller redundancy, specifically arising from the transition from controller-centric to network-centric architectures.

As previously mentioned, this transition has caused increased interest in virtualized controllers. This thesis examines the use of orchestration and containerization within the context of spatial redundancy. The first contribution is an evaluation of orchestration-based failure recovery as both a replacement for, and complement to, traditional redundancy methods, resulting in a layered redundancy approach.

As described in Section 1.2.1, a standby redundancy solution requires a failure detection mechanism to determine if the primary controller has failed. The second contribution of this thesis is a failure detection mechanism designed to ensure consistent process control in cases of network partitioning between redundant controllers.

OPC UA is the envisioned interoperability standard, and the third contribution of this thesis is a study of OPC UA PubSub in a standby controller redundancy context, focusing on the realization of seamless failover. Further, a network-centric controller is likely to connect to a collapsed network that carries both time-sensitive and best-effort traffic. Hence, a controller connected to such a network can receive both types of traffic on the same network interface. Examples of time-sensitive traffic include redundancy-related traffic, such as keep-alive messages and state data. The fourth contribution of this thesis is a method for assigning processing priority within a controller based on priority information in the received traffic.

The fifth contribution introduces an approach that separates the backup role from the direct responsibility of receiving all checkpointed state data, thereby reducing the risk of overwhelming a controller acting as backup for multiple primaries with such network traffic. Finally, the sixth contribution arises from an investigation of existing checkpointing solutions and state data transfer methods (including communication protocols), leading to the proposal and evaluation of a solution specifically designed for state transfer between redundant industrial controllers.

This thesis follows the two-part structure common to a collection of papers. The first part, which is this part, provides an overview and general introduction. Chapter 1 (this chapter) serves as the introduction to this thesis. Chapter 2 provides a more detailed presentation of the domain, terminology, and relevant concepts, including related work. Chapter 3 outlines the research challenges,

states the research question, and describes the research process and methods used. Chapter 4 discusses this thesis's contributions, maps them to the research question, and maps the contribution to the included publications, as well as showing the relation to other publications not included in this thesis. Finally, Chapter 5 concludes the first part with a summary and directions for future work.

The second part of this thesis consists of the included publications.

Chapter 2

Background and Related Work

As described in the introduction, the motivation for revisiting redundancy in this thesis originates from the evolution of control systems toward more interconnected and network-oriented architectures [14]. Ethernet-based communication enables this interconnection, which is leveraged by the OPC UA standard to facilitate inter-vendor data exchange [5, 7].

2.1 Fault Tolerance

As the term fault tolerance suggests, it concerns a system's resilience to faults. It commonly refers to the ability of a system to continue functioning in the presence of faults [10]. But what is a fault? To establish a common understanding, this section revisits the taxonomy and concepts presented by Avižienis et al. [9] and Nelson et al. [15].

The term "error" is commonly used, and in the context of fault tolerance, an error refers to an incorrect internal state that results from the manifestation of a fault. Furthermore, faults manifested as errors can result in externally visible failures, such as a non-redundant controller failing and no longer providing process control. Examples of faults include software bugs, permanently failing hardware components, or transient message loss due to a bit flip from a single-event upset altering an Ethernet frame passing through a network switch.

As implied in the previous paragraph, a fault can be either active or dormant. An active fault is a fault that has caused an invalid system state (an error), such as an undetected bit flip changing the value of a used variable or a software bug. A dormant fault, on the other hand, is a fault that has not yet caused an error, such as a software bug in an execution path that has not yet been executed or an improper implementation of shared resources that only

manifests under certain execution interleaving patterns. As a side note, one of our works, not included in this thesis, addresses the possibility of detecting such faults using Static Code Analysis (SCA) tools [16]. A fault that manifests as an error can lead to a failure. A failure is a deviation from the expected functionality. For example, a control application that fails to provide updated values to the controlled process.

Spatial redundancy, involving hardware duplication, is a common fault-tolerance method in the control system domain [17, 12]. As mentioned, this thesis addresses various aspects of this fault tolerance method.

2.1.1 Redundancy

The following sections use controllers as illustrative examples, given the domain of this thesis. However, the concepts described generally apply to any computing device facing similar needs. The type of redundancy this thesis addresses is spatial redundancy. Spatial redundancy comes in various forms, one of which is Triple Modular Redundancy (TMR), often used in the aerospace domain [18]. TMR consists of three active controllers producing outputs, with a voting mechanism, i.e., the voter, selecting the majority output in case of discrepancies. TMR is a specialization of N-modular redundancy, where N represents the number of active controllers. Typically, N is an odd number so the voter can form a majority decision in case of output differences from the controllers.

In the context of industrial control systems, standby redundancy is the more common redundancy pattern [12, 13]. Standby redundancy typically means that the backup does not produce outputs used for control. Instead, as the name implies, the backup is in standby mode. The degree of backup-readiness varies from cold, warm, and hot [11]. Cold standby means having a spare unit that can replace the failed one, i.e., a unit kept unpowered in storage. Warm standby means that the backup is powered on and can assume the primary role in case of failure, but it does not execute all the functions associated with the primary role. In hot standby, the backup also executes all functions associated with the primary role. The significant difference between the primary and backup in hot standby is that only the primary provides output.

A standby redundancy solution requires two functions: (i) Failure Detection and (ii) State Replication [10, 19]. As the name might give way, failure detection is a backup's means to detect that the primary has failed. The state replication, i.e., the checkpoint and state transfer mechanism, allows a backup to recover from where the primary failed. The following sections elaborate on these functions from the perspective of industrial controller redundancy.

As mentioned, the reason for revisiting these functions is the transition from controller-centric systems to network-centric systems [8]. This transition increases deployment alternatives by eliminating the dependency on designated, purpose-built redundancy links, which might be the case today [20, 21, 22], and instead utilizes standard Ethernet networking.

2.1.1.1 Failure Detection

In a standby redundancy configuration, failure detection is essential for a backup to identify primary failure [10, 19]. Failure detection and diagnostics are also crucial for preventing undetected system deterioration, such as an unnoticed backup failure. This thesis primarily addresses the failure detection of a primary from a backup.

As described in Section 2.1, a failure is the manifestation of a fault that leads to an error and a deviation from the expected result, the actual failure [9]. The consequences of a failure can vary in terms of external observability. Typically, unless explicitly stated otherwise in this thesis, a fail-silent semantics is assumed. Fail-silent semantics assume that the failed controller does not continue to provide output [23]. Another well-known failure model is Byzantine faults, which can lead to Byzantine failures. Byzantine failures are characterized by multifaceted behavior, which means, for example, that the system could show correct and incorrect behaviors towards different parts of the system at the same time [24]. The Byzantine failure model originates from the seminal Byzantine general's problem formulated by Lamport et al. [25].

Failure detection is distinguishing a functioning node, service, or component from one that has crashed (failed). Failure detectors are commonly utilized in distributed systems to identify which participants are operational [26]. In the theoretical realm of failure detectors, Chandra and Toueg introduced the unreliable failure detector abstraction, categorized by its completeness and accuracy properties, explained below [26]. Subsequently, Chandra, Hadzilacos, and Toueg prove that failure detectors belonging to the class $\Diamond W$ (eventual weak) are the weakest mechanism for providing consensus in an asynchronous distributed system [27]. An asynchronous system is one in which message transmission and execution durations are unbounded [26].

As mentioned, Chandra et al. define two properties for failure detectors: completeness and accuracy. Completeness defines the detection capability of failure detectors. If all functioning failure detectors indicate every failed node as failed, it fulfills the strong completeness property. If every crashed node is detected by at least one functioning failure detector as failed, it meets the criteria for weak completeness. A failure detector that always suspects all

nodes can achieve strong completeness, regardless of whether the nodes have failed or not. Therefore, a correctness property, called accuracy, is also needed. Accuracy specifies the false-positive properties of failure detectors. A failure detector with no false positives has strong accuracy. Fulfilling weak accuracy requires that there is at least one functioning node that is never falsely suspected of being faulty. Failure detectors in the $\Diamond W$ class achieve weak completeness and, after some time, weak accuracy [27]. Achieving failure detection with strong completeness and accuracy is impossible in asynchronous systems [26, 28].

Control systems are typically real-time systems, meaning that not only does the output need to be correct, but it also needs to be delivered at the correct time to be accurate and useful. Distributed systems where message delivery and execution occur within a known bounded time are called synchronous systems [29]. Thus, real-time control systems generally fulfill the synchronous system model's properties of bounded execution and message delivery time. However, failure can happen, such as network failures, causing, for example, controllers in a redundant pair to become isolated from one another. In such scenarios, the message delivery time might no longer be bounded, at least not to a known time. In other words, a control system meets the criteria of a synchronous system during most of its operational time, but not necessarily all the time. Hence, control systems generally fit the description of a partly synchronous system model [30].

Message-based failure detection divides into two approaches: push-based or pull-based [30, 28]. The pull-based approach involves a "are you alive" query from the supervising failure detector to the supervised, which responds with an answer. The absence of an answer is interpreted as a failure. On the other hand, the push-based approach involves the supervised continuous sending of heartbeat messages to the supervising failure detector. Again, the absence of heartbeats is interpreted as a failure of the supervised. The literature describes numerous failure detection algorithms; the following are just a few examples. Dwork et al. [30] propose a pull-based algorithm for partially synchronous systems that adapts to potentially unknown but bounded message transfer times. Other examples include utilizing application message exchanges as a complement to heartbeats to reduce network load [31], probabilistic approaches [32], and Quality of Service (QoS) properties for failure detectors [33].

Failure detection implicitly performs a redundancy role selection when there is only one backup, as the sole backup should assume the primary role upon primary failure. If multiple backup candidates are available, an election mechanism can complement failure detection to assign the primary role to an

appropriate candidate. This scenario is commonly known as the leader election problem in the context of distributed systems, with the Bully algorithm being one of the most well-known examples [34]. Failure detection and leader election can be integrated to provide a redundancy role selection mechanism suitable for a DCN context, a topic we explored in related work [35].

This thesis addresses failure detection suitable for controller redundancy, aiming to ensure that only one of the controllers in the redundant pair is primary, even in the event of network partitioning. In other words, a failure detection that prioritizes consistency over availability, a choice that distributed systems, including a controller redundancy pair, must make when faced with partitioning according to the Consistency, Availability, and Partition tolerance (CAP) theorem [36]. Requiring a majority from a quorum, often implemented through consensus protocols, is a common consistency strategy tolerant to network partitioning, as described further in Section 2.1.1.2. A partition that is unable to form a majority is prevented from performing operations that might risk the consistency. Witness solutions that utilize a network accessible entity, such as the cloud or a shared disk, are employed in IT services like Windows Failover Cluster to break ties between equally sized partitions [37]. Emerson's DCS Delta V serves as an SCADA level example of Windows Failover Cluster usage (and thereby witness use) in DCS context [38].

The failure detection method proposed in this thesis addresses failure detection under network faults between a redundant controller pair, resulting in equally sized partitions. The proposed solution leverages existing network equipment as a tiebreaker to determine the primary role, thereby eliminating the need to introduce an additional witness node. The proposed failure detector prioritizes consistency over availability. Similarly, in terms of the failure-detection properties proposed by Chandra et al., it prioritizes accuracy over completeness [36, 26].

In related but separate work, we investigate the use of the Media Redundancy Protocol (MRP) to distinguish between controller failures and network failures within bounded time constraints by utilizing MRP's periodic supervision messages [39]. Dorsch et al. employ Bidirectional Forward Detection (BFD) within Software Defined Networking (SDN) to achieve rapid link-failure detection within smart-grid contexts [40]. BFD, defined by the Internet Engineering Task Force (IETF) in Request For Comments (RFC) 5880, supports configurable supervision intervals (link heartbeat periods), with Dorsch et al. setting it as short as one millisecond [40, 41]. Utilizing MRP or BFD with short supervision intervals could facilitate low-latency tie-breaking through the failure detection proposed in this thesis, though this remains a topic for future research.

While on the topic of potential future research in the context of related failure-detection mechanisms, in another related but separate publication, we present an initial exploration into designing a network switch-hosted tie-breaker (witness) mechanism for high-integrity controllers in safety-critical applications [42]. Additionally, inspired by the failure detection proposed in this thesis, we have conducted model checking to verify the proposed algorithm's properties [43]. In the same work, we present a leasing-based variant based on the same algorithm. Future research could build upon the work in [42] and explore the feasibility of incorporating a high-integrity version of the lease-based variant in network switches.

2.1.1.2 State Replication

State replication is the second function needed to realize standby redundancy [10]. The method of synchronizing redundant instances, often referred to as replicas in a distributed system context, can vary. One approach is to use a deterministic Replicated State Machine (RSM) and replicate the events driving the state transitions; this is called active replication [44]. Employing consensus protocols like Paxos [45, 46, 47], RAFT [48], or Viewstamped Replication (VSR) [49, 50] ensures that the RSM receives events in the same order. To the best of the author's knowledge, active replication based on consensus protocol-driven event replication is not used to synchronize redundant industrial controllers.

Consensus protocols are more commonly used in IT contexts, where one well-known example is etcd, a distributed key-value storage system that utilizes Raft for replication [51, 52]. As further described in Section 2.2, Kubernetes is one of the most well-known container orchestration systems, and Kubernetes uses etcd for replication of, for example, configuration-related data, etc. [53]. However, etcd is designed for replicating key-value pairs, not for continuous synchronization of whole application states (bulk data), such as the internal state of a control application. Moreover, as Hark et al. note, cost-constrained smaller industrial system installations may only have two nodes, which makes quorum-based solutions, such as etcd, challenging in this context [54]. They address this challenge by evaluating alternative deployment strategies and exploring substitutes for etcd within Kubernetes.

Passive replication is the alternative to active replication [44]. Passive replication is more common in standby redundancy [12]. Checkpointing, the collection of internal states, is fundamental in passive replication [55]. The primary system transfers its internal, checkpointed states to the backup. The backup uses these states to assume the primary role, preventing historical or

outdated output from reaching the controlled process [17]. In other words, this approach allows the backup to take over the primary role with the most recent state used to provide an output, i.e., to make a seamless failover from the perspective of the controlled process [3].

The Very-Low Overhead Checkpointing System (VeloC) and Fault Tolerance Interface (FTI) are two libraries providing checkpointing solutions for fault tolerance in high-performance computing [56, 57]. Checkpoint and Restore In Userspace (CRIU) is a Linux utility designed explicitly for checkpointing application data to files [58]. In the realm of redundant industrial controllers, vendors typically employ proprietary methods for checkpointing and transferring controller application states between redundant controllers. Stattelmann et al., evaluate various compiler-aided checkpointing strategies for industrial controllers [3].

In the era of network-centric control systems, cloud or edge-hosted controllers are emerging as viable alternatives [13, 59]. The edge can be a computationally competent device located at the edge of the network, still on-premises and not as geographically distant as cloud-based computational power provided by a remote data center [60]. This emergence brings flexible deployment patterns, where computationally powerful edge devices, like the O-PAS described ACP, could potentially replace many traditional controllers [61, 62]. Hence, one could imagine cost-effective redundancy deployments where a computationally competent device serves as a backup for more than one primary.

This thesis revisits the concept of associating the backup with state storage and proposes an alternative that utilizes storage offered by other controllers or nodes in the network. The motivation is to reduce the risk that a sole backup gets congested with state replication data. Furthermore, this thesis investigates checkpointing solutions employed in industrial and virtualized environments, with a focus on the protocol used for transferring state data. Based on the result of the protocol and checkpointing investigation, this thesis presents a solution designed for transferring state data between a primary and backup controller in a redundancy context.

2.2 Orchestration

Elasticity is one of the desirable properties of cloud-provided services and can be simplified as the ability to adjust to change [63]. This ability includes coping with faults, which lies in the interest of thesis. But what provides this elasticity? One part of the answer to that question is virtualization, and another part of the answer is the management of the virtual instances, further

elaborated below [64].

Virtualization is a key ingredient in cloud elasticity since virtualization isolates and abstracts the underlying server farm hardware from the utilizing software, allowing, for example, multiple and mixed operating systems to share the same hardware while remaining isolated from each other in Virtual Machines (VM) [65]. The number of VMs can be scaled up and down to match the need more easily than physical machines.

Containers offer another, OS-based and more lightweight, form of virtualization than VMs [66]. Containers introduce negligible overhead [67] and have, for almost a decade, been considered a promising technology for virtualization of industrial applications with real-time requirements [68]. A perception shared by Struhár et al. in their survey on real-time containers applicability [69].

Virtualization alone does not provide the elasticity mentioned above. Management of the virtual instances, such as containers, is also necessary, and this is where orchestration comes into play. Container orchestration concerns automated management, deployment, scaling, and failure handling of container instances [66]. The main interest of this thesis is the failure handling mechanisms provided by orchestrators and the applicability of those in the context of controller redundancy.

One of the most well-known container orchestration and management systems is Kubernetes [53]. Kubernetes consists of a control plane containing the logic for monitoring the compute node cluster and driving it toward the desired state. A similar architecture was proposed by Goldschmidt et al. for control systems and flexible controllers running containerized applications [70].

As one can read between the lines in Section 2.1.1.2, a controller, or DCN, is stateful, and this applies to a VDCN as well. Stateful means that the output depends on the input and the current internal state. Schmidt et al. introduced a Kubernetes operator for checkpointing and measured the failure recovery time of stateful applications to be around ten seconds [71]. This duration is too long to serve as a controller redundancy replacement, where recovery times ranging from ten to hundreds of milliseconds are required, depending on the domain and automation solution [72, 13]. Kampa et al. investigated the use of Remote Direct Memory Access (RDMA) as a state transfer mechanism for redundant, virtualized controllers [62]. RDMA over Converged Ethernet (RoCE), as the name implies, is RDMA over Ethernet [73]. Koziolek et al. introduce a Kubernetes extension (operator) for state transfer using OPC UA Client/Server, a state transfer used when upgrading the controller application of virtual controller [74, 75].

This thesis evaluates Kubernetes failure recovery mechanisms, similar to

the work by Schmidt et al. [71] but with the difference that the work in this thesis targets real-time VDCN. In addition, the evaluation includes the recovery time of both single and redundant VDCNs. The work presented in this thesis demonstrates that combining VDCN redundancy with Kubernetes failure recovery provides a layered redundancy, thereby reducing redundancy deterioration in the event of failure.

2.3 Processing of Network Traffic

The processing of network traffic is not a redundancy or fault tolerance topic per se. However, the state transfer and failure detection functionality mandated by standby redundancy utilizes the network [10, 19, 76]. Hence, network traffic processing is a cornerstone in realizing spatial standby redundancy using the network.

On a converged network where OT-traffic coexists with IT-traffic, the best-effort IT communication shall not prevent bounded delivery time of OT messages with real-time requirements [77]. TSN provides means to ensure that time-critical traffic retains its real-time properties even on converged networks [7]. TSN Scheduled Traffic (TSN-ST), i.e., IEEE 802.1Qbv, prescribes the use of time-aware gates in network switches to ensure that frames can be scheduled end-to-end without facing unacceptable queue-induced latency [78]. However, building a schedule that fulfills the latency requirements of all involved communicators is challenging and remains one of the primary TSN challenges [79], although promising scheduling heuristics do exist [80]. The adoption of TSN in automation solutions is still scarce, and interfacing with brownfield systems are likely necessary [1, 79].

Industrial controllers connected to converged networks send and receive traffic with different real-time requirements (e.g., priority, deadline, etc.). If not properly managed, traffic processing degrades the timing behavior of application tasks on the controller, potentially leading to deadline misses [81]. Therefore, different types of traffic may need different processing priorities in the controller, a challenge that has received limited attention [77]. This issue is becoming even more relevant in the era of VDCNs in containerized contexts, where multiple VDCNs can share and compete for the same resources. VDCNs can share both the physical network infrastructure and virtual networks, which can have different performance implications depending on the implementation [82]. Garbugli et al. present Kubernetes support to enable low-latency access to underlying TSN networks for virtualized controllers [83].

In a non-virtualized context, Behnke et al. propose a multi-queue Network Interface Card (NIC) that assigns incoming Ethernet frames to different queues based on metadata [84]. In subsequent work, Behnke et al. implement early demultiplexing of incoming Ethernet frames in the Network Interface Controller (NIC) driver. Demultiplexing into queues served by a single network task. The network task's priority is adjusted dynamically based on the received frames' priority [85].

This thesis presents a mechanism that utilizes widely available NIC features to direct incoming Ethernet frames to different queues based on priority indicating values in the received frames. Network tasks with different priorities (higher and lower) serve the queues. Compared to the approaches above, this solution works with Commercial Off-The-Shelf (COTS) NICs and does not require TSN amendments.

2.4 OPC UA

OPC UA is a comprehensive specification that, for example, encompasses information models and communication protocol specifications, first released in 2008 by the OPC Foundation, which was founded in 1997 [86]. Originally, OPC stood for OLE for Process Control, referring to Microsoft's Object Linking and Embedding; today, OPC stands for Open Platform Communication, and UA stands for Unified Architecture. IEC 62541 standardizes OPC UA.

The OPC UA specification consists of multiple parts, covering areas such as information modeling, services, alarm handling, safety, and security [87]. Part 14, PubSub, describes a publisher-subscriber communication pattern, with or without a central broker. The broker-less variant relies on the network infrastructure to handle the message brokering using multicast [88]. Broker-less PubSub is the communication method prescribed by OPC UA for real-time cyclic process value exchange between controller, I/O, and devices.

Today, there are many Ethernet-based and non-Ethernet-based alternatives for field communication between controllers and remote I/O and devices [5]. These fieldbuses often require a gateway (or similar) for interprotocol/interfieldbus data exchange. The OPC Foundation released the OPC UA Field eXchange (UAFX) specification to address these incompatibility challenges and further enhance field communication interoperability in 2022 [89].

The real-time properties of OPC UA PubSub in combination with TSN have been demonstrated, with the work by Grüner et al. serving as one example [90]. However, research on fault tolerance, particularly spatial redundancy in OPC UA PubSub end nodes, is limited. Neumann et al. examine the requirements for a real-time capable OPC UA PubSub device but do not address fault

tolerance [91]. Cupek et al. explore fault tolerance and spatial redundancy in an OPC UA Client-Server context, but not for PubSub [92].

In fact, at the time of writing, no prior work has been found that addresses OPC UA PubSub in combination with a spatial standby redundancy solution. This thesis contributes by addressing failover behavior in controllers and devices using broker-less OPC UA PubSub.

Chapter 3

Research Overview

3.1 Motivation and Challenges

The work presented in this thesis is the result of a close collaboration between industry and academia, with the author having been an industrial practitioner for nearly two decades. For most of that time, the author has been involved in the development of the embedded-system products that constitute the core of DCSs, namely controllers, I/Os, and communication interfaces providing fieldbus connectivity. Products that can be deployed in spatial redundancy settings to avoid single-point-of-failures [22, 93, 94]. Hence, redundancy has been a part of the author's everyday work life for quite some time, motivating the author's interest in revisiting the topic.

From a non-personal perspective, the motivation for this thesis is the architectural transformation and technology shift described in the earlier chapters. A change that motivates a revisit of standby redundancy-related functions, for the functionality to stay relevant in a new architectural context where previously non-existent technology, or technology reserved for the IT domain, is utilizable in the OT domain. This thesis addresses four challenges (Chl.), elaborated below, motivated by the architectural transformation and technology shift described above.

Chl. 1: **Orchestration and Failure Recovery.** Industrial controller systems increasingly use Ethernet rather than OT-specialized fieldbuses [8, 6]. Less use of OT-specialized fieldbus hardware allows flexible deployment with the use of container virtualization and orchestrators [68, 70, 71]. Orchestration frameworks like Kubernetes offer automated failure recovery mechanisms [53]. In future industrial systems, orchestrators may aid the controller management by, for example, deploying services and

applications on the controllers to match the needs of the automation solution. Introducing orchestrators in the OT domain brings many challenges. This thesis addresses a challenge related to fault-tolerance and redundancy, namely to learn how orchestrator-level and controller-level failure recovery mechanisms can complement each other.

Chl. 2: **Failure Detection and Network Partitioning.** As industrial systems move away from specialized fieldbuses toward Ethernet-based network-centric architectures, the standby redundancy functions, i.e., state replication and failure detection, should preferably only rely on the same Ethernet-based communication means, to not decrease deployment alternatives [76, 6]. As described in Section 2.1.1.1, message-based failure detection methods commonly assume failure of the supervised, i.e., the primary controller in our redundancy use case, from the absence of supervision messages. However, such an absence may also result from a network failure, causing a partitioning rather than a controller failure. This duality in the underlying reason for message absence introduces a research challenge related to ensuring a deterministic and consistent system behavior even in a network partitioning scenario between redundant controllers.

Chl. 3: **OPC UA PubSub and Spatial Redundancy.** As discussed in Section 2.4, OPC UA is receiving attention as the future interoperability standard for DCSs. While the OPC UA specifications define varying levels of service redundancy for Client/Server communications [95], they provide no redundancy guidance for PubSub communication [88]. The above is the motivation for the research challenge related to understanding the behavior, identifying potential shortcomings, and mitigating those for OPC UA PubSub in spatial redundancy scenarios, particularly regarding how publishers and subscribers interact under failover scenarios.

Chl. 4: **Checkpointing and State Transfer.** Besides failure detection, state replication is essential for standby redundancy [10]. As discussed in Section 2.1.1.2, this typically involves checkpointing (capturing internal states) and state transfer (sending these states to backup controllers). The shift toward network-centric architectures with hardware-agnostic redundancy mechanisms motivates revisiting existing checkpointing and state transfer techniques [76]. Hence, the transition motivates the following research challenge, addressed in this thesis: investigating existing checkpointing and state transfer mechanisms from OT and IT con-

texts to consider their applicability for industrial controller redundancy, including adapting and developing solutions for state transfer over Ethernet networks.

The above-mentioned research challenges can fill many theses and, by that, keep multiple PhD students busy for several years. In other words, they are comprehensive. The following section uses them as a motivation and foundation for the research goal and research questions addressed in this thesis.

3.2 Research Goal and Research Questions

Section 3.1 provides the motivation and challenges this thesis addresses. The motivation and challenges that led us to the following overarching research goal for this thesis: **"To explore the challenges and opportunities for achieving fault-tolerance through spatial redundancy in industrial control systems transitioning to network-centric architectures."**

The overarching research goal, along with the challenges outlined by Chl. 1 - Chl. 4 in Section 3.1, are distilled into more precise research questions, listed below.

RQ 1: How can container orchestrators complement or replace redundancy mechanisms in industrial controllers for fault-tolerance purposes?
Motivation: This research question addresses **Chl. 1**. Containers and orchestrators have long been vital technologies in cloud computing, but have not been widely adopted in the OT domain [68, 96]. With the convergence of IT and OT and industrial control systems becoming increasingly network-centric, these technologies now appear to be viable options in the OT domain [70, 74, 62]. Orchestrators like Kubernetes include fault-tolerance capabilities [53]. Exploring how orchestrators can complement or potentially replace traditional standby redundancy mechanisms in industrial controllers is needed to feasibly leverage this technology's benefits in industrial environments, for fault-tolerance purposes.

RQ 2: How can failure detection mechanisms, suitable for redundant industrial controllers, distinguish controller failures from network partitions to maintain consistent control in a spatially redundant setup?
Motivation: This research question addresses **Chl. 2**. Network-centric

controllers preferably utilize standard Ethernet-based networks for redundancy functions, such as state replication and failure detection [76]. However, network partitioning can prevent communication between redundant controllers, potentially causing both partitions to simultaneously have the primary role if conventional failure detection mechanisms are used. According to the CAP theorem, see Section 2.1.1.1, such partitioning scenarios leading to multiple primary controllers correspond to preserving availability but sacrificing consistency [97]. Sacrificing consistency by allowing two or more active primaries within the same redundant controller set is often undesirable in industrial contexts [21].

RQ 3: How can a spatially redundant OPC UA PubSub backup controller or device take over as primary during failover without disrupting communication with interdependent PubSub nodes?
Motivation: This research question addresses **Chl. 3**. OPC UA PubSub aspires to become the interoperability standard, as well as the field exchange protocol of the future [87, 89, 98]. In critical domains where downtime is highly undesirable, OPC UA PubSub is expected to be deployed in spatially redundant setups to facilitate communication between redundant controllers and devices. This question aims to investigate the behavior of OPC UA PubSub in a standby redundancy context, focusing on failure recovery when the primary controller or device fails and the backup assumes the primary role.

RQ 4: How can processing of time-sensitive network traffic be prioritized over best-effort traffic on industrial controllers?
Motivation: This research question indirectly addresses all the research challenges, since they are network-dependent. However, the question is mostly related to **Chl. 2** and **Chl. 4**. Failure detection and state transfer yield time-sensitive traffic that requires suitable processing prioritization to avoid undesirable latency. A controller connected to a converged network is likely to receive and produce network traffic with varying levels of time sensitivity [81, 77]. For example, failure detection heartbeat messages are highly time-sensitive, whereas log file retrieval is less so. This question addresses how traffic can be assigned processing priority corresponding to the message's time sensitivity.

RQ 5: How can a distributed architecture for state storage be designed to reduce bottlenecks in deployments where a single backup serves multiple primary controllers?

Motivation: This research question primarily addresses **Chl. 4**, but it also relates to **Chl. 1**. Controller redundancy in a network-centric context is not necessarily limited to one backup per primary. Instead, a computationally powerful edge device could serve as a backup for multiple controllers [13]. Or, in an orchestrator-managed cluster of controllers, one controller could potentially act as the backup for multiple primaries. Once that controller becomes the primary, the orchestrator could designate the backup role to another suitable node in the cluster. However, in such cases, this single backup would receive aggregated state replication traffic from all the primaries it supports, thereby increasing the risk of network bottlenecks.

RQ 6: How can checkpointed internal state data be transferred reliably, securely, and with bounded transfer times between redundant controllers running multiple applications?
Motivation: This research question addresses **Chl. 4**. As previously mentioned in Section 2.1.1.2, a controller and its applications are stateful; hence, for a backup to resume the operation of a failed primary, the backup needs continuous updates of the primary's internal states. Motivated by the previously mentioned entrance of IT into the OT context, one of the overall purposes of this question is to learn from checkpointing solutions in the IT domain. From that, the goal is to find a solution for state transfer that enables time-bounded, secure, and reliable state data transfer from the primary to the backup controller.

3.3 Research Process

Figure 3.1 illustrates the research process used for all the included publications. This process follows the Design Science Research Process (DSRP) methodology [99]. We have grouped this process into three main steps: (i) formulating a research challenge, (ii) prototyping, experimenting, and evaluating, and (iii) publishing, as shown in Figure 3.1.

The process begins by identifying a research challenge that stems from an industrial problem, typically raised by the development organization of the industrial partner. Once a challenge or problem is identified, a literature review is conducted to learn the state-of-the-art and state-of-practice. If an existing solution is found in the literature, the problem identification is revisited and revised, or a new problem is addressed. If no existing solution is found, the scouting for knowledge to formulate a path to a solution begins. This path typically includes steps like providing a design or an algorithm, followed by

developing a prototype used for experimentation and evaluation against the solution's objective.

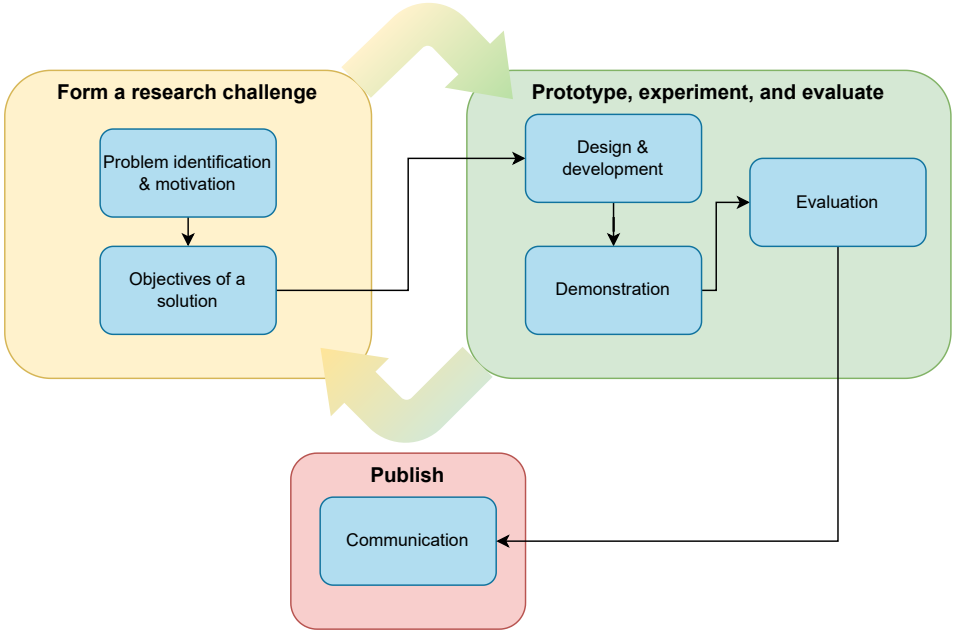


Figure 3.1: Illustration of the Design Science Research Process (DSRP) applied in this thesis, as outlined by Peffers et al. [99].

The research presented in this thesis is primarily experimental, addressing identified problems with algorithms and prototypes used in experiments. The ambition has been to design pragmatic experiments, making them feasible to perform yet realistic enough to provide reasonable confidence in evaluating the demonstrated properties and capabilities. Given the nature of research, knowledge gained in later steps might necessitate revisiting earlier steps. However, eventually, the process reaches a point where newly gained knowledge can be communicated.

Communication involves publishing. Since most of the work in this thesis is industry-oriented, most publications are published at conferences with a strong industry presence, such as Emerging Technology and Factory Automation (ETFA). The process restarts with a new problem after completing a publication, possibly inspired by the recent publication.

The above is the overall description of the research process followed; the

following subsection describes how each included publication aligns with the process.

3.3.1 DSRP Alignment - Publication A

Problem identification and motivation. The problem addressed in paper A originates from **Chl. 1**, addresses **RQ 1**, and is motivated by the industry's need to understand how an orchestrator can complement controller redundancy. A search for related literature that addresses the problem was conducted as part of this step.

Objectives of a solution. From the problem, we deduced a quantitative objective: to determine the reparation time of failed controllers, redundant and non-redundant, when managed by an orchestrator. Reparation time was defined as the duration during which the virtualized controller, the VDCN, is unavailable to external devices that depend on communication with the cluster-hosted VDCN. A repaired VDCN should also resume with its latest state. A second design objective was to understand how to deploy a virtualized industrial controller in a bare-metal cluster, which is a prerequisite for achieving the first objective.

Design and development. A prototype of orchestrated industrial controllers was designed and developed. The prototype utilized OPC UA-based communication, using both Client/Server and PubSub to support different communication models. The inclusion of OPC UA communication enabled external devices to measure the VDCN downtime and thereby recovery time.

Demonstration. To measure recovery time, a single and redundant configured VDCN was deployed on the designed and developed Kubernetes-managed bare-metal cluster. Failures were introduced by automatically and randomly rebooting the nodes hosting the primary and single VDCN.

Evaluation. The evaluation involved measuring the time required to restore communication with the VDCN, as well as verifying whether the VDCN resumed operation with its latest state without issues.

Communication. The results are described in publication A and presented at a conference, see Section 4.3.

3.3.2 DSRP Alignment - Publication B

Problem identification and motivation. The problem addressed in paper B originates from **Chl. 2** and addresses **RQ 2**, specifically, it comes from the industry's need to ensure well-defined behavior even in the presence of network partitioning that isolates the redundant controllers from each other.

Objectives of a solution. The objective, deduced from the problem, is to present a failure detection mechanism suitable for industrial controller redundancy that, to a higher degree than conventional failure detection, distinguishes between an islanding situation and a primary failure. The solution should prioritize consistency to preserve deterministic external behavior when making the CAP theorem-enforced tradeoff between consistency and availability in case of partitioning [97].

Design and development. Two artifacts were developed: (i) a failure-detection algorithm, and (ii) a proof-of-concept prototype realization of the developed algorithm.

Demonstration. The prototype was deployed on two computers connected via a redundant network to simulate a redundant controller pair. The controller pair was connected to simulated I/O, which both received values from and sent values to the controller pair.

Evaluation. The prototype was compared against conventional failure detection under various network failure scenarios. The behavior of the redundant controller pair towards the simulated I/O was analyzed and compared.

Communication. The results are described in publication B and presented at a conference, see Section 4.3.

3.3.3 DSRP Alignment - Publication C

Problem identification and motivation. The problem addressed in paper C originates from **Chl. 3** and addresses **RQ 3**, specifically, how a redundant OPC UA PubSub using controller (or device) can failover without disrupting the OPC UA PubSub communication. As mentioned earlier, see Section 2.4, OPC UA PubSub is a candidate for future field communication and needs to operate in spatially redundant settings. The work began as an investigation of OPC UA PubSub during failover in spatial redundancy configurations, where potential issues were identified related to the failover not being seamless.

Objectives of a solution. The solution's objective is straightforward: to ensure bumpless failover of redundant controllers and devices using OPC UA PubSub. The second, equally important, objective is to preserve standard compliance. Alternative solutions are discussed.

Design and development. A standard-compliant solution was identified, and a prototype design evaluating that solution was described and developed.

Demonstration. The prototype was deployed and tested in simulated failover scenarios, where publishers provided updated values to subscribers.

Evaluation. The prototype solution to the identified problem was compared to OPC UA PubSub implementations that did not include the prototype

solution.

Communication. The results are described in publication C and presented at a conference, see Section 4.3.

3.3.4 DSRP Alignment - Publication D

Problem identification and motivation. The problem addressed in paper D originates mainly from **Chl. 2** and **Chl. 4**, and it addresses **RQ 4**. Given the network dependency of a network-centric controller, processing network traffic becomes a central function. Additionally, when time-sensitive redundancy functions such as state replication and failure detection rely on the network, ensuring suitable processing priority of traffic is essential.

Objectives of a solution. The objective of the solution, deduced from the problem, is to differentiate the processing priority given to time-sensitive traffic and best-effort traffic to reduce latency potential induced by best-effort traffic on the time-sensitive traffic. Furthermore, processing best-effort traffic with high priority could lead to best-effort traffic-induced latency on high-priority tasks, an effect that the solution should reduce.

Design and development. A design is presented where a commonly available mechanism in standard Ethernet controllers is used to direct traffic to different threads based on priority-indicating values in the received Ethernet frame. The design is implemented and realized on VxWorks, a commonly used real-time OS in industrial contexts [100].

Demonstration. A prototype was deployed and tested under different traffic combinations, in conjunction with tasks of varying priorities and execution time, emulating the execution of control applications.

Evaluation. The prototype, which includes differentiation of traffic processing based on priority levels indicated in the received Ethernet frames, was compared to an implementation lacking such differentiation. The evaluation measured both the effect of best-effort traffic on time-sensitive, critical tasks, and the latency in the reception of time-sensitive traffic.

Communication. The results are described in publication D and presented at a conference, see Section 4.3.

3.3.5 DSRP Alignment - Publication E

Problem identification and motivation. The problem addressed in paper E originates from **Chl. 4**, and it addresses **RQ 5**. A potential future deployment scenario could involve multiple active controllers sharing a single backup controller. Using the conventional state replication approach in such a scenario

would result in all controllers replicating to that single backup, potentially causing it, or the shared network paths, to become a bottleneck.

Objectives of a solution. The objective of the solution, deduced from the problem, is to provide a state replication mechanism suitable for industrial controllers that distributes state replication-induced traffic more evenly, reducing the probability that the backup becomes the bottleneck due to an inability to handle the traffic load.

Design and development. An architecture and design for a distributed solution is described. This solution was also implemented on top of VxWorks.

Demonstration. A prototype was deployed on a set of virtual machines simulating redundant controllers, with one of these controllers acting as the sole backup.

Evaluation. The distributed state replication solution of the prototype was compared against the conventional state replication approach, using different set sizes of primary controllers sharing a single backup.

Communication. The results are described in publication E and presented at a conference, see Section 4.3.

3.3.6 DSRP Alignment - Publication F

Problem identification and motivation. The problem addressed in paper F originates from **Chl. 4**, and it addresses **RQ 6**. As mentioned, the internal states must be available to the backup when it becomes the primary, so that the backup can continue from where the primary left off. The data that constitutes these states must be retrieved and transferred reliably and securely to the backup. While reliable and secure real-time communication is not new per se, a challenge related to the redundancy use case is the many-to-one relationship between the multiple applications hosted by a controller and the single underlying network connecting the primary and backup. The use of this network should be managed to avoid overutilization and ensure bounded transfer times with a known transfer failure rate.

Objectives of a solution. The objective of the solution, deduced from the problem, is to provide a performant, reliable, and secure transfer mechanism for checkpointed state data. A comprehensive search, covering both IT and OT contexts, was conducted to identify a suitable solution. This search included checkpointing solutions as well as presenting a set of features desired from a protocol used to transfer state data, followed by matching existing protocols against these features. The objective of the solution is to meet the proposed set of desired features.

Design and development. The results of the searches are presented. Fol-

lowing that, a protocol design and a proposed VxWorks integration are described. A prototype was developed based on the proposed design. The security mechanisms were not included in the prototype.

Demonstration. The prototype was run on two mini-PCs running VxWorks. These two mini-PCs represent the redundant controller pair. Different state data were transmitted using varying data sizes and intervals, combined with different degrees of packet loss.

Evaluation. The proposed protocol was compared against two of the most suitable candidates identified during the protocol search. The comparison was conducted using different data sizes and packet loss rates.

Communication. The results are described in publication F and are targeted for journal publication, see Section 4.3.

3.4 Research Framed in Industrial Context

As mentioned, this thesis is the result of close collaboration with industry, as the author is an industrial PhD candidate. By this point in the thesis, it should be clear to the reader that the challenges addressed in the research stem from industry. Hence, the proposed solutions target the domain and context of industrial control systems. As such, the findings may have limited generalizability. However, as Briand et al. argue, context-driven research is valuable because it presents relevant use cases that address real-world problems [101]. Addressing real problems and shedding light on them has been the overarching goal of this thesis.

As an industrial practitioner who crossed paths with academia to pursue a PhD after many years in the industry, the author believed that bringing industrial challenges to academia was the most meaningful way to contribute, rather than tackling already established academic challenges. Therefore, the research conducted in this thesis is highly applied and is strongly influenced by industry. That is also a contributing factor to the overall and recurring theme of this thesis, the theme of "revisiting." As mentioned, fault tolerance is a mature research area, not receiving as much attention as research in areas such as artificial intelligence at the time of writing. Even so, given that technology constantly evolves, it can be motivated (as argued by this thesis) to revisit established concepts to re-ground them in new architectures and technologies.

The close collaboration with industry also means that the work has been, and could be, applied to real industrial systems. For example, in Publication A, we used real industrial control systems. However, this also means that the level of detail we can disclose is sometimes limited. As a result, in most of our

work, we used abstractions of the controller functionality that could be shared and described in more detail.

3.5 Relation to the Licentiate Thesis

As part of the author's journey as a PhD candidate, a licentiate thesis has been completed and defended [102]. The topic of the licentiate thesis is industrial control systems dependability, and it is titled *"Dependable Distributed Control System: Redundancy and Concurrency Defects"*. As the title suggests, the licentiate thesis addressed dependability, which is a broad topic, and the included publications covered aspects such as localization of concurrency-related bugs and spatial redundancy-related topics like failure detection. Below is the list of the publications included in the licentiate thesis that, like this thesis, is a collection of publications:

Lic. 1: *Concurrency defect localization in embedded systems using static code analysis: an evaluation*
Bjarne Johansson, Alessandro V. Papadopoulos, Thomas Nolte.
In 30th IEEE International Symposium on Software Reliability Engineering (ISSRE), 2019 [16].

Lic. 2: *Heartbeat Bully: Failure Detection and Redundancy Role Selection for Network-Centric Controller*
Bjarne Johansson, Mats Rågberger, Alessandro V. Papadopoulos, Thomas Nolte.
In 46th Annual Conference of the Industrial Electronics Society (IECON), 2020 [35].

Lic. 3: *Kubernetes Orchestration of High Availability Distributed Control Systems*
Bjarne Johansson, Mats Rågberger, Alessandro V. Papadopoulos, and Thomas Nolte.
In 23rd IEEE International Conference on Industrial Technology (ICIT), 2022 [**Paper A**].

This doctoral thesis continues the dependability path, focusing on spatial redundancy in network-centric control system architectures.

Chapter 4

Contributions

This thesis addresses the research question described in Section 3.2, deduced from the challenges describe in Section 3.1, mainly focusing on spatial controller redundancy, in a network-centric control system. This chapter provide a summary of the contribution, as well as mapping of the research challenges, research question, to contributions and publications.

4.1 Contribution Summary

Below is a summary of the contributions.

C1: Experimental measurements of failure recovery times for single and redundant containerized controllers, VDCNs, hosted within a Kubernetes cluster. These measurements provide insights into recovery times achievable through orchestration. Specifically, when paired with controller redundancy, the orchestrator contributes to system robustness by quickly restoring redundancy after failures, thereby minimizing the period of vulnerability.

C2: A failure detection algorithm designed to prevent multiple primary controllers in redundant controller pairs during network partitions. The algorithm, validated experimentally, ensures consistency by allowing at most one primary controller while lowering the availability trade-off dictated by the CAP theorem.

C3: An investigation of OPC UA PubSubs' seamless failover properties in spatial standby redundancy deployments, identifying challenges and exploring potential solutions. The contribution includes a discussion of

these solution alternatives and an experimental evaluation demonstrating one selected approach.

C4: A method for directing incoming Ethernet frames to network-stack processing in network tasks with priority matching message priority. The proposed solution operates without requiring TSN and is compatible with COTS network interfaces. The contribution includes a prototype used for experimental evaluation.

C5: An architecture that enables a primary controller to replicate its state to dedicated state storage that does not reside on the primary and does not necessarily reside on the backup controller, thereby decoupling state management from the backup role. This contribution also includes an experimentally evaluated prototype implementation.

C6: A protocol for transferring checkpointed state data between spatially redundant industrial controllers. The contribution includes an architectural description and an experimental evaluation. It also includes a presentation of checkpointing-related work in IT and OT environments and an examination of existing communication protocols and their suitability for state transfer between a primary and backup controller.

4.2 Contribution Mapping

Table 4.1 shows the mapping between research questions and contributions, while Table 4.2 illustrates the mapping between contributions and publications. Although the questions, contributions, and publications are neatly aligned in these tables, some aspects of this thesis research story get lost in translation when represented only in tabular form. Therefore, the remainder of this section elaborates on how each contribution developed, mapping the relationships with research challenges, research questions, and publications, including publications not included in this thesis. Figure 4.1 provides an overview, which is further explained below for each contribution.

Contribution C1, as illustrated in Figure 4.1, results from addressing research question 1 (**RQ 1**), which was derived from research challenge 1 (**Chl. 1**). **C1** is described and published in publication A (**Pub. A**). The work presented in **Pub. A** inspired **RQ 5**, as **C1** demonstrated that redundancy can be automatically restored. This prompt restoration makes an architecture in which multiple primaries share a single backup more feasible, since a "consumed" backup (due to a primary failure) can be promptly replaced by deploying a

Table 4.1: Question – contribution mapping.

	RQ 1	**RQ 2**	**RQ 3**	**RQ 4**	**RQ 5**	**RQ 5**
C1	X					
C2		X				
C3			X			
C4				X		
C5					X	
C6						X

Table 4.2: Contribution – publication mapping.

	C1	**C2**	**C3**	**C4**	**C5**	**C6**
Paper A	X					
Paper B		X				
Paper C			X			
Paper D				X		
Paper E					X	
Paper F						X

new backup. Furthermore, **C1** led to a patent application, resulting in granted patent 2 (**GP 2**).

Contribution C2 results from addressing **RQ 2**, which was derived from **Chl. 2**. As illustrated in Figure 4.1, **Chl. 2** was inspired by an earlier publication not included in this thesis, publication **X11** [35]. **X11** addressed failure detection, and the contribution from **X11** resulted in the granted patent **GP 2**. **C2** is described and published in **Pub. B**. The work and contribution associated with **Pub. B** inspired additional publications, **X4** [42], **X5** [39] and **X8** [43], which are not included in this thesis. Furthermore, **C2** led to patent application **PA 1**.

Contribution C3 results from addressing **RQ 3**, which was derived from **Chl. 3**. **C3** is described and published in **Pub. C**.

Contribution C4 results from addressing **RQ 4**, which was inspired by **Chl. 2** and **Chl. 4**. **C4** is described and published in **Pub. D**. Furthermore, **C4** was used as part of the solution that constitute contribution **C6**, described in **Pub. F**.

Contribution C5 results from addressing **RQ 5**, which was derived from **Chl. 4**. **C5** is described and published in **Pub. E**. The work towards **Pub. E** and **C5** inspired patent application **PA 2**.

Contribution C6 results from addressing **RQ 6**, which was derived from

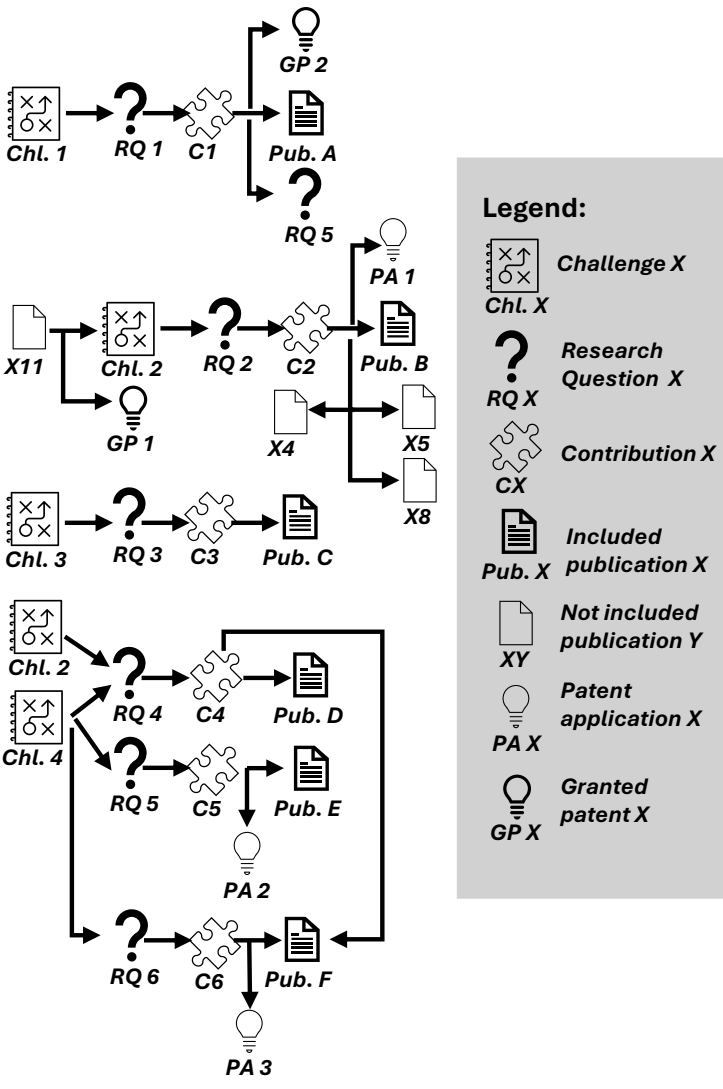


Figure 4.1: Challenges, questions, publications and contributions mapping.

Chl. 4. **C6** is described and published in **Pub. F**, parts of **C4** were used in the work towards **C6** and **Pub. F**. Furthermore, **C6** led to patent application **PA 3**.

4.3 Included Publications

This section outlines the publications included in this thesis. Table 4.2 maps the publication to the contributions.

4.3.1 Paper A

Title: Kubernetes Orchestration of High Availability Distributed Control Systems
Authors: Bjarne Johansson, Mats Rågberger, Alessandro V. Papadopoulos, and Thomas Nolte.
Status: Published in the proceedings of the 23rd IEEE International Conference on Industrial Technology (ICIT), 2022.
Abstract: Distributed control systems transform with the Industry 4.0 paradigm shift. A mesh-like, network-centric topology replaces the traditional controller-centered architecture, enforcing the interest of cloud-, fog-, and edge-computing, where lightweight container-based virtualization is a cornerstone. Kubernetes is a well-known container management system for container orchestration in cloud computing. It is gaining traction in edge- and fog-computing due to its elasticity and failure recovery properties. Orchestrator failure recovery can complement the manual replacement of a failed controller and, combined with controller redundancy, provide a pseudo-one-out-of-many redundancy. This paper investigates the failure recovery performance obtained from an out-of-the-box Kubernetes installation in a distributed control system scenario. We describe a Kubernetes based virtualized controller architecture and the software needed to set up a bare-metal cluster for control systems. Further, we deploy single and redundant configured containerized controllers based on an OPC UA compatible industry middleware software on the bare-metal cluster. The controllers expose variables with OPC UA PubSub. A script-based daemon introduces node failures, and a verification controller measures the downtime when using Kubernetes with an industry redundancy solution.
Paper Contributions: The paper presents an overview and description of the software components needed to set up a Kubernetes cluster for hosting containerized controllers. Two different controller configurations are used: a redundant for high availability and a singular as reference. Node failures are injected, and downtime is measured for the redundant and single controller.

The combination of Kubernetes failure recovery in collaboration with the controller redundancy forms a layered redundancy that reduces deterioration due to failures and automatically restores redundancy. A discussion around controller availability, with the gathered measurements as input, concludes the paper.
Patent: The concept, in which redundancy mechanisms in industrial controllers and orchestrator-based failure recovery collaborate to enable quick failover and rapid repair, led to Granted Patent 2 [103].
My role: I was the main driver and author of this work, collaborating with my supervisors and industry mentor, who provided valuable input and feedback.

4.3.2 Paper B

Title: Consistency Before Availability: Network Reference Point based Failure Detection for Controller Redundancy
Authors: Bjarne Johansson, Mats Rågberger, Alessandro V. Papadopoulos, and Thomas Nolte.
Status: Published in the proceedings of the 28th International Conference on Emerging Technologies and Factory Automation (ETFA), 2023.
Abstract: Distributed control systems constitute the automation solution backbone in domains where downtime is costly. Redundancy reduces the risk of faults leading to unplanned downtime. The Industry 4.0 appetite to utilize the device-to-cloud continuum increases the interest in network-based hardware-agnostic controller software. Functionality, such as controller redundancy, must adhere to the new ground rules of pure network dependency. In a standby controller redundancy, only one controller is the active primary. When the primary fails, the backup takes over. A typical network-based failure detection uses a cyclic message with a known interval, a.k.a. a heartbeat. Such a failure detection interprets heartbeat absences as a failure of the supervisee; consequently, a network partitioning could be indistinguishable from a node failure. Hence, in a network partitioning situation, a conventional heartbeat-based failure detection causes more than one active controller in the redundancy set, resulting in inconsistent outputs. We present a failure detection algorithm that uses network reference points to prevent network partitioning from leading to dual primary controllers. In other words, a failure detection that prioritizes consistency before availability.
Paper Contributions: This paper presents a failure detection algorithm for redundant controllers that prioritizes consistency over availability. It addresses the risk that conventional failure detectors may allow multiple active primaries during a network partition. The algorithm is heartbeat-based

and utilizes an external network reference point as a primary role tie-breaker. It leverages the fact that redundant controllers are typically deployed with redundant network paths, allowing for continued communication even in the event of a single network failure. Under specific conditions, the algorithm can dynamically reassign the reference point, reducing the availability tradeoff following from prioritizing consistency, as dictated by the CAP theorem. The paper contributes a failure detection algorithm along with an evaluation and comparison of its performance against a conventional failure detector across various partitioning scenarios.
Patent: As mentioned, the related publication X11 [35] is one of the inspirations for the work that led to Paper B. The work with X11 led to Granted Patent 1 [104]. The combination of failure detection with a redundancy role leasing function, preferably hosted in network equipment and runtime changeable in case of failure, led to Patent Application 1 [105].
My role: I was the main driver and author of this work, collaborating with my supervisors and industry mentor, who provided valuable input and feedback.

4.3.3 Paper C

Title: OPC UA PubSub and Industrial Controller Redundancy
Authors: Bjarne Johansson, Olof Holmgren, Martin Dahl, Håkan Forsberg, Alessandro V. Papadopoulos, and Thomas Nolte.
Status: Published in the proceedings of the 29th International Conference on Emerging Technologies and Factory Automation (ETFA), 2024.
Abstract: Industrial controllers constitute the core of numerous automation solutions. Continuous control system operation is crucial in certain sectors, where hardware duplication serves as a strategy to mitigate the risk of unexpected operational halts due to hardware failures. Standby controller redundancy is a commonly adopted strategy for process automation. This approach involves an active primary controller managing the process while a passive backup is on standby, ready to resume control should the primary fail. Typically, redundant controllers are paired with redundant networks and devices to eliminate any single points of failure. The process automation domain is on the brink of a paradigm shift towards greater interconnectivity and interoperability. OPC UA is emerging as the standard that will facilitate this shift, with OPC UA PubSub as the communication standard for cyclic real-time data exchange. Our work investigates standby redundancy using OPC UA PubSub, analyzing a system with redundant controllers and devices in publisher-subscriber roles. The analysis reveals that failovers are not subscriber-transparent without synchronized publisher states. We discuss solutions and experimentally validate

an internal stack state synchronization alternative.
Paper Contributions: This paper presents a study of OPC UA PubSub in the context of controller redundancy, with a focus on failover behavior. It identifies weaknesses that can lead to communication disruptions during failover. Different improvement alternatives are proposed and discussed. One of these alternatives is implemented and experimentally evaluated, and its performance is compared to an original (non-modified) version.
My role: I was the main driver and author of this work, collaborating with my supervisors and industry mentor, who provided valuable input and feedback.

4.3.4 Paper D

Title: Priority Based Ethernet Handling in Real-Time End System with Ethernet Controller Filtering
Authors: Bjarne Johansson, Mats Rågberger, Alessandro V. Papadopoulos, and Thomas Nolte.
Status: Published in the proceedings of the 48th Annual Conference of the Industrial Electronics Society (IECON), 2022.
Abstract: This work addresses the impact of best-effort traffic on network-dependent real-time functions in distributed control systems. Motivated by the increased Ethernet use in real-time dependent domains, such as the automation industry, a growth driven by Industry 4.0, interconnectivity desires, and data thirst. Ethernet allows different network-based functions to converge on one physical network infrastructure. In the automation domain, converged networks imply that functions with different criticality and real-time requirements coexist and share the same physical resources. The IEEE 60802 Time-Sensitive Networking profile for Industrial Automation targets the automation industry and addresses Ethernet network determinism on converged networks. However, the profile is still in the draft stage at the time of writing this paper. Meanwhile, Ethernet already provides attributes utilized by network equipment to prioritize time-critical communication. This paper shows that Ethernet Controller filtering with prioritized processing is a prominent solution for preserving real-time guarantees while supporting best-effort traffic. A solution capable of eliminating all best-effort traffic interference in the real-time application is exemplified and evaluated on a VxWorks system.
Paper Contributions: This paper presents a strategy for dispatching incoming Ethernet frames for processing by a network task with a priority matching that of the received Ethernet frame. The method enables elevated (differentiated) processing priority for prioritized traffic, and the paper

describes an implementation of the technique in a testbed that demonstrates its ability to reduce the impact of best-effort traffic on high-priority, time-sensitive, real-time tasks.
My role: I was the main driver and author of this work, collaborating with my supervisors and industry mentor, who provided valuable input and feedback.

4.3.5 Paper E

Title: Partible State Replication for Industrial Controller Redundancy
Authors: Bjarne Johansson, Olof Holmgren, Alessandro V. Papadopoulos, and Thomas Nolte.
Status: Published in the proceedings of the 25th IEEE International Conference on Industrial Technology (ICIT), 2024.
Abstract: Distributed control systems are part of the often invisible backbone of modern society that provides utility services like water and electricity. Their uninterrupted operation is vital, and unplanned stops due to failure can be expensive. Critical devices, like controllers, are often duplicated to minimize the service stop probability, with a secondary controller acting as a backup to the primary. A seamless takeover requires that the backup has the primary's latest state, i.e., the primary has to replicate its state to the backup. While this method ensures high availability, it can be costly due to hardware doubling. This work proposes a state replication solution that doesn't require the backup to store the primary state, separating state storage from the backup function. Our replication approach allows for more flexible controller redundancy deployments since one controller can be a backup for multiple primaries without being saturated by state replication data. Our main contribution is the partible state replication approach, realized with a distributed architecture utilizing a consensus algorithm. A partial connectivity-tolerant consensus algorithm is also an additional contribution.
Paper Contributions: This paper introduces the concept of partible state storage in the context of controller redundancy. Unlike conventional approaches, it decouples the storage of checkpointed state data, retrieved from the primary, from a designated backup, enabling distributed storage across the cluster. The paper proposes a partible state storage architecture that includes a consensus algorithm tolerant to partial connectivity, based on VSR. A prototype implementation is developed and used to evaluate the partible state storage approach in comparison to the conventional approach, comparing performance across varying numbers of primaries sharing a single backup.
Patent: The work on distributed state storage for load-balancing purposes

described in Paper E inspired Patent Application 2 [106]. The idea utilizes collected states to evaluate application changes when changes are introduced. Specifically, two application versions can be executed simultaneously, and their states can be compared without requiring both versions to run on the same controller. Thus, Patent Application 2 enables node-independent application evaluation.
My role: I was the main driver and author of this work, collaborating with my supervisors and industry mentor, who provided valuable input and feedback.

4.3.6 Paper F

Title: Checkpointing and State Transfer for Industrial Controller Redundancy
Authors: Bjarne Johansson, Björn Leander, Olof Holmgren, Alessandro V. Papadopoulos, and Thomas Nolte.
Status: Journal submission under review, September 2025.
Abstract: Industrial controllers are moving from controller-centric to network-centric architectures, where lightweight containerization is increasingly adopted in operational technology. Many industrial domains require high reliability, often achieved through spatial standby redundancy with duplicated controllers where one is the active primary and the other a standby backup. In such setups, the standby backup must seamlessly take over control when the primary fails. Hence, the backup needs to be up-to-date with respect to the primary's internal state. The retrieval of internal states is commonly known as checkpointing. We review checkpointing approaches used in virtualized and industrial settings and derive a set of desired features for state-transfer protocols. We then assess existing communication protocols against these features and experimentally evaluate the two strongest contenders under no-loss and packet-loss conditions, measuring recovery performance. The analysis reveals that no existing protocol meets all the desired features. To address this gap, we introduce a new state-transfer protocol that satisfies all identified features. In experiments, it demonstrates good performance under packet loss, with only a slight reduction in throughput compared to the identified top contender protocols that we used for comparison.
Paper Contributions: This paper investigates checkpointing and state transfer solutions suitable for controller redundancy. It presents the results of a search for checkpointing methods used in both the OT and IT domains, with a focus on the protocols used for state data transfer. Additionally, the paper presents a list of desired features for a protocol intended for state transfer purposes. This feature list is then used to evaluate a set of existing protocols

for their suitability in the industrial controller redundancy use case. Two top candidates are selected and experimentally evaluated in scenarios that, to some extent, mimic state transfer operations. The paper's main contribution is the design, integration, and evaluation of a new protocol that meets the identified desired feature. This protocol is experimentally compared against the two top candidates.

Patent: The work presented in Paper F inspired Patent Application 3 [107], which describes a concept for scheduling state data transfers from multiple applications with varying periods. The idea enables deterministic shared use of a node's network resources. It includes a reliability model that considers retransmission budgets, as well as the overhead introduced by security and safety mechanisms.

My role: I was the main driver and author of this work, collaborating with my supervisors, co-authors, and industry mentor, who provided valuable input and feedback. Björn Leander was the main contributor of the security related parts.

Chapter 5

Conclusions and Future Directions

This chapter provides a summary of this thesis contributions and presents the conclusions. A discussion of future work, derived from the research challenges, concludes the chapter and this part of the thesis.

5.1 Summary and Conclusions

When reading the summary of the contributions and the conclusions derived from addressing the research questions leading to the contributions, you, the reader, might get the impression that there is nothing more to be done—that the work is completed in an absolute sense. That is, of course, not the case; the description of future work is simply reserved for the next section, Section 5.2.

The first challenge addressed in this thesis concerned orchestration-aided failure recovery in an industrial controller setting. To address this challenge, we formulated a research question around whether orchestrators can complement or even replace redundancy mechanisms in industrial controllers. The first contribution came from the pursuit of that question, a description of a bare-metal cluster hosting VDCNs (virtualized controllers), configured as single or redundant, and orchestrated by Kubernetes. Failures, in the form of reboots, were continuously and randomly injected to trigger the failure recovery mechanisms. Recovery times were measured from devices dependent on communication with the failed (rebooted) VDCNs. The conclusion of this work is that controllers relying solely on Kubernetes' failure detection and recovery mechanisms experience healing times that are too long to serve as a feasible replacement for traditional redundancy. However, Kubernetes failure recovery

can synergistically complement controller redundancy, enabling rapid redundancy restoration after a failure.

The second challenge concerned failure detection in network partitioning scenarios. From this challenge, we derived a research question aimed at finding a suitable failure detection mechanism for redundant industrial controllers that can maintain consistent process control despite partitioning. To address this, we developed a heartbeat-based failure detection algorithm that uses an external reference when determining the primary role. This external reference can be changed if needed, leveraging the fact that redundant controllers are commonly deployed with network redundancy. The conclusion is that such a failure detection mechanism can be designed and implemented, even using COTS network equipment as the external reference.

The third challenge addressed OPC UA PubSub in the context of spatial controller redundancy. We derived the third research question from this challenge, which focuses on achieving disruption-free failover of OPC UA PubSub using controllers and devices. The resulting contribution is a presentation of OPC UA PubSub behavior during failover, from both the publisher and subscriber perspectives. We identified that disruption-free failover cannot be guaranteed in all configurations. The contribution includes alternative approaches to address these shortcomings, including the experimental demonstration of one. The conclusion is that OPC UA PubSub, in some configurations, may face challenges in providing disruption-free failover without explicit redundancy support.

The fourth challenge involves ensuring that the backup can use the internal state of the primary so that it can assume the primary role without disruption. Like the second challenge related to failure detection, this challenge requires real-time networking in the sense of bounded communication times. These two challenges inspired the fourth research question addressed in the thesis: how can processing of time-sensitive traffic be prioritized over best-effort traffic? To explore this, we developed a solution to direct network traffic based on priority information in incoming frames to network tasks of suitable priority. The conclusion is that such a mechanism is feasible and can be deployed using COTS Ethernet network interfaces on real-time operating system platforms such as VxWorks, and that is the fourth contribution.

The fifth contribution results from addressing the fifth research question, which is inspired by the work leading to the first contribution and motivated by challenge four. The fifth research question concerns the design of an architecture that allows state data to be replicated to a node other than the primary, though not necessarily the backup. The goal is to prevent a single backup from becoming a bottleneck for state data transfer in deployments where multiple

primaries share a single backup. The resulting contribution is a distributed architecture enabling state data distribution across the cluster. The conclusion is that such an architecture is feasible and can reduce bottleneck effects.

The sixth contribution also originates from the fourth challenge and addresses research question six, which focuses on finding a suitable solution for transferring checkpointed state data from a primary to a backup. "Suitable" here means fulfilling features derived from redundancy-related needs. The contribution is a protocol designed to meet these needs. A prototype implementation is compared against two well-known communication protocols, which were found to be top candidates by matching desired state transfer features against the properties provided by the protocols. The conclusion is that while the proposed protocol shows slightly lower performance in loss-free transfers, it significantly outperforms the alternatives under packet loss conditions.

In summary, we have addressed challenges and provided contributions related to spatial controller redundancy. Two of these challenges concern essential functions for standby redundancy: failure detection and state transfer. A third challenge focused on the failover behavior of OPC UA PubSub, which is critical given OPC UA's envisioned role in future automation systems. The fourth challenge addressed network traffic processing, a crucial aspect considering that redundancy functions rely heavily on network communication.

5.2 Future Directions

Each of the challenges presented in Section 3.1, that yielded the contributions summarized above, presents directions for future work, exemplified below.

Orchestration and Failure Recovery. As mentioned, the conclusion of our work on orchestrators complementing or replacing redundancy functionality showed that Kubernetes failure recovery takes too long to serve as a redundancy replacement. However, that does not mean it must remain this way; there are likely many potential improvements that could reduce recovery time, perhaps even enough to make it a feasible replacement. Since Kubernetes relies on virtualized networks and overlays, the performance characteristics of these, particularly under contention, form another highly relevant research challenge.

Failure Detection and Network Partitioning. This thesis addresses failure detection under network partitioning, proposing a heartbeat-based algorithm that prioritizes consistency by utilizing an external reference point within the network. In related work, not included in this thesis, we extended the algorithm with a redundancy role leasing mechanism, which we formally proved

to guarantee at most one primary in publication **X8** [43]. This concept was further explored when discussing the design of a high-integrity system for safety-critical applications in publication **X4** [42]. That work represents a starting point that could be further developed. Additionally, the proposed algorithms in an orchestrated context could be investigated, can the orchestrator, as a central element in the system architecture, contribute to failure detection?

OPC UA PubSub and Spatial Redundancy. As mentioned, our work showed that spatially redundant controllers and devices utilizing OPC UA PubSub might not guarantee a disruption-free failover. We discussed alternatives and experimentally evaluated one with a prototype. Future work could explore how platform-agnostic redundancy support for OPC UA PubSub could be integrated into the open-source stack, open62541 [108].

Checkpointing and State Transfer. This thesis presents a design that enables state data from primary controllers to be distributed to nodes other than the backup node. Future work in this area includes evaluating the mechanism in a cluster managed by an orchestrator. Other directions include determining how to distribute the state efficiently, including defining what "efficient" means in this context. "Efficient" encompasses various parameters, including the time it takes for the backup to retrieve the state and the network load incurred by storing and fetching the state, among others. This thesis also proposes a protocol designed for state transfer between redundant industrial controllers. The design outlines security integration. However, the prototype implementation does not include any security measures. Hence, evaluating a prototype that includes security measures is suitable for future work. Furthermore, investigating the integration of the proposed protocol and its scheduling with the failure detection mechanisms to enable scheduling, security, and retransmission of heartbeat messages for failure detection is another example of relevant future work.

The above list of future work is not intended to be exhaustive; it should rather be seen as a set of examples, examples identified while writing this thesis and having the pleasure of being yet another dwarf sitting on the shoulders of giants and observing the winds of change [109].

Bibliography

[1] Daniel Hallmans, Mohammad Ashjaei, and Thomas Nolte. Analysis of the TSN standards for utilization in long-life industrial distributed control systems. In *IEEE Int. Conf. Emerg. Tech. & Fact. Autom. (ETFA)*, pages 190–197, 2020.

[2] Martin A Sehr, Marten Lohstroh, Matthew Weber, Ines Ugalde, Martin Witte, Joerg Neidig, Stephan Hoeme, Mehrdad Niknami, and Edward A Lee. Programmable logic controllers in the context of industry 4.0. *IEEE Transactions on Industrial Informatics*, 17(5):3523–3533, 2020.

[3] Stefan Stattelmann, Stephan Sehestedt, and Thomas Gamer. Optimized incremental state replication for automation controllers. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–8. IEEE, 2014.

[4] O-pas standard, version 2.1. `https://publications.opengroup.org/c230`. Accessed: 2025-05-06.

[5] Stefano Vitturi, Claudio Zunino, and Thilo Sauter. Industrial communication systems and their future challenges: Next-generation ethernet, iiot, and 5g. *Proceedings of the IEEE*, 107(6):944–961, 2019.

[6] HMS. Industrial network market shares 2025 according to hms networks. `https://www.hms-networks.com/news/news-details/27-05-2025-hms-networks-report-industrial-trends-2025`. Accessed: 2025-06-04.

[7] Dietmar Bruckner, Marius-Petru Stănică, Richard Blair, Sebastian Schriegel, Stephan Kehrer, Maik Seewald, and Thilo Sauter. An introduction to OPC UA TSN for industrial communication systems. *Proc. IEEE*, 107(6):1121–1131, 2019.

[8] Björn Leander, Bjarne Johansson, Tomas Lindström, Olof Holmgren, Thomas Nolte, and Alessandro V. Papadopoulos. Dependability and security aspects of network-centric control. In *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2023.

[9] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan 2004.

[10] Algirdas Avižienis. Design of fault-tolerant computers. In *Proceedings of the November 14-16, 1967, fall joint computer conference*, pages 733–743, 1967.

[11] Elena Dubrova. *Fault-tolerant design*. Springer, 2013.

[12] Andrei Simion and Calin Bira. A review of redundancy in plc-based systems. *Advanced Topics in Optoelectronics, Microelectronics, and Nanotechnologies XI*, 12493:269–276, 2023.

[13] T. Hegazy and M. Hefeeda. Industrial automation as a cloud service. *IEEE Transactions on Parallel and Distributed Systems*, 26(10):2750–2763, Oct 2015.

[14] Bojana Bajic, Aleksandar Rikalovic, Nikola Suzic, and Vincenzo Piuri. Industry 4.0 implementation challenges and opportunities: A managerial perspective. *IEEE Systems Journal*, 15(1):546–559, 2020.

[15] Victor P. Nelson. Fault-tolerant computing: Fundamental concepts. *Computer*, 23(7):19–25, 1990.

[16] Bjarne Johansson, Alessandro V Papadopoulos, and Thomas Nolte. Concurrency defect localization in embedded systems using static code analysis: An evaluation. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 7–12. IEEE, 2019.

[17] Jacek Stój. Cost-effective hot-standby redundancy with synchronization using ethercat and real-time ethernet protocols. *IEEE Trans. on Autom. Science and Eng.*, 18(4):2035–2047, 2020.

[18] Y.C. Yeh. Triple-triple redundant 777 primary flight computer. In *1996 IEEE Aerospace Applications Conference. Proceedings*, volume 1, pages 293–307 vol.1, 1996.

[19] Jacques Losq. *Influence of fault detection and switching mechanisms on the reliability of stand-by systems*, volume 75. Digital Systems Laboratory, Stanford Electronics Laboratories, Stanford Univ., 1975.

[20] PACSys. Pacsystems™ rx3i hot standby cpu redundancy. https://emerson-mas.my.site.com/communities/en_US/Documentation/PACSystems-Hot-Standby-CPU-Redundancy-Users-Manual, 2023. Accessed: 2024-07-22.

[21] Siemens. Siemens system manual s7-1500r/h redundant system. https://cache.industry.siemens.com/dl/files/833/109754833/att_965668/v4/s71500rh_manual_en-US_en-US.pdf, 2024. Accessed: 2025-05-07.

[22] ABB. Ac 800m controller hardware product guide. https://library.e.abb.com/public/61d046e80d875c7dc1257b5d000eb9bd/3BSE036352-510_A_en_AC_800M_5.1_Controller_Hardware_Product_Guide.pdf, 2024. Accessed: 2025-05-07.

[23] David Powell, Gottfried Bonn, Douglas T Seaton, Paulo Verissimo, and François Waeselynck. The delta-4 approach to dependability in open distributed computing systems. In *FTCS*, volume 18. Citeseer, 1988.

[24] Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg. Byzantine fault tolerance, from theory to reality. In *International Conference on Computer Safety, Reliability, and Security*, pages 235–248. Springer, 2003.

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

[26] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[27] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, 1996.

[28] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. A new adaptive accrual failure detector for dependable distributed systems. In *In ACM Symposium on Applied Computing (SAC 2007*, pages 551–555, 2007.

[29] Mikel Larrea, Sergio Arévalo, and Antonio Fernndez. Efficient algorithms to implement unreliable failure detectors in partially synchronous

systems. In *Distributed Computing: 13th International Symposium, DISC'99 Bratislava, Slovak Republic September 27–29, 1999 Proceedings 13*, pages 34–49. Springer, 1999.

[30] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[31] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. A lazy monitoring approach for heartbeat-style failure detectors. *2008 Third International Conference on Availability, Reliability and Security*, pages 404–409, 2008.

[32] D. Dzung, R. Guerraoui, D. Kozhaya, and Y. Pignolet. Never say never – probabilistic and temporal failure detectors. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 679–688, May 2016.

[33] Wei Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32, Jan 2002.

[34] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. Comput.*, 31(1):48–59, January 1982.

[35] B. Johansson, M. Rågberger, A. V. Papadopoulos, and T. Nolte. Heartbeat bully: Failure detection and redundancy role selection for network-centric controller. In *IECON*, 2020.

[36] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[37] Microsoft. What is a quorum witness? `https://learn.microsoft.com/en-us/windows-server/failover-clustering/what-is-quorum-witness`. Accessed: 2025-07-11.

[38] Deltav virtualization hardware for hyperconverged infrastructure - product data sheet. `https://www.emerson.com/documents/automation/product-data-sheet-deltav-virtualization-hardware-for-hy pdf`. Accessed: 2025-07-15.

[39] Kasra Ekrad, Sebastian Leclerc, Bjarne Johansson, Inés Alvarez Vadillo, Saad Mubeen, and Mohammad Ashjaei. Real-time fault diagnosis of node and link failures for industrial controller redundancy. In *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4. IEEE, 2024.

[40] Nils Dorsch, Fabian Kurtz, Felix Girke, and Christian Wietfeld. Enhanced fast failover for software-defined smart grid communication networks. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2016.

[41] Dave Katz and David Ward. Bidirectional Forwarding Detection (BFD). RFC 5880, June 2010.

[42] Bjarne Johansson, Olof Holmgren, Håkan Forsberg, Thomas Nolte, and Alessandro V Papadopoulos. Towards high-integrity redundancy role leasing. In *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4. IEEE, 2024.

[43] Bjarne Johansson, Bahman Pourvatan, Zahra Moezkarimi, Alessandro Papadopoulos, and Marjan Sirjani. Formal verification of consistency for systems with redundant controllers. *arXiv preprint arXiv:2403.18917*, 2024.

[44] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993.

[45] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

[46] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, pages 51–58, 2001.

[47] Harald Ng, Seif Haridi, and Paris Carbone. Omni-paxos: Breaking the barriers of partial connectivity. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 314–330, 2023.

[48] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pages 305–319, 2014.

[49] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems.

In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17, 1988.

[50] Barbara Liskov and James Cowling. Viewstamped replication revisited. 2012.

[51] etcd - a distributed, reliable key-value store for the most critical data of a distributed system. `https://etcd.io/`. Accessed: 2025-07-16.

[52] Satya Ram Tsaliki. Etcd notification latency reduction using approximate breadth first search abfs graph algorithm. *Available at SSRN 5195304*, 2023.

[53] Kubernetes webpage. `https://kubernetes.io/`. Accessed: 2025-04-02.

[54] Rhaban Hark, Heiko Koziolek, Vladimir Yussupov, and Nafise Eskandani. Kubernetes high-availability software architecture options for two-node clusters in iot applications. In *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*, pages 69–76. IEEE, 2025.

[55] Jack Dongarra, Thomas Herault, and Yves Robert. *Fault tolerance techniques for high-performance computing*. Springer, 2015.

[56] Veloc: Very-low overhead checkpointing system. `https://github.com/ECP-VeloC/VELOC`. Accessed: 2025-07-17.

[57] Fti: Fault tolerance interface. `https://github.com/leobago/fti/tree/master`. Accessed: 2025-07-17.

[58] Criu webpage. `https://criu.org/Main_Page`. Accessed: 2025-07-17.

[59] Thomas Goldschmidt, Mahesh Kumar Murugaiah, Christian Sonntag, Bastian Schlich, Sebastian Biallas, and Peter Weber. Cloud-based control: A multi-tenant, horizontally scalable soft-PLC. *2015 IEEE 8th International Conference on Cloud Computing*, pages 909–916, 2015.

[60] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

[61] Lorenzo Rosa, Andrea Garbugli, Lorenzo Patera, and Luca Foschini. Supporting vplc networking over tsn with kubernetes in industry 4.0. In

Proceedings of the 1st Workshop on Enhanced Network Techniques and Technologies for the Industrial IoT to Cloud Continuum, pages 15–21, 2023.

[62] Thomas Kampa, Amer El-Ankah, and Daniel Grossmann. High availability for virtualized programmable logic controllers with hard real-time requirements on cloud infrastructures. In *2023 IEEE 21st International Conference on Industrial Informatics (INDIN)*, pages 1–8. IEEE, 2023.

[63] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, San Jose, CA, June 2013. USENIX Association.

[64] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Elasticity in cloud computing: state of the art and research challenges. *IEEE Transactions on services computing*, 11(2):430–447, 2017.

[65] Mohammad Masdari, Sayyid Shahab Nabavi, and Vafa Ahmadi. An overview of virtual machine placement schemes in cloud computing. *Journal of Network and Computer Applications*, 66:106–127, 2016.

[66] Emiliano Casalicchio. Container orchestration: A survey. *Systems Modeling: Methodologies and Tools*, pages 221–235, 2019.

[67] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *IEEE Int. Symp. Perf. Analysis of Syst. and Software*, pages 171–172, 2015.

[68] Alexandru Moga, Thanikesavan Sivanthi, and Carsten Franke. Os-level virtualization for industrial automation systems: Are we there yet? In *SAC*, 2016.

[69] Václav Struhár, Moris Behnam, Mohammad Ashjaei, and Alessandro V. Papadopoulos. Real-Time Containers: A Survey. In *Fog-IoT*, 2020.

[70] Thomas Goldschmidt, Stefan Hauck-Stattelmann, Somayeh Malakuti, and Sten Grüner. Container-based architecture for flexible industrial control applications. *J. Syst. Arch.*, 84, 2018.

[71] Henri Schmidt, Zeineb Rejiba, Raphael Eidenbenz, and Klaus-Tycho Förster. Transparent fault tolerance for stateful applications in kubernetes with checkpoint/restore. In *2023 42nd International Symposium on Reliable Distributed Systems (SRDS)*, pages 129–139. IEEE, 2023.

[72] Waqas Ali Khan, Lukasz Wisniewski, Dorota Lang, and Jürgen Jasperneite. Analysis of the requirements for offering industrie 4.0 applications as a cloud service. In *2017 IEEE 26th international symposium on industrial electronics (ISIE)*, pages 1181–1188. IEEE, 2017.

[73] Dániel Géhberger, Dávid Balla, Markosz Maliosz, and Csaba Simon. Performance evaluation of low latency communication alternatives in a containerized cloud environment. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 9–16. IEEE, 2018.

[74] Heiko Koziolek, Andreas Burger, PP Abdulla, Julius Rückert, Shardul Sonar, and Pablo Rodriguez. Dynamic updates of virtual plcs deployed as kubernetes microservices. In *European Conference on Software Architecture*, pages 3–19. Springer, 2021.

[75] Heiko Koziolek, Andreas Burger, and Abdulla Puthan Peedikayil. Fast state transfer for updates and live migration of industrial controller runtimes in container orchestration systems. *Journal of Systems and Software*, 211:112004, 2024.

[76] Björn Leander, Bjarne Johansson, Saad Mubeen, Mohammad Ashjaei, and Tomas Lindström. Redundancy link security analysis: An automation industry perspective. In *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2024.

[77] Ilja Behnke and Henrik Austad. Real-time performance of industrial iot communication technologies: A review. *IEEE Internet of Things Journal*, 2023.

[78] Norman Finn. Introduction to time-sensitive networking. *IEEE Comm. Stand. Mag.*, 2(2):22–28, 2018.

[79] Lucia Lo Bello and Wilfried Steiner. A perspective on IEEE time-sensitive networking for industrial communication and automation systems. *Proc. IEEE*, 107(6):1094–1120, 2019.

[80] Daniel Bujosa, Mohammad Ashjaei, Alessandro V Papadopoulos, Thomas Nolte, and Julián Proenza. Hermes: Heuristic multi-queue

scheduler for tsn time-triggered traffic with zero reception jitter capabilities. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, pages 70–80, 2022.

[81] Ilja Behnke, Lukas Pirl, Lauritz Thamsen, Robert Danicki, Andreas Polze, and Odej Kao. Interrupting real-time iot tasks: How bad can it be to connect your critical embedded system to the internet? In *2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC)*, pages 1–6, 2020.

[82] S. Qi, S. G. Kulkarni, and K. K. Ramakrishnan. Understanding container network interface plugins: Design considerations and performance. In *LANMAN*, pages 1–6, 2020.

[83] Andrea Garbugli, Lorenzo Rosa, Armir Bujari, and Luca Foschini. Kubernetsn: a deterministic overlay network for time-sensitive containerized environments. In *ICC 2023-IEEE International Conference on Communications*, pages 1494–1499. IEEE, 2023.

[84] Ilja Behnke, Philipp Wiesner, Robert Danicki, and Lauritz Thamsen. A priority-aware multiqueue nic design. In *In Proceedings ofthe 35th Annual ACM Symposium on Applied Computing (SAC)*, 2022.

[85] Ilja Behnke, Christoph Blumschein, Robert Danicki, Philipp Wiesner, Lauritz Thamsen, and Odej Kao. Towards a real-time iot: Approaches for incoming packet processing in cyber–physical systems. *Journal of Systems Architecture*, 140:102891, 2023.

[86] Opc foundation history webpage. `https://opcfoundation.org/about/opc-foundation/history/`. Accessed: 2025-05-08.

[87] Opc 10000-1 - ua specification part 1: Overview and concepts. `https://reference.opcfoundation.org/Core/Part1/v105/docs/`. Accessed: 2025-05-05.

[88] Opc 10000-14 - ua specification part 14: Pubsub 1.05.03. `https://reference.opcfoundation.org/Core/Part14/v105/docs/`. Accessed: 2025-05-05.

[89] Opc 10000-80 - uafx part 80: Overview and concepts 1.00. `https://reference.opcfoundation.org/UAFX/Part80/v100/docs/`. Accessed: 2025-05-05.

[90] Sten Grüner, Alexander E Gogolev, and Jens Heuschkel. Towards performance benchmarking of cyclic opc ua pubsub over tsn. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2022.

[91] Rebekka Neumann, Christian von Arnim, Michael Neubauer, Armin Lechler, and Alexander Verl. Requirements and challenges in the configuration of a real-time node for opc ua publish-subscribe communication. In *2023 29th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*, pages 1–6. IEEE, 2023.

[92] Rafal Cupek, Kamil Folkert, Marcin Fojcik, Tomasz Klopot, and Grzegorz Polakow. Performance evaluation of redundant opc ua architecture for process control. *Transactions of the Institute of Measurement and Control*, 39(3):334–343, 2017.

[93] ABB. Ac 800m communication interfaces - ci871a. `https://800xahardwareselector.automation.abb.com/product/ci871a`. Accessed: 2025-05-08.

[94] ABB. Select i/o. `https://new.abb.com/control-systems/system-800xa/800xa-dcs/hardware-controllers-io/select-i-o`. Accessed: 2025-05-08.

[95] Opc 10000-4 - ua specification part 4: Services 1.05.03. `https://reference.opcfoundation.org/Core/Part4/v105/docs/`. Accessed: 2025-05-05.

[96] Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro Vittorio Papadopoulos. React: Enabling real-time container orchestration. In *26th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, Sep. 2021.

[97] Eric Brewer. Cap twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29, 2012.

[98] Dietmar Bruckner, Richard Blair, M Stanica, A Ademaj, W Skeffington, D Kutscher, S Schriegel, R Wilmes, K Wachswender, L Leurs, et al. Opc ua tsn a new solution for industrial communication. *Whitepaper. Shaper Group*, 168:1–10, 2018.

[99] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems

research. *Journal of management information systems*, 24(3):45–77, 2007.

[100] Vxworks - real-time operating system. https://www.windriver.com/products/vxworks. Accessed: 2025-05-15.

[101] Lionel Briand, Domenico Bianculli, Shiva Nejati, Fabrizio Pastore, and Mehrdad Sabetzadeh. The case for context-driven software engineering research: generalizability is overrated. *IEEE Software*, 34(5):72–75, 2017.

[102] Bjarne Johansson. Dependable distributed control system : Redundancy and concurrency defects, 2022.

[103] Bjarne Johansson and Mats Rågberger. Methods and means for failure handling in process control systems, February 11 2025. US Patent 12,224,679.

[104] Bjarne Johansson, Mats Rågberger, and Anders Rune. Method for failure detection and role selection in a network of redundant processes, September 5 2023. US Patent 11,748,217.

[105] Bjarne Johansson, Olof Holmgren, and Mats Rågberger. Safe failover between redundant controllers, 2023. European Patent Application EP23164720A.

[106] Bjarne Johansson, Olof Holmgren, and Stefan Sällberg. Managing introduction of updates in a process control function, 2024. European Patent Application EP24182353.

[107] Bjarne Johansson, Björn Leander, and Olof Holmgren. State transfer scheduling of internal states in an industrial environment, 2024. European Patent Application EP24182353.

[108] Open 62541 - project page. https://www.open62541.org/. Accessed: 2025-05-05.

[109] Karen Bollermann and Cary Nederman. John of salisbury. 2016.

II

Included Papers

Chapter 6

Paper A: Kubernetes Orchestration of High Availability Distributed Control Systems

Bjarne Johansson, Mats Rågberger, Alessandro V. Papadopoulos, and Thomas Nolte.

In 23rd IEEE International Conference on Industrial Technology (ICIT), 2022.

Abstract

Distributed control systems transform with the Industry 4.0 paradigm shift. A mesh-like, network-centric topology replaces the traditional controller-centered architecture, enforcing the interest of cloud-, fog-, and edge-computing, where lightweight container-based virtualization is a cornerstone. Kubernetes is a well-known container management system for container orchestration in cloud computing. It is gaining traction in edge- and fog-computing due to its elasticity and failure recovery properties. Orchestrator failure recovery can complement the manual replacement of a failed controller and, combined with controller redundancy, provide a pseudo-one-out-of-many redundancy. This paper investigates the failure recovery performance obtained from an out-of-the-box Kubernetes installation in a distributed control system scenario. We describe a Kubernetes based virtualized controller architecture and the software needed to set up a bare-metal cluster for control systems. Further, we deploy single and redundant configured containerized controllers based on an OPC UA compatible industry middleware software on the bare-metal cluster. The controllers expose variables with OPC UA PubSub. A script-based daemon introduces node failures, and a verification controller measures the downtime when using Kubernetes with an industry redundancy solution.

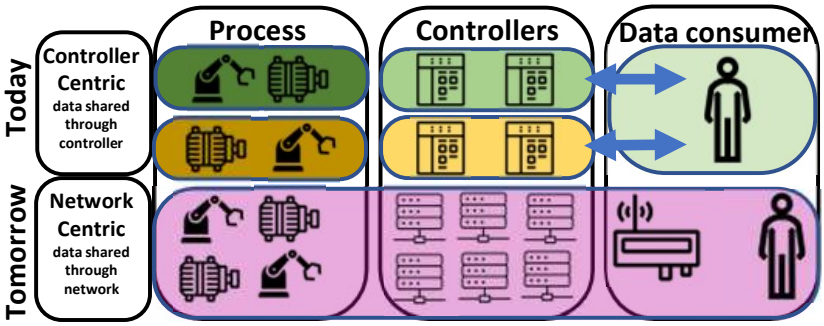


Figure 6.1: A simplified view of a controller-centric and network-centric system.

6.1 Introduction

A Distributed Control System (DCS) is a large-scale control system with multiple Distributed Controller Nodes (DCN) interconnected. A traditional DCN consists of dedicated hardware running the real-time controller FirmWare (FW). A high availability DCN is often achieved with hardware duplication – i.e., two DCNs, an active (primary) and a passive (backup). If the primary fails, the backup takes over the primary role, providing a one-out-of-two (1oo2) redundancy. The controlled process dictates the critical upper bound takeover time, which translates to around 500 ms for DCS in process automation [1]. Manual replacement of a failed DCN is required to restore redundancy.

The Industry 4.0 [2] data thirst drives DCS towards a network-centric architecture with an increased possibility of information and data retrieval. Figure 6.1 shows a simplified view of a traditional controller-centric system and network-centric system. The interconnectivity provided by a network-centric architecture allows data exchange between all devices connected to the network. Access to data produced near the process, i.e., the I/O, sensors, and actuators, does not need to involve the DCN.

Interconnectivity and interoperability are key concepts in the Open Process Automation™ Standard [1] (O-PAS). The O-PAS standard for DCN communication utilizes the OPC-UA [2] model making OPC UA suitable as communication means for our virtualized controller.

Virtualization is a cornerstone in realizing the computational elasticity provided by cloud-, fog-, and edge-computing. Containers are a lightweight and

[1] https://publications.opengroup.org/p190
[2] https://opcfoundation.org/

more performant virtualization alternative to Virtual Machines (VM) [3].

The widespread use of containers has led to container orchestration management systems such as Docker Swarm, Marathon on Mesos, and Kubernetes. The central functionality provided by the orchestrator is situation-aware scheduling and deployment of containers on the available resources.

We study the failure recovery properties provided by a vanilla out-of-the-box Kubernetes installation in a DCN context and the additional plugins needed to set up a bare-metal cluster hosting Virtualized DCN (VDCN). Kubernetes failure recovery, combined with 1oo2 VDCN redundancy, provides a pseudo-one-out-of-N (1ooN) VDCN redundancy and complements manual replacement of failed DCNs.

6.2 Related Work

DCNs are embedded real-time systems, i.e., the temporal aspect of function output is as important as the output itself. Therefore, container performance is of primary concern.

Struhár et al. [4] survey the usage of real-time containers and conclude that tool support, communication, and shared resources are open challenges. Even though challenges remain, ongoing research on real-time containers has been developed over the past few years [5, 6]. Felter et al. [3] show that the container overhead for CPU and memory utilization is negligible, but there can be a performance impact on I/O intensive applications. A similar conclusion is reached by Watada et al. [7], who also identify several challenges, i.e., persistent storage, complex networking, and orchestration management.

Fog computing addresses the inherent communication latency associated with geographically distant cloud computing by utilizing computational resources that are geographically closer [8], implying that the temporal aspect is vital in fog computing. Bellavista et al. [9] show that Single Board Computers (SBC), such as Raspberry Pis, are viable as fog computing nodes. They emphasize that providing real-time guarantees in a system with complex temporal resource utilization patterns is challenging. In a virtualized environment, resource contention can occur even if the utilized resources are different. Kim et al. [10] show that a network-bound application can saturate the CPU with `softirq` processing induced by the network communication.

Domain-specific container scheduling prerequisites have driven scheduling-related research. Eidenbenz et al. [11] evaluate three different Kubernetes scheduling integration alternatives to reduce communication latency and conclude that the Kubernetes native scheduling is the better alternative. Further, Eidenbenz et al. evaluate failover times, similar to our

work, but just with Kubernetes native approach, and conclude that it is not fast enough. Vayghan et al. [12] propose an enhanced Kubernetes controller that gives a shorter downtime if a stateful application fails by having a redundant, passive instance ready to resume when told so by the controller.

Struhár et al. [6] introduce monitoring of real-time properties utilized by a Kubernetes scheduler extension to strengthen the temporal aspects. Großmann et al. [13] develop a resource utilization measurement tool with a small footprint. Using this tool, they compare the resource utilization between Kubernetes and Docker Swarm. Docker Swarm is less resource-demanding, but they also highlight that the comparison is not fair since Kubernetes provide more functionality.

A DCN in a network-centric context relies on network connectivity for various purposes such as communication with field devices, i.e., the network is fundamental. In a containerized context, the network is typically partly virtualized. Container Network Interface (CNI) and Container Network Model (CNM) are two specifications, with corresponding libraries, plugins, and interfaces that a container runtime can utilize to configure the network. Kubernetes supports CNI, and CNI is a *de-facto* standard [14]. There exist many CNI plugins, and researchers have studied the performance of some of them. Qi et al. [15] categorize a selection of CNI plugins in four categories and perform a benchmark, measuring throughput and Round-Trip Time (RTT). Depending on the plugin and the type of communication, the performance degradation ranges from a fraction of a percent up to 30% [14, 16, 15, 17].

As for automation-related controller virtualization research, Hegazy et al. [1] show that Automation as a Service (AaaS) is feasible with a latency compensating control algorithm. However, recouping for the latency is impossible when an RTT shorter than communication time to the remote cloud is required, for example, a quick reaction to a discrete event. Goldschmidt et al. [18] presented and benchmarked a containerized controller architecture, concluding that the introduced overhead is insignificant.

To the best of our knowledge, no related work combines orchestrator failure recovery with VDCN redundancy. The combination results in a pseudo-1ooN VDCN redundancy. Our contribution is the description of the components needed to realize a Kubernetes orchestrated cluster for hosting single and redundant VDCN combined with the measurement and evaluation of failure recovery performance.

6.3 System Description

Docker is a well-known container runtime; examples of other alternatives are rkt[3] and LXC[4]. We select Docker as the container runtime, mainly due to its popularity and performance [3, 5, 18]. Rodriguez et al. [19] presented an overview of orchestration systems. We chose Kubernetes since it is relatively mature with a large open-source community. Marathon on Mesos is a relevant alternative due to its high-availability properties. We identify Marathon on Mesos and Kubernetes dependability evaluation as potential future work and focus solely on Kubernetes in this work.

6.3.1 Kubernetes Components and Architecture

Control plane is the name for the logical consolidation of the cluster control logic, i.e., the brain of the orchestrator. Compute node is the name for the nodes, physical or virtual, doing the actual work. Kubernetes offer a high-availability setup that prevents a single point of failure to bring down the control plane functionality. This work focuses on compute node failure and notes that evaluating control plane failure in a DCS context is relevant future work.

The central component in Kubernetes is the *Pod*. A Pod is a collection of one or more containers that are co-scheduled and co-located. Kubernetes do not schedule or deploy containers directly; Kubernetes operates on Pods.

The main components in the Kubernetes architecture, divided into control plane and compute node components, are shown in Figure 6.2, and briefly described below. Control plane components:

- `kube-apiserver`: the frontend of the cluster, all cluster interaction, including configuration, takes place through the `kube-apiserver`.
- `kube-scheduler`: assigns pods to nodes based on scheduling constraints and node resource availability.
- `kube-controller-manager`: control loops driving the actual state towards the desired state.

Compute node components:

- `kubelet`: Kubernetes node agent that monitors the node and the Pod deployed containers.
- `kube-proxy`: maintainer of node network rules, allowing inter-pod communication.

[3] `https://github.com/rkt/rkt/`
[4] `https://linuxcontainers.org/`

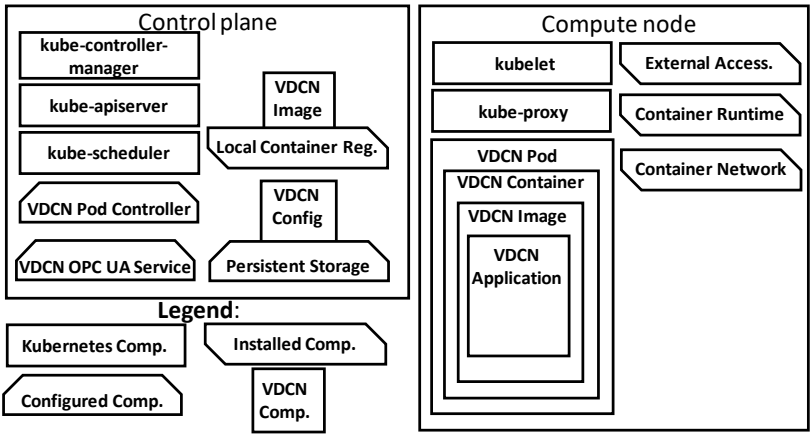


Figure 6.2: A Kubernetes-based VDCN cluster architecture.

6.3.2 Kubernetes DCN Cluster Architecture

A VDCN Kubernetes cluster requires additional components, configuration of Kubernetes components, and VDCN specific components. Figure 6.2 show the architecture. We divide the components into three categories and give a short overview below and a more detailed description in Section 6.4.

The installed components are: (i) the Container Runtime (CR) and Local Container Registry (LCR), (ii) the Container Network, for intra-cluster container communication, and (iii) the External Access, for inter-cluster communication.

The configured Kubernetes objects are: (i) Persistent Storage, (ii) VDCN Pod, containing a container instantiated from the VDCN Image, (iii) VDCN Pod Controller, for managing VDCN Pod instances, and (iv) VDCN OPC UA Service, for OPC UA Server endpoint lookup amongst VDCN Pods, i.e., intra-cluster OPC UA traffic routing.

VDCN components include: (i) VDCN Application, i.e., the controller software, (ii) VDCN Image, containing the VDCN Application image, and (iii) VDCN Configuration, containing the specific VDCN configuration.

6.4 Components

In this section, we describe the cluster components to provide a holistic view of a bare-metal cluster capable of hosting redundant and single VDCNs, that together with Kubernetes, constitute the VDCN cluster.

Docker runtime and registry

The Docker runtime pulls the container image from the LCR and starts the containerized process, the VDCN Application. Upon termination, the runtime cleans up the allocated resource.

The LCR serves as a cluster repository for container images, i.e., the VDCN images.

Container network

Container Network (CN) is the network that connects containers, intra-, and inter-node. The CN can be the physical network directly, set up with IP address routing and Ethernet switching, i.e., the underlay network, e.g., a traditional switched Ethernet network. A CN can also be a virtual network built upon the underlay using tunneling protocols such as VXLAN[5], i.e., an overlay network.

An example of an underlay network is a network created using the `macvlan` driver. The `macvlan` driver creates a virtual Ethernet interface, with an additional MAC address tied to a physical Ethernet interface. By making the virtual `macvlan` interface accessible from the container network namespace, the container gets access to the network.

A more common approach to allow the container to partake in network communication is to use a Virtual Ethernet Device (`veth`) adapter pair, `veth` are always created in pairs. One of the adapters resides in the container namespace and is therefore visible from the containerized application. The other `veth` lives in the host namespace. Together, the `veth` pair form a tunnel from the container namespace to the host namespace.

CNI Plugins are software components that comply with the CNI specification and provide a CN. For example, Flannel[6], Weave[7] and Cilium[8] create a VXLAN based overlay while Calico[9] and Kube-router[10] uses IP-in-IP[11] [15].

To summarize and relate CNI to Kubernetes context, each Pod has an IP address. The CNI plugins are responsible for providing the IP address and realizing the Pod-to-Pod communication within the cluster.

Qi et al. [15], evaluate the performance of Flannel, Cilium, Weave, Calico, and Kube-router and conclude that there is no all-around winner performance-

[5] `https://tools.ietf.org/html/rfc7348`
[6] `https://github.com/flannel-io/flannel`
[7] `https://www.weave.works/oss/net/`
[8] `https://cilium.io/`
[9] `https://www.projectcalico.org/`
[10] `https://www.kube-router.io/`
[11] `https://tools.ietf.org/html/rfc1853`

wise. Cilium has the best intra-host performance, while Kube-router and Calico are more performant in inter-host communication.

The VDCN utilizes UDP multicast, described in the following sections. Searching the internet and available CNI-plugins project pages tells us that Calico has multicast support on the roadmap, but it is currently not implemented. Weave is the only plugin we found with multicast support; hence Weave is the plugin we use for the VDCN cluster.

External access

The CN setup the intra-cluster communication. For inter-cluster ingress traffic, Kubernetes provides three alternatives[12]:

- NodePort: expose the service on a statically allocated port on each node's IP. I.e., a node IP address combined with the static port is the externally exposed access point.
- Load balancer: load balances and directs the traffic to the service endpoint. The load balancer specifies the external access point, and typically the cloud provider provides the load balancer.
- External IPs: ingress traffic reaching a cluster node, on an IP address that matches the IP address specified in the Kubernetes external IP service specification, is routed to the service endpoint by Kubernetes Services.

A VDCN cluster use case where external access is needed is when an OPC UA Client outside the cluster requests services from a cluster VDCN OPC UA Server. The OPC UA Client should always reach the same VDCN on the same IP address, provided that the VDCN is available.

How feasible are the different alternatives for realizing the above? NodePort requires that the client outside the cluster re-connects to a new IP address in case of failure of the node owning the IP address the client currently uses. NodePort also requires mapping between the original port and the port used for exposing the service. NodePort does not ensure that the OPC UA Client only needs to know one IP address per DCN. Hence, NodePort is not an alternative.

External IP is not per se managed by Kubernetes; the cluster administrator must ensure that the external IP address exists and routes to a node in the cluster. Kubernetes forwards cluster ingress traffic with a destination IP matching the external IP to the endpoint designated for the port. For example, external

[12] `https://kubernetes.io/docs/concepts/services-networking/service/`

IP could ensure that an OPC UA Client only needs to know one IP address per VDCN; however, it would require the cluster administrator to set up a solution tolerant to node failures.

Load balancers, as mentioned, are typically provided by the cloud provider hosting the cluster. However, an on-site, bare-metal cluster does not necessarily utilize the cloud, and it is not desirable to route time-critical traffic through the cloud provider. Hence, the load balancer alternative requires a bare-metal load balancer.

We have been able to identify three bare-metal load balancers, MetalLB [13], PorterLB [14] and PureLB [15]. The selection and evaluation of the load balancer is a potential work on its own. For the work presented here, we conclude that using a load balancer with network redundancy capabilities and IP Address Management (IPAM) would make the load balancer alternative the better of the three presented alternatives. MetalLB provides both; hence the load balancer alternative with MetalLB as the bare metal load balancer is the one we use.

MetalLB supports two modes: (i) layer 2 mode, and (ii) Border Gateway Protocol (BGP) mode. In layer 2 mode, all incoming traffic pass through one of the cluster nodes kube-proxy, the leader node. From kube-proxy and onward, it is the internal Kubernetes service endpoint handling. MetalLB elects a new leader node if the leader node fails, and MetalLB will send gratuitous ARP packets, announcing that the IP address association changed to the MAC address of the new leader.

MetalLB BGP mode requires a router; MetalLB uses BGP to announce multiple routes, routes leading to different nodes in the cluster, i.e., the load balancing is the multipath handling in the router. When the traffic reaches the cluster node, the handling is the same as in layer 2 mode.

We use MetalLB in layer 2 mode, leaving load balancing related questions as possible future work. MetalLB provides IPAM, and the IP address managed are provided to MetalLB as an IP address pool. The specification of an externally accessible Kubernetes Service contains an IP address from the MetalLB IP address pool.

Persistent storage

Traditionally, memory on the DCN provides the DCNs persistent storage, for example, a non-volatile RAM or an SD card. The DCN stores the configuration, application, and current state in the persistent storage, which allows the

[13] `https://metallb.universe.tf/`
[14] `https://porterlb.io/`
[15] `https://gitlab.com/purelb/purelb`

DCN to resume operation after a failure, such as a power failure. A VDCN in a Kubernetes cluster is deployable on multiple nodes. Hence the persistent storage needs to be accessible from the nodes that host the VDCN.

Kubernetes provides the possibility to use various storage solutions. Volume is the Kubernetes term for file storage. A Volume, from a Kubernetes Pod perspective, is just a directory. Kubernetes do not care how that directory comes into existence. Setting up the storage is the cluster administrator's responsibility.

Kubernetes manages the lifetime of the Volume. There are two types of Volumes, Volume and Persistent Volume (PV). A Volumes lifespan is the same as the Pod', i.e., when the Pod ceases to exist, Kubernetes destroy the Volume. On the other hand, the PV lifespan is independent of the Pod. A Persistent Volume Claim (PVC) is the mean for a Pod to claim a PV. The PVC specifies the Pods requirements on the PV, such as size, access modes, etc.

In our experiment setup, we use a Network File System (NFS)[16] hosted on the control plane node to provide storage. The storage is not redundant – but that is not crucial for the evaluation since the control plane reschedules the Pods, i.e., no control plane, no Pod rescheduling.

VDCN Pod

The VDCN Pod is the Kubernetes Pod encapsulation of the VDCN Container. The VDCN Pod claims a PV using a PVC; the VDCN Pods in the test setup claim 100 MB that is read and writable. Due to multicast not being supported by MetalLB, the VDCN Pod has access to the node (host) network directly.

VDCN Pod controller

The VDCN Pod controller is the name we have given to denote the functionality we achieve by utilizing Kubernetes for controlling the VDCN Pods. An application running on a Kubernetes cluster is a workload, e.g., the VDCN is a workload. Workload resources are Kubernetes objects that specify the desired state for a Pod or Pods. Kubernetes controllers, executed in the context of the kube-controller-manager, strive to maintain the workload resource desired state.

A Kubernetes Deployment is a workload resource type for managing Pods. A Deployment strives to ensure that at least as many Pods as specified in the Deployment description are available in the cluster. In addition, if the Pods

[16] `https://tools.ietf.org/html/rfc3010`

use PV, all the Pods created by the same Deployment share the same PV. Thus, Deployments are well suited for stateless applications.

Statefulset is another workload resource type that ensures that, at most, the number of Pods specified in the Statefulset description is available in the cluster. The Statefulset creates the Pods in a predetermined order with a known identity. If the Pods use PV, each Pod gets its PV.

The VDCN Pod Controller is the Kubernetes controller with a Statefulset describing the desired state. We use Statefulset as the workload resource since we want stricter control of the number of VDCN instances running than the Deployment can provide to avoid situations where two or more VDCN with the same identity are active but in different states.

Our testbed cluster uses two separate VDCN Pod Controllers, i.e., Statefulsets, one for the single VDCN and one for the redundant. For the single VDCN, the number of Pods is one. The number of Pods in the redundant VDCN is two since the redundant VDCN is a pair. We use Pod anti-affinity to ensure that Kubernetes does not schedule both VDCN of the redundant pair on the same node.

VDCN OPC UA Service

Kubernetes Services is the front-end of a cluster-hosted application function. The containers running inside Pods are the *Service endpoint*. Pods' IP addresses and whereabouts are not static; they can change from one moment to another. Kubernetes Services is the mechanism to find the Pod that offers the Service for the requested function, independent of the current deployment. Kubernetes Services is the intra-cluster solution to find the endpoint. The kube-proxy handles the Service endpoint lookup on each node, watches the control plane for Service and endpoint updates through the kube-api, and updates the node iptables accordingly.

The VDCN has three network communication dependent functions: (i) the cyclic exchange of variables over OPC UA PubSub, (ii) acyclic communication using OPC UA Client Server, and (iii) the redundancy communication. OPC UA PubSub and the redundancy communication use UDP multicast and do not need a Kubernetes service. The network IGMP support provides the means to match publishers with subscribers.

OPC UA Connection Protocol (UACP) is the abstract protocol that describes the full-duplex communication channel between client and server. OPC UA supports TCP, HTTPS, and WebSocket as the UACP underlying transport protocols, and the VDCN OPC UA Client-Server uses TCP. In other words, the OPC UA Client-Server communication is unicast-based, point-to-point.

A request addressed to a VDCN OPC UA Server can originate from an OPC UA client inside or outside the cluster. The external handling described above ensures that the request reaches a cluster node. When the request has reached a cluster node, the Kubernetes Service handling provides the endpoint reaching means.

Our example setup consists of three VDCNs, the single configured and the redundant pair. We use two VDCN OPC UA Services, one for the single VDCN and one for the primary VDCN. The redundancy state of the redundant VDCN is application-specific. To allow Kubernetes to redirect the traffic to VDCN in primary mode, we need Kubernetes to update the routes depending on the application state. A Kubernetes mechanism for that is the probes, probes that probe the application's state. The application tailors the application end of the probe for its need.

Kubernetes provides three types of probes. The Liveness-probe determines if the application is responsive (alive) or not. If not, Kubernetes can restart the container. The Startup-probe tells Kubernetes that the container application has started, and the Readiness-probe tells if the container application is ready to accept traffic. If the Readiness-probe result is negative, the probed application is removed from the list of potential service endpoints. The VDCN Application uses the readiness probe to direct traffic to the primary VDCN in the redundant VDCN configuration; the backup VDCN Application reply negatively to Kubernetes Readiness-probe requests.

VDCN Application

In a traditional DCN, the VDCN Application is the FW capable of executing the control loop logic. The VDCN Application used in our testbed is an ABB proprietary software, i.e., a modern DCN FW. It consists of three main parts, an OPC UA stack for industrial use, a middleware, and the control loop logic. The OPC UA stack provides the OPC UA communication means, and the middleware offers functionality to the control logic. The middleware functionality relevant for this testbed is redundancy-related. Finally, the control logic in the VDCN Application consists of a cyclic task with a configurable interval time. The cyclic task updates configured variables each iteration and exposes the updated variables externally using OPC UA PubSub.

In addition to the cyclic OPC UA PubSub communication, the VDCN Application also contains means for OPC UA Client-Server request-based acyclic communication. An OPC UA Server is the VDCN Application side of the request-based, acyclic OPC UA communication, exposing Remote Procedure Calls (RPC) callable from an OPC UA Client.

As the name implies, OPC UA PubSub is a publisher-subscriber-based solution. The publishers do not directly connect to the subscribers, and vice versa. Two models are supported, broker-based and broker-less. A broker-based publisher sends messages to a central broker from which subscribers subscribe. Two concrete broker-based solutions are supported, Message Queue Telemetry Transport (MQTT) and Advanced Message Queuing Protocol (AMQP). The broker-less model relies on properties provided by the network, specifically multicast and broadcast possibilities. UDP multicast is the supported realization of the broker-less model. A network infrastructure supporting IGMP ensure that published message only is forwarded to the subscribers. Network infrastructure without IGMP support broadcast the messages, i.e., published messages reach the whole broadcast domain. The VDCN application uses the broker-less OPC UA PubSub model realized with UDP multicast.

The OPC UA PubSub publishing and subscribing function run in a task of its own - unsynchronized with the producer/consumer of the exchanged variable values. Figure 6.3 shows a conceptual view of the data flow between the tasks.

The VDCN Application redundancy mode is configurable as single or redundant. In single mode, there is no backup ready to resume operation in case of failure. The single configured VDCN Application stores the dynamic state (variable values etc.) on a file located in the PV, allowing a re-deployed single VDCN to resume operation from the last stored state.

The redundant VDCN Application runs in a one-out-of-two (1oo2) setup. One VDCN Application is active, publishing updated variable values using OPC UA PubSub, and the other is passive, ready to resume operation in case of failure of the active. We refer to the active as the primary and the passive as the backup.

Two mechanisms are fundamental in a redundant setup where only one is active, the state transfer and the failure detection. The state transfer provides

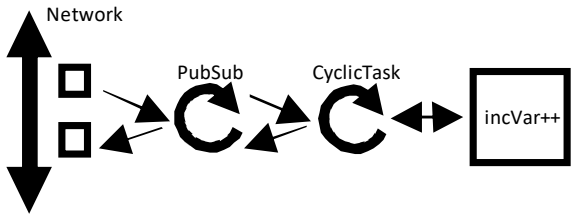


Figure 6.3: The VDCN Application is involved in the cyclic exchange.

the backup with the primary's latest checkpointed dynamic state, allowing a backup to resume the role as primary, without historic signal values outputted. The VDCN application utilizes an ABB proprietary state transfer mechanism based on UDP multicast. Heartbeat Bully [20] over UDP multicast constitutes the failure detection and role selection mechanism.

VDCN Image

The container image. When instantiated by the container runtime, the VDCN image of the VDCN Application becomes the VDCN. The VDCN image is built with Docker and pushed to the LCR.

VDCN Configuration

The VDCNs are configurable, and VDCN Pod PV holds the configuration files, ensuring that they are accessible from each node that hosts the VDCN. Section 6.5 describes the specific configuration used in the test setup, such as task cycle times and the variables exchanged.

6.5 Execution and Result

The purpose is to measure the failure recovery time of single and redundant VDCNs, deployed in VDCN Pods orchestrated by Kubernetes. First, we let Kubernetes deploy the VDCN Pods on the cluster compute nodes while bringing down the nodes hosting the primary or single VDCN after a random time. Then, Kubernetes failure detection and rescheduling re-deploy the VDCN affected by the node failure. In the redundant VDCN case, the backup VDCN resumes operation as primary, while Kubernetes re-deploy a new backup VDCN. A Verification DCN sample the signal values and gather statistic related to the cyclic exchanged variables, see Section 6.5.1.3, i.e., it checks the cyclic communication. The Verification OPC UA Client test the acyclic communication, see Section 6.5.1.4.

6.5.1 Testbed

Four main parts constitute the testbed, the cluster, a failure daemon, a cyclic communication verification node (Verification DCN), and an acyclic verification client (Verification OPC UA Client). Figure 6.4 shows the testbed deployment and Table 6.1 list the used software.

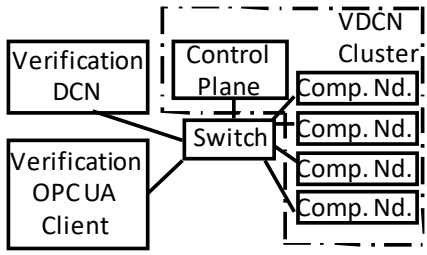


Figure 6.4: Testbed deployment.

Table 6.1: Software used.

Name	Version	Comment
Ubuntu Server	20.04	Control plane OS
Raspberry Pi OS	10	Compute node OS
PREEMPT_RT	4.4	Compute node kernel patch
KubeAdm	1.21	Kubernetes installer
Kubernetes	1.21	Kubernetes version
Docker	20.10.1	Container runtime
Weave	2.8.1	CNI plugin
MetalLb	0.9.6	Bare-metal load balancer

6.5.1.1 Cluster

The compute nodes in the testbed consist of four Raspberry Pi 4B, with four GB RAM. The control plane runs on a 2GHz Intel I7 I7-9700T PC, with 16 GB RAM.

6.5.1.2 Failure Daemon

The failure daemon is a `systemd` daemon installed on all compute nodes that check if a VDCN Pod runs on the node. If it does, and the VDCN Application is in single or primary mode, the failure daemon shuts down the node after a random time of 5 minutes, $\pm$10s.

6.5.1.3 Verification DCN

The Verification DCN hardware is a 2GHz Intel I7 I7-9700T PC, with 16 GB RAM, running VxWorks 7.0 and the verification application (VA). The VA checks VDCN output values and measures the time between updates. The VA

is the same application as the VDCN Application but with a different cycle time configuration.

6.5.1.4 Verification OPC UA Client

The Verification OPC UA Client runs on a Windows 10 PC. Every 10s, it establishes a connection to the VDCN OPC UA servers, one to the primary and one to the single VDCN, measuring the time between two successful connection attempts.

6.5.2 Exchanged Variables

The variables published by the VDCN and monitored by the VA in the Verification DCN are: (i) $incVar$, 32-bit unsigned integer, incremented by the VDCN each iteration, and (ii) $nodeId$, a string identity of the node currently hosting the specific VDCN. We expect application size to affect a VDCN similar to a DCN since container overhead is neglectable [3]. Furthermore, we deploy one VDCN per node. Hence variable handling and communication come down to resource utilization and prioritization within the VDCN and node, and the critical VDCN task has real-time priority. Ensuring real-time properties when running multiple VDCN in the context of containers/pods on one node, with more extensive use of virtualized networks, is future work.

6.5.3 Task Interval and Updating Period

We base the calculation on the simplification of the VDCN Application shown in Figure 6.3. In reality, the span will be larger due to task interleaving patterns with other high-priority tasks in the VDCN Application. The PubSub and Cyclic task in the VDCN Application has 5 and 20ms cycles. The corresponding cycle time in the VA is 1ms for both.

With the cycle times above, the VDCN publishing interval of updated variable values is in the range $PubUpdIntv \in (20, 25)$ms. The interval in which the Cyclic task in the VA can receive and check variable values is $VaCheckIntv \in (1, 2)$ms. We denote the VA measured update interval of $incVar$, $VaUpdIntv = PubUpdIntv \pm VaCheckIntv$, thus being in the range $VaUpdIntv \in (18, 27)$ms. In relation to interval times used, the network propagation time is deemed negligible and not included.

The redundant VDCN failure detection mechanism is Heartbeat Bully [20]. The backup VDCN expects heartbeat messages regularly from the primary VDCN. The time between primary VDCN failure to the backup VDCN resuming the primary role depends on the heartbeat interval and the number of miss-

ing heartbeats allowed. The heartbeat interval used is 10ms, and the number of missed heartbeats allowed is two. The VDCN Application polls the heartbeats receiving status, adding a heartbeat interval to the detection time, resulting in the primary failure detection time range $PriFailDetTime \in (30, 50)$ms. On top of that, the VDCN has a resume primary role functionality with an execution time interval of $BePriExec \in (12, 34)$ms that contributes to the total failover time. Resulting in failover time range $FoT \in (42, 84)$ms.

The $UpdIntvFail$ is the longest time between VA observed updates of $incVar$ when a failover happens we have that:

$$UpdIntvFail = 2 \cdot PubUpdIntv + FoT \pm VaCheckIntv$$

resulting in the range $UpdIntvFail \in (80, 136)$ms.

6.5.4 Kubernetes Settings

Kubernetes reschedule Pods on failure and the failure detection and reaction time are configurable, and those setting impacts the single configured VDCN $UpdIntvFail$ and redundant VDCN timeframe without a backup. The Kubernetes `kubelet` monitors the Pod and updates the status of the Pod and node to the `kube-apiserver` on the control plane. By default, the `kubelet` reports the node status every 10s, and the `kube-apiserver` has a grace period of forty seconds before setting the node status to not-available.

The default Pod eviction timeout is five minutes, and when a node failure is detected, Kubernetes reschedules the Pod after the eviction timeout expiration. Statefulsets are used for Pod management, i.e., the VDCN Pod controller. We use a Pod eviction timeout of 3s, specified in the Statefulset specification. A VDCN Pod hosted in a failing node is rescheduled after 43s; hence $UpdIntvFail$ for the single VDCN is higher than 43s.

When a node running stateful Pods fails, Kubernetes do not schedule a new stateful Pod by default. Since missing updates from that node's `kubelet` could be a consequence of network partitioning. Kubernetes cannot be confident that the node and Pod are gone. Hence, Kubernetes requires the cluster administrator to delete the failed node stateful Pods manually in a node failing situation.

We assume that the network is redundant and reliable with minimal risk of network partitioning. Even though our testbed network is not redundant, the network on an actual site is likely to be. Hence, we configure Kubernetes to reschedule stateful Pods by setting their termination grace period to 0s.

Table 6.2: Measured interupdate time.

Mode	Primary				Single			
	Min	Max	Avg	SD	Min	Max	Avg	SD
Normal	3.0ms	36ms	20ms	1.3ms	3.0ms	224 ms	20ms	1.8ms
Failure	83ms	129ms	110ms	10.5ms	43.5s	74.7s	55.4s	8.1s

6.5.5 Result

The test ran for ten hours and accumulated 200 node failures, one hundred failures each on primary and single VDCNs hosting nodes and 3.6 million interupdate measures without node failures in between.

Table 6.2 shows the interupdate times per VDCN and mode. The Normal mode row shows the interupdate time measured during the failure-free periods and the Failure mode row when two updates span a node failure of the node hosting the VDCN publishing the variable. The Normal interupdate time is a reference for update time in a normal situation. The measured interupdate (or update interval) times without failure are on average 20ms, and that is in the expected range $VaUpdIntv$ from Section 6.5.3. The single VDCN interupdate time standard deviation is higher than from the primary. The NFS-based PV state storage induces a longer execution time for the single VDCN. Max and min interupdate measured from the primary are outside the theoretical limit due to task interleaving patterns in the VDCN Application used that we do not address in the theoretical simplification. The SD tells that the vast majority is within the expected interval.

The interupdate time during a primary node failure, i.e., the failure recovery time, is within the expected range with an average of 110ms. As a comparison, the Kubernetes controller-based recovery time archived by Vayghan et al. [12] is in the range of 1.5 seconds. The single VDCN minimum interupdate time during node failure is 43.5 seconds, reached when the Pod is scheduled on the same node again. That can happen since the failure injection is a reboot. After the reboot, the node is failure-free again. In that case, the container image is not pulled from the LCR, reducing the time. The max of 74.7s includes pulling the image from the LCR.

OPC UA Client connection reestablishment to the new primary VDCN took a max of 41 seconds with an average of 40 seconds, which is the pod eviction time. For the single VDCN, the connection reestablishment times are the same as the failure interupdate times, which is feasible since that is the time it took Kubernetes to replace the failed single VDCN.

6.5.6 Availability Discussion

We base the replacement scenarios in this section on input from experienced engineers working close to end-users. When a DCN fails today, it needs manual replacement and the replacement time depends on the situation. The best circumstance is when the failed DCN is close to both a spare DCN and maintenance personal that can exchange the broken DCN. In that case, a replacement within an hour is optimistic but realistic. A less favorable scenario is a failure in a remote location. It could take time for maintenance personnel to reach the site, causing repair time to range from several hours to many days. With an orchestrated VDCN cluster, the orchestrator could redeploy the failed VDCN on compute nodes with enough available capacity and reduce the replacement time.

A commonly used measure in reliability contexts is Mean Time To Failure (MTTF), a statistical measure of the probability of failure within a specific period [21]. DCN MTTF depends on the hardware components used, temperature, and more. Based on public product information[17,18], we use a mid-range MTTF approximation of twenty years for the DCN and compute node hardware in the following discussion. Depending on the product and equipment type, it can be higher or lower. Note that the above user manuals use the term Mean Time Between Failures (MTBF) as MTTF, since the listed values do not account for repair time. Hence, MTTF is a more accurate term to use since it does not include repair time [21].

Availability is the proportion of time that a system is available, often denoted in the number nines in the availability percentage; for example, 99.99% has four nines. A DCN replacement mean-time, Mean Time To Repair (MTTR) of one hour with MTTF of twenty years translates to an availability of five nines and an MTTR of twenty-four hours to four nines.

The average time for a redeployment of VDCN upon failure on the compute node currently hosting it is 55 seconds, see Table 6.2, which translates to an availability of seven nines for the single configured VDCN. A four nines availability level gives a yearly downtime of roughly 52 minutes, five nines and seven nines correspond to 5 minutes and 3 seconds downtime respectively and annual.

A redundant DCN has high availability, with the twenty years MTTF and the average takeover time from Table 6.2 the availability level is nine nines.

[17] `https://search.abb.com/library/Download.aspx?DocumentID=3BSE091397&Action=Launch`

[18] `https://support.industry.siemens.com/cs/attachments/16818490/mtbf.zip`

The orchestration benefit is the automated and quicker backup VDCN return, resulting in a pseudo-1ooN redundancy, with N being the number of compute nodes in the cluster capable of hosting a redundant configured VDCN.

An orchestration benefit is the increased availability that follows the quicker replacement of a failed DCN. Other potential benefits are flexible maintenance. For example, a VDCN can be moved to another compute node while upgrading the base software of the former. Even if one compute node fails, there might still be enough spare capacity to counter further failures, allowing process operation until the next scheduled maintenance with high confidence in the availability.

6.6 Conclusion and Future Work

By describing the components needed when setting up a bare-metal Kubernetes cluster for VDCN, we provide a holistic view of the system and show the multitude of component alternatives available. We created a testbed consisting of compute nodes, on which we deployed two VDCN configurations, a single and a redundant. Outside the cluster, we had a DCN verifying the VDCN OPC UA PubSub published variables and Windows application on a PC confirming the OPC UA Client/Server communication.

The result shows that Kubernetes hosted VDCN are feasible for both single and redundant VDCN. The measured redeployment of the single VDCN is too long to be a redundancy replacement. As stated in Section 6.1 takeover time below 500 ms can be needed for DCS in process automation. Nevertheless, it can still serve as a faster alternative to a manual replacement of a failed node. If shorter downtimes are required, a 1oo2 setup is feasible, where Kubernetes also ensure a quicker reinstatement of a backup VDCN than manual replacement, resulting in a pseudo-1ooN VDCN redundancy.

Kubernetes provide much more extensive customization alternatives than we have utilized. Further work could evaluate the feasibility of further customization of Kubernetes for VDCNs, such as the faster discovery of failed nodes. Kubernetes scheduling in the DCS context is another natural extension of this work, for example, finding and deploying the VDCN to a suitable node capable of fulfilling the dependability needs dictated by the VDCN.

We showed the plurality of different network virtualization alternatives. Reliable, deterministic communication is essential for DCS. VDCN in a containerized Kubernetes managed context relies on virtualized network functions provided by CNI plugins and load balancers. Ensuring dependability when utilizing virtualized network is a future challenge, especially when sharing the underlying resources between VDCN or other entities on the compute node.

Bibliography

[1] T. Hegazy and M. Hefeeda. Industrial automation as a cloud service. *IEEE Transactions on Parallel and Distributed Systems*, 26(10):2750–2763, Oct 2015.

[2] R. Drath and A. Horch. Industrie 4.0: Hit or hype? [industry forum]. *IEEE Industrial Electronics Magazine*, 8(2):56–58, June 2014.

[3] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *IEEE Int. Symp. Perf. Analysis of Syst. and Software*, pages 171–172, 2015.

[4] Václav Struhár, Moris Behnam, Mohammad Ashjaei, and Alessandro V. Papadopoulos. Real-Time Containers: A Survey. In *Fog-IoT*, 2020.

[5] Alexandru Moga, Thanikesavan Sivanthi, and Carsten Franke. Os-level virtualization for industrial automation systems: Are we there yet? In *SAC*, 2016.

[6] Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro V. Papadopoulos. React: Enabling real-time container orchestration. In *ETFA*, 2021.

[7] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu. Emerging trends, techniques and open issues of containerization: A review. *IEEE Access*, 7, 2019.

[8] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Mobile Cloud Comp.*, pages 13–16, 2012.

[9] Paolo Bellavista and Alessandro Zanni. Feasibility of fog computing deployment based on docker containerization over raspberrypi. In *ICDCN*, 2017.

[10] E. Kim, K. Lee, and C. Yoo. On the resource management of kubernetes. In *ICOIN*, pages 154–158, 2021.

[11] R. Eidenbenz, Y. Pignolet, and A. Ryser. Latency-aware industrial fog application orchestration with kubernetes. In *FMEC*, pages 164–171, 2020.

[12] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. A kubernetes controller for managing the availability of elastic microservice based stateful applications. *J. Syst. and Soft.*, 175:110924–, 2021.

[13] M. Großmann and C. Klug. Monitoring container services at the network edge. In *ITC*, pages 130–133, 2017.

[14] J. Yoon, J. Li, and S. Shin. A measurement study on evaluating container network performance for edge computing. In *APNOMS*, pages 345–348, 2020.

[15] S. Qi, S. G. Kulkarni, and K. K. Ramakrishnan. Understanding container network interface plugins: Design considerations and performance. In *LANMAN*, pages 1–6, 2020.

[16] N. Kapočius. Performance studies of kubernetes network solutions. In *eStream*, pages 1–6, 2020.

[17] H. Zeng, B. Wang, W. Deng, and W. Zhang. Measurement and evaluation for docker container networking. In *CyberC*, pages 105–108, 2017.

[18] Thomas Goldschmidt, Stefan Hauck-Stattelmann, Somayeh Malakuti, and Sten Grüner. Container-based architecture for flexible industrial control applications. *J. Syst. Arch.*, 84, 2018.

[19] Maria A Rodriguez and Rajkumar Buyya. Container-based cluster orchestration systems: A taxonomy and future directions. *Soft., Pract. & Exp.*, 2019.

[20] B. Johansson, M. Rågberger, A. V. Papadopoulos, and T. Nolte. Heartbeat bully: Failure detection and redundancy role selection for network-centric controller. In *IECON*, 2020.

[21] Elena Dubrova. *Fault-tolerant design*. Springer, 2013.

Chapter 7

Paper B: Consistency Before Availability: Network Reference Point based Failure Detection for Controller Redundancy

Bjarne Johansson, Mats Rågberger, Alessandro V. Papadopoulos, and Thomas Nolte.

In 28th International Conference on Emerging Technologies and Factory Automation (ETFA), 2023.

Abstract

Distributed control systems constitute the automation solution backbone in domains where downtime is costly. Redundancy reduces the risk of faults leading to unplanned downtime. The Industry 4.0 appetite to utilize the device-to-cloud continuum increases the interest in network-based hardware-agnostic controller software. Functionality, such as controller redundancy, must adhere to the new ground rules of pure network dependency. In a standby controller redundancy, only one controller is the active primary. When the primary fails, the backup takes over. A typical network-based failure detection uses a cyclic message with a known interval, a.k.a. a heartbeat. Such a failure detection interprets heartbeat absences as a failure of the supervisee; consequently, a network partitioning could be indistinguishable from a node failure. Hence, in a network partitioning situation, a conventional heartbeat-based failure detection causes more than one active controller in the redundancy set, resulting in inconsistent outputs. We present a failure detection algorithm that uses network reference points to prevent network partitioning from leading to dual primary controllers. In other words, a failure detection that prioritizes consistency before availability.

7.1 Introduction

Distributed Control Systems (DCS) progress toward more network-oriented architectures where switched Ethernet in combination with OPC UA[1] constitute the interoperability communication backbone in future automation installations [1]. A progression observable through the increase of Ethernet solutions and decrease of fieldbus installations [2, 3].

The trajectory to network-centric architectures yields that DCS controllers, denoted Distributed Control Nodes (DCN) by the Open Process Automation Forum (OPAF), and Fieldbus Communication Interfaces (FCI) will rely more on Ethernet. Ethernet enables new controller deployment alternatives, such as the execution of control applications in a virtualized context in the cloud or in orchestrated embedded clusters [4, 5, 6].

DCN redundancy is an example of functionality that must refrain from customized hardware dependency to avoid reducing deployment alternatives. Today, standby redundancy with hardware duplication is a typical redundancy pattern in a DCS context. One DCN is the active primary DCN, and the other is the passive backup DCN, ready to take the primary role in case of failure of the current primary. Only the primary DCN provides output values to I/O connected to the process and physical world.

A common failure detection method is a heartbeat, a cyclic message sent within a known interval from the supervisee to the supervising [7]. The supervising node interprets heartbeat timeout as a supervisee failure. We denote such a conventional heartbeat-based failure detection Conv. FD.

In a DCN redundancy context, the supervisee is the primary DCN, and the backup DCN is the supervising. The backup DCN in a DCN redundancy pair using Conv. FD interprets a heartbeat timeout as a primary DCN failure and resumes the primary role. However, a timeout is not necessarily a consequence of a primary DCN failure; it could follow from a network failure causing a network partitioning between the primary and backup DCN. Hence, in a network partitioning situation, with a DCN redundancy using Conv. FD, the partitioning causes the backup DCN to become primary while the former primary remains primary, resulting in dual primaries. A consequence of dual primaries is that both the DCNs in the DCN redundancy pair control I/O values, causing inconsistency.

DCN redundancy is typically used with network redundancy to reduce the probability of communication failures. However, the partitioning probability is not zero, not even with duplicated networks. Hence, DCN redundancy based on Conv. FD gives a dual primary probability larger than zero.

[1] `https://opcfoundation.org/`

If the DCN redundancy cannot ensure a single primary, it is better to have no output than conflicting outputs from dual primaries. This prevents fluctuations in the controlled process because I/O channels that expect DCN updated values will output preconfigured values when updates are missing. This setting is named Output Set as Predetermined (OSP) [8] in ABB I/O system context, and the input channel equivalent is Input Set as Predetermined (ISP), which the DCN control application uses when input values are absent.

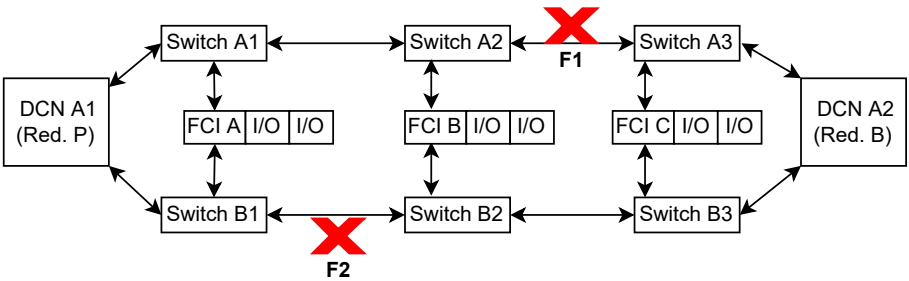


Figure 7.1: Example of a network failure causing a partitioning where both DCN will become primary and drive output signals. The DCN redundancy recovery time is typically between ten to hundred milliseconds.

Figure 7.1 shows a partitioning situation, with two network failures $F1$ and $F2$ between the redundant DCN pair. Both the DCNs (DCN A1 and DCN A2) take the primary role when using Conv. FD and provide output values in the resulting dual primary situation. FCI B gets output values from both DCNs. Values that differ since the DCNs reach two different FCIs. DCN A1 reaches FCI A and B, while DCN A2 reaches FCI B and C. Hence, the two DCNs will use different input values. DCN A1 will use the ISP values instead of actual I/O values from FCI C, while DCN A2 will use the actual I/O values from FCI C. Vice versa applies to FCI A and the DCNs.

We address this problem by proposing a failure detection algorithm that utilizes a Network Reference Point (NRP) external to the DCNs to mitigate dual primary situations due to network partitioning while striving to keep high availability. We use the name NRP Failure Detection (NRP FD) for the proposed algorithm.

The paper is structured as follows, Section 7.2 presents the related work, and Section 7.3 describes NRP FD. Section 7.4 compares NRP FD and Conv. FD concerning the availability and consistency tradeoff, followed by an experimental comparison of NRP FD and Conv. FD described in Section 7.5. Lastly, Section 7.6 summarizes the paper.

7.2 Related Work

The Consistency, Availability, and Partition tolerance (CAP) theorem [9, 10] state that in case of partitioning, a distributed system can be either consistent or available, not both. Lee et al. present a modified version of CAP, the Consistency, Availability, and apparent Latency (CAL) theorem [11, 12]. Using CAL, Lee et al. quantify consistency and availability compromises under latency requirements. NRP FD preserves consistency under network partitioning by ensuring one or no primary, i.e., consistency before availability, further described in Section 7.3.

Failure detection is crucial in a standby redundancy solution, and existing work ranges from the introduction of unreliable failure detection concept [13] and failure detection Quality of Service (QoS) attributes [14] to heartbeat optimizations and improvements [15, 16]. However, none of these works addresses differentiation between node and network failure, which, as pointed out by van Steen [17], a failure detector ideally should do. Distinguishing node and network failures could solve the problem we address.

We have not found any scientific failure detection publication addressing the differentiation between node and network failures in wired networks, but two patents that do. Charny et al. [18] uses an alternative path to the node when failing to reach the node on the first path, a solution that is similar to the neighboring using approach van Steen [17] discuss. I.e., querying neighbors of the suspected node to learn if they can reach it. Filsfils et al. [19] describe a Bidirectional Forwarding Detection (BFD) based approach to distinguish network from node failures using multiple BFD sessions over various disjoint paths. BFD [20] is a protocol for quick link failure detection between adjacent nodes, typically used by routers.

Ritter et al. [21] present a similar approach for ad-hoc mobile wireless networks. A beacon node sends heartbeats to all other nodes. The beacon's closest neighbor, the buddy node, supervises the beacon faster than the heartbeat interval. If the buddy node does not hear the beacon, it tries an alternative path, and if that also fails, the beacon node is assumed to have failed. Otherwise, the network is considered partitioned. Our work addresses partitioning without alternative paths. The work described above would treat such partitioning as a node failure and cause a dual primary situation, which we want to avoid.

Failure detection in a redundant DCN context, with only one backup, constitutes an implicit leader election. The Bully algorithm [22] is one of the more famous leader election algorithms, and many variants exist [23, 24, 25]. However, they will all elect a leader per partition, i.e., provide availability before consistency.

Quorum consensus protocols like Paxos [26, 27] and Raft [28] provide consistency and tolerates $(N-1)/2$ faults, where N is the number of nodes. A redundant DCN only requires two individual DCNs, $N = 2$, i.e., a consensus protocol-based redundancy would not tolerate one fault if $N = 2$, making quorum-based DCN redundancy meaningless.

Active redundancy means that all the nodes in the redundant set are active [29, 30]. However, it pushes the decision of which data to use to a selection function such as a voter, a selection that needs to be made on the data consumers to be partition tolerant, forcing DCN redundancy data handling to all data consumers.

None of the related work covered solves the problem we address, a real-time capable failure detection that prioritizes consistency without requiring a DCN quorum while minimizing the availability compromise.

7.3 Network Reference Point Failure Detection

7.3.1 Overview

NRP FD is a heartbeat-based failure detection with additions. We describe it in the context of a redundant DCN pair. A failed DCN is assumed to stop sending heartbeats. Generalization and adaptation to more flexible redundancy patterns are future work.

NRP FD provides consistency over availability, meaning that both DCNs will not be primary due to multiple network failures, but none might.

The NRP FD additions are (i) the usage of the NRP and (ii) the optional utilization of temporal properties of received heartbeats. We use the word network to describe a communication path between primary and backup DCN. NRP FD does not dictate any requirement on the number of networks connecting the primary and backup. In the description, we use a redundant network (two networks), exemplified in Figure 7.2. NRP FD assumes that the NRP is an individually accessible node in the network infrastructure between primary and backup; in practice, a managed switch. Each DCN in the redundant pair has an NRP candidate per network; see Figure 7.2. How NRP FD is made aware of the NRP candidates is outside the algorithm's scope. Section 7.3.2 describes the considerations that apply to the NRP candidates. Only one of the NRP candidates is the NRP; selected by the primary.

Algorithm summary; the primary sends heartbeats on all networks connecting the primary and the backup DCNs, containing the IP address of the NRP that the primary has selected. If the backup does not observe any heartbeat within a timeout period, it checks if it can reach the NRP, and if it can, it

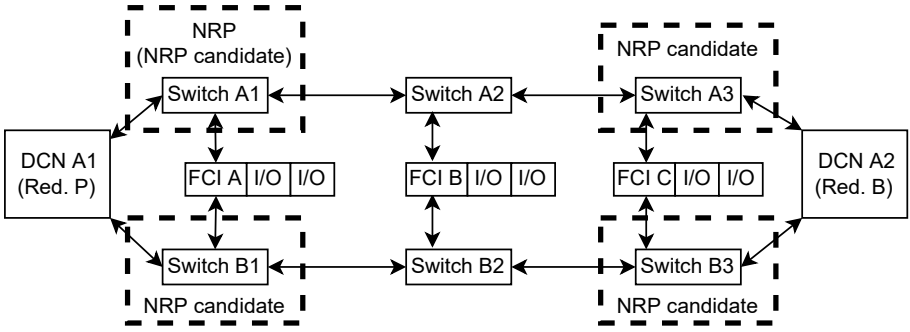


Figure 7.2: NRP in a system context where each DCN has one candidate per network and the primary has appointed one NRP (switch A1).

becomes the new primary, and if it cannot, it remains passive.

We use the term PINGNRP for the NRP reachability test. NRP FD is agnostic to the implementation behind PINGNRP. To comply with COTS switches, the PINGNRP is limited to communication means and protocols supported by most managed industrial COTS switches; in practice, this means using ICMP[2] ping.

An ICMP ping does not have a bounded response time, but NRP FD can guarantee a bounded reaction time by utilizing temporal aspects of heartbeats received on the different networks. NRP FD can assume that a heartbeat silence is due to a primary failure if heartbeats timeout simultaneously on multiple networks and skip the PINGNRP. Note that this is optional handling with the cost of a small risk of treating simultaneous network failures as a failure of the primary.

A backup that does not use the simultaneous timeout optimization or only has one functioning network always uses the PINGNRP to determine if it should become the primary in case of heartbeat silence.

In addition to the PINGNRP performed by the backup, NRP FD prescribes that the primary checks if it can reach the NRP; if it can't, it tries to elect a new NRP from the NRP candidates. If that fails, it surrenders the primary role.

If a backup does not receive heartbeats on the network of the NRP but on other networks, the backup checks if the NRP is reachable. If it is not, the backup proposes a switch of NRP to the primary.

Figure 7.3 gives an overview of NRP FD, further described in Section 7.3.3.

[2] `https://www.rfc-editor.org/rfc/rfc792`

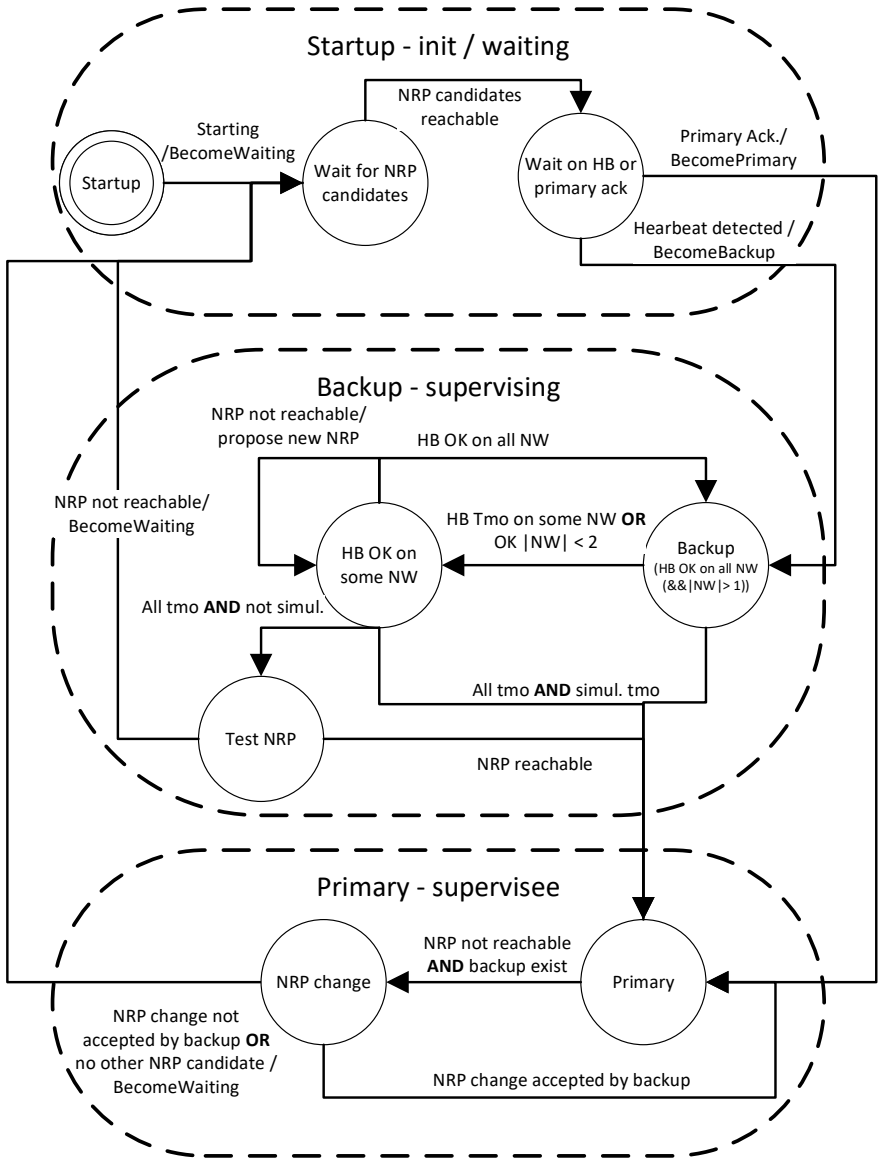


Figure 7.3: NRP FD state machine.

7.3.2 Assumptions

NRP FD requires that the NRP (i) has no common cause failure with the DCN and (ii) that the NRP is uniquely addressable.

The NRP must not have any common cause failure with the DCN, such as being powered by the same power supply or using the same CPU for processing requests, such as ICMP pings. That could be the case if the NRP were an embedded switch mounted on the DCN hardware.

The DCN is assumed to have an IP address per network, and the IP addresses of the partner DCN per network, are known for each DCN. The NRP candidates' IP addresses are pre-configured and unique, making the NRP candidates uniquely addressable.

7.3.3 Detailed Description

We begin the detailed description with the startup and the transition to the specific roles, followed by NRP selection. Then, we continue with the actual failure detection, the (i) backup - supervision handling, and the (ii) primary - supervisee handling. Finally, the NRP change handling wraps up the section.

7.3.3.1 Startup - Init / Waiting

Algorithm 1 describes the startup and initial handling. It consists of the function BECOMEWAITING that does three things (i) wait for available NRP candidates, (ii) wait for acknowledgment to become primary or a heartbeat from an existing primary, (iii) transition to the primary or backup role.

WAITFORNRPCANDIDATES waits for at least one NRP candidate to be available. Available means answering to PINGNRP. When at least one NRP candidate is available, the next step is to wait for either a heartbeat from an existing primary or an OK-to-be-primary acknowledgment. The function HBORPRIMARYACK performs this wait. An operator or maintenance engineer issues the primary acknowledgment, okaying the DCN to become primary, mitigating the risk that a DCN starting up in a partitioned network becomes primary. The person issuing the acknowledgment must ensure that this is not the case.

7.3.3.2 BecomePrimary - Transition to the Primary Role

The transition to the primary role only consists of selecting an NRP from the NRP candidates ($reachableCandidates$ from Algorithm 2), Line 20-23 in Algorithm 1. Any NRP candidate reachable is selectable.

Algorithm 1 Startup - init / waiting

```
1:  BECOMEWAITING( )
2:  function BECOMEWAITING( )
3:      WAITFORNRPCANDIDATES( )
4:      do
5:          do
6:              hbOrAckSts ← HBORPRIMARYACK( )
7:          while hbOrAckSts ≠ HbOrPrimaryAck
8:          if hbOrAckSts = AckToBePrimary then
9:              BECOMEPRIMARY( )
10:         else                                  ▷ HB seen, become backup
11:             BECOMEBACKUP( )
12:         end if
13:     while not(isPrimary OR isBackup)
14: end function
15: function WAITFORNRPCANDIDATES( )
16:     do
17:         MONITORNRP( )                         ▷ See Algorithm 2
18:     while reachableCandidates = {∅}
19: end function
20: function BECOMEPRIMARY
21:     SELECTNRP( )                              ▷ Select the NRP to use.
22:     isPrimary ← TRUE
23: end function
24: function BECOMEBACKUP
25:     SENDIMHERETOPRIMARY( )
26:     isOwnIpInHb ←WAITFORONWADDRINHB(Tmo)
27:     if isOwnIpInHb == TRUE then
28:         isBackup ← TRUE
29:     else
30:         isBackup ← FALSE
31:     end if
32: end function
```

7.3.3.3 BecomeBackup - Transition to the Backup Role

The backup informs the primary of its presence, Line 25-27 in Algorithm 1. Then, the backup waits for the primary to acknowledge its presence by adding the IP address of the backup to the heartbeat field *Backups Known*, see Table 7.1. This ensures that the backup does not resume the primary role due to a failure when the primary is unaware of the backup. Since a primary without backups will remain primary even if the NRP reachability is lost. A backup is not a backup unless it sees its address in the heartbeat field *Backups Known*, continuously checked while in the backup state.

7.3.3.4 NRP Selection and Candidate Monitoring

The set $reachableCandidates$, contains the reachable NRP candidates; see Algorithm 2. The primary selects an NRP from the $reachableCandidates$ set as the NRP. The $reachableCandidates$ set is updated with a suitable interval, for example, a few times per minute. Each network has an NRP candidate. The algorithm description does not cover how NRP FD becomes aware of NRP candidates; however, Section 7.5.1 presents alternatives. The function GETCANDFORNW represents NRP candidate retrieval for a specific network nw.

Algorithm 2 NRP candidate monitoring

```
1: function MONITORNRP( )
2:     reachableCandidates ← {∅}
3:     for all nw ∈ AllNetworks do
4:         candidate ← GETCANDFORNW(nw)
5:         isReachable ← PINGNRP(candidate)
6:         if isReachable then
7:             reachableCandidates ← reachableCandidates ∪ {candidate}
8:         end if
9:     end for
10: end function
```

7.3.3.5 Backup - Supervising

The backup continuously, with a cycle time preferably longer than the heartbeat interval, announces its presence to the primary, see Algorithm 3 Line 2. Further described in Section 7.3.3.6.

Algorithm 3 Backup - supervising

```
1:  while isBackup do
2:      SendImHereToPrimary( )                          ▷ Longer interval.
3:      hbSts ← ChkHbStsOnAllNw( )
4:      if hbSts == tmoAllSimul then
5:          BecomePrimary( )
6:      else if hbSts == tmoAllNotSimul then
7:          isNrpReachable ← PingNRP( )
8:          if isNrpReachable then
9:              BecomePrimary( )
10:         end if
11:     else if hbSts == tmoSomeNotAll then
12:         isNrpReachable ← PingNRP( )
13:         if not isNrpReachable then
14:             AskPrimaryToSwitchNRP( )
15:         end if
16:     end if
17:     isBackup ← isOwnIpInHb                          ▷ See Alg. 1
18: end while
```

The backup checks the heartbeats on all used networks, Line 3. If heartbeats timed out on all networks and two or more timed out simultaneously, NRP FD assumes that the cause is a failure of the primary rather than two (or more) independent network failures in a short time frame, Line 3-5. Treating a simultaneous timeout of heartbeats as a primary failure is an optimization, a way to reduce decision time by avoiding PingNRP when a failover time shorter than the response time of the PingNRP is required. If the PingNRP have a sufficiently low upper bounded reply time, Line 3-5 could be removed and be covered by the handling described on Line 6-10.

Equation 7.1 defines the simultaneous timeout, where $\Delta hbTmo$ is the interval for considering two heartbeats to be simultaneous, $hbTmoT$ is the time of the timeout, and NW is the set of all networks connecting the primary and backup. $hbTmoT$ is zero in case of no timeout.

$$\begin{aligned} & i, j \in NW, \forall hbTmoT_i, \forall hbTmoT_j, \\ & i \neq j, hbTmoT_i \neq 0, hbTmoT_j \neq 0 \mid \\ & |hbTmoT_i - hbTmoT_j| \leq \Delta hbTmo \end{aligned} \tag{7.1}$$

If heartbeats timeout on all networks but not simultaneously (or if the optimization is not used), the NRP reachability test determines if the backup should become the primary, Line 6-10.

If heartbeats timed out on some, but not all, networks, the backup tests the NRP reachability. If unreachable, the backup asks the primary to change NRP, Line 11-16; further described in Section 7.3.3.7.

7.3.3.6 Primary - Supervisee

The primary sends heartbeats on all networks, with the configured interval containing the address of the current NRP; see Algorithm 4 Line 3.

Algorithm 4 Primary - supervisee

```
1:  nrpAddr ← GETNRPADDR( )
2:  while isPrimary do
3:      SENDHEARTBEAT(nrpAddr)
4:      isNRPReachable ← PINGNRP(nrpAddr)
5:      if not isNRPReachable then
6:          newNRPAddr ← GETNEWREACHABLENRP( )
7:          if IsValid(newNRPAddr) then
8:              if isBupAvailable then
9:                  ASKBUPTOCHGNRP(newNRPAddr)
10:                 isNRPChgOk ← NRPCHGRESP(Tmo)
11:                 if isNRPChgOk then
12:                     nrpAddr ← newNRPAddr
13:                 else
14:                     BECOMEWAITING( )                ▷ See Alg. 1
15:                 end if
16:             else
17:                 nrpAddr ← newNRPAddr
18:             end if
19:         else
20:             if isBupAvailable then
21:                 BECOMEWAITING( )                    ▷ See Alg. 1
22:             end if
23:         end if
24:     end if
25:     isBupAvailable ← ISBACKUPPRESENT( )
26: end while
```

The primary also checks the NRP reachability, Line 4. A reachable NRP does not require any further actions. The remaining parts of Algorithm 4 pseudocode cover the unreachable NRP scenario, Line 5-24. If a backup and an alternative NRP candidate exist, the primary tries to switch NRP, Line 7-18.

The primary must ensure that the backup accepts an NRP change before changing NRP. Hence, the primary requests the backup to change NRP and waits for a response with a timeout, Line 9-15. The primary leaves the primary role if it does not receive a positive confirmation in time. If no backup exists, the primary just switches NRP.

If no reachable NRP candidate exists, Line 19-23, the primary leaves the primary role if the backup is present and remains primary if the backup is not present. The primary expects the backup to report its presence within a timeout, see Line 25, allowing the primary to vacate the primary role only if a backup is present. The backup presence timeout is preferably at least one order of magnitude larger than the heartbeat interval since backup presence detection does not affect the failover time.

Table 7.1: Heartbeat message fields.

Name	**Description**
NRP	Address of the current NRP.
Backups Known	The addresses of the backups known by the primary.
Iteration (seq.) number	Identifies the iteration/cycle the the heartbeat was sent. Incremented each cycle by the primary.

7.3.3.7 NRP Change Handling

The backup and primary must never use different NRPs; an NRP change must never violate that. It is beneficial to change the NRP in two situations. The first is when the primary fails to reach the NRP. In that situation, the primary proposes a new NRP to the backup as shown in Algorithm 4. A backup that accepts the new NRP starts using it after it sees the new NRP address in a heartbeat. Hence, during the time between acceptance and heartbeat receiving, the backup can not use the NRP reachability to become primary. The backup can only know that the acceptance message is delivered to the primary once it sees a confirmation. The confirmation is the changed NRP address in received heartbeats. If the NRP address is not changed, the backup reverts to using the

former NRP address after two heartbeat cycles by using the iteration number in the heartbeat message, see Table 7.1.

The second situation is when the backup cannot reach the NRP. A backup that cannot reach the NRP cannot take the primary role using the NRP reachability test. Hence, in that situation, the backup suggests that the primary switches the NRP.

7.4 Consistency and Availability Comparison

In this section, we present a comparison of consistency and availability between Conv. FD and NRP FD using switch-failure scenarios. We use two topologies, $T1Sw$ with one switch per network between the DCN redundancy pair and $T3Sw$ with three switches, depicted in Figure 7.2.

We assume that all switches, and thereby the NRP and NRP candidates, are the same make and model and have the same MTTF. We use a switch MTTF of 75 years, same as the Westermo managed-switch Lynx[3]. The DCN MTTF is assumed to be lower since DCNs are likely more complex hardware-wise. We assume a DCN MTTF of 20 years.

The reliability function $R(t) = e^{-\lambda t}$ and the corresponding failure function $F(t) = 1 - R(t)$ give the failure probabilities. We use a run time of ten years, i.e., $t = 10$. The scenarios are multiple faults, requiring the failure of one network path between the primary and backup and another failure. One week is the assumed reparation time for the first failure. Heartbeats are sent from the primary to the backup every 50 milliseconds, Line 3 in Algorithm 4, and the backup presence timeout $IsBackupPresent$, Line 25, is one second. Table 7.2 presents the probabilities for the scenarios described below.

The scenarios where NRP FD prioritizes consistency before availability by vacating the primary or backup role potentially negatively impact availability. Therefore, these are the scenarios used for comparison. In other words, we compare the probability of scenarios that lead to no primary using NRP FD with the likelihood of scenarios leading to dual primary using Conv. FD.

NRP FD vacates the primary role if the NRP is not reachable, a backup is present, and the request to change NRP fails, see Line 5-24 in Algorithm 4. The NRP change request can fail for two reasons: partitioning or backup failure. In the partitioning case, the backup will become primary unless the second failure is a failure of the NRP itself, denoted $NRPFdNRPAndNWFail$, failure combination $F1F4$ in

[3] https://www.westermo.se/-/media/Files/Data-sheets/westermo_ds_lynx_100-and-200-series_2205_en_revg.pdf

Table 7.2: Availability and consistency loss probabilities.

Scenario	Probability	
	NRP FD No primary	Conv. FD Dual primary
$NRPFdNRPAndNWFail$	$T1Sw$: $3.2 * 10^{-3}\%$ $T3Sw$: $8.5 * 10^{-3}\%$	0% [1]
$NRPFdNRPAndBupFail$	$T1Sw$: $1.7 * 10^{-8}\%$ $T3Sw$: $1.7 * 10^{-8}\%$	0%
$NRPFdBupNotRchNRP$	$T1Sw$: $2.0 * 10^{-9}\%$ $T3Sw$: $5.2 * 10^{-9}\%$	0%
$ConvFdNwPart$	See note[2]	$T1Sw$: $3.2 * 10^{-3}\%$ $T3Sw$: $2.5 * 10^{-2}\%$

[1] Partitioning due to NRP failure is included in $ConvFdNwPart$ for Conv. FD.

[2] Partitioning due to failure of any switch is covered by $NRPFdNRPAndNWFail$ for NRP FD.

Figure 7.4 is an example of this. For $NRPFdNRPAndNWFail$ to result in a no primary situation, the NRP must fail while the other network path is broken. Hence, the probability increases with the reparation time. The second reason and scenario for an NRP change request to fail is a backup failure simultaneously to the primary loss of NRP reachability, denoted $NRPFdNRPAndBupFail$. For $NRPFdNRPAndBupFail$ to cause a non-primary situation, the NRP must fail before the $IsBackupPresent$ expires, see Line 25 in Algorithm 4.

If the backup cannot reach the NRP, it cannot resume the primary role using the PINGNRP. Therefore, it proposes an NRP change; see Line 12-15 in Algorithm 3. If the primary fails before it has changed the NRP, the backup won't takeover, denoted $NRPFdBupNotRchNRP$. We use 100 milliseconds as the NRP change time, i.e., twice the heartbeat interval.

Conv. FD causes a dual primary situation in any network partitioning scenario, denoted $ConvFdNwPart$.

Table 7.2 shows that the inconsistency probability for Conv. FD is higher than NRP FD's availability loss probability for topology $T3Sw$. The main reason for that is that a Conv. FD causes a dual primary situation for any partitioning situation, i.e., if any of the switches between the primary and backup fail on the two networks. NRP FD only causes a no primary situation when the specific failure mentioned above occurs. For $T1Sw$, the probability of availability loss with NRP FD is marginally higher than the probability for

consistency loss using Conv. FD.

7.5 Implementation and Evaluation

7.5.1 Implementation

The NRP candidates are the switches, see Table 7.3 and Figure 7.4, that NRP FD learns about using a dynamic approach based on Link Layer Discovery Protocol (LLDP). LLDP allows neighboring devices to announce their presence. The switches reveal their existence and IP addresses with LLDP, and NRP FD in each DCN learns about adjacent switches on each network. Those adjacent switches are the NRP candidates, from which NRP FD selects an NRP.

LLDP provides vendor-specific extension possibilities. Hence, LLDP could announce the support of low latency PINGNRP that NRP FD could utilize if available. For example, a future switch supporting a real-time PINGNRP could reveal that through LLDP, and NRP FD could use this knowledge to avoid using simultaneous heartbeat timeout handling when switches that support real-time capable PINGNRP are used.

Our implementation uses ICMP ping as the PINGNRP, and our test shows that our switches typically reply to ping with a sub-millisecond response time.

7.5.2 Evaluation

Figure 7.4 shows the evaluation setup we use. The Input sig. provider emulates an input I/O. It generates a sine wave with a frequency of two Hertz and an update rate of 100 updates per second. Outputting a new value every ten milliseconds on network A and B, using UDP multicast.

The control application that runs on the redundant DCN pair (DCN A1 and DCN A2) expects Input sig. provider values, and outputs values to the Output sig. consumer. The control application checks for an updated value every five ms. If received, it outputs the value received; if it does not receive a value after three iterations, it outputs the ISP value zero.

The Output sig. consumer emulates an output I/O and uses the last received value as OSP. It is a Windows application that plots the received value for visualization purposes; Figure 7.6 shows the Output sig. consumer value plot when OK.

We compare our NRP FD and Conv. FD implementation and show the resulting consistency difference between NRP FD and Conv. FD in network partitioning situations by breaking the network in the places marked $F1$ - $F4$

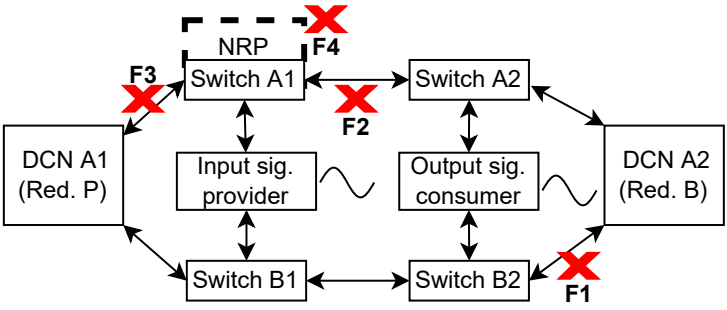


Figure 7.4: The evaluation setup used. F1 - F4 marks the network faults induced.

Table 7.3: Software and hardware used.

Node name	**Hardware**	**Software**
DCN A1	MSI, Core i3	VxWorks 7.0
DCN A2	Lenovo, Core i5	VxWorks 7.0
Input sig. provider	Intel NUC, Core i3	VxWorks 7.0
Output sig. consumer	Lenovo, Core i7	Windows 10
Switches	Zyxel, GS1900-8	V2.40

in Figure 7.4. Fault $F1$ breaks network B and is always the first fault. We combine $F1$ with $F2$ - $F4$ separately, resulting in three different fault combinations, (i) $F1F2$, (ii) $F1F3$, and (iii) $F1F4$.

Figure 7.5 shows the Output sig. consumer plot with fault combination $F1F2$ using Conv. FD. DCN A1 receives values from Input sig. provider while DCN A2 does not. Hence, DCN A1 outputs the sine wave values, and DCN A2 uses the ISP value and outputs a constant zero. Output sig. consumer receives values from both DCN A1 and DCN A2, resulting in the distorted sine wave shown in Figure 7.5. With the same fault combination, $F1F2$, NRP FD ensures that DCN A1 remains primary and DCN A2 passive. Hence, the Output sig. consumer receives consistent values, shown in Figure 7.6.

The fault combination $F1F3$ results in a dual primary situation using Conv. FD. The difference from $F1F2$ is that DCN A1 and DCN A2 receive values from the Input sig. provider. Hence, the inconsistency in the output values is less than for $F1F2$, as shown in Figure 7.7, but it still shows. With NRP FD, DCN A2 becomes primary, and DCN A1 vacates the primary role, keeping consistency as shown in Figure 7.6.

Fault combination $F1F4$ covers NRP failure ($F4$) and the Conv. FD result

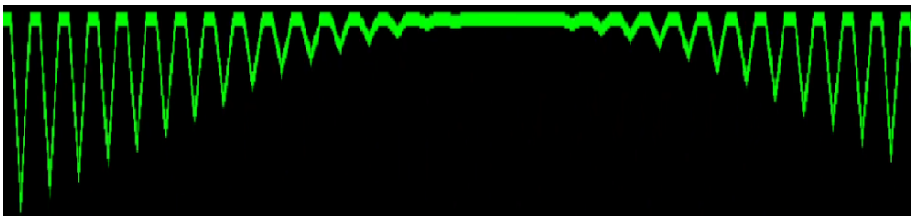


Figure 7.5: Conv. FD based redundancy causes dual primaries in partitioning situation $F1F2$. Output sig. consumer receives different (inconsistent) values from DCN A1 (over the lower network) and DCN A2 (over the uper network).

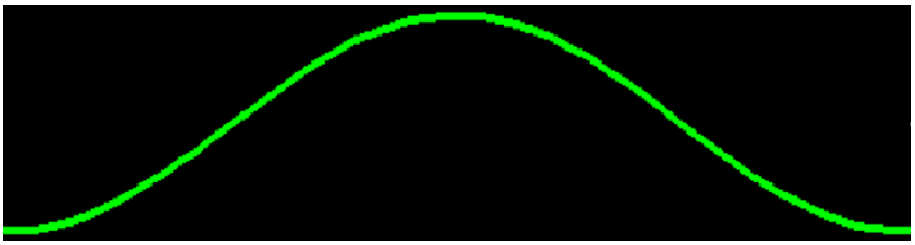


Figure 7.6: Correctly received and plotted values by Output sig. consumer. NRP FD ensures one primary; hence, Output sig. consumer receives the correct sine wave values even under partitioning.

is the same as for $F1F2$, shown in Figure 7.5. Fault combination $F1F4$ using NRP FD results in DCN A1 vacating the primary role; it cannot reach the NRP nor elect a new NRP since it cannot communicate with DCN A2. DCN A2 does not become primary since it can't reach the failed NRP. Hence, no primary and Output sig. consumer, do not get any values and therefore use the OSP value, shown in Figure 7.8.

Performance is also an important aspect, and the penalty using NRP FD comes from the additional PINGNRP, realized with ICMP ping in our evaluation. However, the overhead is typically less than a millisecond. We used a heartbeat period of five milliseconds and required two missing heartbeats to indicate a failure. With that configuration, we did not see any difference when measuring the time between two consecutive updates in the Output sig. consumer when primary failed.

More elaborated performance measurements are future work. Future implementation optimized for performance and combined with a time bounded low latency implementation in the switches for the PINGNRP.

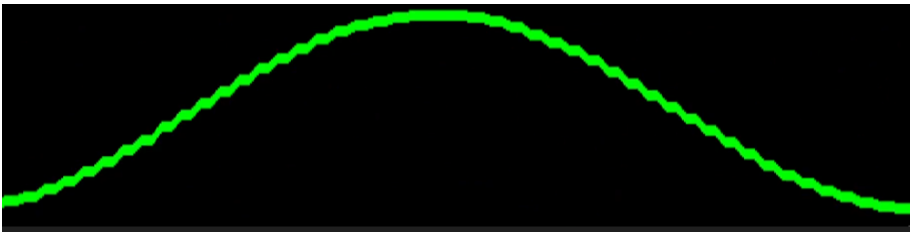


Figure 7.7: Conv. FD caused dual primary, where both DCN use the same input values, hence inconsistency is less than for $F1F2$, shown in Figure 7.5, but still visible when compared to Figure 7.6.

Figure 7.8: No value received - Output sig. consumer plots the last value, as OSP.

7.6 Summary and Future Work

This paper presents NRP FD, a failure detection algorithm that prioritizes consistency, in contrast to the conventional failure detection, Conv. FD, that prioritizes availability. The target use case, and the description base of the NRP FD algorithm, is controller redundancy. With a theoretical comparison between NRP FD and Conv. FD we showed that the NRP FD is less likely to lose availability than Conv. FD is to lose consistency. Further, with a testbed and implementations of NRP FD and Conv FD, we showed the consistency gain NRP FD gives over Conv. FD by injecting failures causing network partitioning between the redundant pair.

Potential continuations include incorporating a hard real-time, low-latency PINGNRP support in the switches to perform a more exhaustive and challenging failover performance test. With tailored switch support, the heartbeat and PINGNRP could be integrated to allow NRP FD to guarantee at most one primary, not only under persistent network partitioning but also under transient disturbance. Future work would aim to prove that property using model checkers and probabilistic network models.

Furthermore, a low-latency PINGNRP can also be used for performant network supervision and breakage localization if combined with a topology map. Additional future work is to combine NRP FD with failure detection and role selection targeting other redundancy configurations than 1oo2, such as Heartbeat bully [31].

Bibliography

[1] The dcs of tomorrow - abb's process automation system vision whitepaper. https://new.abb.com/control-systems/control-systems/envisioning-the-future-of-process-automation-systems/automation-system-whitepaper. Accessed: 2023-03-20.

[2] Prakash Ganesan, S Gunasekaran, Ar Ashwin Balaji, S Jini Fathima, and S Suryaprakash. Comparative analysis on industrial iot communication protocols and its future directives. In *2021 International Conference on Advancements in Electrical, Electronics, Communication, Computing and Automation (ICAECA)*, pages 1–6. IEEE, 2021.

[3] Gábor Soós, Dániel Ficzere, and Pál Varga. Investigating the network traffic of industry 4.0 applications–methodology and initial results. In *2020 16th International Conference on Network and Service Management (CNSM)*, pages 1–6. IEEE, 2020.

[4] Thomas Goldschmidt, Mahesh Kumar Murugaiah, Christian Sonntag, Bastian Schlich, Sebastian Biallas, and Peter Weber. Cloud-based control: A multi-tenant, horizontally scalable soft-PLC. *2015 IEEE 8th International Conference on Cloud Computing*, pages 909–916, 2015.

[5] T. Hegazy and M. Hefeeda. Industrial automation as a cloud service. *IEEE Trans. Par. and Distr. Syst.*, 26(10):2750–2763, 2015.

[6] Bjarne Johansson, Mats Rågberger, Thomas Nolte, and Alessandro V Papadopoulos. Kubernetes orchestration of high availability distributed control systems. In *Proc. ICIT*, 2022.

[7] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. A new adaptive accrual failure detector for dependable distributed systems. In *In ACM Symposium on Applied Computing (SAC 2007*, pages 551–555, 2007.

[8] Abb ability™ system 800xa control and i/o overview. https://library.e.abb.com/public/1a8bb0db5cf4474a95a113f96c18a5c2/3BSE047351%20en%20K%20ABB%20Ability%20System%20800xA%20Control%20and%20IO%20Overview.pdf. Accessed: 2023-03-20.

[9] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[10] Eric Brewer. Cap twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29, 2012.

[11] Edward A Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. Trading off consistency and availability in tiered heterogeneous distributed systems. *Intelligent Computing*, 2:0013, 2023.

[12] Edward A Lee, Ravi Akella, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. Consistency vs. availability in distributed real-time systems. *arXiv preprint arXiv:2301.08906*, 2023.

[13] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[14] Wei Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32, Jan 2002.

[15] C. Fetzer, M. Raynal, and F. Tronel. An adaptive failure detection protocol. In *Proceedings 2001 Pacific Rim International Symposium on Dependable Computing*, pages 146–153, Dec 2001.

[16] M. G. Gouda and T. M. McGuire. Accelerated heartbeat protocols. In *Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No.98CB36183)*, pages 202–209, May 1998.

[17] Maarten Van Steen. *Distributed systems*. Citeseer, 2017.

[18] Anna Charny, Robert Goguen, Carol Iturralde, Elisheva Hochberg, and Jean Philippe Vasseur. Distinguishing between link and node failure to facilitate fast reroute, July 26 2011. US Patent 7,986,618.

[19] Clarence Filsfils and David D Ward. Technique for distinguishing between link and node failure using bidirectional forwarding detection (bfd), December 20 2011. US Patent 8,082,340.

[20] Dave Katz and David Ward. Bidirectional Forwarding Detection (BFD). RFC 5880, June 2010.

[21] Hartmut Ritter, Rolf Winter, and Jochen Schiller. A partition detection system for mobile ad-hoc networks. In *2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004.*, pages 489–497. IEEE, 2004.

[22] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. Comput.*, 31(1):48–59, January 1982.

[23] A. Arghavani, E. Ahmadi, and A. T. Haghighat. Improved bully election algorithm in distributed systems. In *ICIMU 2011 : Proceedings of the 5th international Conference on Information Technology Multimedia*, pages 1–6, Nov 2011.

[24] Minhaj Khan, Neha Agarwal, Saurabh Jaiswal, and Jeeshan Ahmad Khan. An announcer based bully election leader algorithm in distributed environment. In Pushpak Bhattacharyya, Hanumat G. Sastry, Venkatadri Marriboyina, and Rashmi Sharma, editors, *Smart and Innovative Trends in Next Generation Computing Technologies*, pages 664–674, Singapore, 2018. Springer Singapore.

[25] Seok-Hyoung Lee and Hoon Choi. The fast bully algorithm: For electing a coordinator process in distributed systems. In Ilyoung Chong, editor, *Information Networking: Wireless Communications Technologies and Network Applications*, pages 609–622, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[26] Leslie Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. 2019.

[27] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[28] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pages 305–319, 2014.

[29] Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM journal of research and development*, 6(2):200–209, 1962.

[30] Elena Dubrova. *Fault-tolerant design*. Springer, 2013.

[31] B. Johansson, M. Rågberger, A. V. Papadopoulos, and T. Nolte. Heartbeat bully: Failure detection and redundancy role selection for network-centric controller. In *IECON*, 2020.

Chapter 8

Paper C: OPC UA PubSub and Industrial Controller Redundancy

Bjarne Johansson, Olof Holmgren, Martin Dahl, Håkan Forsberg, Alessandro V. Papadopoulos, and Thomas Nolte.

In 29th International Conference on Emerging Technologies and Factory Automation (ETFA), 2024.

Abstract

Industrial controllers constitute the core of numerous automation solutions. Continuous control system operation is crucial in certain sectors, where hardware duplication serves as a strategy to mitigate the risk of unexpected operational halts due to hardware failures. Standby controller redundancy is a commonly adopted strategy for process automation. This approach involves an active primary controller managing the process while a passive backup is on standby, ready to resume control should the primary fail. Typically, redundant controllers are paired with redundant networks and devices to eliminate any single points of failure. The process automation domain is on the brink of a paradigm shift towards greater interconnectivity and interoperability. OPC UA is emerging as the standard that will facilitate this shift, with OPC UA PubSub as the communication standard for cyclic real-time data exchange. Our work investigates standby redundancy using OPC UA PubSub, analyzing a system with redundant controllers and devices in publisher-subscriber roles. The analysis reveals that failovers are not subscriber-transparent without synchronized publisher states. We discuss solutions and experimentally validate an internal stack state synchronization alternative.

8.1 Introduction

Automation solutions are crucial in modern society and pivotal in infrastructure for critical utility services such as power and freshwater distribution. At the core of these automation solutions is the controller, which interacts with the physical environment through Input and Output (I/O) devices. The controller processes data from input devices to assess the system's status and directs output devices to achieve desired outcomes, forming what is known as the control loop. In certain domains, such as offshore oil and gas production, halts can incur significant costs, particularly unexpected halts due to hardware failures. Hence, the reliability requirements for the components constituting the control loop are high in such domains.

A widely adopted strategy to mitigate the risk of unplanned stops caused by hardware failures is the implementation of spatial redundancy. This involves duplicating critical hardware components such as devices, communication systems, and controllers. The aim is to ensure that a single fault does not lead to a system halt, effectively eliminating what is known as a Single Point of Failure (SPoF). By having redundant components, the system can continue to operate even if one component fails, thereby enhancing the system's reliability and reducing the risk of unexpected halts.

The predominant industrial controller redundancy solution is standby redundancy [1]. Standby controller redundancy means that one of the controllers in the controller pair is assigned the primary role, meaning that it is the active controller driving the process [2]. The other controller in the pair is the backup, which is passive until the primary controller fails. In such a situation, the backup assumes the primary role and continues to run the control application. The same principle commonly applies to redundant devices, where one is the primary, and the other is the backup.

The automation domain is experiencing a paradigm shift driven by Industry 4.0's demand for data, increased interoperability, and interconnectivity. OPC UA is identified as the enabling standard for interoperability [3]. The PubSub part of the OPC UA standard details a publish-subscribe model suitable for cyclic real-time communication between controllers and I/O devices [4]. Further, OPC UA PubSub is the communication foundation in the OPC UA Field eXchange (UAFX) standard, targeting field device communication [5, 6].

Our contribution is the OPC UA PubSub analysis through the standby redundancy lens. Using controller and device redundancy as an analysis basis, we identify challenges in publisher failover when using the standard's normative configuration for real-time exchange. The issue is subscriber expectancy on message information populated by the publisher, which typically

lacks replication in backup publishers. Based on our analysis, we propose alternative solutions and validate our findings through experiments, including a basic test where we synchronize publisher internals.

The paper is organized as follows: Section 8.2 present related work; Section 8.3 provides an overview of OPC UA, especially PubSub; Section 8.4 discusses OPC UA PubSub's behavior in selected controller redundancy failure scenarios; Section 8.5 explores OPC UA PubSub redundancy adaptations, complemented by an experimental evaluation in Section 8.6; and finally, Section 8.7 concludes with a summary and future directions.

8.2 Related Work

This work explores OPC UA PubSub in the context of controller and device redundancy. Redundancy is a means of fault tolerance. Fault tolerance, defined by Avizienis et al. as the preservation of operation in the face of faults, is a critical aspect of dependability [7]. The field of fault tolerance research, especially in embedded and industrial systems, is extensive. For instance, Ballesteros et al. introduce an architectural model that dynamically adjusts resilience by dynamic allocation of communication and computational resources according to task criticality [8]. Vitucci et al. investigate hardware design techniques that strengthen product reliability [9]. In the flourishing field of artificial intelligence, Nouioua et al. examine the use of machine learning for network fault detection [10].

Given the reliability requirements of industrial networks, several fault tolerance approaches exist, offering various types of spatial, temporal, or informational redundancy. Álvarez et al. comprehensively survey these mechanisms in industrial networks [11] and Danielis et al. [12] survey reliability aspects of industrial, Ethernet-based, protocols. Neither of the two surveys cover OPC UA PubSub. Nast et al.'s survey of communication protocols industrial applicability covers OPC UA PubSub and treats reliability as a requirement, but does not consider redundancy [13].

Regarding controller redundancy, Simion et al. note that standby modes—either hot or warm—are prevalent redundancy patterns in industrial controllers [1]. The distinction between hot and warm standby lies in the level of backup readiness. However, the backup controls the process in neither warm nor hot standby mode. Additionally, Stój et al. present a cost-effective approach to controller redundancy utilizing EtherCAT [14]. None of these controller redundancy-related works cover OPC UA PubSub.

In the context of OPC UA PubSub, Neumann et al. investigate the requirements that an OPC UA PubSub field device must meet [15]. However,

their study does not address reliability aspects. Additionally, the integration of OPC UA PubSub with Time Sensitive Networks (TSN) has been examined by various researchers, demonstrating the feasibility of achieving low latency in real-time communication [16, 17, 18, 5].

Redundancy, in the context of OPC UA, is considered by Ismail et al. that describe a redundant OPC UA server architecture based on ZooKeeper as the underlying replication means [19]. Additionally, Cupek et al. detail the implementation of an OPC UA server in Java, focusing on redundancy aspects [20]. However, these works are related to redundancy for OPC UA Servers, which differs from OPC UA PubSub, further described in Section 8.3.

The related work mentioned does not address OPC UA PubSub and redundancy. To our knowledge, this study is the first to examine OPC UA PubSub in the context of controller redundancy.

8.3 OPC UA

Established in 2008, OPC UA is a comprehensive standard for interoperability across various parts of industrial automation, encompassing machine-to-machine communication, commissioning, and engineering [21]. It introduces a platform-independent and service-oriented architecture and an information model where data and services are accessible via attributes and methods on objects within an information collection called AddressSpace. Remote access to AddressSpace exposed information typically uses OPC UA Client Server [22]. OPC UA Client Server is not designed for real-time, low-latency communication but specifies server redundancy handling. OPC UA prescribes OPC UA PubSub for cyclic, real-time communication, though it does not specify PubSub redundancy handling.

The OPC UA's PubSub part details a publish-subscribe communication model, complementing the client-server model and supporting deterministic cyclic process value exchange [5, 16, 18]. PubSub utilizes a Message Oriented Middleware (MOM) to decouple publishers and subscribers [4]. The MOM can be broker-based, where a broker connects publishers and subscribers, or broker-less, relying on network equipment to act as a broker via multicast groups. This work focuses on broker-less PubSub over User Datagram Protocol (UDP), which targets real-time cyclic data exchange.

8.3.1 OPC UA PubSub - Internals

This section outlines the internals of OPC UA PubSub as defined by the standard [4]. Figure 8.1 depicts the objects and their interconnections. The Pub-

lishSubscribe object is the root for all PubSub objects, aggregating one or more PubSubConnections. A PubSubConnection defines the transport protocol, e.g., UDP, and specifies the destination address, either unicast or multicast. Each PubSubConnection may have multiple WriterGroups. WriterGroups encapsulates the data from DataSetWriter into a NetworkMessage. The DataSetWriter fetches and formats the data from PublishedDataSet for publishing. On the subscriber side, ReaderGroup is the receiving counterpart of the WriterGroup, and DataSetReader, analogous to DataSetWriter, unpacks the data and updates the SubscribedDataSet.

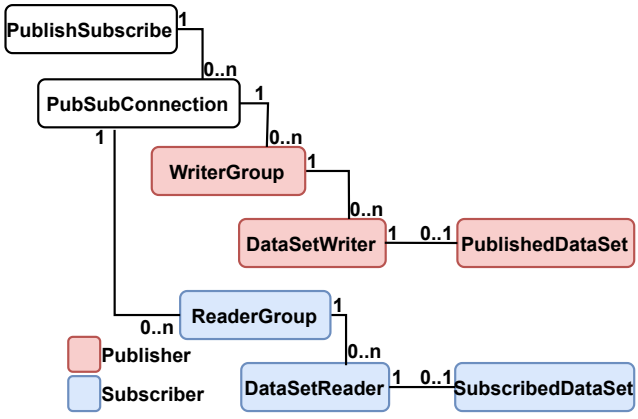


Figure 8.1: OPC UA PubSub objects.

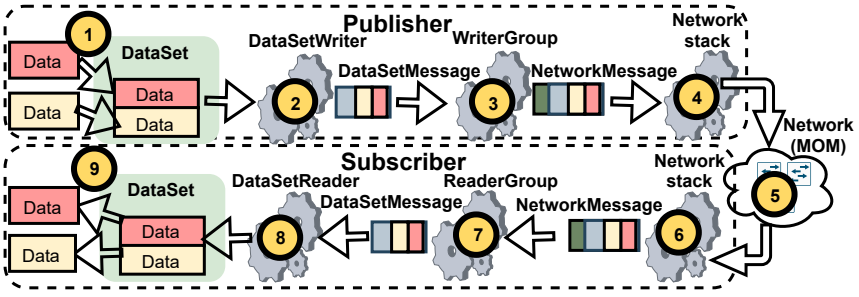


Figure 8.2: OPC UA PubSub data flow from publisher to subscriber.

Next, to describe the OPC UA PubSub internals, we'll follow the data from a publisher to a subscriber, depicted in Figure 8.2. The initial step (1) collects the DataSet data for publication. The data acquisition method depends on the type of publisher, for example, sampling I/O values or reading a variable in

a control application. The PublishedDataSet defines the data source and type of data, including the DataSetMetaData, which is essential for subscribers to interpret the received DataSetMessage.

Step (2) encapsulates the DataSet into a DataSetMessage using the DataSetWriter. The DataSetWriter DataSetMessage creation offers flexibility in fields selected for inclusion into the DataSetMessage. Which fields to include is controlled by the DataSetFieldContentMask. For example, fields like ConfigurationVersion and DataSetMessage SequenceNumber can be included, as detailed in Figure 8.3.

Next, the WriterGroup (3) encapsulates the DataSetMessage into a NetworkMessage. A single WriterGroup can receive DataSetMessages from multiple DataSetWriters, allowing a NetworkMessage to carry several DataSetMessages. As the DataSetWriter, the WriterGroup offers flexibility in which fields to include in the NetworkMessage, such as sequence numbers. The PublishingInterval parameter of the WriterGroup determines the frequency of publishing.

In step (4), the WriterGroup sends the NetworkMessage to the publisher's network stack. The broker-less middleware utilizes the network for message broking (5). The publisher can target the message to a specific subscriber using a unicast IP address or address multiple subscribers simultaneously with a multicast address. The network equipment, assumed to be layer two network switches in case of real-time exchange, ensures that the published message reaches the subscribers.

Upon arrival at the subscriber (6), the network stack verifies that the message is meant for this subscriber on the node level, i.e., confirming that the destination address matches a multicast address or the subscriber's unicast address (IP or MAC address).

Next, the ReaderGroup processes the incoming NetworkMessage (7), discarding any messages not intended for this subscriber by verifying the PublisherID in the NetworkMessage. It then extracts the DataSetMessage from the NetworkMessage and forwards the DataSetMessage to the appropriate DataSetReader.

The DataSetReader (8) uses the DataSetMetaData to decode the DataSetMessage and update the DataSet with the data received. The DataSetReader monitors the interval between two DataSetMessages. Suppose no new DataSetMessage appears within the period defined by the MessageReceiveTimeout parameter of the DataSetReader. In that case, the DataSetReader enters an error state. When in error state, the DataSetReader sets the data quality on data update by the DataSetReader to bad, indicating to any dependent application that the data is outdated. Finally, the specific

application determines how the received and updated data are utilized (9).

8.3.2 OPC UA PubSub Protocol - UADP

As discussed, OPC UA PubSub offers various alternatives for the underlying protocol and MOM, ranging from direct, broker-less Ethernet operation to broker-based solutions. This work focuses on the UDP-based alternative, where published network messages are encapsulated in UDP packets on Ethernet, known as the Unified Architecture Datagram Protocol (UADP) [4].

UADP targets cyclic real-time data exchange, such as controller and device communication. The standard outlines recommended message layouts and header configurations. Our description adheres to these recommendations, excluding security enhancements, which are left for future exploration. Figure 8.3 illustrates the normative UADP NetworkMessage fields for cyclic real-time communication, per the standard [4].

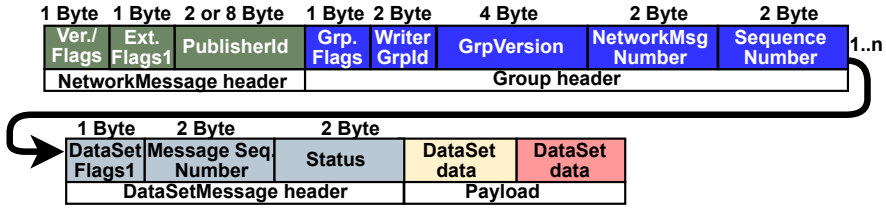


Figure 8.3: UADP NetworkMessage layout, with the normative header fields for cyclic data exchange.

The first field in the NetworkMessage header is the Version/flags byte, which specifies the UADP version and includes flags that indicate the presence of other header fields. The standard suggests that Extended Flags 1, PublisherId, and GroupHeader are included in the cyclic real-time normative NetworkMessage.

The second field, ExtendedFlags1 (ExtFlags1), further defines which additional header fields to expect. For instance, it determines whether the PublisherId is a 16-bit or 64-bit value, with 64-bit being the default. The PublisherId is the third field, a unique publisher identifier within the MOM.

Next is the Group header, containing WriterGroup information. The first field within this header is the GroupFlags (Grp. Flags), which, similar to the NetworkMessage header, indicates the presence of certain fields in the Group header. Again, the normative fields are illustrated in Figure 8.3. The WriterGroupId (WriterGrpId) uniquely identifies the WriterGroup within the

publisher. The GroupVersion (GrpVersion) notes the time of the last layout change to the data encapsulated by the WriterGroup, such as changes in header fields or DataSet reconfigurations. The NetworkMessageNumber (NetworkMsgNumber) is utilized if multiple NetworkMessages are required to transmit all DataSets managed by the WriterGroup, and the SequenceNumber increments with each message. The subscriber discards the messages that are deemed outdated by the sequence number comparisons. If no messages are received from the publisher within a time exceeding a predetermined "fail-time" (MessageReceiveTimeout), the receiver should be prepared to accept any sequence number. This mechanism ensures resilience in scenarios where the publisher fails and subsequently recovers. However, the subscriber tags the SubscribedDataSet data quality as bad since it is not updated within the MessageReceiveTimeout.

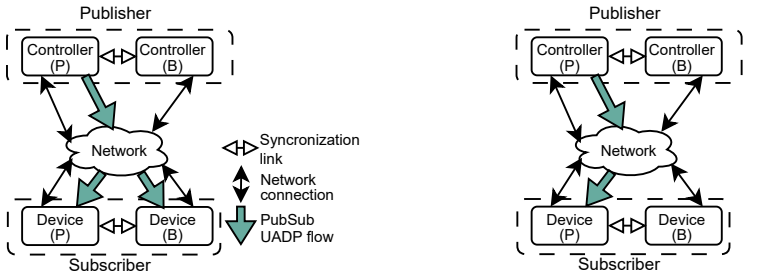
The DataSetMessage header and the DataSetMessage payloads come after the Group header and carry the published values/data. The first field in the DataSetMessage header is DataSetFlags1, specifying the subsequent header fields that are present. Next is the MessageSequenceNumber, a sequence number unique to the DataSetMessage, updated by the DataSetWriter for each DataSetMessage. The Status field follows the MessageSequenceNumber, providing quality information about the data/values within the DataSetMessage, indicating whether the data is good, bad, or uncertain. Last is the payload, comprising the application-specific data exchanged.

8.4 PubSub and Controller/Device Redundancy

In this section, we analyze OPC UA PubSub in the context of controller and device redundancy, using two configurations as depicted in Figure 8.4. The distinction between the two configurations is the type of UADP connection utilized for PubSub, i.e., multicast or unicast. As mentioned, the UADP PubSub configuration used is the normative for cyclic real-time data exchange as detailed in Section 8.3.2.

With the two configurations illustrated in Figure 8.4, we investigate failure recovery, i.e., failover, using four different failure scenarios. Those are:

- $\mathrm{PC_M}$ - Primary controller failure with multicast PubSub.
- $\mathrm{PC_U}$ - Primary controller failure with unicast PubSub.
- $\mathrm{PD_M}$ - Primary device failure with multicast PubSub.
- $\mathrm{PD_U}$ - Primary device failure with unicast PubSub.



(a) Multicast. (b) Unicast.

Figure 8.4: Redundant controller publishing to the redundant subscribing device.

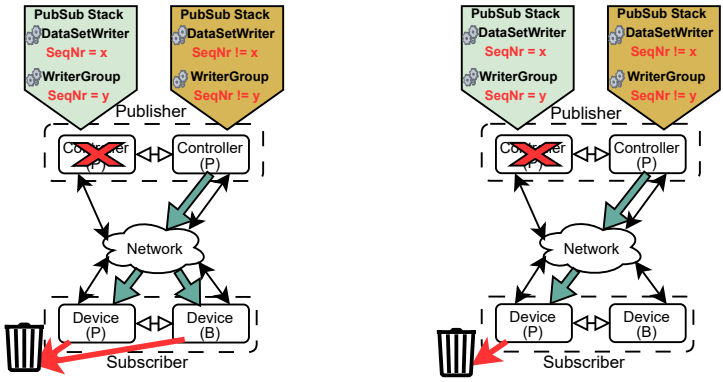
To keep the explanation and illustration as simple as possible, we assume that the controller is the publisher and the device the subscriber, even though a controller and, likewise, a device could be both subscriber and publisher or any other possible combination. We use the four failure scenarios as the basis for the following subsections.

The assumptions for our failure consequence analysis and their justifications are as follows: the backup can, through the synchronization link, detect if the primary fails and resume the primary role. The application states, particularly the control loop states, are synchronized with the backup. However, the internal states of the OPC UA PubSub stack are not synchronized. The rationale is that control system manufacturers often develop controller runtimes, whereas communication stacks, such as OPC UA, are typically third-party software integrated into the system. Therefore, synchronizing internal stack states is not commonly practiced. Furthermore, we assume the backup is configured identically to the primary regarding OPC UA PubSub-related settings.

8.4.1 Primary Controller Failure with Multicast PubSub - $\mathrm{PC_M}$

As illustrated in Figure 8.5a and elaborated in Section 8.3.1, the WriterGroup and DataSetWriter hold internal states that contribute to the composition of the NetworkMessage, such as the sequence numbers. These sequence numbers are part of the dynamic state data within the WriterGroup and DataSetWriter, and they change with each message transmitted.

When the backup controller takes over as primary, the necessary actions depend on the capabilities of the utilized stack. The backup may need to instantiate the WriterGroup and DataSetWriter, or, if the stack permits, these components could be pre-configured but inactive, allowing for a quicker tran-



(a) Muticast - $\mathrm{PC_M}$. (b) Unicast- $\mathrm{PC_U}$.

Figure 8.5: Primary controller failure, i.e., publisher failure.

sition to an operational state upon taking over as primary. Upon activation, the new primary begins publishing with the internal states and sequence numbers from its own WriterGroup and DataSetWriter, as shown in Figure 8.5a. Hence, the sequence number in the messages from the new primary is not resumed from where the former primary failed, causing the subscribing device's ReaderGroup to discard them as outdated. In a rare scenario where the backup takes over just before a sequence number wraparound, the first message from the new primary might have the expected sequence number, allowing the message to go through to the DataSetReader. However, the DataSetReader will likely reject the DataSetMessage due to an old MessageSequenceNumber.

DataSetMessages are discarded until the MessageReceiveTimeout expires. At this point, as mentioned in Section 8.3.2, the DataSetReader enters an error state, marking data quality as bad but resetting the expectation for sequence numbers. The subsequent DataSetMessage from the new primary is accepted, but this acceptance comes too late for a seamless transition, as data quality has already been compromised due to the expiration of the MessageReceiveTimeout.

Although multicast allows a backup subscribing device to receive data directly from the primary controller publisher, this doesn't address the sequence number expectation mismatch between subscriber and publisher due to the publisher failover.

8.4.2 Primary Controller Failure with Unicast PubSub - $\mathrm{PC_U}$

The outcome of the $\mathrm{PC_U}$ failure scenario is identical to that of $\mathrm{PC_M}$, because the failure originates at the publishing end in both examples, and the subscriber's sequence number expectations are the same. The only distinction is that in the $\mathrm{PC_U}$ scenario, only the primary device receives the published message. Nevertheless, this difference does not affect the outcome of the failure scenario; see Figure 8.5b.

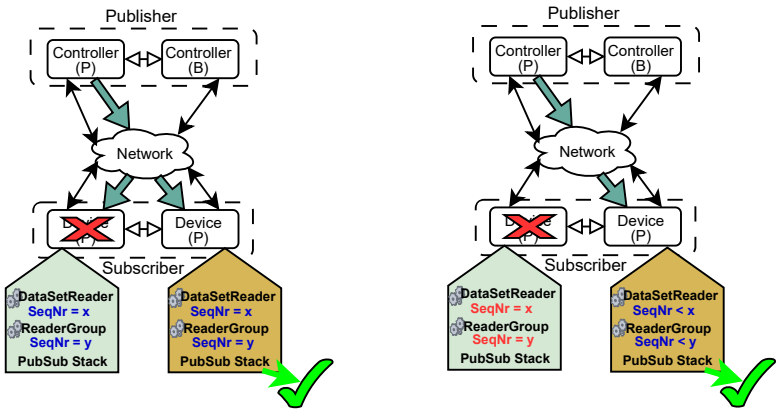
8.4.3 Primary Device Failure with Multicast PubSub - $\mathrm{PD_M}$

This scenario covers the failure of a subscribing primary device in a multicast configuration as depicted in Figure 8.6a. We assume both devices are appropriately configured and have their OPC UA PubSub stacks initialized to subscribe to the multicast published data. Hence, the primary and backup devices receive the data published by the primary controller. However, this requires the pair to ensure consistency. One alternative could be to discard the updated values on the backup when they reach the application layer where the redundancy roles are known. An alternate strategy would be to prevent the backup device from receiving any updates by only activating its subscription once it is required to take over as the primary. This approach would mirror the $\mathrm{PD_U}$ scenario.

In this, the $\mathrm{PD_M}$ scenario, the new primary's ReaderGroup and DataSetReader are already aligned with the former primary's since both devices have been receiving the same messages. Therefore, when the backup takes the primary role, it can seamlessly accept and process the published values from the controller.

8.4.4 Primary Device Failure with Unicast PubSub - $\mathrm{PD_U}$

In contrast to $\mathrm{PD_M}$, in this scenario, the backup device, when stepping into the primary role due to the failure of the former primary, hasn't received the latest values published by the controller. Upon starting to receive messages, the subscriber—now the new primary—has no preconceived expectations regarding the sequence numbers. Specifically, the ReaderGroup, having not received any NetworkMessage from the publishing controller previously, holds no anticipation about the sequence numbers from the publishing controller's WriterGroup. As a result, it would accept the incoming NetworkMessage and forward the contained DataSetMessages to the DataSetReader. Similarly, the DataSetReader, with no prior expectations regarding the MessageSequenceNumber in the DataSetMessage, would also accept the incoming message. In this scenario, the transition to receiving subscribed data by the new primary



(a) Multicast. (b) Unicast.

Figure 8.6: Primary device failure, i.e., subscriber failure.

device would be seamless, ensuring no interruption in data reception despite the role change.

8.4.5 Summary

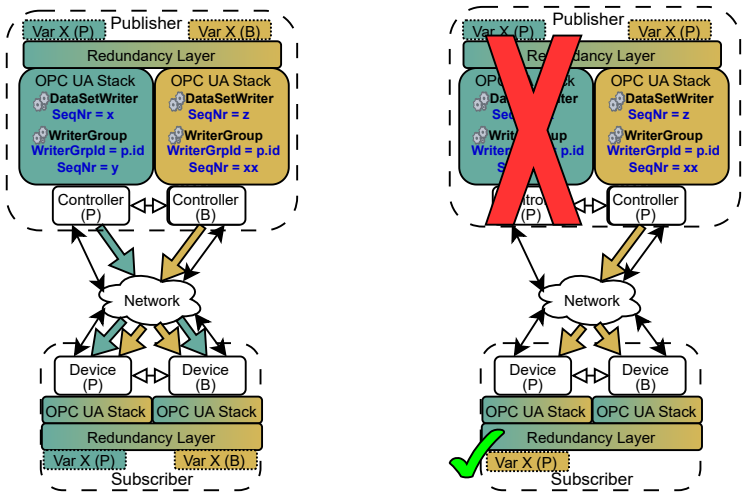
Table 8.1 summarizes the redundancy and failure scenarios discussed above. It shows that a primary subscriber's failure recovery ($\mathrm{PD_M}$, $\mathrm{PD_U}$) in a redundant pair can be transparent to the OPC PubSub data using layers in the device. However, the recovery of a publisher failure ($\mathrm{PC_M}$, $\mathrm{PC_U}$), e.g., the controller in our discussion, results in bad data quality status, which is undesirable.

Table 8.1: Failure and recovery scenario summary.

Scenario:	$\mathrm{PC_M}$	$\mathrm{PC_U}$	$\mathrm{PD_M}$	$\mathrm{PD_U}$
Result:	FAIL	FAIL	OK	OK

8.5 Improvement Alternatives

As summarized in Table 8.1, a publisher failure causes the subscriber to reject DataSetMessages and NetworkMessages from the new primary's DataSetWriter and WriterGroup. This section examines three strategies for seamless publisher failover: (i) the PubSub redundancy layer, (ii) stack synchronization, and (iii) standard extension alternatives, detailed further



(a) Before failure. (b) After failure.

Figure 8.7: PubSub redundancy layer establishing parallel publications.

in subsequent sections. While our examples use multicast publishers, the strategies also apply to unicast publishers.

8.5.1 PubSub Redundancy Layer

In the PubSub redundancy layer alternative, PubSub-related redundancy management occurs in a layer above OPC UA PubSub, termed the redundancy layer. As depicted in Figure 8.7a, each controller within the pair establishes a WriterGroup and DataSetWriter, while each device in the redundant pair configures a corresponding ReaderGroup and DataSetReader. Synchronization between the pair is managed at the redundancy layer, not within the OPC UA PubSub stacks.

Several approaches exist for the redundancy layer. One method involves embedding redundancy state information within the transmitted data. Embedding redundancy state information in the published data is similar to PROFINET's redundancy approach, which creates parallel logical connections between controller and device, distinguishing one as primary and the others as backups [23]. At the subscriber, the redundancy layer can opt to process data solely from the primary, similar to PROFINET's strategy, potentially minimizing the backup's activity. Note that the redundancy state information is carried in the application-specific data, whereas in PROFINET, this information is part of the protocol.

Another approach allows both controllers in the redundant pair to publish updates and actual data, enabling the subscriber to utilize data from either the primary or the backup, with the option to switch based on error indications from the corresponding DataSetReader or ReaderGroup. This necessitates the backup being up-to-date and publishing data at appropriate intervals, as illustrated by variable X in Figure 8.7a.

A third approach is to avoid parallel publications, accept the delays, and hide potential quality degradation resulting from MessageReceiveTimeout expiration in the redundancy layer. With this approach, the redundancy layer would manage data updates. This approach hides the resumption of subscription from the new primary publisher within the redundancy layer. This strategy is most suitable for RawData since the standard does not prescribe quality handling for RawData. A MessageReceiveTimeout would otherwise lead to subscribed data being marked with bad quality.

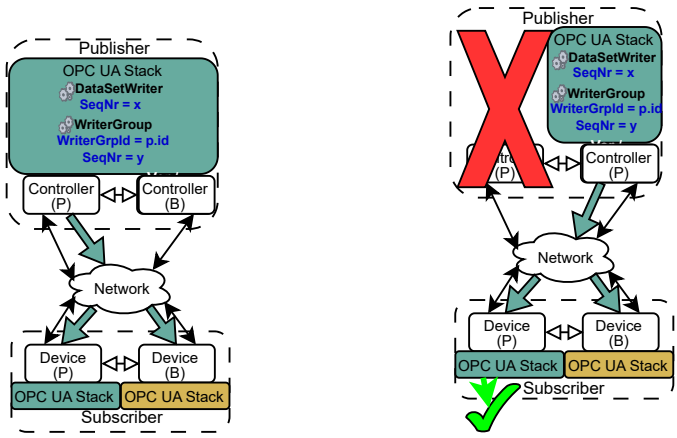
As exemplified, the redundancy layer is realizable in various incompatible ways. Hence, the redundancy layer alternative will likely need standardization to maintain interoperability between vendors.

8.5.2 Stack Synchronization

The stack synchronization entails synchronizing the internal states of the OPC UA PubSub stack from the primary publishing controller to the backup publisher, as illustrated in Figure 8.8. The synchronization allows the stack instance running on the backup to resume with the latest state of the primary stack instance. Specifically, sequence numbers need to be synchronized to the backup before transmitting the message. With this strategy, the backup publisher can continue publishing using the same internal state as the former primary's DataSetWriter and WriterGroup. Therefore, from the subscriber's perspective, the failover due to the failure is transparent. This approach's advantage is its transparency to the subscriber. The downside, however, is the need for synchronization support within the OPC UA PubSub stack's internal workings.

8.5.3 Standard Extension

As mentioned in Section 8.5.1, PROFINET achieves redundancy through parallel logical connections between controllers and devices, designating a primary connection for data exchange and monitoring others to prevent failures leading to undetected redundancy deterioration [23]. A redundancy layer manages these connections, seamlessly switching the primary



(a) Before failure.

(b) After failure.

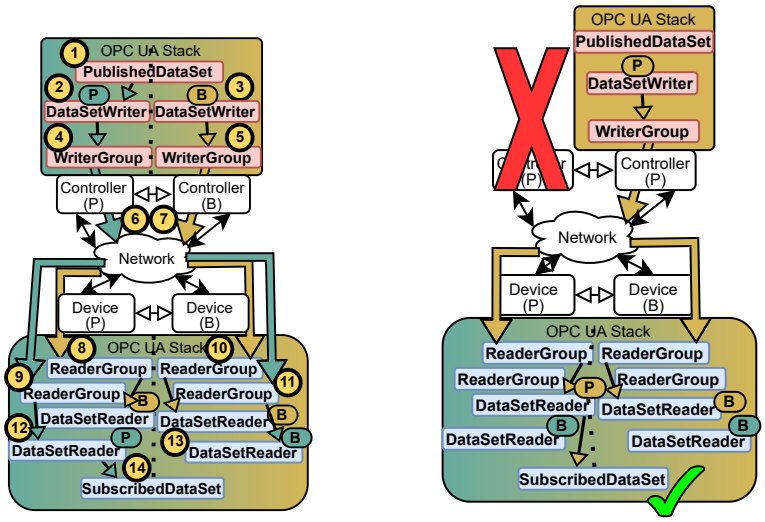
Figure 8.8: Stack synchronization in the publisher creates a seamless appearance, making the stack instances appear as one to the subscriber, hence depicted as a single entity. On the subscriber side, there's no necessity for internal stack state synchronization; thus, we depict it as two distinct instances.

connection as needed, thereby decoupling redundancy management from the application layer. The OPC UA PubSub standard could similarly incorporate a redundancy model like PROFINET. On a high abstraction level, this section presents one alternative to integrate similar redundancy features into OPC UA PubSub.

The extension includes (i) a RedundancyState field in the DataSetMessageHeader to indicate if the message is from a primary publisher and (ii) a redundancy state for DataSetWriter and DataSetReader. DataSetFlags2, as DataSetFlag1, indicates field presence. A bit in DataSetFlags2 will represent the presence of the RedundancyState field, with DataSetFlag1 indicating the presence of DataSetFlags2. The extension is detailed in Figure 8.9a.

The PublishedDataSet (1) represents the data set to be published, synchronized between the primary and backup. The primary DataSetWriter (2), labeled P, creates the DataSetMessage, setting the introduced RedundancyState field to primary in the DataSetMessageHeader. A backup DataSetWriter (3) in the backup controller also publishes, duplicating the data or sending a placeholder, with RedundancyState field set as backup.

The primary controller's WriterGroup (4) embeds the primary DataSetWriter's DataSetMessage into a NetworkMessage. Similarly, the backup controller's WriterGroup (5) encapsulates the DataSetMessage into



(a) Before failure.

(b) After failure.

Figure 8.9: Publisher and subscriber redundancy example with the described OPC UA PubSub standard extension.

a NetworkMessage. The primary and backup WriterGroups have distinct WriterGroupIds that ideally follow a convention that allows corresponding pairs to be identified.

The NetworkMessages are multicast (6-7), ensuring that both devices in the redundant pair receive all messages. Alternatively, additional WriterGroups using unicast could also ensure that both devices in the redundant pair receive all messages without using multicast.

ReaderGroups (8-11) on the redundant device receive NetworkMessages from both controllers, allowing connection monitoring and means to prevent undetected redundancy deterioration. The ReaderGroups forward DataSetMessages to the DataSetReaders (12-13). Each device has two ReaderGroups to handle messages from both publishing controllers in the redundant controller pair. Only the primary state DataSetReader updates the SubscribedDataSet (14), and only with DataSetMessages where the RedundancyState field equals primary.

If the primary publisher fails, the backup DataSetWriter becomes the primary, continuing to publish the PublishedDataSet, as shown in Figure 8.9b. The specifics of this transition, particularly changing the DataSetWriter's redundancy state, are likely implementation-dependent.

8.6 Experimental Evaluation

Section 8.4 looked at OPC UA PubSub in a redundancy context, identifying that publisher failovers aren't inherently transparent to subscribers, see Table 8.1. Consequently, Section 8.5 explored alternatives, identifying the synchronization of internal stack states as the approach that maintains standard compatibility without necessitating specialized handling by the subscriber. This section experimentally tests these findings, employing the multicast scenarios $\mathrm{PC_M}$ and $\mathrm{PD_M}$, using the transport protocol and UADP message configurations outlined in Section 8.3.2 and Figure 8.3.

For the experiment, we use the open source OPC UA stack open62541 (v1.4.0) [24, 25] running on Ubuntu 20.04.6 LTS using VMWare. One Virtual Machine (VM) acts as the publishing controller, and the other as the redundant subscriber device. We simulate publisher failure recovery (failover) by halting and restarting the publisher in the same VM and process. Further, we simulate subscriber failure and recovery by restarting the subscriber. The exchanged data consists of ten four-byte values the publisher publishes every 100 ms. The test implementation and modifications to the open62541 stack are available on GitHub [26].

8.6.1 Implementation

The open62541 PubSub implementation doesn't verify sequence numbers. It checks neither the Group header nor the DataSetMessage sequence numbers. To address this, we added checksum verification as per the standard. We added checking of the Group header sequence number, updated by the WriterGroup and checked by the ReaderGroup, and checking of the DataSetMessageHeader sequence number, updated by the DataSetWriter and verified by the DataSetReader. Messages with incorrect checksums are discarded.

For stack synchronization, we implemented a simple yet representative solution allowing a resumed instance to continue with the sequence number last used. This implementation is sufficient for our experiment, where we simulate the failure by stopping the publisher and resuming a new one in the same VM and process. For more details, refer to the implementation [26].

8.6.2 Experiment and Result

We conducted the experiments using the setup previously described and three different variants of the open62541 stack: (i) the original open62541 version - ORG, (ii) sequence number adherence - SEQ, and (iii) synchronization and sequence number adherence - SYNC. Table 8.2 displays the results, with OK

indicating a seamless recovery from the subscriber's perspective and FAIL indicating a non-seamless recovery.

The results, as detailed in Table 8.2, reflect the analysis from Section 8.4 highlighting the challenges with publisher failures. While the original open62541 version (ORG) shows OK for publisher failure scenarios ($\mathrm{PC_M}$), this is attributed to the stack's non-adherence to sequence numbering; it ignores them. In the SEQ variant, publisher recovery is not transparent to the subscriber; the new primary publisher uses sequence numbers perceived as outdated by the subscriber, leading to message rejection. Conversely, the SYNC variant enables the new primary publisher to resume with sequence numbers aligned with subscriber expectations, resulting in a successful, seamless recovery.

Table 8.2: Failure recovery result for different scenarios and stack variants.

Scenario	**Stack variant**		
	ORG	**SEQ**	**SYNC**
$\mathrm{PC_M}$	OK[1]	FAIL	OK
$\mathrm{PD_M}$	OK	OK	OK

8.7 Conclusion and Future Work

This work has examined OPC UA PubSub within the context of controller and device redundancy, focusing on the standard's recommended messaging configuration for real-time, cyclic data exchanges. The type of exchange that is typical in industrial settings. We explored four failure scenarios in a redundant controller and device setup using OPC UA PubSub, assessing the transparency of failovers, where the backup should take over seamlessly without impacting the application.

Our analysis revealed that the publisher redundancy is not transparent—highlighting a gap where the redundant publisher fails. We proposed three alternatives to address this in order to provide a seamless publisher failover. Further, we conducted experiments using the open62541 stack, implementing one of the suggested alternatives to validate our discussions. We conclude that achieving publisher redundancy in a way that is transparent to subscribers is feasible but requires stack support.

[1]OK due to the stack not adhering to the, by the standard, prescribed sequence number verification.

Future work includes adding general platform-independent redundancy support in open62541, as well as an in-depth evaluation and implementation of a redundancy layer similar to that of PROFINET, allowing for status monitoring to prevent redundancy deterioration from going undetected.

Bibliography

[1] Andrei Simion and Calin Bira. A review of redundancy in plc-based systems. *Advanced Topics in Optoelectronics, Microelectronics, and Nanotechnologies XI*, 12493:269–276, 2023.

[2] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993.

[3] Dietmar Bruckner, Richard Blair, M Stanica, A Ademaj, W Skeffington, D Kutscher, S Schriegel, R Wilmes, K Wachswender, L Leurs, et al. Opc ua tsn a new solution for industrial communication. *Whitepaper. Shaper Group*, 168:1–10, 2018.

[4] Opc 10000-14 - ua specification part 14: Pubsub 1.05.03. `https://reference.opcfoundation.org/Core/Part14/v105/docs/`. Accessed: 2025-05-05.

[5] Sten Grüner, Alexander E Gogolev, and Jens Heuschkel. Towards performance benchmarking of cyclic opc ua pubsub over tsn. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2022.

[6] Opc 10000-80 - uafx part 80: Overview and concepts 1.00. `https://reference.opcfoundation.org/UAFX/Part80/v100/docs/`. Accessed: 2025-05-05.

[7] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan 2004.

[8] Alberto Ballesteros, Manuel Barranco, Julián Proenza, Luís Almeida, Francisco Pozo, and Pere Palmer-Rodríguez. An infrastructure for enabling dynamic fault tolerance in highly-reliable adaptive distributed embedded systems based on switched ethernet. *Sensors*, 22(18):7099, 2022.

[9] Carlo Vitucci, Daniel Sundmark, Marcus Jägemar, Jakob Danielsson, Alf Larsson, and Thomas Nolte. Fault management impacts on the networking systems hardware design. In *IECON 2023-49th Annual Conference of the IEEE Industrial Electronics Society*, pages 1–8. IEEE, 2023.

[10] Mourad Nouioua, Philippe Fournier-Viger, Ganghuan He, Farid Nouioua, and Zhou Min. A survey of machine learning for network fault management. *Machine Learning and Data Mining for Emerging Trend in Cyber Dynamics: Theories and Applications*, pages 1–27, 2021.

[11] Inés Álvarez, Alberto Ballesteros, Manuel Barranco, David Gessner, Sinisa Djerasevic, and Julian Proenza. Fault tolerance in highly reliable ethernet-based industrial systems. *Proc. IEEE*, 107(6):977–1010, 2019.

[12] Peter Danielis, Jan Skodzik, Vlado Altmann, Eike Bjoern Schweissguth, Frank Golatowski, Dirk Timmermann, and Joerg Schacht. Survey on real-time communication via ethernet in industrial automation environments. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–8. IEEE, 2014.

[13] Michael Nast, Hannes Raddatz, Benjamin Rother, Frank Golatowski, and Dirk Timmermann. A survey and comparison of publish/subscribe protocols for the industrial internet of things (iiot). In *Proceedings of the 12th International Conference on the Internet of Things*, pages 193–200, 2022.

[14] Jacek Stój. Cost-effective hot-standby redundancy with synchronization using ethercat and real-time ethernet protocols. *IEEE Trans. on Autom. Science and Eng.*, 18(4):2035–2047, 2020.

[15] Rebekka Neumann, Christian von Arnim, Michael Neubauer, Armin Lechler, and Alexander Verl. Requirements and challenges in the configuration of a real-time node for opc ua publish-subscribe communication. In *2023 29th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*, pages 1–6. IEEE, 2023.

[16] Julius Pfrommer, Andreas Ebner, Siddharth Ravikumar, and Bhagath Karunakaran. Open source opc ua pubsub over tsn for realtime industrial communication. In *2018 IEEE 23rd international conference on emerging technologies and factory automation (ETFA)*, volume 1, pages 1087–1090. IEEE, 2018.

[17] Andreas Eckhardt and Sebastian Müller. Analysis of the round trip time of opc ua and tsn based peer-to-peer communication. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 161–167, 2019.

[18] Patrick Denzler, Thomas Frühwirth, Daniel Scheuchenstuhl, Martin Schoeberl, and Wolfgang Kastner. Timing analysis of tsn-enabled opc ua pubsub. In *2022 IEEE 18th International Conference on Factory Communication Systems (WFCS)*, pages 1–8. IEEE, 2022.

[19] Ahmed Ismail and Wolfgang Kastner. Coordinating redundant opc ua servers. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2017.

[20] Rafal Cupek, Kamil Folkert, Marcin Fojcik, Tomasz Klopot, and Grzegorz Polakow. Performance evaluation of redundant opc ua architecture for process control. *Transactions of the Institute of Measurement and Control*, 39(3):334–343, 2017.

[21] Opc 10000-1 - ua specification part 1: Overview and concepts. `https://reference.opcfoundation.org/Core/Part1/v105/docs/`. Accessed: 2025-05-05.

[22] Opc 10000-4 - ua specification part 4: Services 1.05.03. `https://reference.opcfoundation.org/Core/Part4/v105/docs/`. Accessed: 2025-05-05.

[23] High availability - guideline for profinet version 1.2 - date feburary 2020 - order no.: 7.242.

[24] Florian Palm, Sten Grüner, Julius Pfrommer, Markus Graube, and Leon Urbas. Open source as enabler for opc ua in industrial automation. In *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–6. IEEE, 2015.

[25] Open 62541 - project page. `https://www.open62541.org/`. Accessed: 2025-05-05.

[26] The experiment source code - fork of open62541. `https://github.com/Burne77a/open62541-red-inv`. Accessed: 2025-05-05.

Chapter 9

Paper D: Priority Based Ethernet Handling in Real-Time End System with Ethernet Controller Filtering

Bjarne Johansson, Mats Rågberger, Alessandro V. Papadopoulos, and Thomas Nolte.

In 48th Annual Conference of the Industrial Electronics Society (IECON), 2022.

Abstract

This work addresses the impact of best-effort traffic on network-dependent real-time functions in distributed control systems. Motivated by the increased Ethernet use in real-time dependent domains, such as the automation industry, a growth driven by Industry 4.0, interconnectivity desires, and data thirst. Ethernet allows different network-based functions to converge on one physical network infrastructure. In the automation domain, converged networks imply that functions with different criticality and real-time requirements coexist and share the same physical resources. The IEEE 60802 Time-Sensitive Networking profile for Industrial Automation targets the automation industry and addresses Ethernet network determinism on converged networks. However, the profile is still in the draft stage at the time of writing this paper. Meanwhile, Ethernet already provides attributes utilized by network equipment to prioritize time-critical communication. This paper shows that Ethernet Controller filtering with prioritized processing is a prominent solution for preserving real-time guarantees while supporting best-effort traffic. A solution capable of eliminating all best-effort traffic interference in the real-time application is exemplified and evaluated on a VxWorks system.

9.1 Introduction

Distributed Control Systems (DCS) are transcending into the Industry 4.0 era, where data and information are valuable optimization-enabling assets. Data collection requires connectivity and communication, pushing the automation industry towards network-centric solutions, implying that the network, to some extent, replaces the controller as the information center of the control system. Following in the tracks of network-centric systems are increased interest for interconnectivity and interoperability, key concepts in the Open Process Automation™ Standard[1] (O-PAS). O-PAS prescribes OPC UA[2] as the interoperable communication standard.

Upper bound end-to-end communication time of real-time traffic is a challenge for converged Operation Technology (OT) Ethernet networks when competing for network resources against low-priority, best-effort traffic induced by noncritical functions. A challenge addressed by Time Sensitive Networking (TSN) amendments to the IEEE 802.1Q Ethernet networking standard. The IEEE 802.1Qbv amendment for Scheduled Traffic (TSN-ST) brings forth short and bounded end-to-end communication time on converged networks using time-scheduled communication. TSN offers solutions but is still in the early stages of industry adoption. For example, the Industrial Automation IEEE 60802 TSN profile[3] is at the time of writing this paper still in the draft stage, and the support in industrial network equipment is scarce [1].

Ethernet networking does not require TSN to provide Quality of Service (QoS). The Priority Code Point (PCP), introduced in the late '90s in the IEEE 802.1D-1998 and later incorporated in IEEE 802.1Q, already offers that. PCP provides priority information utilizable by the OSI layer two network infrastructure (i.e., switches) to determine forwarding precedence.

In this paper, we address the challenge of preserving the correctness of network-dependent real-time functions in end systems when coexisting with non-real-time programs reliant on best-effort traffic. We illustrate the problem with a simulated DCN application running on VxWorks. A DCN application with OPC UA PubSub-based real-time communication dependencies. We identify prioritized frame processing aided by Ethernet Controller filtering as a prominent solution, which is the paper's contribution, together with the verification of the solution.

The paper is organized as follows. Section 9.2 presents the related work. Section 9.3 describes the filtered enabled prioritized frame processing, fol-

[1] `https://publications.opengroup.org/p190`
[2] `https://opcfoundation.org/`
[3] `https://1.ieee802.org/tsn/iec-ieee-60802/`

lowed by a quantitative evaluation in Section 9.4, and a discussion in Section 9.5. Section 9.6 outlines conclusions and future work.

9.2 Related Work

TSN combined with OPC UA is identified by Bruckner et al. [2] as the future of communication in the automation domain. TSN consists of multiple amendments to IEEE 802.1 [3] and Lo Bello et al. [4] provide an overview and discuss open issues from an automation mindset, where configuration ease is one of the highlighted challenges. As mentioned in the introduction and shown, for example, by Zhao et al. [5], TSN-ST provides low latency, deterministic end-to-end communication, but the scheduling problem is not a trivial problem [6, 7]. Hallmans et al. [1] highlight that industry TSN adoption is still low, and the support provided by industrial network equipment is still scarce, motivating why we do not consider TSN in the end system handling in this work.

Already in 2002, after the introduction of PCP, end system handling of prioritized traffic was addressed by Skeie et al. [8] with software-based priority queue to reduce latency, and the need for QoS in the end system highlighted by Thyrbom et al. [9]. Since then, many studies have been made on network stack performance on Linux. For example, Larsen et al. [10] studied TCP latency on Linux in a data center environment and found that the latency is around 10μs in point-to-point communication.

Beifuß et al. [11] measures latency and constructs a model to predict latency in the Linux network stack to find optimization knobs to turn. Describing four main optimization points: (i) copying between user- and kernel space, (ii) usage of preallocated buffer, (iii) polling to reduce interrupt frequency, and (iv) processing of batches instead of single frames. The performance study performed by Ramneek et al. [12] reaches a similar conclusion: buffer allocation and interrupt handling can be expensive in terms of CPU usage.

Priority inversion due to processing of low priority packets with high priority can potentially impact high priority real-time execution badly [13]. A challenge addressed by Lee et al. [14] and further improved by Blumschein et al. [15]. Both use an early software demuxer to classify packages and prioritize the handling of the incoming packet according to the priority of the receiving task. To further aid that approach, Behnke et al. propose a multi-queue network interface [16]. In contrast, our work focuses on unmodified network stacks and presents Ethernet Controller enabled filtering to enable processing priority matching the received frame's QoS.

9.3 Prioritization Filtering

The Ethernet Controller (EC) and the Media Access Controller (MAC) handles, with the Physical Layer (PHY), the transmission and reception of Ethernet Frames. When a frame is received, the EC stores the frame in a RxQueue and raises an interrupt to the CPU, and the OS Network stack process the frame further. However, as pointed out by earlier work, the network stacks do not, by default, treat the incoming frames according to priority, which may result in priority inversion or latency [14, 16].

ECs provide filtering options and the possibility to direct traffic to different Rx queues based on those filtering options. In this section, we base our EC examples on Intel I211[4], which has PCP filtering capabilities and two Rx queues. Other ECs, such as Intel I350, have eight Rx queues, allowing even better filtering granularity. Different ECs also support different filtering possibilities, from Ethernet header information to filtering on the higher protocol-layer information, typically IP-header information. We denote the configurable properties on which to take the filter decision, the filtering property (FP).

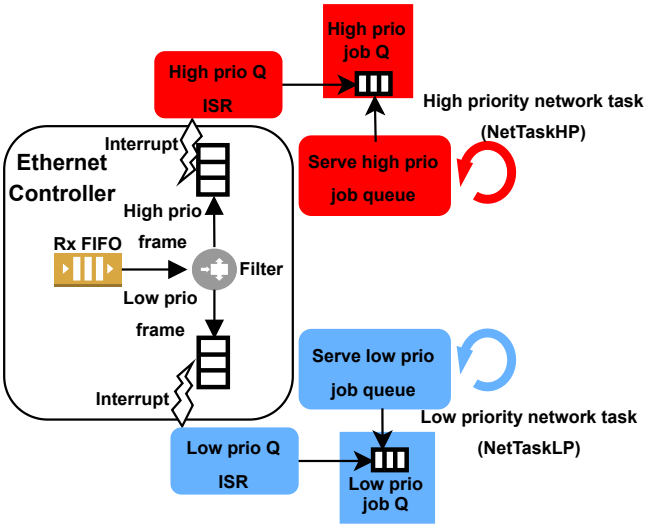


Figure 9.1: Priority based filtering - an example with two priorities, high and low.

[4] https://cdrdv2.intel.com/v1/dl/getContent/333017

Figure 9.1 summarizes the idea, elaborated next. Based on the FP, the EC determines the frame priority. The EC queue high priority frames on the high priority Rx queue and raises the corresponding interrupt. The high priority ISR post the request to serve the incoming frame to the high priority job queue served by the high priority network task. Low-priority frames are handled similarly but follow the low-priority path. If the low priority task cannot read frames from the low priority Rx queue faster than they arrive, the low priority queue will eventually become full. It is then essential that the EC drop low priority frames that can't fit into the low priority queue to avoid filling the RxFIFO and causing high priority frame drops.

The principle described can also handle multiple network interfaces, where the filtering can ensure that the most suitable network task, priority vise, processes incoming frames.

9.3.1 QoS Prioritization Filtering on VxWorks

VxWorks[5] is a widely used and well-known commercial RTOS that has been around for more than 35 years and installed more than two billion times. The solution was also tested on Linux but left out due to page limitations.

Enabling priority packet filtering in VxWorks requires two things, enabling driver support and providing two network tasks with different priorities. The driver support added consists of (i) support for multiple Rx queues and (ii) filter configuration support. The filtering configuration of the EC also configures the EC to drop frames if the destination Rx Queue is full to prevent the RxFifo, from filling up and causing high priority frame drop due to the Rx Queue for low priority frames being full.

Two network tasks with different priorities are configured, denoted $NetTaskHP$ and $NetTaskLP$. The $NetTaskHP$ is the high-priority network task, given a priority of 20. The $NetTaskLP$ is the low priority network task and has priority 50. The priority values given here are the ones used in the evaluation system, described in section 9.4.3.

9.4 Quantitative Evaluation

9.4.1 Evaluation Setup

The evaluation setup consists of the four nodes, listed in Table 9.1. Table 9.2 lists the software. C1 runs the Evaluation Application (EA), explained in Section 9.4.3. C3 and C4 are nodes addressing C1 with low-priority traffic. The

[5] `https://www.windriver.com/products/vxworks`

Table 9.1: Hardware used.

Node	Hardware	Ethernet
C1	MSI Intel 2.4GHz I3-7100U 256GB RAM	EP1 I211 EP2 I219
C2	Lenovo Mini PC 2GHz Intel I7 I7-9700T 16 GB RAM	EP1 I219
C3,C4	Raspberry Pi 4B 1.5GHz ARM Cortex A72	EP1 Broadcom 2711
Switch	Zyxel GS1900-8	10 Gbps

high-priority data exchange between C1 and C2 is brokerless OPC UA PubSub over UDP[6]. The Switch is an OSI level two Ethernet switch configured to give precedence based on the PCP field in the Ethernet frame. The network consists of two virtual local area networks (VLAN), with VLAN ID (VID) 1 and 2, see Figure 9.2.

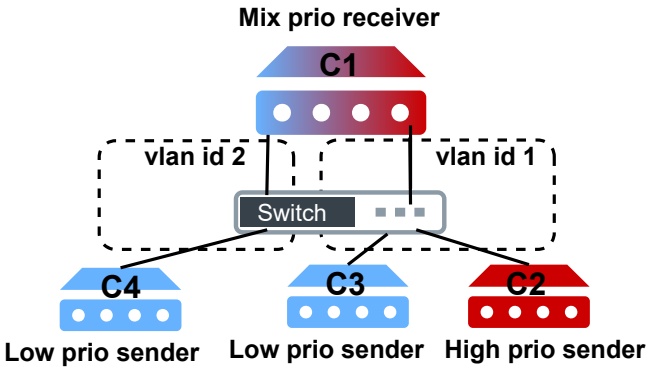


Figure 9.2: Evaluation setup topology.

9.4.2 Network Configuration

C1 has two ECs connected to two RJ45 ports. An I211 is the Ethernet Port 1 (EP1) EC, and an I219 manages Ethernet Port 2 (EP2). EP1 connects to VID 1 and EP2 to VID 2. The OPC UA PubSub communication from C2 is the prioritized traffic. Ethernet frames carrying the PubSub UDP frames have the

[6] https://reference.opcfoundation.org/v105/Core/docs/Part14/

Table 9.2: Software versions used.

Name	Version	Comment
VxWorks	21.07	OS C1 and C2
Raspberry Pi OS	10	OS on C3 and C4
Open62541	1.0.1	OPC UA PubSub stack on C1 and C2

PCP field set to six, and the low priority traffic frames have PCP set to zero. The links are 1 Gbps full-duplex.

9.4.3 Evaluation Application

The EA consists of two applications/subsystems, the real-time and non-real-time applications. The real-time application simulates a control application concerning CPU load, determinism, and dependency on input values.

The C1 CPU has four cores $\{P_1, P_2, P_3, P_4\}$. Each real-time task τ_i is a 4-tuple $\langle C_i, T_i, P_i, A_i \rangle$. C_i is the worst-case execution time, T_i is the period (and deadline), i.e., shortest inter-release time, P_i is the priority, where a lower value is a higher priority, and A_i is core affinity, $A_i \in \{0, P_1, P_2, P_3, P_4\}$ and $A_i = 0$ means no affinity, i.e., the task can be scheduled on all four cores.

The real-time application consists of two sets of tasks, $HP = \{\tau_1^{HP}, \tau_2^{HP}, \tau_3^{HP}, \tau_4^{HP}\}$ and $MP = \{\tau_1^{MP}, \tau_2^{MP}, \tau_3^{MP}, \tau_4^{MP}\}$. Each set contains as many tasks as there are cores, that is, four. *HP* contain the high priority tasks and *MP* the medium priority tasks.
$\forall_{\tau_i^{HP}} \forall_{\tau_i^{MP}}, \tau_i^{HP} \in HP, \tau_i^{MP} \in MP | \tau_i^{HP}.P < \tau_i^{MP}.P$

The application has two multiprocessor using modes, Partitioned Scheduling (PS) mode, where the real-time tasks are pinned to a specific core, using the task affinity property $\tau.A$.
$\forall_{\tau_i^{HP}} \forall_{\tau_i^{MP}}, \tau_i^{HP} \in HP, \tau_i^{MP} \in MP, i \in 1, \ldots, 4 | \tau_i^{HP}.A = P_i, \tau_i^{MP}.A = P_i$
The other mode is Global Scheduling (GS), where the scheduler is free to schedule the real-time tasks on all cores.
$\forall_{\tau_i^{HP}} \forall_{\tau_i^{MP}}, \tau_i^{HP} \in HP, \tau^{MP} \in MP, i \in 1, \ldots, 4 | \tau_i^{HP}.A = 0, \tau_i^{MP}.A = 0$
Figure 9.3 gives a conceptual overview of the EA and system.

At each invocation, the *HP* task requires an updated value from C2 and consumes all previous values received. Values are exchanged with OPC UA PubSub, C2 is the publisher, and C1 is the subscriber. Each of the four sender tasks in C2 publishes an updated value every 5th ms. In C1, for each τ_i^{HP}, there is an event-driven OPC UA PubSub Subscriber task that shares the affinity and priority of the τ_i^{HP}, that stores the received values in a FIFO. A FIFO read by

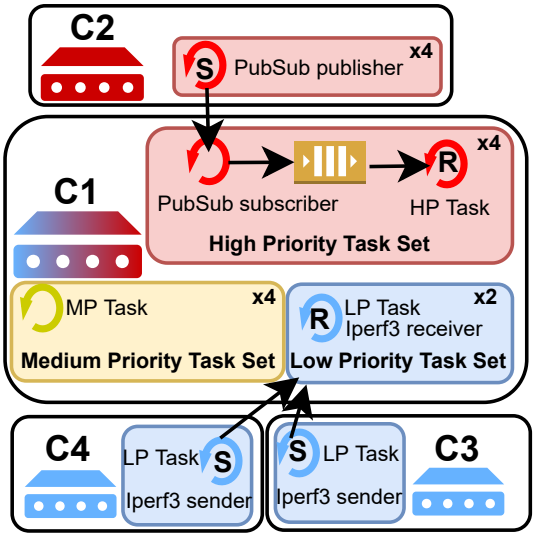


Figure 9.3: Conceptual view of the evaluation application.

τ_i^{HP}, shown in Figure 9.3.

The *MP* task does not have network dependency. An analogy is an application dependent on local I/O values. The *MP* tasks have lower priority since they can have a longer execution time than the *HP* task, allowing the *HP* task to preempt the *MP* tasks. Table 9.3 shows priorities and execution times, elaborated further in Section 9.4.4.1.

Table 9.3: Task parameters.

Name	Priority (P)	Period (T)	Exec. time (C)	CPU utilization
HP	20	10ms	1ms	10%
MP	40(HN), 50(EN) 60(LN)	20ms	0-16ms	0-80%
LP	100	Event driven	Comm. dep.	Comm. dep.
NetTask	50	Event driven	Comm. dep.	Net. dep.

The Low Priority (*LP*) application and tasks represent non-time-critical applications, dependent on data produced in C3 and C4. For example, reception of system maintenance files as preparation for a system upgrade, application change, and less critical process values. We use `Iperf3` [7] to emulate this

[7] `https://iperf.fr/`

application.

9.4.4 Evaluation Variants

By using different priorities, execution times, and best-effort traffic, we measure the correctness of the EA in terms of deadline misses (overruns) and missed high-priority values updates from C2.

9.4.4.1 Execution Time

The *HP* tasks have a fixed execution time of one ms. We use different execution times for the *MP* tasks for two reasons. Firstly, to observe how *MP*.C impacts the NetTask and the *HP* dependencies on received values from C2. Secondly, the NetTask interference on *MP* when *MP*.C increases. Table 9.3 show the *MP* tasks C range and the corresponding utilization. Note that the execution time, C, specified is the CPU time the task requires to complete, i.e., the Worst-Case Execution Time (WCET) and execution time are always the same.

The *HP* and *MP* CPU utilization range is between 10% and 90%. OPC UA PubSub subscriber adds approximately four percent, in addition to the time shown in Table 9.3. Figure 9.4 shows that it is not possible to schedule *MP* tasks with $C = 16$ms and $T = 20$ms combined with *HP* tasks with $C = 1$ms and $T = 10$ms with the interference of a high priority NetTask, the NetTaskHP, and the PubSub task. The execution times illustrated for the PubSub and NetTaskHP are likely higher than in reality, but other high-priority executions are left out, such as scheduling overhead. A WCET analysis would give us the exact limits, but that is beyond the scope of this paper. The above is the motivation behind the upper limit of *MP*.$C = 16$ms; it is over the limit.

9.4.4.2 Priority

As shown in Table 9.3 we use three priority levels for *MP*. These are Higher than NetTask (HN) with priority 40, Equal to the NetTask (EN) with priority 50, and Lower than the NetTask (LN) with priority 60. The three levels are selected to show the interference between NetTask and real-time application tasks. *MP* priority HN, can cause *MP* execution to block network handling and delay the values communicated to *HP*. *MP* priority EN, can cause the *MP* execution time to affect the network handling, similar to HN, since VxWorks, by default, will not preempt the same priority. Finally, when *MP* is lower in priority than the NetTask, LN, the execution of the *MP* will not interfere with the network communication. However, network handling can block *MP* and cause *MP* to overrun.

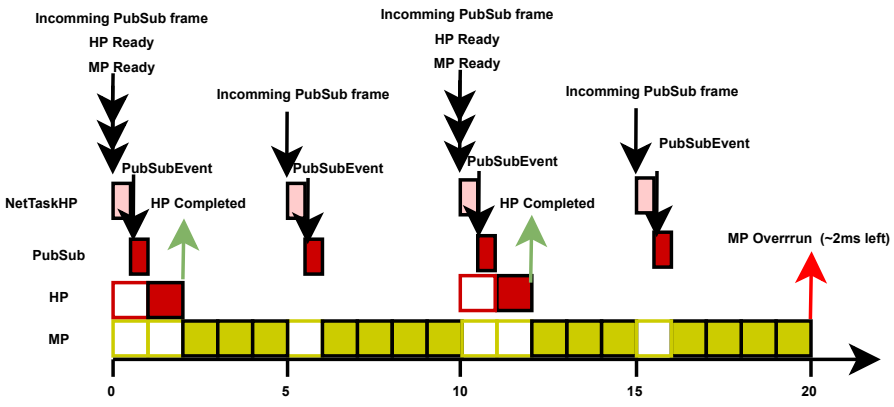


Figure 9.4: Scheduling example of a *HP* and *MP* task.

9.4.4.3 Network Traffic

The high priority network traffic consist of the variable exchange over OPC UA PubSub from C2 to C1. The different types of low priority, best-effort traffic are summarized in Table 9.4. With No *LP* we mean no low-priority

Table 9.4: Low priority network communication types.

Abbreviation	Sender	Protocol	Bandwidth
No LP	None	-	0
1 TCP	C3	TCP	$<$ 1Gbps
2 TCP	C3,C4	TCP	$<$ 1Gbps
1 UDP	C3	UDP	$<$ 400Mbps
2 UDP	C3,C4	UDP	$<$ 400Mbps

traffic. The 1 Gbps link and the receiver processing possibilities limit the TCP bandwidth utilization. The UDP max bandwidth utilization is limited to 400 Mbps, about 34 frames per millisecond and 40% of the available bandwidth.

9.4.4.4 Scheduling Variants

The VxWorks scheduler is a priority-based preemptive scheduler that, by default, uses GS and does not pin tasks to cores, with some exceptions, such as the NetTask, that have a core affinity for performance reasons. The default affinity for the NetTask is P_1. The EA can run in two scheduling modes, PS and GS. In PS mode $\tau_1^{HP}.A = P_1, \tau_1^{MP}.A = P_1$, i.e., τ_1^{HP} and τ_1^{MP} are pinned to

the core P_1, the same core as the NetTask. Their uncompromising coexistence with the NetTask on the core P_1 makes the EA in the PS mode more likely to encounter failures faster due to overuse of P_1. The affinity to P_1 prevents those tasks from utilizing the potentially available CPU time on other cores.

9.4.5 VxWorks results – no Filtering

This section presents the results of running the variants discussed without prioritization filtering. The data comes from one-minute per variant runs. Fig-

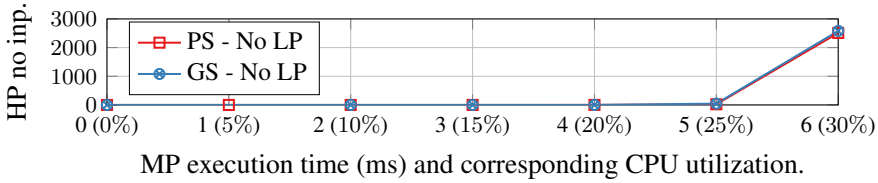


Figure 9.5: *MP* tasks with higher priority than NetTask (HN) result in *HP* cycles without input updates.

ure 9.5 shows that *MP* with HN priority, cause *HP* to lack input data when *MP*.C increases. *MP* execution blocks the NetTask, and C2 PubSub values do not reach *HP* during *MP*.C.

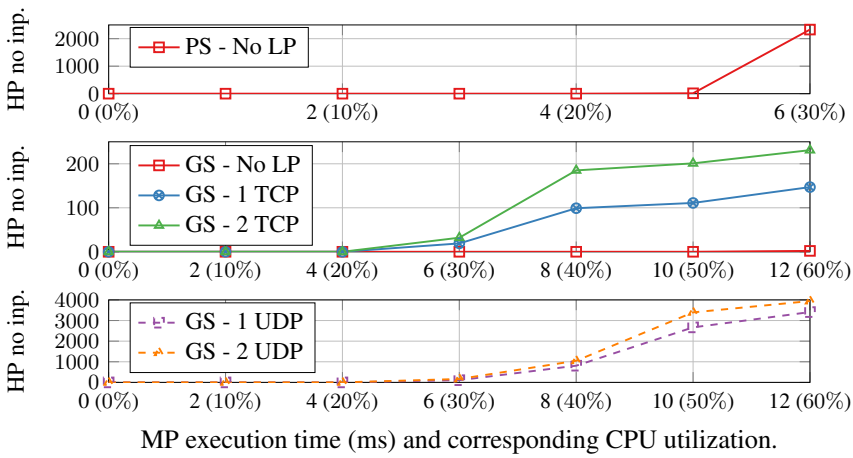


Figure 9.6: *MP* tasks with equal priority to the NetTask (HN) result in *HP* cycles without input updates.

Figure 9.6 shows the result when *MP* priority EN. The result follows the same pattern for EA in PS mode as for *MP* priority HN. For EA in GS mode,

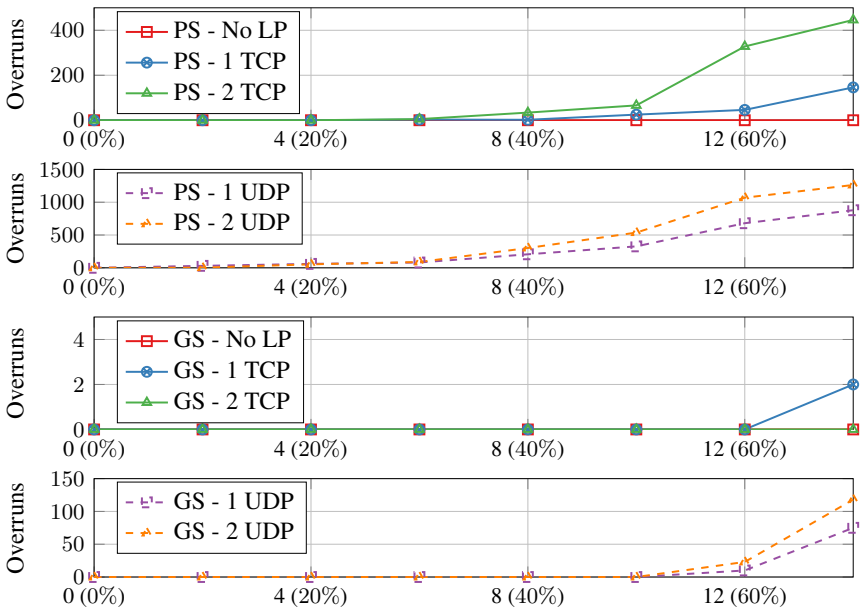


Figure 9.7: *MP* lower priority than NetTask - *MP* misses deadlines (overruns).

HP gets the data each cycle without low priority traffic. A *MP* task with priority EN, equal to NetTask priority, won't preempt and block the NetTask; the scheduler migrates the *MP* task to another available core if any. With low-priority traffic, *HP* lacks updates when $MP.C$ increases.

With *MP* priority LN, $MP.C$ does not affect *HP* reception of values. However, NetTask execution can prolong the *MP* response time. Figure 9.7 shows when *MP* start to miss deadlines and overruns for the different types of low priority, best-effort traffic.

Again, we see that the EA in PS mode fails before the EA in GS mode. Notable is also that the UDP traffic is more of a challenge than TCP most likely due to the TCP flow control easing the burden on the receiver.

Figure 9.8 shows the average time the *MP* tasks are in the ready state. Ready is the state a VxWorks task is in when it is ready to execute, but the CPU is busy executing higher priority tasks/interrupts. The NetTask interference on the *MP* task is higher in PS mode but limited to the *MP* task that shares the core with NetTask. The *MP* average ready time in PS mode when receiving low-priority TCP traffic is higher for lower $MP.C$. Potential due to *LP* tasks (`iperf3`) getting more execution time, resulting in a larger TCP flow control window. Except for that TCP variant, UDP traffic causes the highest

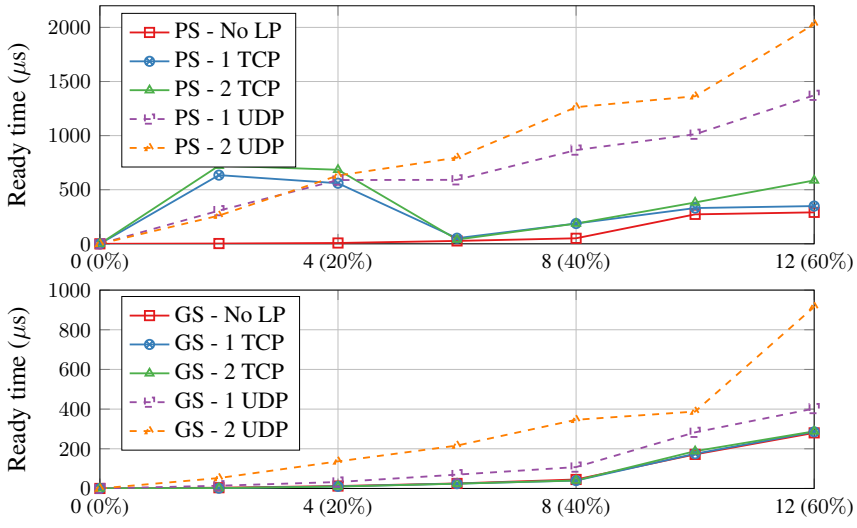


Figure 9.8: Average time *MP* tasks are in ready state - blocked for execution by higher priority tasks. The difference between No *LP* and the graphs for *LP* traffic illustrates the *LP* traffic impact on *MP*.

interference on the *MP* tasks.

9.4.6 Evaluation System – with Filtering

Table 9.5: Task parameters.

Name	Priority (P)	Period (T)	Exec. time (C)	CPU utilization
HP	20	10ms	1ms	10%
MP	40	20ms	0-16ms	0-80%
LP	100	Event driven	Event driven	Comm. dep.
NetTaskHP	20	Event driven	HP Comm. dep.	HP comm. dep.
NetTaskLP	50	Event driven	LP Comm. dep.	LP comm. dep.

We apply the prioritization filtering mechanism described in Section 9.3.1 on VxWorks running in C1. Table 9.5 shows the task priorities when using filtering. NetTaskHP handles the high-priority traffic and NetTaskLP processes the low-priority traffic.

9.4.7 VxWorks Result – with Filtering

NetTaskHP handles the high-priority network traffic, the data that *HP* tasks depend on, and NetTaskHP has a higher priority than the *MP* tasks. Hence *MP* execution does not cause lost / late inputs for *HP*.

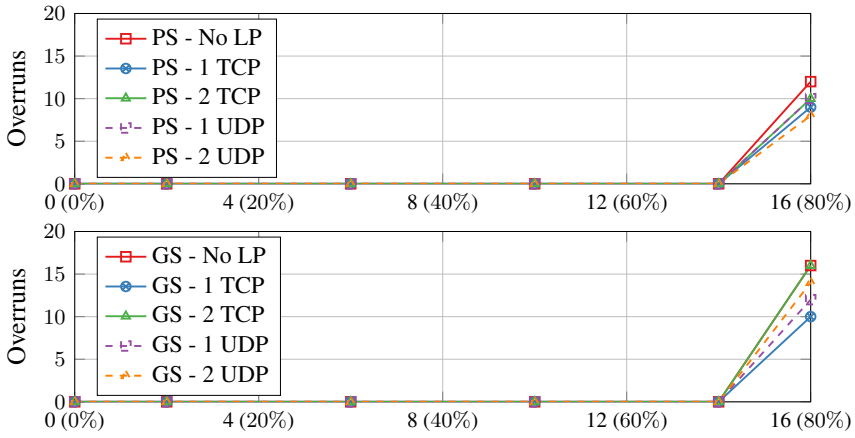


Figure 9.9: *MP* deadline misses with packet filtering enabled.

Figure 9.9 shows that with priority filtering of the incoming frames, the *MP* deadline missing limit is higher; it now occurs at 16ms (80% utilization), the upper limit of what is feasible. Figure 9.10 shows that the time the *MP* task is in a ready state is not affected by the low priority traffic. Hence, with the help of filtering in the EC, a network-dependent real-time application can be free from interference from less critical, best effort, and low priority network traffic.

9.5 Discussion

Section 9.4.7 shows that prioritization filtering can eliminate best-effort traffic impact on the real-time functions. Even though the EA is a simulated application designed to show the priority inversion problem that emerges when handling incoming traffic with different criticality, end-systems on converged networks benefit from eliminating the priority inversion problem. The elimination of priority inversion due to best-effort traffic interference on critical real-time task increase the dependability of the system. How much depends on properties like CPU utilization, communication patterns, etc., domain and

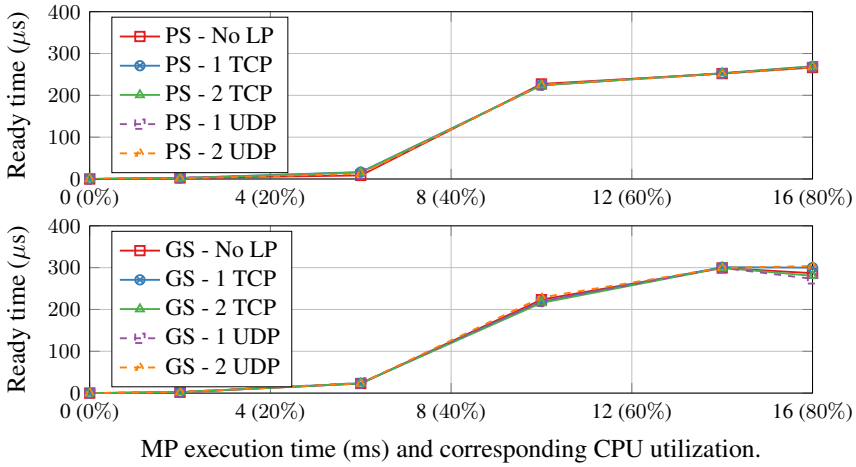


Figure 9.10: Average pended time for all medium priority tasks - with packet filtering.

solution-specific properties. Other potential benefits are reduced latency and decreased probability of dropping high priority frames due to full queues.

High NetTaskHP priority poses a potential risk. For example, a Denial of Service (DoS) attacker could generate high-priority traffic that starves out other high-priority execution. However, if real-time functions are network-dependent, network handling is likely to have high priority. If that is the case, filtering might reduce the DoS attack surface since prioritization directs low priority, best-effort traffic to lower priority processing. A potential hardening strategy could be to limit the nodes trusted for high-priority processing and filter not only on a QoS property but also on node identities. Such as MAC- or IP-addresses. However, cybersecurity is a vast topic on its own. We realize that the challenges and potential future work could further evaluate how to use the Ethernet Controllers for security purposes.

The traffic load used can be discussed; 400 Mbps UDP traffic might be much. However, consider that we only used two Ethernet ports and two clients. A modern IPC, such as the APC 910 from B&R[8], has support for six and more one Gbps ports served by a similar CPU as the one in C1, a quad-core Intel I3.

[8] https://www.br-automation.com/en/products/industrial-pcs/automation-pc-910/

9.6 Conclusion and Future Work

In this paper, we identified hardware-aided filtering of incoming frames to network processing with appropriate priority as a prominent solution. We described the steps needed to realize priority processing filtering on VxWorks. Finally, we evaluated the solution on VxWorks using a simulated controller application consisting of several real-time and non-real-time tasks with different priorities and network dependencies. The results show that prioritization filtering eliminates the best-effort traffic impact on the application's real-time functionality.

Relevant future work is to evaluate this approach on Linux combined with virtualization and prioritization handling in converged virtual networks. Another natural extension of this work is to take a holistic approach that incorporates outgoing traffic since outgoing traffic also requires network task processing.

Bibliography

[1] Daniel Hallmans, Mohammad Ashjaei, and Thomas Nolte. Analysis of the TSN standards for utilization in long-life industrial distributed control systems. In *IEEE Int. Conf. Emerg. Tech. & Fact. Autom. (ETFA)*, pages 190–197, 2020.

[2] Dietmar Bruckner, Marius-Petru Stănică, Richard Blair, Sebastian Schriegel, Stephan Kehrer, Maik Seewald, and Thilo Sauter. An introduction to OPC UA TSN for industrial communication systems. *Proc. IEEE*, 107(6):1121–1131, 2019.

[3] Norman Finn. Introduction to time-sensitive networking. *IEEE Comm. Stand. Mag.*, 2(2):22–28, 2018.

[4] Lucia Lo Bello and Wilfried Steiner. A perspective on IEEE time-sensitive networking for industrial communication and automation systems. *Proc. IEEE*, 107(6):1094–1120, 2019.

[5] Luxi Zhao, Paul Pop, and Silviu S. Craciunas. Worst-case latency analysis for IEEE 802.1Qbv time sensitive networks using network calculus. *IEEE Access*, 6:41803–41815, 2018.

[6] Silviu S Craciunas, Ramon Serna Oliver, Martin Chmelík, and Wilfried Steiner. Scheduling real-time communication in IEEE 802.1 Qbv time

sensitive networks. In *Int. Conf. Real-Time Networks and Systems*, pages 183–192, 2016.

[7] Paul Pop, Michael Lander Raagaard, Marina Gutierrez, and Wilfried Steiner. Enabling fog computing for industrial automation through time-sensitive networking (tsn). *IEEE Comm. Stand. Mag.*, 2(2):55–61, 2018.

[8] T. Skeie, S. Johannessen, and O. Holmeide. The road to an end-to-end deterministic ethernet. In *IEEE Int. Workshop on Factory Communication Systems*, pages 3–9, 2002.

[9] Linus Thrybom and Gunnar Prytz. QoS in switched industrial ethernet. In *IEEE Conf. Emerg. Tech. & Fact. Autom. (ETFA)*, pages 1–8, 2009.

[10] Steen Larsen, Parthasarathy Sarangam, Ram Huggahalli, and Siddharth Kulkarni. Architectural breakdown of end-to-end latency in a TCP/IP network. *Int. journal of parallel programming*, 37(6):556–571, 2009.

[11] Alexander Beifuß, Daniel Raumer, Paul Emmerich, Torsten M Runge, Florian Wohlfart, Bernd E Wolfinger, and Georg Carle. A study of networking software induced latency. In *Int. Conf. and Workshops on Networked Systems (NetSys)*, pages 1–8, 2015.

[12] Ramneek, Seung-Jun Cha, Seung Hyub Jeon, Yeon Jeong Jeong, Jin Mee Kim, and Sungin Jung. Analysis of Linux kernel packet processing on manycore systems. In *IEEE Region 10 Conf. TENCON*, pages 2276–2280, 2018.

[13] Ilja Behnke, Lukas Pirl, Lauritz Thamsen, Robert Danicki, Andreas Polze, and Odej Kao. Interrupting real-time iot tasks: How bad can it be to connect your critical embedded system to the internet? In *2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC)*, pages 1–6, 2020.

[14] Minsub Lee, Hyosu Kim, and Insik Shin. Priority-based network interrupt scheduling for predictable real-time support. *Journal of Computing Science and Engineering*, 9(2):108–117, 2015.

[15] Christoph Blumschein, Ilja Behnke, Lauritz Thamsen, and Odej Kao. Differentiating network flows for priority-aware scheduling of incoming packets in real-time iot systems. In *25th IEEE International Symposium on Real-Time Distributed Computing (ISORC)*, 2022.

[16] Ilja Behnke, Philipp Wiesner, Robert Danicki, and Lauritz Thamsen. A priority-aware multiqueue nic design. In *In Proceedings ofthe 35th Annual ACM Symposium on Applied Computing (SAC)*, 2022.

Chapter 10

Paper E: Partible State Replication for Industrial Controller Redundancy

Bjarne Johansson, Olof Holmgren, Alessandro V. Papadopoulos, and Thomas Nolte.

In 25th IEEE International Conference on Industrial Technology (ICIT), 2024.

Abstract

Distributed control systems are part of the often invisible backbone of modern society that provides utility services like water and electricity. Their uninterrupted operation is vital, and unplanned stops due to failure can be expensive. Critical devices, like controllers, are often duplicated to minimize the service stop probability, with a secondary controller acting as a backup to the primary. A seamless takeover requires that the backup has the primary's latest state, i.e., the primary has to replicate its state to the backup. While this method ensures high availability, it can be costly due to hardware doubling. This work proposes a state replication solution that doesn't require the backup to store the primary state, separating state storage from the backup function. Our replication approach allows for more flexible controller redundancy deployments since one controller can be a backup for multiple primaries without being saturated by state replication data. Our main contribution is the partible state replication approach, realized with a distributed architecture utilizing a consensus algorithm. A partial connectivity-tolerant consensus algorithm is also an additional contribution.

10.1 Introduction

Distributed Control Systems (DCS) are the backbone of many large-scale automation solutions, especially in critical domains where unplanned downtime can have significant financial and operational repercussions. Central to these systems are controllers with redundancy mechanisms that minimize the risk of unplanned downtime. Commonly, this redundancy is achieved through hardware duplication, where one controller operates as the active primary and another as a standby backup, ready to take over in case of a primary failure. In a DCS setting, controllers are often termed Distributed Controller Nodes (DCN), a term interchangeable with 'controller' in this paper.

With the advent of Industry 4.0, there has been a notable transition from specialized fieldbuses to more flexible, networked solutions, enhancing system interconnectivity. Network-based architecture enables flexible redundancy schemes, such as one backup for multiple primaries, a redundancy pattern that increases fault tolerance with a reduced hardware footprint. However, seamless backup takeover requires state replication from the primaries, a task the backup's bandwidth could limit.

Central to the DCS is the DCN-driven control application, which manages the physical process's state. The application samples the process state by reading values from input I/O connected to sensors and determines appropriate actions based on these samples. These actions then dictate the output values sent to the output I/O, interfacing with the real-world process. Figure 10.1 shows this sequence—often described as 'copy-in, execute, and copy-out.'

As mentioned, a primary DCN replicates the redundant DCN control application state to the backup. The application state data size depends on the application and can vary between a few bytes to many megabytes.

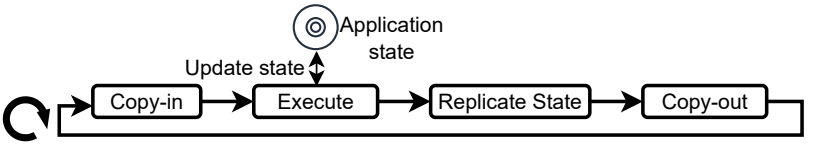


Figure 10.1: Typical control application task execution steps.

The latest state is needed to resume the operation of an application seamlessly. Figure 10.2 depicts a redundancy deployment with multiple primaries and one backup using a naive state replication where all applications replicate their state to a single backup (DCN 5). The network capacity of the backup in terms of bandwidth becomes a potential bottleneck in such a deployment.

The required bandwidth is the replicated state data size multiplied by repli-

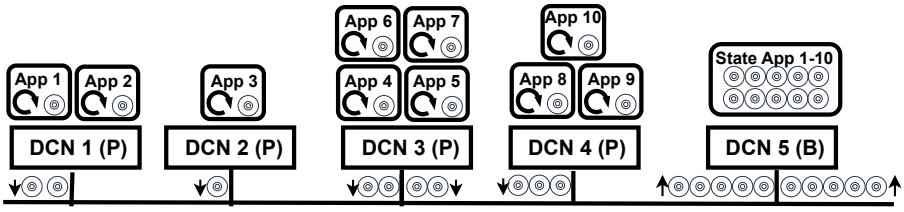


Figure 10.2: Naive state replication approach - the backup stores every application's state.

cation frequency. If A is the set of all redundant applications, sds_i is the size of the state data, and rt_i is the allowed state replication time for application i and $i \in A$. Then, Eq. 10.1 shows the total bandwidth required, bw, to replicate the states. If bw is larger than the bandwidth the backup provides, the puzzle is not solvable with the naive approach.

$$bw = \sum_{i=1}^{|A|} sds_i \left(\frac{1}{rt_i} \right) \tag{10.1}$$

This paper aims to answer the question of how the aggregate state replication data sent to the backup DCN can be reduced. By reducing this traffic, the risk of redundancy arrangements being limited due to backup network resources will also decrease. To address this challenge, the paper proposes a new method called Partible State Replication (PSR), which separates the backup role from the recipient of primary application states. With PSR, states can be replicated to any DCN, serving as our direct contribution. Further enhancing fault tolerance, we decentralize storage allocation handling for PSR, leading to our secondary contribution: a consensus protocol targeting a DCN cluster.

The paper is organized as follows: Section 10.2 reviews related work. Section 10.3 provides an overview of PSR, elaborated upon in Section 10.4. The consensus protocol is detailed in Section 10.5, while Section 10.6 presents our implementation and evaluation findings. Finally, we conclude and discuss future work in Section 10.7.

10.2 Related Work

Passive standby redundancy is the prevailing DCN redundancy mechanism [1, 2, 3]. Prior research has explored diverse DCN redundancy concepts, including cloud-hosted redundant controllers, orchestrator utilization, and architectures

centered on forming redundant solutions from non-redundant Commercial Off-The-Shelf (COTS) Programmable Logic Controllers (PLC) [4, 5, 6].

Achieving standby redundancy via hardware duplication is costly, especially with high-end DCNs designed for redundancy [6]. In contrast to the related work mentioned above, we propose a partible synchronization between primary and backup to enable a cost-effective redundancy. This partible approach entails segregating state storage from the backup role, further detailed in Section 10.3.

The data replication research landscape is vast; examples include deduplication and placement strategies [7, 8, 9]. Our contribution is a placement-enabling architecture aimed at reducing the network resource load on backup nodes. Exploring optimal placements for redundant DCN applications remains an avenue for future research. Like our work, Bakhshi et al. [10] provide a distributed persistent state storage architecture for containerized applications. However, their solution replicates the states to all nodes, likely increasing bandwidth demand.

PSR is a decentralized distributed system. Common in fault-tolerant distributed systems is active replication using Replicated State Machines (RSM) synchronized using a replicated request log [11, 12]. Consensus protocols, like the well-known Paxos, ensure ordered delivery of requests to the RSMs [13, 14]. While influential, Paxos is intricate; hence, Raft offers a simpler alternative [15]. Raft divides time into terms, each with a dedicated leader. Another quite well-known consensus protocol is Viewstamped Replication (VSR), which employs views comparable to Raft's terms [16, 17, 18].

Omni-Paxos, a variant of Paxos, addresses a shortcoming in protocols like Raft, Paxos, and VSR, which can lose progression under partial connectivity scenarios [19]. An example of partial connectivity is a three-peer system where only one peer connects to all others, inhibiting direct communication between the two remaining peers. This situation can hinder progress in VSR and Raft. Omni-Paxos resolves this by implementing Quorum-Connected (QC) as a criterion for leader election. QC means a connection to a quorum of peers.

ZooKeeper Atomic Broadcast (ZAB) is a replication protocol that prioritizes performance by relaxing the guaranteed order slightly [20, 21, 16].

PSR and the above protocols assume fail-stop semantics; Castro et al. propose a practical version of a Byzantine fault-tolerant protocol [22].

Industrial control systems—especially those necessitating redundancy—prioritize high dependability [6]. Solutions tolerant to partial connectivity are more likely to show higher availability. Also, as argued by Ongaro et al. [15], an algorithm where there is one dedicated leader, and that leader is the most up-to-date partaker, is easier to understand. Hence,

with inspiration from the abovementioned protocols, we propose a consensus protocol that, like Omni-Paxos, is partial connectivity tolerant but built upon a VSR foundation instead of Paxos. VSR, like Raft, ensures that the leader is up-to-date with the latest entries after synchronization, and VSR deterministically elects a leader and ensures that this is the only leader. We call the proposed protocol Viewstamped Replication - Quorum Connected (VSR-QC), further described in Section 10.5.

10.3 Partible State Replication

This section provides a high-level introduction and overview of PSR, the problem addressed with PSR, the assumptions, and requirements.

Overview: In the naive state replication method, the backup is required to manage the aggregate bandwidth necessary for synchronizing the state of every application for which it serves as the backup DCN, as detailed in Eq. 10.1. PSR reduces the state replication bandwidth required from a backup by distributing the replicated state storage amongst the DCNs in the DCN cluster. The DCN cluster is the set of DCNs that forms the resource pool available for state replication. Figure 10.3 demonstrates the distributed state storage facilitated by PSR, where DCNs 1-4 function as primary DCNs, managing the primary instances of the applications, while DCN 5 acts as a backup for all these primaries.

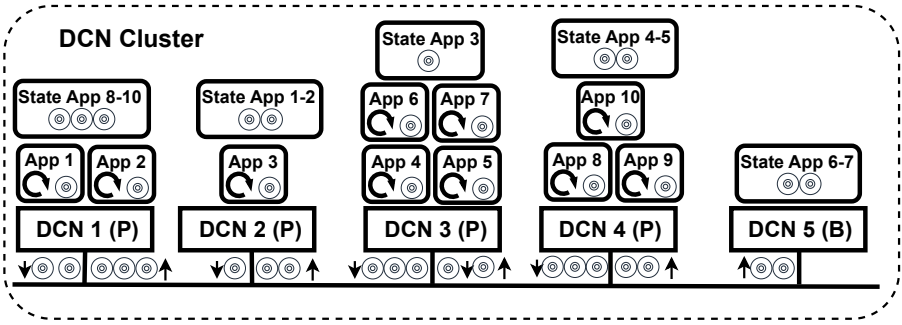


Figure 10.3: An example of a PSR using DCN cluster of five DCNs. DCN 1-4 host primary applications, and DCN 5 is backup for DCN 1-4. All primary applications replicate their state somewhere, but not all to the same DCN.

Assumptions: This work does not cover the allocation of applications to the DCNs, nor the allocations of DCNs to a DCN cluster. We assume applications reside in persistent storage and start upon DCN startup. Additionally,

the backup is assumed to have adequate resources to maintain applications on warm standby. I.e., allow the backup application instance to detect a failure of the primary instance and resume the primary state. The system operates under a non-Byzantine failure-recovery semantic.

Requirements: Each primary application instance must have a designated location for state storage, and each backup instance needs to be able to access this storage. Efficient state fetching and storing are crucial, especially in applications with short cycle times. "Short" is relative, but shorter is better for faster control loops, with updates several times per second being a common minimum [4].

PSR must avoid central mechanisms for pairing application state storage. The cluster should operate independently and recover from faults without a central server, enhancing fault tolerance.

PSR must provide the capability to add (register) and remove (deregister) applications for state storage, i.e., provide dynamic properties. Active applications request and consume available storage; removed applications return storage. Similarly, DCNs register their storage capability upon activation and update it upon change, ensuring they don't become over-allocated. In other words, DCNs report their available capacity, applications declare their resource needs when registering, and PSR tries to find a matching storage for each application.

10.4 Architecture

This section outlines the PSR architecture, detailing its internal components and their interactions in key use cases. Although we refer to DCN, this term is interchangeable with any computing device. The focus is on storage and state replication, but the described principles and mechanisms can be applied to other scenarios, like allocating application execution based on available computational resources.

10.4.1 Components

The PSR architecture comprises three main components, as depicted in Figure 10.4: (i) Application Redundancy Functions (ARF), (ii) Partible State Replication Manager (PSRM), and (iii) Cluster Consensus Manager (CCM). Section 10.5 provides a detailed discussion of the CCM. For the context of this section, it suffices to understand that the CCM provides consistent replication across all DCNs via a consensus algorithm.

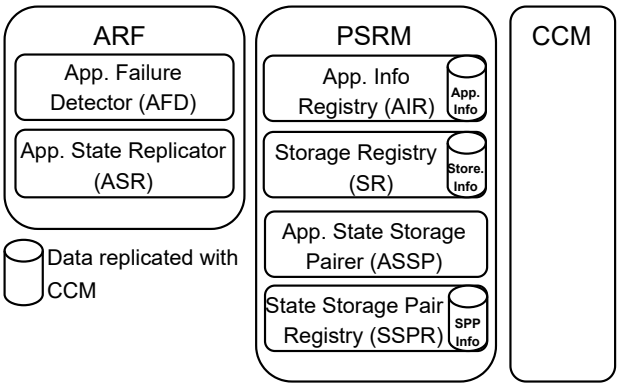


Figure 10.4: High-level architecture view of the PSR components.

The ARF offers redundancy functions essential for a redundant application, including failure detection and state replication. It comprises two subcomponents: the Application Failure Detector (AFD) and the Application State Replicator (ASR). AFD manages failure detection, where the backup's AFD monitors the primary and alerts in case of failure. ASR is responsible for state replication, transferring the application state to the designated storage, and enabling the backup application to fetch the state to resume with the primary's latest state if needed.

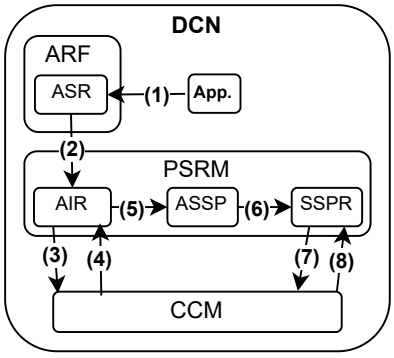
The PSRM consists of four sub-components: the Application Info Registry (AIR), the Storage Registry (SR), the Application State Storage Pairer (ASSP), and the State Storage Pair Registry (SSPR).

The AIR's responsibility is threefold. The first is to gather the local applications' state replication needs. The second is replicating the collected information in the cluster. The third is to keep a registry of all the application's state storage needs in the cluster. See Figure 10.5a.

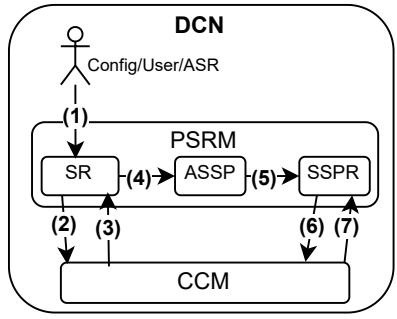
The SR's responsibility is also threefold, like AIR. SR is the AIR counterpart for storage. It gathers the storage capability provided by the local DCN, replicates this information in the cluster, and holds a registry with all the available storage in the cluster. See Figure 10.5b.

The ASSP does the actual paring; it uses the application's state replication needs to find a state storage for each application. ASSP uses the information in AIR and SR to do the pairing. The ASSP instance running on the DCN with the leader CCM does the pairing; see Section 10.5.

The SSPR keeps the registry of application storage allocation, i.e., the application-storage pair. The ASSP updates the SSPR if any change in the requested or available storage impacts the pairing made. Such as the adding



(a) Application registering (or deregistering) (adding or removing).



(b) Storage (DCN) registering (or deregistering) (adding or removing).

Figure 10.5: Component interaction when (a) adding/removing an application or (b) storage when a DCN startup.

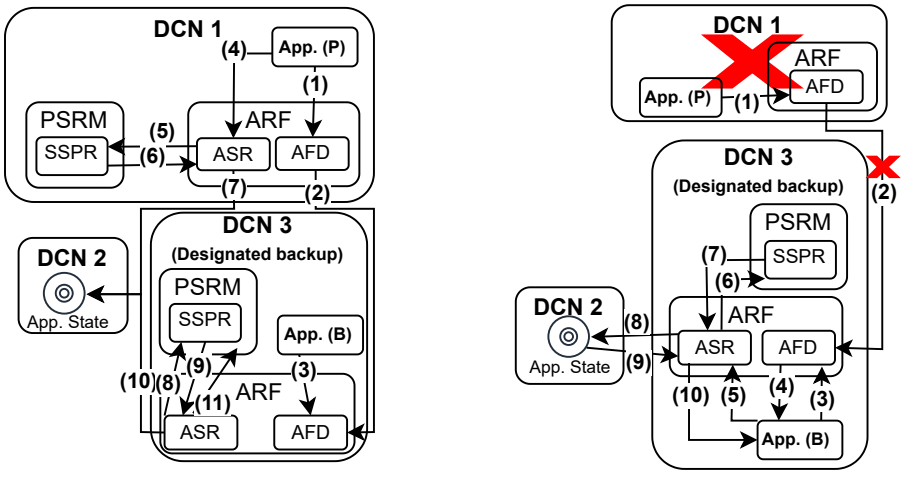
or removing of DCNs or applications. SSPR uses the CCM to replicate the paring information in the cluster. See Figure 10.5a.

10.4.2 Use Cases

This section shows the interaction between the different components for four key use cases.

10.4.2.1 Application Start (Application Registering/Deregistering)

A redundant application registers itself with the ASR at startup (1); see Figure 10.5a. ASR registers the application information in PSRM through AIR (2). AIR uses the CCM to replicate the application information in the cluster (3). When the CCM has successfully replicated application information in the cluster, all AIR instances are informed and update their registry (4). AIR notifies the ASSP when there is a change in the registered applications (5). A change in the CCM leader state also triggers ASSP to evaluate the current pairings. The ASSP located with the CCM leader pairs the application with storage located on a DCN other than the primary application. Once ASSP has made a paring (or removed one), it asks SSPR to update (6). SSPR requests the CCM to replicate the updated pairing in the cluster (7). When the pairing information has replicated in the cluster, the CCM in all DCNs informs the SSPR of the changed pairing information (8).



(a) Normal operation.

(b) Failing DCN scenario.

Figure 10.6: Component interaction during (a) normal operation and (b) failure of DCN running a primary instance of an application.

10.4.2.2 DCN Startup (Storage Registering/Deregistering)

A DCN providing storage registers its information upon startup, as shown in Figure 10.5b. It informs the SR in PSRM (1), and the SR replicates this storage information using CCM (2). CCM ensures all SR instances are updated (3). When SR detects a change in information, it notifies ASSP (4). If the storage update requires a change in the application state storage pairing, ASSP in the DCN with the leader CCM updates and passes this information to SSPR (5). SSPR then replicates this pairing across the cluster using CCM (6), which distributes the updated pairing to SSPR on all DCNs (7).

An alternative scenario occurs when storage becomes unavailable, triggering ASR to mark the storage as unavailable, leading ASSP to reassign alternative storage to the impacted application.

10.4.2.3 Normal Operation

Figure 10.6a illustrates the standard operation process. The application communicates its status to the AFD (1). The AFD in DCN 1 then sends a heartbeat to its backup counterpart in DCN 3 (2). As long as the AFD in DCN 3 receives confirmation that DCN 1's primary instance is operational (3), the backup application in DCN 3 remains on warm standby. The primary application instance replicates its state through ASR (4), which retrieves storage

information from SSPR (5-6) and then stores the application state data, for example, in DCN 2 (7). The ASR in the backup continuously verifies access to the storage and the validity of the state data, retrieving storage location from SSPR (8-9) and checking its accessibility and data validity (10). Any change in storage accessibility is reported to PSRM and SR (11), initiating the process shown in Figure 10.5b. Both primary and backup ASRs perform this accessibility check.

10.4.2.4 DCN Failure

Figure 10.6b depicts a DCN failure and the subsequent actions by the backup application instance in DCN 3 to take the primary role. If the primary application instance fails to trigger the AFD, the warm standby application in DCN 3 becomes the primary (1-4). Upon assuming the primary role (4), it requests the latest state from the ASR in ARF (5). The ASR then inquires about SSPR for the storage location (6-7), retrieves the latest state from the storage (e.g., DCN 2) (8-9), and supplies the retrieved state to the application (10).

10.5 CCM - Cluster Consensus Manager

PSR utilizes CCM for cluster consensus, employing VSR-QC as its consensus protocol. This section details VSR-QC and outlines the CCM sub-components for implementing CCM with VSR-QC.

10.5.1 VSR-QC Protocol

VSR-QC, influenced by VSR [18], Raft [15], and Omni-Paxos [19], operates under a non-Byzantine failure-recovery model. It assumes protocol instances don't continuously crash and recover. Messages may be lost, but the system is generally synchronous, following a partially synchronous model. When describing the protocol, we use the term VSR-QC instance rather than server, node, or DCN since VSR-QC can run multiple instances per node / DCN.

Like Raft, a VSR-QC instance can be in either of the three states $Follower$, $Electing$, or $Leader$, shown in Figure 10.7.

The leader is the VSR-QC instance in state $Leader$ and it is the driver of the replication. The leader is elected from the set of followers. A follower can only become a leader if it is QC and elected by a majority. Each instance has a unique and persistently stored identification, id.

Followers are VSR-QC instances in state $Follower$. Each VSR-QC instance maintains its own instance of the replicated log. The replicated log is

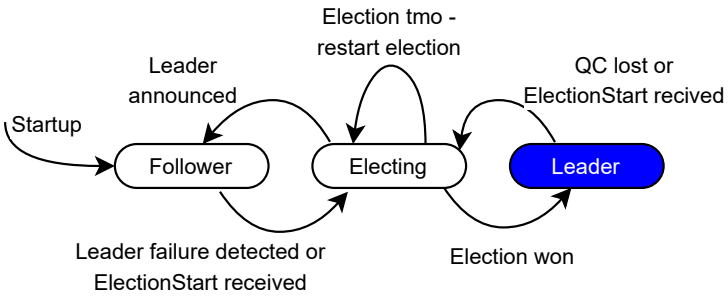


Figure 10.7: The VSR-QC state machine.

an ordered list of replicated requests. The log is orderly replicated to all functioning VSR-QC instances in the replica group. The replica group is the set of VSR-QC instances forming the distributed replication. VSR-QC as VSR, Raft, and Paxos tolerate f faulty VSR-QC instances in the replica group. Hence, $2f + 1$ is the minimum replica group size to be f fault tolerant.

VSR-QC utilizes the view concept [18, 15]. A view is an integer, $ViewNumber$, incremented each time a leader election process is started.

The protocol's functionality is explained through four scenarios: (i) normal operation, (ii) leader election, (iii) synchronization, and (iv) failure detection, concluded with a brief discussion on configuration.

10.5.1.1 Normal Operation

The normal operation of VSR-QC is similar to VSR [18], illustrated in Figure 10.8 and summarized below.

A DCN (or other client) issues a request by sending a $\langle Request, rid, msg \rangle$ message to the leader (1). Step 3 in Figure 10.5a is a PSR request example. The rid is a tuple comprising the client ID and a request number, forming a unique request ID, rid. The rid prevents double processing of requests in case of a leader failure while a request is uncompleted. The payload of the request is msg. Unprocessed requests, identified by rid, prompt the leader to dispatch a $\langle Prepare, v, n, r, m \rangle$ message to all followers (2), where v is the current $ViewNumber$, n the $OpNumber$, and r and m are the rid and msg. The $OpNumber$ is an integer incremented by the leader for each finalized request.

Followers process $Prepare$ messages sequentially in $OpNumber$ order. Upon having all prior log entries, a follower adds the new entry, stores the rid, and sends a $\langle PrepareOK, v, n \rangle$ back to the leader. If preceding entries are missing, the follower attempts synchronization (see Section 10.5.1.3), with-

holding $PrepareOK$ until all previous and current entries are stored in the log.

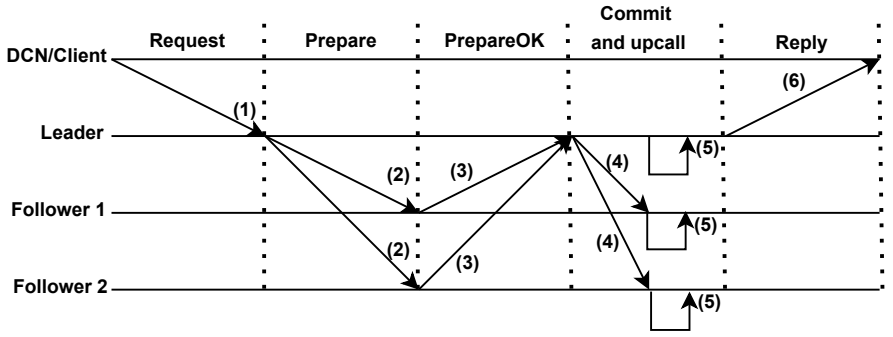


Figure 10.8: Normal replication flow of VSR-QC.

The leader waits for replies from f followers with $PrepareOK$ (f followers plus the leader constitute a majority). After receiving at least f $PrepareOK$, the request is stored in the replicated log of a majority, and the leader issues a commit with the $\langle Commit, v, k \rangle$ message (4), where k is the $CommitNumber$. The $CommitNumber$ is the highest $OpNumber$ that has been committed. Committed entries can not be changed or removed.

After sending the $Commit$, the leader performs the upcall to the distributed application (5). The upcall is the term for passing the request to the distributed application layer, PSR, in our case, exemplified in step 4 in Figure 10.5a. The followers issue the upcall when they receive the $Commit$ if all previous entries are committed.

10.5.1.2 Leader Election

As mentioned in Section 10.2, VSR does not handle partial connectivity because it requires QC voters [18, 19]. VSR-QC is partial connection tolerant because VSR-QC uses QC as leader criteria, but it does not require that the voters are QC, similar to Omni-Paxos [19].

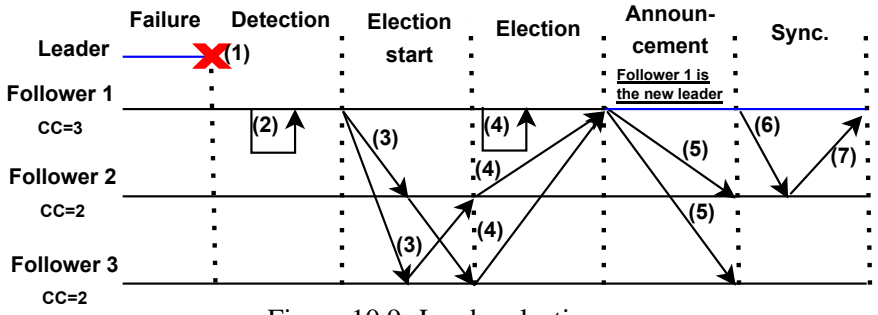


Figure 10.9: Leader election.

The failure detection catches leader failure and determines which instances are QC or not; see Section 10.5.1.4. Figure 10.9 shows a leader election sequence that starts with the failure of the current leader (1). Eventually, the failed leader is detected by another VSR-QC instance, in this case, Follower 1 (2).

Follower 1 detects the failed leader and initiates an election by entering the $Electing$ state, increments its $ViewNumber$, sends an $\langle ElectionStart, v \rangle$ message to all other VSR-QC instances (3), and starts the election timeout timer. The $ElectionStart$ message contains v set to the current (just incremented) $ViewNumber$ of the VSR-QC instance.

Follower 2 receives the $ElectionStart$ message from Follower 1 and enters the $Electing$ state if the received v is higher than $ViewNumber$ and assigns v to its $ViewNumber$. Follower 2 also sends an $ElectionStart$ message to all other instances when entering the $Electing$ state and starts the election period timer.

Suppose no leader has presented itself directly via the $ElectionComplete$ message or indirectly via the heartbeat. In that case, when the election period timer expires, the election process restarts by incrementing the $ViewNumber$ again and re-entering the $Electing$ state.

If a new $ElectionStart$ message with v higher than $ViewNumber$ is received before the election period has ended, the election period restarts and the above-described actions repeat.

In $Electing$, VSR-QC instances cast one vote per election period ($ViewNumber$) using $\langle ElectionVote, lid, v, n, k \rangle$, addressed to the prospective new leader. This message includes lid (the prospective leader's id), $ViewNumber$, $OpNumber$, and $CommitNumber$, as depicted in step (4) in Figure 10.9.

Voting is based on the Connectivity Count (cc), updated by the failure detection; see Section 10.5.1.4. The cc reflects the number of connected VSR-QC instances. Votes are given to the instance with the highest cc over the QC limit. In case of equal cc values, the tie is broken by id, favoring the lowest id.

A VSR-QC instance receiving an $ElectionVote$ message enters the $Electing$ state if the received v is higher than $ViewNumber$ and performs the above-described action when entering state $Electing$. In state $Electing$ it counts all $ElectionVote$ messages with $v = ViewNumber$ received within the election period as valid votes. If it gets an $ElectionVote$ with a v higher than its $ViewNumber$, it re-enters $Electing$, resets the vote count, restarts the election timer, and updates its $ViewNumber$ to v.

A VSR-QC instance in $Electing$ that receives $f + 1$ valid votes accepts that it is the new leader and enters the $Leader$ state and announces itself

as the new leader by sending out a $\langle ElectionComplete, lid, v, n, k \rangle$ message where lid is id, v is $ViewNumber$, n the $OpNumber$, and k the $CommitNumber$. Figure 10.9 (4) shows Follower 1 obtaining the majority of votes, becoming the new leader, and making the announcement with the $ElectionComplete$ message (5). The other VSR-QC instance in state $Electing$ enters the $Follower$ state when receiving $ElectionComplete$, or a $Heartbeat$ indicating a leader, with a v equal or higher than $ViewNumber$.

A leader that loses QC leaves the leader role and initiates a new election by sending the $ElectionStart$ message; see Figure 10.7.

The new leader must ensure it has the latest log entries, which it does by requesting the entries it is missing, if any, from the most up-to-date follower. The n k in the $ElectionVote$ message has informed the leader about the most up-to-date follower, and it is to that follower the leader requests a synchronization, step (6) and (7) in Figure 10.9. Synchronization is further described in Section 10.5.1.3.

10.5.1.3 Synchronization

VSR-QC requires no persistent storage to store the log; it assumes that a majority never fails at the same time. However, if desired, VSR-QC, as the VSR inspiration, can be modified to use persistent storage and reduce the synchronization needed upon recovery [18].

This section describes synchronization steps to bring a VSR-QC instance that, for whatever reason, has become outdated in synchronization again. We divided the synchronization description into three steps: (i) detection, (ii) follower synchronization, and (iii) (newly elected) leader synchronization.

A follower detects that it is not synchronized when receiving $Prepare$ or a $Heartbeat$ message with a v and n higher than the follower's $ViewNumber$ and $OpNumber$. A leader detects lagging when it has received a majority of votes by comparing the v and n in the received $ElectionVote$ message with its $ViewNumber$ and $OpNumber$.

A follower that is out of synchronization uses the $\langle SyncMeReq, i \rangle$ message where i is the id of the follower. The receiving VSR-QC instance, regardless of its current role, will reply with the $\langle SyncMeReply, v, n, k, l \rangle$ message, where v,n,k, and l is the $ViewNumber$, $OpNumber$, $CommitNumber$, and log entries of VSR-QC instance i.

The leader is the most updated VSR-QC instance since it is the designated receiver of client requests and is the driver of advancement. However, a new leader might not possess the latest entries immediately after the election. Therefore, as described in Section 10.5.1.2, a newly elected leader's

first step is synchronizing itself with the latest entries. The information in the $ElectionVote$ concludes which is the most updated VSR-QC instance in the majority. The leader sends a $SyncMeReq$ message to one of the most updated VSR-QC instances to retrieve the log and perform any missing commits and upcalls. After synchronization completion, the leader starts accepting and processing client requests.

Do note there are several ways to make the synchronization handling more efficient; some are discussed in the VSR description [18].

10.5.1.4 Failure Detection

All the VSR-QC instances send $Heartbeat$ to one another. A concrete realization example of such an exchange is a multicast group dedicated to failure detection within the replica group.

The $\langle Heartbeat, v, n, k, i, p, c \rangle$ message conveys each instance's replication and connectivity status. Replication status includes $ViewNumber(v)$, $OpNumber(n)$, $CommitNumber(k)$, instance id (i), and the leader's id (p), with p set to zero if no leader is identified. Hence, this message gossips the leader's identity. Connectivity status is represented by c, indicating which instances are connected in a simple bit-field format, where each bit corresponds to a VSR-QC instance.

Based on these heartbeats, VSR-QC instances update their Connectivity Count (cc). Only QC followers can start elections, and votes are given to the highest cc, averting continuous re-elections. The leader will relinquish its role if it loses QC status.

10.5.1.5 Configuration

Adding and removing VSR-QC instances to the replica group is not covered for page conservation reasons. We envision that the mechanisms VSR uses for adding/removing members are suitable for VSR-QC as well [18].

10.5.2 Components

Figure 10.10 displays the components of CCM, with the CCM Service Abstraction (CSA) acting as an interface. CSA's role is to provide an easy-to-use interface to the replicated services like PSR and abstract the underlying consensus protocol. CSA functions include issuing requests, upcall registration, and leader checks. CCM's architecture, as shown in Figure 10.10, comprises four sub-components: (i) Leader Elector (LE), (ii) Failure Detector (FD), (iii) Group Member Manager (GMM), and (iv) Log Replicator (LR). LE and FD

handle leader election and failure detection (Section 10.5.1.2 and 10.5.1.4), GMM manages and updates replica group membership, and LR, detailed in Section 10.5.1.1, manages replication, performs upcalls for new log entries, and handles synchronization.

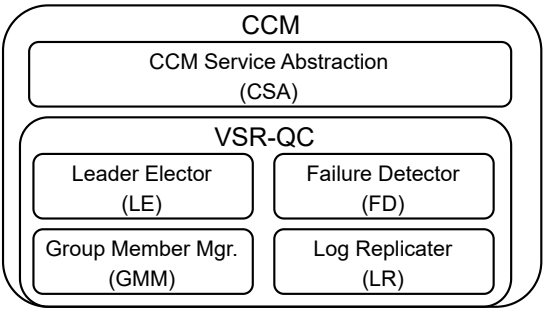


Figure 10.10: CCM Components.

10.6 Implementation, Execution and Result

10.6.1 Implementation

We developed a PSR prototype for VxWorks based on the architecture described in the previous sections. The prototype is available on GitHub [23].

The prototype version of the ASSP application-storage pairing algorithm pairs applications with available storage based on the sequence of their registration. The primary goal of the algorithm is to find storage on another DCN than the DCN hosting the primary application instance; the secondary goal is to find the least utilized storage. The prototype version pairs an application requiring storage with the first found unused storage. If no unused storage exists, it searches for the first storage used by only one other application. If that fails, no storage exists for the application. The exploration of more advanced pairing algorithms is future work.

The ARF implementation of the AFD uses a UDP-based heartbeat message protocol. The heartbeat messages are sent on requests from the application. The ASR state replication uses a UDP-based message encapsulating the most recent state as reported by the application. This message is directed to the state-storing DCN, exemplified by DCN 2 in Figure 10.6a. The ASR is responsible for operating a storage server on the storage DCN. This server receives the incoming state messages and preserves the most recent within RAM.

The prototype includes a Test Application (TAPP), a redundant application that can function as either primary or backup. We utilize the TAPP to assess PSR's performance and to draw comparisons with the naive state replication approach depicted in Figure 10.2. In its backup role, the TAPP remains in warm standby, meaning it's loaded into RAM and primed to switch to the primary role when the AFD detects a failure of the original primary instance of TAPP. The primary TAPP instance requests that the AFD send heartbeats and the ASR transmit its latest state to the storage. The state includes a sequence number that is incremented in each iteration. The state size and period time are adjustable.

When a TAPP instance takes the primary role, it requests the latest state using its local ASR. The local ASR, in turn, sends a state request message to the ASR on the storage DCN to obtain the most recent state, as illustrated in Figure 10.6b.

To conduct failover testing, the primary TAPP instance is instructed to cease operation, which prompts the backup instance to assume the primary role upon the AFD heartbeat timing out. In transitioning to the primary role, the TAPP anticipates a specific state from the ARF. Knowing the sequence number of the latest state before a commanded shutdown, the TAPP can confirm whether it has successfully retrieved the most recent state.

10.6.2 Setup and Execution

We utilize virtual machines running on VMware 17 as DCNs. Each virtual machine has one CPU, one core, and two GB of RAM. These machines are hosted on a Lenovo ThinkPad P15, featuring a 2.7 GHz Intel I7 processor and 48 GB of RAM. The bandwidth of the virtual network interface connecting the virtual machines is limited to 1024 Kbps. This limitation is imposed to make evaluations feasible in a virtualized environment without overloading the host computer. On these virtual DCNs, we run the PSR prototype, including the TAPP, on VxWorks 21.07.

Our experiments involve five different redundancy patterns: (i) one primary and one backup (1p), (ii) two primaries and one backup (2p), (iii) three primaries and one backup (3p), (iv) four primaries and one backup (4p), and (v) five primaries and one backup (5p). The primaries run the TAPP instance in the primary mode, while the backups host the backup TAPP, as depicted in Figure 10.3. In the 1p configuration, there are four TAPP instances: two running as primaries on the primary DCN and two as warm standby backups. This pattern continues, resulting in 8 instances for 2p, 12 for 3p, 16 for 4p, and so on.

Each TAPP instance operates on a 40-millisecond cycle, sending a heartbeat (via AFD) and replicating its state (using ASR). The volume of state data replicated in each cycle is adjustable. We begin our tests with 128 bytes of state data, increasing it in 128-byte increments until network resource overutilization on one or more DCNs causes the test to fail. At each increment, we simulate a controlled failure of the TAPP instance on DCN 1 by commanding it to stop.

A test is considered to have failed when a backup TAPP instance erroneously transitions to the primary role due to AFD not receiving heartbeats – a consequence of network congestion from state replication traffic. Similarly, a test fails if a triggered failure doesn't result in the backup retrieving the latest state. We define the point at which tests begin to fail as the 'failure threshold.'

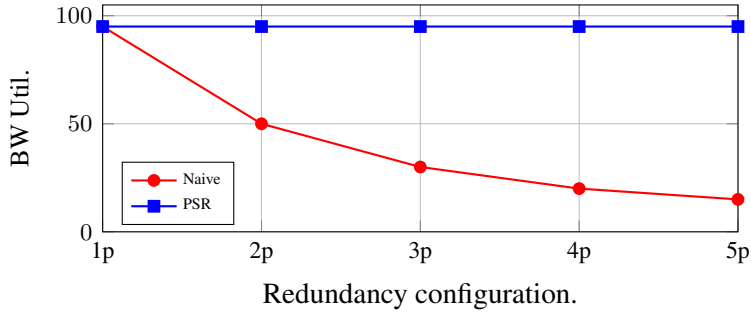
In our tests, we run the system in PSR mode, employing PSR for state replication. Each participating virtual DCN contributes storage for two TAPP instances in this mode. Therefore, in the 1p configuration, two DCNs provide storage. However, for fault tolerance, the primary can only use storage on the backup DCN, not on itself. In the 2p setup, three DCNs offer storage, assigned to TAPP by PSR, demonstrating the concept illustrated in Figure 10.3. In contrast, the naive mode only uses the backup for storage, as shown in Figure 10.2.

10.6.3 Result

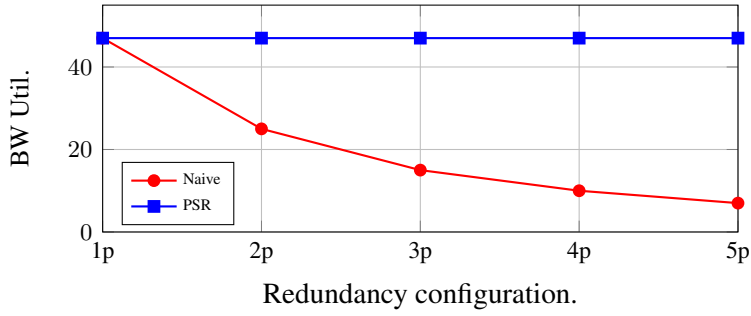
The graphs presented in Fig 10.11 illustrate the bandwidth usage for state replication under functioning redundancy, the TAPP state replication increment **before** the failure threshold. In other words, the graphs highlight the point at which an additional 128-byte increment in the TAPP state data usage leads to system failure. Thus, these graphs offer insights into the differing aspects of the state replication bandwidth threshold for both naive state replication and PSR.

For the 1p configuration, the state replication threshold is identical between PSR and the naive approach. This is because, in a 1p setup, the naive and PSR methods replicate the state to the only backup available. However, as illustrated in Figure 10.11a, the bandwidth available to each DCN decreases as we expand the configuration using the naive approach. In contrast, with PSR, the bandwidth remains consistent regardless of the number of DCNs since each added DCN also contributes storage, as shown in Figure 10.11a.

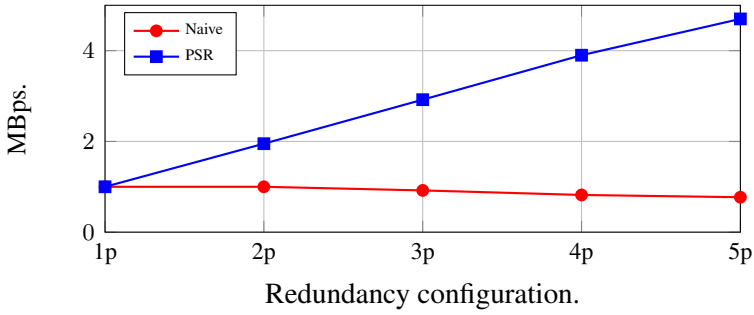
Figure 10.11b displays the bandwidth utilization per application. In our experiment, we consistently deployed two TAPP instances per DCN, meaning the threshold bandwidth utilization for each application is effectively half that



(a) DCN bandwidth utilization.



(b) App. bandwidth utilization.



(c) Total state replication bandwidth used.

Figure 10.11: Graphs showing the threshold bandwidth utilization per DCN, application, and the total state replication bandwidth.

of the DCN.

Finally, Figure 10.11c depicts the total bandwidth used for state replication across all applications in the various configurations. Notably, the total bandwidth usage for PSR increases as more configurations are added. This increase is attributed to each newly added DCN hosting both the TAPP and provides storage. Conversely, in the naive approach, where only the backup provides storage, the total bandwidth usage slightly decreases. This decrease is likely due to the increased number of heartbeat messages and overhead.

10.7 Conclusion and Future Work

This paper introduced an architecture that separates state replication storage from backup in a decentralized system, employing the VSR-QC consensus protocol to maintain consistency. We evaluated the state replication capacity, comparing PSR with naive state replication methods. Our results show that PSR significantly increases the feasible state replication data volume, enabling a single DCN to back up multiple primaries.

Future research goals include bounded, low-latency, state data retrieval mechanisms, and reliability modeling to find cost-efficient deployments for real applications that satisfy given reliability targets. Another future research possibility is optimizing application and state storage pairing, given available resources and response time requirements. A last example of future research is investigating the integration of PSR in a context where a system like Kubernetes orchestrates the DCNs and applications.

Bibliography

[1] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993.

[2] Sandeep Singh, Valavoju Mahesh Chary, and P Abdul Rahman. Dual redundant profibus network architecture in hot standby fault tolerant control systems. In *Int. Conf. on Advances in Eng. & Tech. Research (ICAETR)*, 2014.

[3] Andrei Simion and Calin Bira. A review of redundancy in plc-based systems. *Advanced Topics in Optoelectronics, Microelectronics, and Nanotechnologies XI*, 12493:269–276, 2023.

[4] T. Hegazy and M. Hefeeda. Industrial automation as a cloud service. *IEEE Transactions on Parallel and Distributed Systems*, 26(10):2750–2763, Oct 2015.

[5] Bjarne Johansson, Mats Rågberger, Thomas Nolte, and Alessandro V Papadopoulos. Kubernetes orchestration of high availability distributed control systems. In *Proc. ICIT*, 2022.

[6] Jacek Stój. Cost-effective hot-standby redundancy with synchronization using ethercat and real-time ethernet protocols. *IEEE Trans. on Autom. Science and Eng.*, 18(4):2035–2047, 2020.

[7] Ali Shakarami, Mostafa Ghobaei-Arani, Ali Shahidinejad, Mohammad Masdari, and Hamid Shakarami. Data replication schemes in cloud computing: a survey. *Cluster Computing*, 24:2545–2579, 2021.

[8] Ibrahim Adel Ibrahim, Wei Dai, and Mustafa Bassiouni. Intelligent data placement mechanism for replicas distribution in cloud storage systems. In *IEEE Int. Conf. on Smart Cloud (SmartCloud)*, pages 134–139, 2016.

[9] Hong Zhang, Bing Lin, Zhanghui Liu, and Wenzhong Guo. Data replication placement strategy based on bidding mode for cloud storage cluster. In *Web Information System and Application Conf.*, pages 207–212, 2014.

[10] Zeinab Bakhshi, Guillermo Rodriguez-Navas, and Hans Hansson. Analyzing the performance of persistent storage for fault-tolerant stateful fog applications. *Journal of systems architecture*, 144:103004, 2023.

[11] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[12] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.

[13] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

[14] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, pages 51–58, 2001.

[15] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pages 305–319, 2014.

[16] Robbert Van Renesse, Nicolas Schiper, and Fred B Schneider. Vive la différence: Paxos vs. viewstamped replication vs. zab. *IEEE Transactions on Dependable and Secure Computing*, 12(4):472–484, 2014.

[17] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17, 1988.

[18] Barbara Liskov and James Cowling. Viewstamped replication revisited. 2012.

[19] Harald Ng, Seif Haridi, and Paris Carbone. Omni-paxos: Breaking the barriers of partial connectivity. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 314–330, 2023.

[20] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. {ZooKeeper}: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[21] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 245–256, 2011.

[22] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, number 1999, pages 173–186, 1999.

[23] Psr prototype implementation on github. `https://github.com/Burne77a/psr`. Accessed: 2025-05-05.

Chapter 11

Paper F: Checkpointing and State Transfer for Industrial Controller Redundancy

Bjarne Johansson, Björn Leander, Olof Holmgren, Alessandro V. Papadopoulos, and Thomas Nolte.

Journal submission under review, September 2025.

Abstract

Industrial controllers are moving from controller-centric to network-centric architectures, where lightweight containerization is increasingly adopted in operational technology. Many industrial domains require high reliability, often achieved through spatial standby redundancy with duplicated controllers where one is the active primary and the other a standby backup. In such setups, the standby backup must seamlessly take over control when the primary fails. Hence, the backup needs to be up-to-date with respect to the primary's internal state. The retrieval of internal states is commonly known as checkpointing. We review checkpointing approaches used in virtualized and industrial settings and derive a set of desired features for state-transfer protocols. We then assess existing communication protocols against these features and experimentally evaluate the two strongest contenders under no-loss and packet-loss conditions, measuring recovery performance. The analysis reveals that no existing protocol meets all the desired features. To address this gap, we introduce a new state-transfer protocol that satisfies all identified features. In experiments, it demonstrates good performance under packet loss, with only a slight reduction in throughput compared to the identified top contender protocols that we used for comparison.

11.1 Introduction

Industrial Control Systems (ICS) are undergoing an architectural paradigm shift, a shift from a controller-centric architecture to a network-centric architecture [1]. The distinguishing difference between the two architectures is that the network replaces the controller as the system center. The shift is part of a strive to create interoperable and flexible systems designed to ease data propagation to data-hungry AI-driven forecasting and decision-making systems. Facilitating standards is a cornerstone in inter-vendor interoperability; in the context of ICS, OPC UA is believed to be such a standard [2].

The connectivity provided by the network-centric architecture, in combination with increased Ethernet usage, enables more flexible deployment of controllers, thereby boosting the interest of Information Technology (IT) in Operation Technology (OT) domains [3, 4, 5]. One example of such technologies is lightweight virtualization in the form of containers and the orchestration of those [6, 7, 8]. Containerized controllers can increase the deployment alternatives and provide more flexibility, especially if the controllers are hardware agnostic and not dependent on specialized fieldbuses for communication [6, 5].

ICS automates a broad range of solutions in a wide spectrum of domains. Needless to say, no one wants unplanned production stops due to their control system failing, and for some domains, stops can have severe impacts. Mandating a need to keep the probability of failure low with various fault-tolerance techniques. A conventional way to increase fault tolerance is to duplicate critical devices such as controllers and network paths to form redundant solutions and avoid single points of failure [9]. In the context of ICS and controllers, spatial standby redundancy with hardware duplication is a common redundancy pattern, where one controller serves as the active, primary controller, and the other acts as a standby backup [10, 11]. The redundancy masks primary controller failures from the perspective of field devices relying on control from the redundant controller pair, forming a passive standby redundancy [9]. In the case of primary failure, the backup controller seamlessly assumes the primary role and provides output to the field devices.

For a backup to be able to assume the primary role upon failure of the original primary controller, mechanisms for failure detection and state replication are needed [12, 13]. As the name implies, failure detection is the mechanism used to determine if the primary has failed. The second mechanism, state replication, allows the backup to resume with the internal states needed to continue the primary role transparently for the field devices and, ultimately, the controlled process.

Checkpointing is the process of collecting the internal states; hence, to

have any state data to replicate, the primary first needs to checkpoint. As mentioned, the network-centric transformation and lightweight virtualization concepts drive controller software to be hardware agnostic, including redundancy functions such as state replication [14]. The focus of the work is the transfer of checkpoint data, and the goal is to find a solution suitable for transferring the collected state data of a primary controller over Ethernet to the backup controller.

The state data transfer needs to be secure, and security is a growing concern within the industrial domain, given the increasing system complexity and connectivity of industrial systems that utilize ubiquitous communication protocols. Hence, state transfer protocols need protection from cybersecurity threats, as they are potential attack vectors for availability attacks, and state data may contain sensitive information [15].

We structure the work as a five-step workflow, where each step builds upon the preceding one, as illustrated in Figure 11.1.

Step I presents a literature search and summary covering checkpointing/state-transfer work in industrial controller redundancy and in container/orchestration contexts.[1]

Step II defines desired features for state transfer and compares candidate communication protocols against these features.

Step III presents and performs an experimental evaluation of two protocols with the highest feature coverage.

Step IV presents a protocol conceived for state transfer, together with its design.

Step V presents the implementation and integration on VxWorks, a real-time operating system (RTOS) [16], and the experimental results.

The contributions of the paper are:

C 1: Literature search and summary: a concise overview of checkpointing/state-transfer approaches in industrial redundancy and container/orchestration contexts.

C 2: Protocol feature matching: identification of desired features for communication protocols used for state transfer, and matching these features against selected protocols, where the main emphasis is to identify protocols suitable for state transfer in the industrial controller redundancy use case.

[1] This is a targeted literature scan to identify relevant mechanisms and technologies for our use case; it is *not* a systematic literature review.

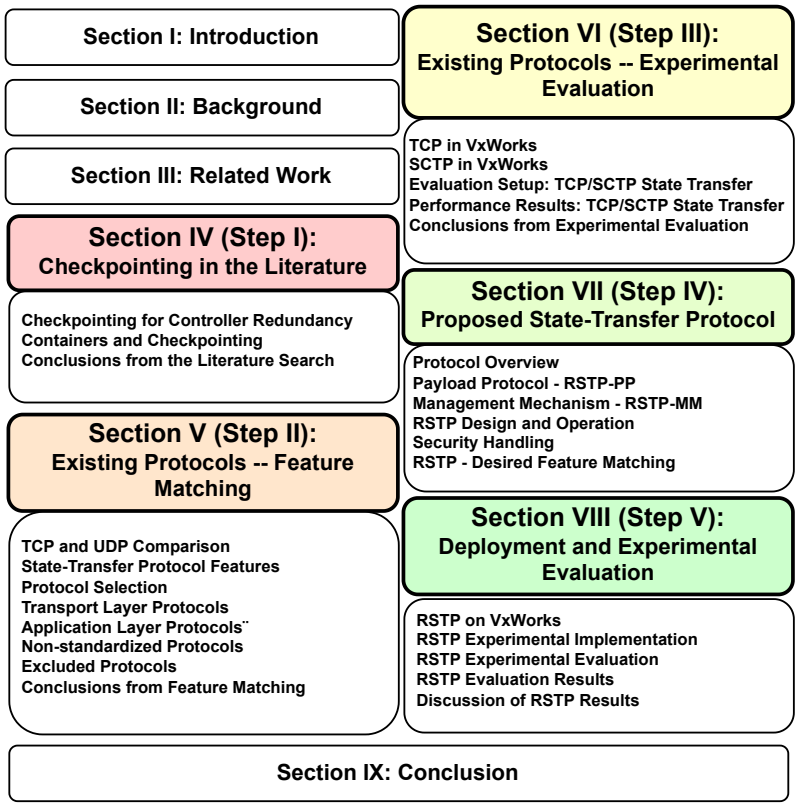


Figure 11.1: Overview of the paper's sections and the five-step workflow (Step I–V).

C 3: Experimental evaluation: experimental evaluation of the two best-matching protocols on VxWorks under loss-free and lossy conditions.

C 4: State-transfer protocol: design and integration of a state-transfer protocol, experimentally evaluated on VxWorks (including multi-application mimicking workloads) and compared to the two best-matching protocols; it exceeds them under loss and supports transmission scheduling to facilitate deadline-driven prioritization.

The paper is organized as follows: Section 11.2 introduces industrial controllers, the execution model, fault tolerance, and container orchestration, and

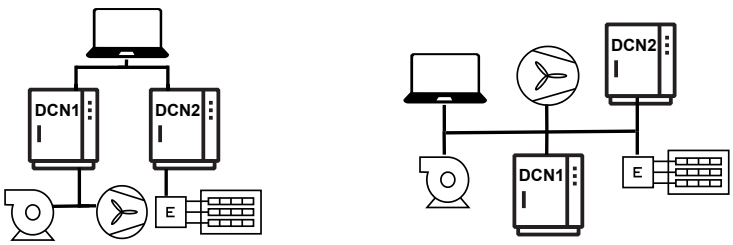
Section 11.3 provide related work. Section 11.4 (Step I) details the checkpointing literature search and summarizes the results. Section 11.5 (Step II) defines desired features for state-transfer protocols, introduces candidate protocols, and assesses them against these features. Section 11.6 (Step III) experimentally evaluates the top candidates and reports results. Section 11.7 (Step IV) presents the proposed protocol, and Section 11.8 (Step V) evaluates it. Section 11.9 concludes. Figure 11.1 illustrates the paper structure.

11.2 Background

This work addresses challenges related to the fault tolerance of industrial controllers. Hence, this section first introduces ICS and their execution models, then introduces fault-tolerance concepts, and finally, briefly introduces orchestration and containers.

11.2.1 Industrial Controllers

Industrial controllers are rugged computers designed for longevity in potentially harsh environments. The controller executes the control logic to drive the process to the desired state by reading and writing values to and from field devices that interface with the physical world. Distributed Control Systems (DCS) are large-scale automation systems comprising interconnected controllers that communicate with each other and field devices to automate an entire site, rather than just a single machine. A Programmable Logic Controller (PLC) is another well-known term for industrial-grade controllers, often featuring built-in Input/Output (I/O) interfaces.



(a) Controller-centric. (b) Network-centric.

Figure 11.2: Controllers, field devices, and upper layers of the automation pyramid in a controller-centric architecture and a network-centric architecture.

Figure 11.2a shows the traditional hierarchical controller-centric architecture, where field devices at the bottom of the figure connect to only one controller, commonly over dedicated fieldbuses [17]. Above the controllers are the high-level systems, such as Supervisory Control and Data Acquisition (SCADA), which provide operator control and overview with fewer real-time requirements and more reliance on IT systems. Figure 11.2b shows the the flattened network-centric architecture, with all system parts connected to a communication backbone, denoted as the O-PAS Connectivity Framework (OCF) by the Open Process Automation™ Standard (O-PAS) [18]. O-PAS refers to controllers as Distributed Control Nodes (DCNs); we use the terms *controller* and *DCN* interchangeably.

As mentioned, the DCN runs the control logic that strives to drive the controlled process to the desired state by getting and providing input and output to field devices. The following section introduces the DCN execution model.

11.2.2 Execution Model

The control logic that executes on the controller is a program, also referred to as an application, typically developed in an engineering tool provided by the DCN manufacturer. The engineering tool enables users to program and develop applications for specific domains and download them to the DCN. The predominant standard for programming DCN applications is IEC 61131-3, and the execution model is cyclic as shown in Figure 11.3 [19, 20].

As shown in Figure 11.3, the execution phase consists of four phases: (i) *Copy-in* (CI), (ii) *Execute* (Exe), (iii) *Replicate State* (RS), and (iv) *Copy-out* (CO). *Copy-in* is the phase where updated values from the field device are made available to the application. These are the values the application uses when executing the control logic in the *Execute* phase. The *Execute* phase updates the internal states of the application, i.e., variables are updated. The updated state needs to be replicated to the backup in case of redundancy. This replication takes place in the *Replicate State* phase. Lastly, the updated values are communicated to the connected field devices, which occurs in phase *Copy-out*.

A controller typically executes a set of control applications, denoted as A, where each application $a \in A$ has a period P_a. Within each period P_a, the application a executes all its phases: CI_a, Exe_a, RS_a, and CO_a. Specifically, application a must complete all phases included in the Execution Phase (EP) tuple $\langle CI_a, Exe_a, RS_a, CO_a \rangle$ during each period. Koziolek et al. define the application slack time as the interval from when application a completes its CO_a phase to its next invocation in the subsequent period [21]. Figure 11.4

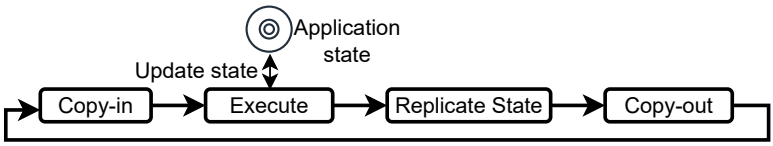


Figure 11.3: Controller application execution sequence.

illustrates the application phases and the phases' dependency on input data and internal state from earlier periods. Figure 11.4 also shows the output from the different phases.

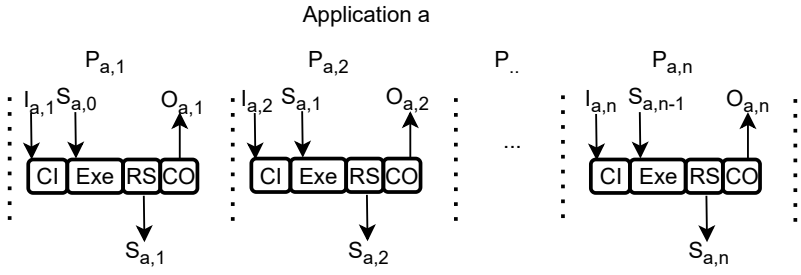


Figure 11.4: Internal state and application execution phases dependency and relation.

$I_{a,n}$ represents the set of input values from CI at period instance n for application a. $S_{a,n-1}$ represents the internal state at the start of execution for application a in period n, while $S_{a,n}$ is the new internal state after execution $Exe_{a,n}$. This new state ($S_{a,n}$) is replicated to the backup and used as the internal state for $Exe_{a,n+1}$ in period $P_{a,n+1}$. Lastly, $O_{a,n}$ represents the externally visible output from the execution of application a in period $P_{a,n}$.

$O_{a,n}$ depends on $I_{a,n}$, $S_{a,n-1}$, and the execution $Exe_{a,n}$. Therefore, to avoid producing historically outdated values during a failover, any failover occurring after the output $O_{a,n}$ must result in outputs that are $O_{a,n}$ or later for all $a \in A$. Consequently, once the primary outputs $O_{a,n}$, the backup must hold an internal state $S_{a,n-1}$ or later. This implies that, upon the primary's output of $O_{a,n}$, state $S_{a,n-2}$ (and older) are outdated. Figure 11.5 illustrates state aging on a backup for application a.

Note that the application state for application a is utilized by a only; hence, a failed state transfer does not directly impact any other application than a.

Inter-application communication between applications is handled in a similar way as application and field device communication, rather than multiple applications being directly dependent on the same state data.

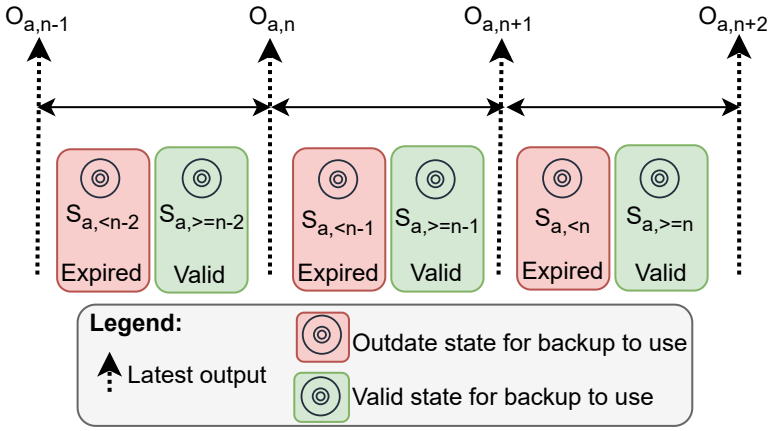


Figure 11.5: Output to field devices and state and aging on the backup.

11.2.3 Fault Tolerance

Fault-tolerance, as the name implies, is about being robust and continuing to operate even in the presence of faults. Spatial standby redundancy is a specific fault-tolerance pattern common in ICS [11].

State replication is a fundamental part of a standby spatial redundancy where one unit is active, and one or more backups are ready to take over in case of failure of the active [12]. We assume a fail-silent semantics, meaning that if a primary fails, it stops providing output [22]. The backup typically supervises the primary by expecting a primary-originated message at known intervals, a so-called heartbeat [23, 24]. The backup typically interprets the absence of heartbeats as a failure of the primary controller. Self-tests, diagnostic checks, and parallel execution with cross-comparison are fault detection methods that also serve to strengthen the fail-silent behavior in industrial control systems [25]. Failure detection is not in the scope, and other failure semantics, such as Byzantine faults, are not considered [26].

Spatial standby redundancy with hardware duplication addresses persistent hardware failures, and failure detection is necessary to enable the backup to recognize that the primary has failed. From the perspective of the field devices,

this is a passive redundancy, where a failover occurs when the backup takes over for a failing former primary, and the change should be transparent from the field devices' perspective [9].

The degree of readiness denotes the level of the standby, divided into cold-, warm-, and hot-standby [9]. Cold standby refers to an unpowered spare that maintenance personnel can use to quickly replace a failing DCN. The difference between warm and hot is the spare's activity level. A warm standby DCN backup does not execute the control applications. Still, it quickly resumes them when becoming primary, and a hot standby DCN executes the control application but does not provide output to the field devices. Our work targets warm and hot standby redundancy.

Retrieving an application state for recovery is commonly referred to as checkpointing [20]. Alternative to checkpointing, for redundancy purposes, are, for example, deterministic Replicated State Machines (RSM), where the events that progress the state machines are transferred rather than checkpointed internal states, i.e., active replication [27]. To repeat events in a deterministic order, consensus protocols such as Raft and Paxos can be used [28, 29]. This work focuses on passive replication, where internal states are checkpointed and transferred rather than the events themselves.

11.2.4 Orchestration and Containers

Containers are an OS-level virtualization technique providing software bundling and resource isolation with low overhead [8]. The performant nature of containers and their deployment flexibility make containers interesting in the ICS context [6]. Docker is one of the most well-known container solutions [30, 31]. Docker has experimental support for checkpointing using Checkpoint/Restore In Userspace (CRIU) [32]. CRIU is a Linux software for checkpointing to disk [33].

Orchestration is a term commonly associated with the automated management of containers. For example, cloud service providers utilize a combination of containers and orchestration for elasticity, i.e., scaling resources to match current needs and handling failures [34]. Kubernetes (K8s) is one of the most well-known container orchestration systems [35].

11.3 Related Work

One of the goals of this work is to explore existing research on state replication in the ICS context, with a focus on the mechanisms used for transferring state data. Table 11.2 lists the identified publications.

Of these works, only a few address redundancy directly. Stattelmann et al. evaluate different checkpointing approaches to reduce data size but do not discuss the protocols used for transferring state data [20]. Stój proposes a state-machine-based hot-standby solution using a non-redundant controller [36], and Zhao et al. describe a redundant architecture [37]. Hegazy et al. present automation-as-a-service with redundancy [3], and Goldschmidt et al. present a container-based architecture that briefly touches on redundancy [6]. Since security is fundamental for ICS, Ma et al. discuss security challenges in redundant controller architectures [38]. These works cover controller redundancy to varying degrees, but none dive deep into state transfer mechanisms.

Johansson et al. propose a distributed architecture to avoid overloading a backup that serves multiple primaries with state data [39]. However, they do not evaluate the performance of the underlying state transfer protocol. Bakhshi et al. propose an architecture for persistent, fault-tolerant state storage for stateful containers in the context of industrial robotics [40]. They use distributed storage and Raft for consistency, but do not evaluate the performance of the underlying state transfer protocol. Nouruzi et al. also focus on mobile industrial robotics and propose an architecture with redundant navigation modules, but do not detail the state replication mechanisms [41].

Another goal of this work is to study checkpointing and state replication mechanisms used in containerized applications, focusing on the mechanisms used to transfer state data to learn if the found approaches suit our redundancy use case. Table 11.4 summarizes the findings, i.e., related work concerning checkpointing in a containerized context.

Koziolek et al. address state transfer in the ICS context between Kubernetes-managed containers, aiming to allow software upgrades without interrupting the control application [21]. The interruption-free upgrade is enabled by transferring internal states from the old version to the container running the new version. They utilize OPC UA Client/Server for state transfer, achieving relatively good performance. However, the suitability of OPC UA Client/Server as a state replication protocol for redundancy use cases is not discussed.

Johansson et al. utilize Kubernetes to manage redundant, containerized DCNs. When a failure occurs, Kubernetes automatically restores redundancy and mitigates service degradation [5]. However, they do not detail the mechanisms used for state transfer between the redundant controllers. Leander et al. present a security analysis of a communication link used for standby redundancy purposes [15].

None of the studies above propose concrete solutions for transferring checkpointed state data. In contrast, our work explores the checkpointing

and redundancy literature to determine the suitability of existing protocols for controller redundancy. We then identify a set of desired features for state-transfer protocols in redundancy scenarios and evaluate a selection of candidate protocols against these criteria. Finally, we experimentally assess the most promising protocols and introduce our own solution, which fulfills all desired features and is likewise evaluated through experimentation.

11.4 Checkpointing in the Literature

To gain an understanding of the checkpointing solutions described in the literature and their applicability to our redundancy use case, we conducted literature searches. The following subsections present the search results.

11.4.1 Checkpointing for Controller Redundancy

To find literature covering checkpointing in an industrial controller redundancy context, we searched WebOfScience and Scopus for redundancy-related work targeting checkpointing and state replication in an industrial controller context using the query shown in Table 11.1. We followed references to widen the search, and the relevant literature was added to the list in Table 11.2. Table 11.2 summarizes the found literature and shows each publication's main topic and to what level it covers redundancy, checkpointing, and transfer mechanisms. As seen in Table 11.2, most publications do not discuss checkpointing or the transfer method. The ones that do are further summarized in Section 11.4.1.1 below.

Table 11.1: Controller redundancy checkpointing literature query.

```
("controller" OR "PLC") AND "redundan*" AND
("state" OR "checkpoint*")
```

11.4.1.1 Summary of Identified Papers

Stattelmann et al. discuss a compiler-aided checkpointing mechanism, where data that has changed since the last checkpoint is stored in a dedicated buffer for transfer to the backup [20]. The article states that state data is transferred, but does not describe how. Ma et al. examine how redundant controller constellations are vulnerable to cyber attacks [38]. They argue that redundancy increases the attack surface, and the work discusses the transfer of checkpoint

Table 11.2: Overview of publications related to checkpointing in industrial controller redundancy context.

Ref.	Topic	Redundancy	Checkpoint	Transfer
	Directly identified from literature search			
[42]	Safety	✗	✗	✗
[43]	Deployment	✗	✗	✗
[20]	Data reduction	✓	✓	❍
[38]	Security	✓	❍	❍
[44]	Simulation	✗	✗	✗
[36]	Cost-eff. red.	✓	✗	✗
[45]	Security	✗	✗	✗
[46]	Time sync.	✗	✗	✗
[41]	Cloud-hosted ctrl.	✓	❍	❍
[47]	Deployment	✓	✗	✗
[39]	State transfer	✓	❍	❍
[48]	System red.	✓	✗	✗
	Indirectly identified via reference tracking			
[37]	System red.	✓	✗	✗
[49]	Security	✓	✗	✗
[50]	App. upgrade	✗	✓	❍
[3]	Cloud-hosted ctrl.	✓	✓	✓
[6]	Architecture	❍	❍	❍
[5]	Orch. red. ctrl	✓	❍	❍
[51]	Cloud-hosted ctrl.	✓	❍	❍
[52]	Migration	✗	✓	✓
[53]	Architecture	✗	✗	✗
[54]	Architecture	✗	✗	✗
[55]	Recovery time	✓	✗	✗
[56]	Live migration	✗	✓	❍
[14]	State transfer	✓	❍	✓

Legend: ✓ Detailed, ❍ Mentioned (no technical details), ✗ Not mentioned

data without providing details on the mechanisms used. Stój proposes a cost-effective redundancy approach using a PLC pair but does not address checkpointing or the transfer of state data [36].

Nouruzi-Pur et al. design a cloud-hosted redundant controller for mobile robots, where the mobile robot is responsible for replicating the state between redundant servers [41]. The internal workings and details of the state transfer are not discussed. Johansson et al. address potential network congestion that may occur when one controller serves as a backup for more than one primary [39]. To mitigate congestion at the backup, the checkpointed state data is transferred to a node other than the primary producing the state, though not necessarily the backup itself. The state transfer protocol is UDP-based, but its details are not described.

Zhao et al. present a redundant system with redundant networks and devices but do not address checkpointing or state handling [37]. Luo et al. present a hot standby solution with redundant PLCs (a quad PLC architecture) but do not discuss checkpointing or state transfer [49].

Wahler et al. present a method for bumpless and fast application updates, where a new application version is started on another node [50]. "Teacher" objects in the runtime of the original application gather state data and send it to "Learner" objects in the runtime of the node hosting the updated application. A monitor compares the results of the two application versions. However, the underlying mechanism for transferring changed state data is not detailed.

Hegazy et al. host the controller application in the cloud and store state information on the device to make it accessible to other controllers in the redundant set [3]. TCP and Modbus TCP are used as communication protocols; however, the article does not evaluate these protocols for state transfer purposes. Goldschmidt et al. present an architecture for containerized controllers and identify redundancy-related use cases as important for the architecture to support, but do not describe checkpointing or state transfer in detail [6].

Johansson et al. investigate Kubernetes-based orchestration as a complement or even a replacement to traditional warm standby redundancy [5]. They mention using a proprietary checkpointing mechanism but do not provide details of the protocol. Kaneko et al. present a redundancy solution where controllers are hosted across multiple geographically distributed data centers, across continents even [51]. Neither checkpointing nor transfer mechanisms are discussed.

Gundall et al. propose a live migration approach where state data is continuously sent from the source node to the destination during migration. Once the state data difference between the source and destination nodes is small enough, the handover is initiated [52]. The details of the protocol used to transfer the

state are not presented.

Grüner et al. and Vogt et al. present architectures for flexible control systems but do not discuss redundancy, checkpointing, or state transfer [53, 54]. Barletta et al. measure the recovery time of stateless applications in a Kubernetes context [55]. Stateless applications do not require checkpointing or state transfer, hence, these topics are not covered.

Govindaraj et al. aim to optimize downtime during live migration [56]. They use a request buffer at the destination server to store and replay requests while transferring checkpointed data. However, the details of the transfer mechanism are not discussed.

Kampa et al. discuss and evaluate Remote Direct Memory Access (RDMA) for transferring state data between two virtualized PLCs (vPLCs) in a redundant deployment [14]. They use two types of vPLCs: a homebrewed mockup and a CODESYS vPLC. The CODESYS vPLC originally employs both TCP and UDP for state transfer; Kampa et al. replace this with RDMA [57, 14]. The RDMA-based state transfer introduced by Kampa et al. demonstrates significantly better performance compared to the original TCP/UDP-based approach. The measured average time for transferring 1 MB of data using TCP/UDP is 295 milliseconds over dual 25 Gbps links. In contrast, the theoretical minimum transfer time over a single 1 Gbps link is approximately 8 milliseconds. This notable gap is not addressed in the discussion, nor is the limitation that CODESYS only supports synchronization of data from a single task [57, 14].

As seen in Table 11.2, the number of works addressing checkpointing in the context of industrial controllers for standby redundancy purposes is quite limited. Fifteen of the listed publications, including those on cold standby, discuss redundancy. Of these, only five describe checkpointing mechanisms, and only Hegazy et al. [3] and Kampa et al. [14] discuss the means of state transfer down to the transport protocol used.

11.4.2 Containers and Checkpointing

Inspired by the growing adoption of containers and orchestration in industrial control systems and real-time systems in general, we search for checkpointing mechanisms in the context of containers and orchestration [55, 5, 6, 56]. We search Scopus and Web of Science using the query in Table 11.3.

Since the search aims to determine whether recent technologies and solutions in the container context can be used for or inspire checkpointing mechanisms in an industrial controller redundancy use case, we limit the search to publications from 2018 to 2024, the year of writing this section.

Table 11.3: Checkpointing in container and orchestration context literature query.

```
("checkpoint*" OR "replicat*" OR "restore") AND
("kubernetes" OR "k8s" OR "k3s" OR "orchestrat*"
OR "container*")
```

Table 11.4 presents the result of the search, gives an overview of the relevant literature, and highlights the main topic of each publication, the method used for checkpointing, the transfer mechanism, and whether the work targets a real-time-dependent solution. The following section, Section 11.4.2.1, gives a summary of the found publications.

11.4.2.1 Summary of Identified Papers

As seen in Table 11.4, CRIU is the dominant solution for checkpointing. When it comes to transfer mechanisms, there is no dominant solution. We group the found literature into the following categories: (i) CRIU with Defined Transfer Methods, (ii) CRIU without Transfer Details, (iii) Custom Solution with Defined Transfer Details, (iv) Custom Solution without Transfer Details, and (v) Consensus Protocol-based Solutions.

CRIU with Defined Transfer Methods: These works utilize CRIU and describe the transfer mechanism. Starting with the work that uses a file transfer protocol. Afshari et al. use CRIU to checkpoint application states and compare the performance between File Transfer Protocol (FTP) and Secure Shell (SSH) when transferring the checkpointed file to the destination node [63]. FTP is quicker, and the transfer times are within the second range for the smallest checkpointed data. FTP is also used by Droob et al., who optimize the number of checkpoints to minimize the performance impact of the checkpointed service while providing fault tolerance [82]. Pu et al. migrate applications if they believe the Quality of Service (QoS) will improve by doing so; they use Secure Copy Protocol (SCP), which utilizes SSH [67].

Chebaane et al. use CRIU checkpointing and Remote Sync (rsync) to offload critical tasks from the device to fog or edge, in a Vehicle-to-Infrastructure (V2I) use case [75]. They use rsync to transfer the checkpointed file. Qiu et al. use rsync on top of Multipath TCP (MPTCP) [80]. MPTCP is a protocol that provides multihoming TCP transfer, that is, multiple paths between communication endpoints [105]. Guitart et al. also move the CRIU checkpoint file using rsync [99].

Widjajarto et al. measure the resource utilization when using CRIU for

Table 11.4: Overview of container checkpointing publications.

Ref.	Topic	Method	Transfer	Real-time
[58]	Migration	CRIU	❍	✗
[59]	Migration	CRIU	ZFS	✗
[60]	Fault tolerance	Incremental	✗	✗
[61]	Recovery time	Custom K8s service	HTTP	✗
[62]	Recovery time	Custom K8s service	HTTP	✗
[63]	Migration	CRIU	FTP, SSH	✗
[64]	Contention	CRIU	MOSIX (TCP)	✗
[65]	Contention	CRIU	✗	✗
[66]	Fault tolerance	CRIU	File share	✗
[67]	Migration	CRIU	SCP	✗
[68]	Migration	CRIU	SCP	✗
[69]	Fault tolerance	Custom	Raft	Robotics
[70]	Fault tolerance	Key-value store	NFS	✗
[71]	Fault tolerance	Apache Kafka	❍	✗
[72]	Utilization	CRIU	❍	✗
[73]	Storage	DB, etcd	❍	ICS
[74]	Migration	CRIU	❍	✗
[75]	Migration	CRIU	rsync	V2I
[76]	Migration	CRIU	❍	✗
[77]	Migration	CRIU	❍	✗
[78]	Forensics	CRIU	No transfer	✗
[7]	App. upgrade	Custom	OPC UA CS	ICS
[79]	Migration	CRIU	❍	✗
[80]	Migration	CRIU	rsync	✗
[81]	Contention	CRIU	No transfer	✗
[4]	Fault tolerance	CRIU	FTP	✗
[82]	Fault tolerance	CRIU	FTP	✗
[83]	Fault tolerance	CRIU	DRBD	✗
[84]	Fault tolerance	Custom	No transfer	✗
[85]	Fault tolerance	Custom	❍	✗
[86]	Fault tolerance	CRIU	❍	✗
[87]	Fault tolerance	CRIU	❍	✗
[88]	Fault tolerance	Custom	No transfer	✗
[89]	Fault tolerance	Custom	Raft	✗
[90]	Migration	Custom (CRIU)	❍	✗
[91]	Migration	CRIU	NFS	✗
[92]	Storage	Custom	❍	✗
[93]	DB persistence	CRIU	No transfer	✗
[94]	Migration	CRIU	NFS	✗
[95]	Migration	CRIU	❍	✗
[96]	Fault tolerance	Custom	❍	✗
[97]	Migration	CRIU	❍	✗
[98]	Migration	CRIU SR-IOV	NFS	✗
[99]	Migration	CRIU	rsync	✗
[100]	Migration	CRIU	❍	✗
[101]	Fault tolerance	CRIU	❍	✗
[102]	Migration	CRIU	❍	✗
[103]	Migration	File replication	❍	✗
[104]	Fault tolerance	CRIU	❍	✗

Legend: ❍ Mentioned (no technical details), ✗ Not mentioned

migration [91]. The checkpointed data is copied with a file copy using Network File System (NFS). Mangkhangcharoen et al. compare CRIU and Distributed MultiThreaded CheckPointing (DMTCP) for checkpointing machine learning applications for migration purposes [94]. NFS is used to transfer the checkpointed data. Prakash et al. use CRIU with single root-input/output virtualization (SR-IOV) to reduce the CPU overhead induced by handling virtual networks [98]. The checkpointed data are transferred using NFS.

Bhardwaj et al. utilize the distributed file Z File System (ZFS) to distribute the checkpointed data [59]. Zhou et al. optimize CRIU and use the Distributed Replicated Block Device (DRBD) to replicate the checkpointed files [83, 101]. Adhipta et al. address shared resource contention when checkpointing, since the checkpointing process requires processing and storage resources [64]. The file is stored on the distributed file system MOSIX [106].

CRIU without Transfer Details: Below are the works that use CRIU but don't detail the checkpointed data transfer. Khan et al. use CRIU for migration in a V2I use case but do not detail how the data is transferred [58]. Müller et al. propose a Kubernetes-based architecture for fault tolerance of stateful applications [66]. Checkpointed data is stored on persistent storage, but the storage details and the transfer of the data to the storage are not detailed.

Ramanathan et al. improve CRIU to handle migration of network connections better [74, 77]. The actual transfer mechanism of the checkpointed state is not described. Ngo et al. propose a delta identifier to reduce the data transferred, but the mechanism for the transfer is not presented [76]. Karhula et al. use checkpointing in a Function as a Server (FaaS) context to save resource utilization by checkpointing and suspending the containerized application that provides the function while the application is waiting for the next job [72]. Lee et al. use CRIU for checkpointing in memory databases [93].

Stoyanov et al. compare different checkpointing methods, and Li et al. use CRIU in a Kubernetes context to checkpoint Virtualized Network Functions (VNF) for migration [79, 95]. Bhardwaj et al. compare the checkpointing performance between containers and virtual machines [97]. Di et al. develop a tool for migrating containers using CRIU [100]. Oh et al. propose a CRIU-based application transparent migration [102]. Schmidt et al. introduce a Kubernetes operator for transparent checkpointing using CRIU in the Kubernetes context [104]. Gharaibeh et al. use checkpointing for forensics purposes, that is, troubleshooting or investigating suspected attack attempts [78].

Venâncio et al. use CIRU to checkpoint and go through different VNF redundancy deployments [86, 87]. Some VNF deployments discussed replicate the checkpointed data to a central database, while others use a dedicated state replicator to distribute state data to replicas.

None of these works detail the mechanisms used for transferring the checkpointed data.

Custom Solution with Defined Transfer Details: The below works are the works that define a custom checkpointing solution and also describe the transfer mechanism used.

Arif et al. describe a checkpointing solution for FaaS, using a key-value storage hosted on an NFS [70].

Koziolek et al. introduce a Kubernetes operator for application updates [7]. The updated states are sent from the old version of the application to the new version using the OPC UA Client-Server (OPC UA CS) protocol. This work does not target redundancy, but the use case is similar; the changed application states are transferred in the execution slack between two invocations of the same task.

Vayghan et al. introduce a custom Kubernetes controller for quicker failure recovery and a Kubernetes service for state replication between the stateful applications over HTTP [61, 62].

Custom Solution without Transfer Details: The works listed below present customized checkpointing solutions without describing the mechanisms used to transfer state data.

Zhang et al. present a custom approach where they incrementally store the dirty pages of a Docker container up to a certain threshold, where the remaining dirty pages are checkpointed, to reduce the time the processes are freezed [60]. Venkatesh et al. propose checkpointing to memory instead of disk to boost performance and reduce I/O contention from disk accesses [81]. Yu et al. propose a CRIU optimization that checkpoints to memory instead of disk. The checkpointed data is transferred, but without providing details [90]. Han et al. also address resource contention when storing checkpointed data by utilizing properties provided by the storage, in their case, SSD disk [65].

Junior et al. replicate container file systems between different data centers but do not discuss the communication protocol used [103]. Stavrinides et al. let each task checkpoint its data, but the data is not transferred [84]. Cai et al. replicate to a double buffer; one page of the buffer is replicated, while the other is updated by the application [85].

Choi et al. propose a checkpointing solution called iContainer, and Luati et al. optimize storage of checkpointed data using a distributed storage [88, 92]. Behera et al. propose a predictive checkpointing solution for High-Performance Computing (HPC) [96]. Jia et al. propose a custom mechanism for checkpointing, where the states are replicated amongst peers [4]. However, the transfer protocol is not detailed.

Denzler et al. compare different architectures for persistent storage

for stateful, containerized applications but do not detail the underlying protocols [73].

Consensus Protocol-based Solutions: Below is the literature that describes solutions that utilize consensus protocols to distribute the checkpointed state.

Bakhshi et al. simulate fault-tolerant persistent storage and analyze the performance of their proposed fault-tolerant persistent storage used for replicated, stateful applications [107, 69]. A storage handling container is responsible for replicating the data amongst all other nodes, using Raft [29]. Netto et al. also use Raft in a Kubernetes context to replicate requests in an orderly manner to the replicas, providing active redundancy [89].

Javed et al. use Apache Kafka to replicate the data produced amongst different processing nodes exemplified in a camera surveillance use case [71].

11.4.3 Conclusions from the Literature Search

The search for literature covering checkpointing solutions in the context of industrial controller redundancy reveals that only one work focuses on the details of state transfer—namely, Kampa et al. and their use of RDMA in a vPLC setting [14]. They use CODESYS as the redundant PLC, which is limited to state transfer from a single task [57]. Furthermore, reliability-related aspects such as packet loss and recovery are not considered.

The literature search for checkpointing solutions in container and orchestration contexts reveals a significant amount of work, as shown in Table 11.4. The majority of the work uses CRIU for checkpoints. A file transfer, in one form or another, is the most common alternative for transferring the checkpointed data.

The work by Koziolek et al., like ours, stems from the ICS context, and the replication of state in application slack time is similar to the need of our redundancy use case, as described in Section 11.2 [7]. They use OPC UA Client/Server as the communication protocol to transfer the collected state data, which is performant enough for the use case they address.

Conclusions:

- Detailed state transfer works targeting ICS redundancy are scarce, especially work considering protocol reliability aspects such as packet retransmissions.
- Container-based work favors CRIU plus file transfer, with limited discussion of real-time properties.

- We found no comparative evaluation for transferring checkpointed state in ICS redundancy (or generally).

As mentioned, none of the found literature compares protocols for transferring the checkpointed data, neither in an ICS redundancy use case nor in general. This finding motivates **Step II**, Section 11.5, where we define and describe features desirable for a state-transfer protocol, against which we match relevant protocols, followed by the experimental evaluation of top protocol candidates in **Step III**, Section 11.6.

11.5 Existing Protocols – Feature Matching

The results from the literature search in Section 11.4 show that there are few available works related to protocols for exchanging state data, particularly in the context of industrial controller redundancy. Motivated by that finding, this section aims to identify suitable protocols for that purpose.

TCP and UDP are the two most widely used transport-layer protocols. The common perception is that TCP is reliable but unsuitable for real-time use; however, what does the existing literature say? In Section 11.5.1, we search for literature comparing the performance of TCP and UDP to address that question as a first substep.

As a second substep, we present three features that are highly desirable for a protocol used for controller redundancy state transfer. We match these features against a set of protocols to evaluate the protocol's suitability for the state transfer use case, which is the primary focus of this step, to identify suitable protocols for the redundancy state transfer use case.

11.5.1 TCP and UDP Comparison

Similarly to how we retrieved the checkpointing-related literature in Section 11.4, we turn to Scopus and Web of Science with the query in Table 11.5 to retrieve literature related to TCP and UDP performance in a real-time context.

Table 11.5: TCP and UDP performance comparison literature query.

```
"tcp" AND "udp" AND ("real-time" OR "real time")
AND "performance" AND "evaluation"
```

Table 11.6 presents the remaining publications after filtering out those not explicitly comparing UDP and TCP. The Topic column shows the focus of

each study (i.e., the targeted challenge), and the Packet Delivery Ratio (PDR) column indicates which protocol (UDP or TCP) was found to have the highest PDR, defined as:

$$PDR = PacketsReceived \div PacketsSent \tag{11.1}$$

The throughput (Tput) column lists the protocol that achieved the highest measured throughput under the measurement conditions, and the latency column indicates the protocol with the lowest measured latency. The Network column specifies the type of network used (e.g., simulated, wired, or wireless).

Table 11.6: TCP and UDP performance comparison.

Ref.	**Topic**	**PDR (highest)**	**Tput (highest)**	**Latency (lowest)**	**Network**
[108]	Dist. RT systems	-	-	UDP	Wired
[109]	Video streaming	TCP	TCP	UDP	Simulated
[110]	Voice streaming	TCP	-	UDP	Wired
[111]	Video streaming	TCP	UDP	-	Simulated
[112]	Congestion ctrl.	-	-	UDP	Simulated
[113]	Voice streaming	TCP	-	UDP	Simulated
[114]	Vehicle comm.	TCP	-	UDP	Wireless
[115]	Microgrid ctrl.	TCP	UDP	UDP	Simulated
[116]	Long-dist.	TCP	TCP	-	Wired
[117]	Session init.	-	UDP	UDP	Simulated

As seen in Table 11.6, three studies did not measure PDR [108, 112, 117], but among those that did, TCP exhibited the highest PDR. TCP is considered more reliable due to its congestion window (CWND) management, receiver window (RWND) flow control, and retransmission of lost packets [118]. In contrast, all studies that measured latency found that UDP offers lower latency. Regarding throughput, three studies favor UDP [111, 115, 117], while two favor TCP [109, 116].

The result suggests that the optimal choice for throughput depends on the specific usage scenario. Using UDP without congestion or flow control mechanisms may risk resource exhaustion, leading to increased packet loss and reduced throughput. Overall, these results confirm that TCP provides reliability, whereas UDP offers lower latency. A state replication protocol should be low-latency, reliable, and deliver high throughput. Additionally, it must be secure, as discussed in Section 11.5.2.3. Hence, we further explore the desired features of a protocol for transferring state data.

11.5.2 State-Transfer Protocol Features

The industrial controller redundancy use case presents challenges that we translate into a set of desired protocol features aimed at addressing these challenges. The following subsections elaborate on and justify these features, emphasizing why these features are desirable for our redundancy state transfer use case.

TSN and similar standards and technologies offer low latency and network resource reservation [2]. However, relying on specific technologies can limit deployment and complicate life cycle management, especially in DCS installations, which may operate for over 40 years [119]. It is, therefore, desirable that the protocol only depends on widespread technology and lower-layer protocols that are part of most modern operating systems' network stack. In other words, the protocol should not depend on niche or fringe technology. Protocols that do not meet this platform-agnostic prerequisite are excluded; relevant protocols omitted for this or other reasons are described in Section 11.5.7.

We divide the desirable features into three different categories: (i) Reliability, (ii) Real-time, and (iii) Security, all of which are essential for a protocol used for transferring state data from primary to backup for redundancy purposes. In addition, we use a three-graded feature fulfillment scale, (i) Absent, (ii) Partly, and (iii) Fully, when listing the protocol fulfillment grade in Table 11.12. Where absent means that there is no support. Partly signifies that the feature is only met under restricted conditions, or via optional profiles/adjacent layers, or similar. Phrased differently, partly indicates that the feature can be provided by the protocol to some degree, but not entirely. Fully means that the feature is fully supported.

Protocols are not static; they evolve (with varying degrees), hence, the feature matching provided in this work might not hold true for future protocol versions. Therefore, the feature matching sections refer to the specifications used to determine the fulfillment grade. We do not consider different implementation variants or potential deviation/customization, only the specification.

11.5.2.1 Reliability

The reliability-related features address the protocol's robustness and fault tolerance. The size of the state data produced by checkpointing varies with the application, ranging from a few kilobytes to megabytes [7, 14]. When a large state is segmented into multiple Ethernet frames, the loss of a single frame should not result in a failed transfer, as that could lead to the backup lacking the latest state. Therefore, the protocol should include a mechanism for recovering lost segments, i.e., a retransmission mechanism providing reliable

delivery of state data. We denote this desired reliability feature, Reliable Delivery (*Rel_RD*). An ordered delivery and a recovery mechanism are needed to fully fulfill the *Rel_RD* feature.

Frame loss can result from disturbances or overfull queues and buffers on the network or the receiver. Flow control mitigates frame loss due to overfull receiver queues [120], regulating the data flow from sender to receiver so that the receiver's buffers are not exhausted. Congestion control mechanisms address network overutilization. Network overutilization can lead to frame loss when bottleneck links receive more traffic than they can handle. Congestion control aims to adjust the sending rate to avoid overloading bottleneck links. Although many congestion control algorithm variants exist, they typically share the common principle of reducing the send rate when congestion is suspected [121]. Congestion control algorithms commonly suspect congestion when acknowledgments are missing or arrive too late. Such dynamic congestion control complicates throughput prediction and makes it harder to accurately foresee the transfer time, as the send rate may vary. This issue is discussed further in Section 11.6.

Given the above, a desirable feature for protocol robustness is a mechanism for managing the receive buffer to reduce the risk of packet loss due to exhausted receiver capacity. We denote this feature as *Rel_RC*. The protocol should also include a mechanism to prevent packet loss resulting from overutilization of network capacity, denoted as *Rel_NC*. Table 11.7 provides an overview of the reliability-related features.

We consider *Rel_RC* fully fulfilled if the protocol includes a mechanism specifically designed to prevent receiver buffer exhaustion. Similarly, we consider *Rel_NC* fully fulfilled if the protocol has a mechanism specifically designed to address network overutilization. *Rel_RC* and *Rel_NC* are partly fulfilled if the protocol includes a feature that achieves a similar result, even if it is not primarily designed to address these specific needs.

Table 11.7: Desired reliability-related features.

Identity	**Description**	**Motivation**
Rel_RD	Reliable Delivery	Tolerance to transient faults
Rel_RC	Receiver Capacity	Avoid loss due to buffer exhaustion
Rel_NC	Network Capacity	Avoid overutilization-induced data loss

11.5.2.2 Real-time

As described in Section 11.2.3, state data must be available at the backup within a bounded time to ensure that it can assume the primary role without outputting outdated data. Therefore, the worst-case transfer time, including retransmissions, must be predictable, preferably low, and, as mentioned, bounded. Hence, motivating the desired feature denoted *RT_PT* - predictable and bounded transfer time.

Given a bounded transfer time and a known application period, it becomes possible to define an expected reception interval. This enables the receiver to detect when new data has not arrived within the anticipated timeframe. Ideally, the protocol itself should handle this monitoring, thereby relieving the application of this responsibility. This capability is represented by the update expectancy feature, denoted *RT_UE*. Section 11.2.2 explains the time span until state data invalidation.

Section 11.2 also explains that a controller may run applications with varying execution periods and state sizes. For example, a controller might host both a small application with a short cycle time and a larger one with a longer cycle time. In such cases, the state transfer for the smaller application should not be delayed by the state transfer induced by the larger one, as this could result in the state not being transferred within the application period. To address this, a prioritization mechanism is desirable, hence motivating the desired feature *RT_PR*.

Table 11.8 provides an overview of the real-time related features described above.

Table 11.8: Desired real-time features.

Identity	**Description**	**Motivation**
RT_PT	Predictable transfer time	Bounded transfer time given bandwidth usage
RT_UE	Update time expectancy	Backup receives state data within period
RT_PR	Prioritization	Long-period transfers must not block shorter ones

11.5.2.3 Security

State data may contain sensitive information, such as internal control application variables. Undetected alteration of state data may cause a backup device

Table 11.9: Desired security features.

Identity	Description	Motivation
Sec_Int	Data integrity	State data cannot be altered without detection
Sec_Auth	Data authenticity	Origin of the data can be verified
Sec_Conf	Data confidentiality	State Data cannot be read by unintended receiver
Sec_Fresh	Data freshness	State data cannot be replayed at a later time without detection

to obtain a false view of the state, which at failover can result in unexpected, faulty behavior of the new primary, including incorrect setting of I/O variables. When state data is being transferred over a shared network, protection mechanisms for the protocol should be included.

In a previously conducted security analysis of a redundancy link for state transfer [15], protocol-level mitigations, as described in Table 11.9 as desired security features, should be supported to provide necessary protection against malicious actors.

A protocol for state replication should have a possibility to support these mitigating mechanisms. The required mechanisms and the strength of the mechanisms may, however, vary based on application-specific requirements, e.g., the expected security level of the IEC 62443 standard [122] to be fulfilled.

The most straightforward way to provide the protocol-level security features would be to encapsulate the state transfer protocol within a security protocol on a lower level, e.g., utilizing Transport Layer Security (TLS) for stream-based protocols, or IPsec or Datagram Transport Layer Security (DTLS) for packet-oriented protocols.

Another approach is to use a standard protocol and apply security features to the payload using various post-protocol-stack mechanisms on the application layer.

IPsec can be used for providing security services on the internet layer, implying that the protective mechanisms will only be from node to node, not from application to application, i.e., it will only give assurance if the communicating nodes are trusted. IPsec provides services for integrity, authenticity, and confidentiality, as well as replay protection.

IPsec is often used in *tunneling mode*, forming Virtual Private Networks (VPNs) between networks separated by an insecure network. However, that

use case is not applicable for providing security services to a state transfer protocol; instead, *transport mode* is the appropriate option. Usually, IPsec protocol support is implemented at the OS level and therefore must be configured at the node level. Consequently, applications relying on the security services may have limited opportunities to enforce or verify that the measures are actually in place.

IPsec in transport mode does not support broadcast or multicast, as it is a point-to-point protocol.

TLS and DTLS are by far the most common security protocols used for providing security services for internet-based communication, denoted (D)TLS when both protocols are implied. Even though named *Transport Layer Security*, (D)TLS is implemented in the application stack, making it part of the presentation layer. (D)TLS provides security services for integrity, confidentiality, and authenticity. TLS is a connection-based protocol that uses a client-server approach and can be run with either single or mutual authentication, which is typically certificate-based. If run in single mode, the client can verify the authenticity of the server, but the server requires additional mechanisms to authenticate the client.

For providing security mechanisms to a state transfer protocol, the suggested approach would be to use mutually authenticated (D)TLS to assure data authenticity. The OPC UA Client/Server is implemented in a manner very similar to how mutually authenticated TLS works. It is worth noting that (D)TLS cannot be added to an existing protocol without adaptation at the application layer. Any protocol used for state transfer that wants the benefits of (D)TLS would need to include some changes, which would have implications on both deployment complexity and execution time.

Similarly to IPsec, (D)TLS cannot support broadcast or multicast traffic.

Post protocol-stack security mechanisms is a method to add the required security mechanisms only for the data, while transporting the data using a standard non-secure protocol. This allows for high flexibility in the security services provided, but may increase the complexity of the implementation. In particular, it is considered a bad practice to implement one's own security protocols, implying that well-known patterns and libraries should be used if adopting this approach.

Secure OPC UA PubSub over UDP is one example of such a post-fix security mechanism, which can provide some of the required security services, e.g., confidentiality, while still supporting UDP multicast on the lower protocol level.

In addition to the well-known security protocol above, some communication protocols and standards we evaluate have security profiles or standard

Table 11.10: Security protocol feature fulfillment.

Protocol	*Sec_Int*	*Sec_Auth*	*Sec_Conf*	*Sec_Fresh*
IPsec	Partly	Partly	Fully	Fully
TLS single auth.	Fully	Partly	Fully	Fully
TLS mutual auth.	Fully	Fully	Fully	Fully
Post-protocol sec.	?	?	?	?
SRTP	Fully	Fully	Fully	Fully
DDS Sec. Spec.	Fully	Partly	Fully	Partly
UASC	Fully	Fully	Fully	Fully
OPC UA SKS	Fully	Partly	Fully	Fully

amendments. If so, it is described for each protocol and summarized, along with the security protocol feature fulfillment, in Table 11.10.

11.5.3 Protocol Selection

The protocols selected are identified from the literature referenced in earlier sections, i.e., in Section 11.4 and Section 11.5.1. In addition, we complemented the list by turning to Google with the query shown in Table 11.11 below.

Table 11.11: Reliable real-time protocol - Google query.

```
reliable real-time data communication protocols
```

We divide the listed protocol into four categories, each with a subsection, as follows. The categories are (i) Transport layer protocols, (ii) Application layer protocols, (iii) Non-standardized protocols, and (iv) Excluded protocols. As the name implies, the transport layer protocol and application layer protocols are protocols described in a standard that fall into either the transport or application layer categories. Non-standardized protocols list protocols described in scientific literature but not standardized. The excluded protocols section lists and motivates the exclusion of the listed protocols from the matching against the desirable features. We match the protocol security with the security protocol from Section 11.5.2.3, which are matched against security features in Table 11.9.

Table 11.12 provides an overview of each protocol's real-time and reliability features. The "Security Integration" column explains how security measures are incorporated, while the "Security Protocol" column indicates the typ-

ical protocols used. If a security protocol is listed under "Security Integration," it means that the use of that specific protocol is mandated.

11.5.4 Transport Layer Protocols

This section presents the desired feature fulfillment of the standardized transport protocol alternatives.

Table 11.12: Protocol feature fulfillment.

Protocol	*Rel_RD*	*Rel_RC*	*Rel_NC*	*RT_PT*	*RT_UE*	*RT_PR*	Security integration	Security protocol prescribed
Standardized transport layer protocols								
TCP	Fully	Fully	Fully	Partly	Absent	Absent	Post-protocol	-
UDP	Absent	Absent	Absent	Fully	Absent	Absent	Post-protocol	-
NORM	Fully	Partly	Fully	Absent	Absent	Absent	Post-protocol	IpSec
RTP	Absent	Partly	Partly	Fully	Partly	Absent	Post-protocol	SRTP (own)
SCTP	Fully	Fully	Fully	Partly	Absent	Partly	Post-protocol	DTLS
QUIC	Fully	Fully	Fully	Partly	Absent	Partly	TLS single auth.	TLS single auth.
Standardized application layer protocols								
DDS	Fully	Partly	Partly	Partly	Fully	Partly	Post-protocol	DDS Security Specification
OPC UA (CS)[1]	Fully	Fully	Fully	Partly	Partly	Absent	Post-protocol	UASC
OPC UA (PubSub)[2]	Absent	Absent	Absent	Fully	Fully	Fully	Post-protocol	OPC UA SKS
Non standardized protocols								
RUDP	Fully	Fully	Absent	Fully	Absent	Absent	Post-protocol	IPsec
RBUDP	Fully	Absent	Fully	Fully	Absent	Absent	Post-protocol	-
PA-UDP	Fully	Fully	Partly	Fully	Absent	Absent	Post-protocol	-
UDT	Fully	Fully	Fully	Partly	Absent	Absent	Post-protocol	-
RUFC	Fully	Fully	Fully	Partly	Absent	Absent	Post-protocol	-
SABUL	Fully	Partly	Partly	Fully	Absent	Absent	Post-protocol	-
Tsunami	Fully	Partly	Partly	Fully	Absent	Absent	Post-protocol	-
Excluded protocols								
AMQP	See Section 11.5.7.1							
COAP	See Section 11.5.7.2							
DCCP	See Section 11.5.7.3							
FASP	See Section 11.5.7.4							
MQTT	See Section 11.5.7.5							
RoCE	See Section 11.5.7.6							
Industrial protocols	PROFINET, EthernetIP, EtherCAT, and ModbusTCP. See Section 11.5.7.7							

11.5.4.1 Transmission Control Protocol - TCP

The Transmission Control Protocol (TCP) was first standardized in the early 1980s [123]. It is a connection-based, reliable protocol that provides an ordered byte stream to its users.

[1] OPC UA CS (Client/Sever) utilizes OPC UA TCP.
[2] OPC UA PubSub utilizes OPC UA UDP.

Reliability features: TCP fully fulfills *Rel_RD* since it provides ordered delivery and detects and retransmits lost data. The receiver advertises the remaining space in its receive buffer, thereby fulfilling *Rel_RC*. Additionally, a TCP node must implement congestion control mechanisms, such as slow start and reduction of transmission rate upon loss detection [123], which means TCP also fulfills *Rel_NC*.

Real-time features: Due to congestion window management and the slow start mechanism, calculating the transfer time for a known data size depends on the Round-Trip Time (RTT), as the congestion window increases upon receiving acknowledgments. Packet loss further decreases the congestion window. Consequently, the transfer time depends on the number of packets lost and when they are lost. Therefore, TCP only partly fulfills *RT_PT*, as further detailed in Section 11.6. There is no concept of data expiration in TCP, meaning *RT_UE* is absent. Additionally, TCP only carries a single stream of data and thus lacks support for prioritization, making *RT_PR* absent. In fact, TCP can suffer from head-of-line blocking, where a lost segment prevents delivery of already received data due to TCP's requirement for in-order delivery [124]. As a result, multiple application layer streams sharing the same TCP connection may experience head-of-line blocking.

Security features: As mentioned, TCP is a connection-oriented byte-stream transport layer protocol that does not provide security features; it relies on post-protocol security. TCP is commonly used with TLS as the post-protocol solution.

11.5.4.2 User Datagram Protocol - UDP

Like TCP, the User Datagram Protocol (UDP) was standardized in the early 1980s [125]. UDP is a connectionless, packet-oriented protocol that delivers individual packets, rather than a byte stream like TCP, to its users.

Reliability features: UDP does not provide retransmission capabilities nor offer mechanisms for managing receiver or network resources to prevent buffer exhaustion. Therefore, *Rel_RD*, *Rel_RC*, and *Rel_NC* are considered absent for UDP.

Real-time features: UDP does not implement congestion control, such as slow start or adaptive sending rates based on acknowledgments. The protocol does not regulate the send rate in any way. Hence, no flow, congestion algorithms, or other UDP-specific mechanisms impact transfer time predictability, and thereby, *RT_PT* is fully fulfilled. UDP does not have any expiration time feature or prioritization capabilities. Thus, *RT_UE* and *RT_PR* are absent.

Security features: UDP is a connectionless transport layer protocol, and

like TCP, it does not provide security features; it relies on post-protocol security. A UDP is packet-oriented, and DTLS is a suitable post-protocol UDP solution.

11.5.4.3 NACK-Oriented Reliable Multicast - NORM

NACK-Oriented Reliable Multicast (NORM) is a connectionless, reliable protocol for bulk data transfer to one or more receivers [126]. NORM supports three categories of data transfer: (i) memory, (ii) file, and (iii) streams. NORM uses Forward Error Correction (FEC) to aid in failure recovery. With FEC, NORM can avoid retransmissions by reconstructing lost data from the error correction information. In addition to FEC, NORM uses Negative ACKnowledgments (NACK) to request the retransmission of lost packets when necessary.

Reliability features: As mentioned above, NORM combines FEC with NACK-based retransmissions to recover from packet loss, thereby fully fulfilling *Rel_RD*. Although NORM does not explicitly exchange receiver buffer capacity information, the sender can announce the size of the data being sent, allowing the receiver to allocate appropriate buffers. Therefore, we categorize *Rel_RC* as partly fulfilled. Like TCP, NORM uses a slow start congestion avoidance mechanism, gradually increasing the transmission rate until packet loss is detected. Since NORM is NACK-based, it uses an explicit message to retrieve round-trip times. The trip times are input to the transmission rate reduction due to packet loss, and since NORM can handle multiple receivers, it gathers the round-trip time from all. Hence, NORM fully fulfills *Rel_NC*.

Real-time features: The congestion control mechanism makes transfer time prediction more difficult, especially since some retransmission timeouts are randomized by design, hence, *RT_PT* is absent. Furthermore, NORM does not have any expectation of timely data updates, nor does it have a prioritization mechanism. Hence, both *RT_UE* and *RT_PR* are absent.

Security features: The protocol specification states that the NORM is compatible with IPsec, at the same time, it recommends application-level integrity [126]. Hence, it has to rely on post-protocol security but does not explicitly mention any other protocol besides IPsec.

11.5.4.4 Real-Time Transport Protocol - RTP

The Real-Time Transport Protocol (RTP) is a connectionless, packet-oriented protocol designed in 1996 for audio and video streaming [127]. Although RTP typically utilizes UDP, it is not limited to UDP. RTP uses the RTP Control Protocol (RTCP) for control [127]. RTCP is also connectionless and usually

operates over UDP. RTCP provides quality feedback, congestion, and flow control for RTP.

Reliability features: RTP targets streaming audio and video, where minor data losses might not significantly impact the experience, and the value of the data diminishes quickly over time. Consequently, RTP does not support ordered delivery or retransmission mechanisms; thus, *Rel_RD* is absent. While RTP does not provide information about receiver-side buffer capacity, its companion protocol, RTCP, offers feedback on packet loss. Send rates can be adjusted based on the packet loss information to reduce loss due to receiver-side buffer overutilization. Hence, RTP partly fulfill *Rel_RC*. Similarly, RTP lacks explicit congestion control mechanisms to prevent network resource exhaustion. However, RTP applications can utilize RTCP's packet loss feedback to reduce send rates and mitigate congestion risks. As a result, *Rel_NC* is also partly fulfilled.

Real-time features: RTP is designed to use UDP and does not enforce rate control, leaving that responsibility to the application; hence, RTP fully fulfills *RT_PT*. RTP utilizes timestamps, allowing an application to determine if the data is too old. However, RTP does not invalidate outdated data; hence, *RT_UE* is partly fulfilled. RTP has no prioritization mechanisms; hence *RT_PR* is absent.

Security features: RTP has a security profile named Secure Real-time Transport Protocol (SRTP) defined in Internet Engineering Task Force (IETF) Request For Comments (RFC) 3711 [128]. Adding SRTP to the RTP is described as a "bump in the stack", i.e., as SRTP resides between the application and the RTP transport layer, in other words, post-protocol from the view of RTP. SRTP provides message authentication, a receiver can verify that the sender is likely to originate from the claimed sender, hence fulfilling *Sec_Auth*. RSTP also describes integrity handling, confidentiality mechanisms, and replay detection prevention, fulfilling *Sec_Int*, *Sec_Conf*, and *Sec_Fresh*.

11.5.4.5 Stream Control Transmission Protocol - SCTP

Stream Control Transmission Protocol (SCTP) is a connection and message-oriented transport protocol from the early 2000s designed to address wishes not fulfilled by TCP and/or UDP, such as reliable transfer without head-of-line blocking by allowing more than one stream of data over a single connection [129]. SCTP offers reliable data transfer per stream and multi-homing; a node can have multiple IP addresses that SCTP can utilize for fault tolerance by using different paths through the network.

SCTP runs directly on top of IP, as UDP and TCP, but it is not as widely

adopted as TCP and UDP. Linux and VxWorks support it, and third-party drivers exist for Windows [130, 131, 132].

Reliability features: SCTP fully supports retransmissions of lost data and provides ordered delivery; hence, SCTP fully fulfill *Rel_RD*. However, ordered delivery is optional. Like TCP, SCTP fully fulfills *Rel_RC*, since each connection (or associations as SCTP connections are called due to the multi-homing capabilities) has a receiver window representing the receiver's capacity. SCTP has one receiver window, even if the association is multi-home; the smallest announced window sets the limit. SCTP also has a congestion window that is dynamically adapted, as TCP does, to avoid network resource exhaustion-induced congestion, hence fully fulfills *Rel_NC*. There is one congestion window per home, i.e., network path.

Real-time features: SCTP congestion handling is basically that of TCP but capable of handling multiple paths as needed with multi-homing support. Due to that, we use the same arguments as for TCP regarding transfer time predictability, namely that the dynamic congestion window handling complicates the transmission time prediction; hence, SCTP only partly fulfills *RT_PT*. SCTP does not offer any expiration time on data; therefore, *RT_UE* is absent. Although SCTP offers different streams, these streams lack prioritization attribute differentiation. However, an application can prioritize the different streams differently; hence, *RT_PR* is partly fulfilled.

Security features: RFC 3436 describes TLS over SCTP [133], and later RFC 6083 describes DTLS over SCTP [134]. TLS over SCTP has limitations due to SCTP being packet-oriented. Hence, DTLS over SCTP security is a later RFC to address those weaknesses. In other words, the security protocol to use on top of SCTP is optional, hence post-protocol.

11.5.4.6 QUIC - QUIC

QUIC (not an acronym) is a connection-oriented protocol that uses UDP for the actual data exchange, designed by Google to improve HTTPS performance [135]. RFC 9000 describes the core parts of the protocol, and IETF RFC 9002 defines the congestion control [136, 137]. QUIC reduces the connection establishment latency, which can significantly reduce the overall latency in use cases with many short-lived connections. QUIC also provides multiple streams, relieving QUIC from the head-of-line blocking problem.

Reliability features: As mentioned, QUIC is a reliable protocol that provides ordered delivery. Retransmission handles packet losses; hence, QUIC fully fulfills *Rel_RD*. QUIC provides flow management by exchanging information about the receiver side receive buffer capacity. Hence, QUIC

fully fulfill *Rel_RC*. QUIC tries to prevent network resource congestion due to overutilization with congestion control, similar to TCP; hence, QUIC fully fulfill *Rel_NC*.

Real-time features: Using the same argument as for TCP concerning transfer time predictability, the slow start and dynamic congestion handle make it harder to predict; even though the QUIC variant is quicker to recover, we say that *RT_PT* is partly fulfilled. QUIC does not provide any expiration time on data; hence, *RT_UE* is absent. QUIC does not prioritize the streams and the transferred data; that is up to the application. However, since QUIC supports different streams, it provides the foundation for the application to prioritize them; hence, QUIC partly fulfills *RT_PR*.

Security features: QUIC requires TLS. TLS is an integrated part of the protocol, and it uses single authentication, where only the server is authenticated.

11.5.5 Application Layer Protocols

This section presents the desired feature matching of standardized application layer protocols.

11.5.5.1 Data Distribution Service - DDS

The Data Distribution Service (DDS) is a middleware that provides distributed applications with a data-centric publish-subscribe communication model [138]. DDS utilizes a UDP-mappable abstract protocol called Real-Time Publish-Subscribe (RTPS) [139]. DDS also has a specification in beta state on how to map DDS onto TSN capable networks [140].

Reliability features: DDS has mechanisms to resend due to loss and provides ordered delivery; hence, it fully fulfills *Rel_RD*. DDS does not provide an exchange of receiver buffer capacity. However, it can run on top of TCP, which does. Hence, DDS partly fulfills *Rel_RC*. The same reasoning applies to network resource utilization. Therefore, DDS can partly fulfill *Rel_NC*.

Real-time features: DDS does not mandate the underlying transport protocol; the transfer time predictability depends on the protocol used. Hence, we say that DDS partly fulfill *RT_PT*. DDS provides a deadline property for subscribed data, invalidating data if not updated within that period, fully fulfilling *RT_UE*. DDS has a prioritization mechanism, but how well they are adhered to depends on the used transport protocol; hence, DDS partly fulfill *RT_PR*.

Security features: The DDS Security Specification defines a security model for DDS [141]. DDS does not mandate the use of the security spec-

ification; therefore, DDS supports post-protocol security integration. However, the DDS Security Specification should be followed to stay compliant with the specification. The specification describes the handling of all the security-related features. DDS, like OPC UA PubSub, recommends using symmetric keys for real-time data exchange to improve performance. Hence, authentication is provided by controlling the key distribution.

11.5.5.2 OPC UA Client/Server - OPC UA TCP

OPC UA Client/Server (also denoted OPC UA CS for space conservation) is a part of the OPC UA standard [142]. OPC UA Client/Server invokes remote procedures exposed by OPC UA servers [143]. OPC UA Client/Server can utilize an abstract protocol, called the OPC UA Connect Protocol (UACP), for platform- and technology-independent reasons. OPC UA also describes the mapping of OPC UA CS to TCP (OPC UA TCP) and HTTPS (OPC UA HTTPS) as underlying protocols. We assume OPC UA TCP for the desired feature matching.

Reliability features: OPC UA TCP, as the name implies and as described above, uses TCP; hence, the fulfillment of reliability features is the same as for TCP. That is, full fulfillment of *Rel_RD* since TCP handles retransmission, TCP also provides receiver buffer management and thereby OPC UA TCP fulfill *Rel_RC*. TCP also has a mechanism to avoid congestion by over-utilizing the network; hence, OPC UA TCP fulfills *Rel_NC*.

Real-time features: Since OPC UA TCP uses TCP, the transfer time predictability argumentation is the same as for TCP; the dynamic congestion window handling makes predictability harder, especially if losses affect the congestion window, hence OPC UA TCP partly fulfills *RT_PT*. OPC UA provides timestamps. Hence, mechanisms exist to detect outdated data, but it's up to the client to utilize them. Hence, we classify feature *RT_UE* as partly fulfilled. OPC UA Client/Server does provide prioritization mechanisms for how a server should handle subscriptions, which is a step in the right direction. However, TCP does not have any prioritization. As mentioned earlier, TCP also suffers from the head-of-the-line block. Hence, *RT_PR* is absent.

Security features: OPC UA is designed to operate in a very heterogeneous industrial landscape. Hence, it provides a flexible use of security mechanisms, where OPC UA nodes can choose to conform to suitable security profiles [144, 143, 145]. Hence, the security integration is post-protocol; however, the selection is limited to comply with the standard. This description assumes OPC UA TCP secured with OPC UA Secure Conversation (UASC). UASC can fulfill all the desired security features, as shown in Table 11.10.

11.5.5.3 OPC UA PubSub - OPC UA UDP

OPC UA PubSub is an additional OPC UA communication model, and as the name implies, it is a publish-subscribe communication model [146]. OPC UA PubSub supports two broker models, brokerless and broker-based. The broker-based model uses Advanced Message Queuing Protocol (AMQP) or Message Queue Telemetry Transport (MQTT). The broker-less alternative is one that targets real-time exchange, such as that between a device and a controller. It utilizes network equipment for brokering, specifically multicast groups on Ethernet and IP. The brokerless OPC UA PubSub can run directly over Ethernet or UDP, and it supports connectionless and unidirectional communication between publisher and subscriber. There is no mandated communication-related feedback from subscribers to publishers. The UA Datagram Protocol (UADP) specifies the brokerless OPC UA PubSub message format, including its headers and their meanings. We base the feature discussion on OPC UA PubSub UADP, which is built on top of UDP.

Reliability features: OPC UA UDP uses UDP and does not mandate any additional resend mechanism; sequence numbers are optional. Sequence number usage can provide ordered delivery; however, since there is no resend option and no alternative to it, *Rel_RD* is absent. Furthermore, OPC UA UDP does not exchange receiver buffer information; hence, *Rel_RC* is absent. *Rel_NC* is also absent, as there are no additional measures for network resource management and congestion avoidance. It is worth noting that mappings between OPC UA PubSub and TSN have been described, which could then reserve network resources and detect overutilization if used [147].

Real-time features: OPC UA UDP uses UDP with no throttling; hence, the transfer time is predictable and fulfills *RT_PT*. Subscribers to published data can error mark that data if not updated within the expected interval, hence fully fulfilling *RT_UE*. OPC UA PubSub also provides prioritization levels that are mappable onto underlying network prioritization mechanisms such as differentiated service code point (DSCP) in the IP header or the priority code point (PCP) in the Ethernet frame [147]. The standard prescribes that the processing of outgoing data with higher priority should precede that of lower priority, hence fully fulfilling *RT_PR*.

Security features: OPC UA PubSub does not enforce any security. Hence, the security integration is post-protocol from the OPC UA UDP perspective. However, the OPC UA prescribed mechanisms should again be used to ensure compatibility. OPC UA PubSub uses Security Key Service (SKS) to provide keys for signing and encrypting messages [146]. We denote this OPC UA SKS. Keys are distributed based on roles. Hence, authentication is provided

by controlling the key distribution. OPC UA SKS can also fulfill the other desired security features, as shown in Table 11.10.

11.5.6 Non-standardized Protocols

This section presents the desired feature matching on non-standardized protocols found in the literature.

11.5.6.1 Reliable UDP - RUDP

Reliable UDP (RUDP) is a reliable, connection-oriented protocol built on top of UDP, as defined in a draft RFC [148]. RUDP provides reliable and ordered delivery and flow control, but no congestion control.

Reliability features: RUDP fully fulfills *Rel_RD*, i.e., ordered delivery, loss detection, and retransmission of lost packets. It exchanges information about how many outstanding packets are allowed before an acknowledgment must be received, serving as the flow control. RUDP does not have any network over-utilization prevention. Hence, fulfilling *Rel_RC*, but *Rel_NC* is absent.

Real-time features: RUDP has no congestion control and a fixed limit for the number of outstanding packets allowed, serving as a flow control mechanism. Hence, the predictability of transfer time only depends on how many packets are lost, not when they are lost. Therefore, RUDP fulfills *RT_PT*. RUDP has no expectancy update time nor prioritization; hence, both *RT_UE* and *RT_PR* are absent.

Security features: RUDP does not mandate any security protocol. Hence, it is post-protocol. The specification mentions that it is IPsec compatible.

11.5.6.2 Reliable Blast UDP - RBUDP

Reliable Blast UDP (RBUDP) is, as the name implies, a reliable protocol for transferring bulk data that uses UDP to avoid TCP congestion handling to increase throughput [149]. RBUDP uses UDP to send data and is configured with a specific send rate to prevent exceeding the bandwidth capacity of the underlying network. It utilizes a secondary management channel over TCP to communicate information about the transfer, including lost packages.

Reliability features: RBUDP has a resend mechanism, and ordered delivery is ensured with numbered packets, fully fulfilling *Rel_RD*. RBUDP has no mechanism for synchronizing receiver-side buffer capacity; therefore, *Rel_RC* is absent. RBUDP adjusts its sending to a specified send

rate that should be selected so that the underlying network is not over-utilized; hence, RBUDP fully fulfills *Rel_NC*.

Real-time features: RBUDP uses the send rate to avoid overutilizing the network; given the send rate (and the packet size), the transfer time is predictable; hence *RT_PT* is fully fulfilled. RBUDP does not provide any update expectancy mechanism or prioritization means; hence, both *RT_UE* and *RT_PR* are absent.

Security features: RBUDP does not describe any security measures. Hence, security integration must be post-protocol.

11.5.6.3 Performance Adaptive UDP - PA-UDP

Performance Adaptive UDP (PA-UDP) targets bulk data transfer, and the authors argue that there is no need for congestion control on a dedicated link; hence, PA-UDP uses UDP for the data transfer [150]. In addition, PA-UDP also uses a TCP channel to communicate information feedback, such as lost messages. PA-UDP includes a rate control dictated by the receiver, as the protocol targets data transfer in use cases where disk access storing the received data is the limiting factor.

Reliability features: PA-UDP provides ordered delivery and retransmission of lost packets, i.e., fully fulfilling *Rel_RD*. PA-UDP does not explicitly exchange buffer size information, but the sender can limit the send rate if buffer exhaustion is at risk; hence, fully fulfilling what *Rel_RC* is about. Assuming the sender and receiver are aware of the capacity of the underlying link, rate control can serve as a means to avoid overutilizing the receiver and the network, even though it was primarily designed to prevent overutilizing the receiver. Hence, PA-UDP partly fulfills *Rel_NC*.

Real-time features: PA-UDP uses UDP with a receiver-set rate control; hence, predicting transfer time is straightforward. Therefore, PA-UDP fully fulfills *RT_PT*. PA-UDP does not provide any update monitoring or prioritization mechanism. Hence, both *RT_UE* and *RT_PR* are absent.

Security features: PA-UDP does not describe any security measures. Hence, security integration must be post-protocol.

11.5.6.4 UDP-based Data Transfer Protocol - UDT

UDP-based Data Transfer (UDT) is a reliable and connection-oriented protocol designed to more effectively utilize high-speed links by introducing an alternative congestion control mechanism compared to TCP [151]. Specifically, using TCP over high-speed links with long distances and long round-trip

times can reduce throughput. UDT addresses this problem, and as the name implies, UDT utilizes UDP.

Reliability features: UDT fully fulfills *Rel_RD*; it handles out-of-order packets as well as resends lost packets. UDT exchanges receiver side capacity, and the sender adjusts to that, fully fulfilling *Rel_RC*. UDT has a dynamic congestion control to avoid congestion due to overutilization of the network; hence, it fully fullfills *Rel_NC*. The UDT congestion control does not react to just one lost packet, thereby avoiding a lossy link and reducing the congestion window due to disturbance rather than congestion.

Real-time features: UDT has, as mentioned, a dynamic congestion control. We use the same argument as for TCP when it comes to the fulfillment of *RT_PT* for UDT. The actual transfer time depends on when the disturbance occurs, not just on the number of losses. Hence, UDT partly fulfill *RT_PT*. UDT does not provide any update monitoring or prioritization. Hence, both *RT_UE* and *RT_PR* are absent in UDT.

Security features: UDT does not describe any security measures. Hence, security integration must be post-protocol.

11.5.6.5 Reliable UDP with Flow Control - RUFC

Reliable UDP with Flow Control (RUFC) is a connection-oriented protocol designed to be a performant alternative that addresses the underutilization that may result from congestion and flow control [152]. RUFC introduces a layer between the application and UDP for evaluating different control algorithms.

Reliability features: RUFC fully handles retransmission and ordered delivery, fully fulfilling *Rel_RD*. It also supports window management, hence receiver buffer capacity control, and fulfills *Rel_RC*. RUFC has traffic shaping support that can do flow control to adapt to the network capacity; therefore, RUFC fulfills *Rel_NC*.

Real-time features: Rate control is optional when using RUFC, and the paper evaluates different types, and neither is as performant as native UDP. The predictability depends on the rate control used; hence, RUFC partly fullfills *RT_PT*. RUFC does not have any update monitoring mechanism or prioritization. Hence, both *RT_UE* and *RT_PR* are absent.

Security features: RUFC does not describe any security measures. Hence, security integration must be post-protocol.

11.5.6.6 Simple Available Bandwidth Utilization Library - SABUL

Simple Available Bandwidth Utilization Library (SABUL) is a reliable and lightweight protocol with flow and rate control [153] SABUL, like RBUDP

and PA-UDP, uses UDP for data exchange and TCP for acknowledgment and rate control; see Section 11.5.6.2 and Section 11.5.6.3. SABUL transmits a fixed number of packages and then waits for reception information from the receiver over the TCP channel.

Reliability features: SABUL handles retransmission and ordering, and the sender is informed about lost messages after transmitting a fixed amount of packages. SABUL fully handles retransmission and ordered delivery and thereby fullfills *Rel_RD*. Since SABUL transmits a fixed amount of packages, this is a rather simplistic receiver buffer management and network resource management; hence, SABUL partly fulfills *Rel_RC* and *Rel_NC*.

Real-time features: Given the rather simplistic transmission control of SABUL, where a predefined number of packages are transmitted before waiting for acknowledgment, predicting the transfer time is straightforward. The transmission time does not depend on when packages are lost, as for TCP; it only depends on how many are lost. Hence, SABUL fully fulfills *RT_PR*. As mentioned, SABUL is designed to be a lightweight protocol. Hence, it does not support any data expiration properties or prioritization. In other words, both *RT_UE* and *RT_PR* are absent.

Security features: SABUL does not describe any security measures. Hence, security integration must be post-protocol.

11.5.6.7 Tsunami

Tsunami is an application protocol designed to achieve faster file transfer than FTP over TCP by FTP over UDP [154]. Tsunami is an application-layer protocol, and like SABUL and others (see Section 11.5.6.6), Tsunami uses UDP for data transfer and TCP for control data. Tsunami examples of control parameters include transfer rate and delay time between transferred data blocks.

Reliability features: Tsunami provides recovery of lost data and ordered delivery; hence, Tsunami fully fulfills *Rel_RD*. Tsunami does not explicitly exchange receiver buffer sizes, but it exchanges desired transfer rate and delay time, which can be set so that the receiver buffer is not exhausted and the underlying network is not overutilized. Hence, Tsunami partly fulfill *Rel_RC* and *Rel_NC*.

Real-time features: As mentioned, Tsunami uses a rate and a delay to avoid congestion. Hence, the transfer time depends on the amount of data to transfer and the number of lost packages, not when the packages are lost. Hence, Tsunami fully fulfills *RT_PR*. Tsunami do not have any expiration date mechanism nor prioritization. Hence, both *RT_UE* and *RT_PR* are absent.

Security features: Tsunami does not describe any security measures.

Hence, security integration must be post-protocol.

11.5.7 Excluded Protocols

This section lists protocols excluded from feature matching because they are deemed unsuitable for the use case, but are relevant enough to warrant their exclusion.

11.5.7.1 Advanced Message Queuing Protocol - AMQP

Advanced Message Queuing Protocol (AMQP) is an application-layer protocol that supports broker-based publish/subscribe and targets enterprise communication between heterogeneous systems [155].

AMQP is excluded since it is a broker-based protocol that targets heterogeneous exchanges through a broker, rather than the real-time point-to-point transfer of larger data sizes.

11.5.7.2 Constrained Application Protocol - COAP

The Constrained Application Protocol (COAP) is an application-layer protocol that exposes RESTful APIs in a resource-constrained manner, compared to HTTPS, targeting resource-constrained devices [156].

COAP is excluded since it primarily targets lightweight communication for more resource-constrained devices, such as reading samples from battery-powered intelligent sensors, rather than real-time bulk data transfers.

11.5.7.3 Datagram Congestion Control Protocol - DCCP

As the name implies, Datagram Congestion Control Protocol (DCCP) is a datagram-based transport layer protocol that supports congestion control [157]. The motivation behind DCCP is to spare applications using datagram protocols the need for congestion control implementation, as congestion control is highly recommended for Internet-bound traffic [158, 159].

DCCP is excluded due to its limited spread and support in operating systems' network stacks.

11.5.7.4 Fast Adaptive and Secure Protocol - FASP

Fast Adaptive and Secure Protocol (FASP) is a proprietary protocol developed by Aspera, now part of IBM [160, 161], targeting high-speed data transfer over

long distances. FASP overcomes some TCP shortcomings, such as lowered bandwidth utilization with increased Round-Trip Time (RTT).

FASP is excluded since it is an IBM proprietary protocol.

11.5.7.5 Message Queue Telemetry Transport - MQTT

Message Queue Telemetry Transport (MQTT) is a broker-based publish/subscribe protocol targeting resource-constrained devices and, as the name implies, targets the exchange of telemetry data, and by that aspiring to be a lightweight protocol [162].

MQTT is excluded since it's a broker-based protocol that primarily targets the exchange of smaller data sizes rather than a real-time exchange of larger data sizes, such as the application's state.

11.5.7.6 Remote memory access over Converged Ethernet - RoCE

Remote direct memory access over Converged Ethernet (RoCE) is a technology used in data centers for high-speed data transfer, often aided with hardware support [163, 164]. RoCE is Infiniband's network and transport layer encapsulated on top of Ethernet. From RoCE version 2, also UDP over IP is supported, and there exist software versions that do not require hardware support, which has been shown to be a performant alternative for container communication [165].

RoCE and RDMA are excluded since they require OS Kernel and/or hardware support.

11.5.7.7 Industrial Protocols

Industrial protocols are protocols developed for an industrial context. PROFINET, EthernetIP, EtherCAT, and Modbus TCP are four of the most widely used and well-known protocols[166, 167].

These protocols are designed for real-time exchange between a controller and devices. Typically, that means reading sensory values from input devices and providing output values to output devices, in other words, small data sizes. The devices are commonly slave devices, and the controllers are the masters. On top of that, the cyclic exchanged data are often confined to fitting into an Ethernet frame [168, 169].

The industrial protocols PROFINET, EthernetIP, EtherCAT, and Modbus TCP are excluded for the above reasons. Namely, they are not designed to exchange large data sizes, but rather to be performant when it comes to reading and updating smaller data sizes.

11.5.8 Conclusions from Feature Matching

From the desired protocol feature matching, summarized in Table 11.12, we see that no silver bullet protocol exists, i.e., no protocol fully fulfills all our desired features. SCTP, QUIC, DDS, and OPC UA TCP are the protocols that provide the best matches.

DDS and QUIC are the least favorable for our industrial controller redundancy use case compared to SCTP and OPC UA TCP. DDS is a middleware, and OPC UA is the middleware used by industrial controllers; see Section 11.2. Hence, DDS is less favorable since it would add another middleware. QUIC is less favorable than SCTP since SCTP is available in VxWorks and Linux; see Section 11.5.4.5.

As both OPC UA TCP and SCTP only partly fulfill *RT_PT*, Section 11.6 evaluates the transfer times and the predictability of those. OPC UA TCP uses TCP. Hence, the evaluation utilizes TCP, further elaborated in Section 11.6.

Conclusions:

- No single protocol satisfies all features.
- **Top candidates:** SCTP and OPC UA Client/Server (OPC UA TCP).
- The real-time feature *RT_PT* is only partly met, motivating the work in **Part III** (see Section 11.6).

11.6 Existing Protocols – Experimental Evaluation

As shown and motivated in Section 11.5.8, OPC UA TCP and SCTP are the top candidates. OPC UA TCP runs on top of TCP, and as described in Section 11.5.5.2, TCP provides the reliability features as well as being the reason for the fulfillment grade of real-time feature *RT_PT* and *RT_PR*. Hence, to learn if protocols based on TCP, such as OPC UA TCP, are suitable for the state transfer use case, we evaluate TCP instead of OPC UA TCP.

In addition to TCP, we evaluate SCTP, the second top candidate. The following subsections present the evaluation of TCP and SCTP as state transfer protocol candidates, focusing on the real-time properties that are only partly fulfilled, as indicated in Table 11.12.

As previously discussed, virtual controllers are gaining interest, presenting both challenges and opportunities, with real-time performance being one of the main challenges [30]. Therefore, systems requiring hard real-time properties will likely run on a real-time operating system. Where applicable, we use the configuration provided by VxWorks when describing the aspects of TCP

affecting transfer time in Section 11.6.1. In Section 11.6.2, we apply the same approach for SCTP. The analysis of the protocols and their implementations serves as the basis for the experimental evaluation described in Section 11.6.3. This evaluation is followed by a discussion of the results and their implications for our redundancy use case.

11.6.1 TCP in VxWorks

VxWorks version 24.03 and the RFCs supported by the VxWorks network stack are the basis for the TCP description in this section [132]. The section describes the TCP-related RFCs and their implementation in VxWorks.

RFC 793 specifies the protocol and is the RFC referenced in the VxWorks documentation, even though it has been obsoleted by RFC 9293 [123, 170]. RFC 2018 introduces Selective ACKnowledgment (SACK), which allows a receiver to inform a sender about segments it has received in the event of losses [171]. SACK enables the sender to avoid retransmitting segments received by the receiver.

RFC 5681 describes the four control algorithms a TCP implementation should adopt with equal or greater conservatism [172]. These algorithms are (i) Slow Start, (ii) Congestion Avoidance, (iii) Fast Retransmit, and (iv) Fast Recovery. The slow start algorithm regulates the number of bytes in flight, i.e., unacknowledged bytes. Two connection-specific variables control this: the congestion window ($cwnd$) and the receiver window ($rwnd$). The number of unacknowledged bytes must never exceed the smaller of $cwnd$ and $rwnd$.

The allowed growth of $cwnd$ depends on whether $cwnd$ is below or above a connection-specific variable called the slow start threshold ($ssthresh$). The slow start algorithm dictates the growth of $cwnd$ when $cwnd$ is lower than $ssthresh$. VxWorks assigns $cwnd$ an initial value equal to two times the maximum segment size (mss), where mss is 1420 bytes. The initial value of $ssthresh$ is arbitrary; VxWorks sets it to 65,535. Consequently, the slow start algorithm is active when a TCP connection is established in VxWorks, and the implementation increments $cwnd$ by mss for each acknowledgment of newly received data. Algorithm 5 summarizes the description above.

In VxWorks, the $rwnd$ size is set to the receiver buffer size (i.e., the buffer size of the receiving socket), which is by default set to 60,000. Congestion avoidance becomes active when $cwnd$ exceeds $ssthresh$. RFC 5681 describes various methods for increasing $cwnd$ during congestion avoidance; in VxWorks, $cwnd$ is incremented for each acknowledgment of newly received data. Thus, receiving acknowledgments is crucial when $cwnd$ is small, as it permits more data to be in flight.

TCP supports delayed acknowledgments to reduce the overall number of acknowledgments. The rules for delay are as follows: an acknowledgment should not be delayed for more than 500 milliseconds and should be sent for at least every second full-sized segment [172]. In VxWorks, the default delay time is 200 milliseconds, and the system also allows configuration so that an acknowledgment is sent immediately if a segment with the push (PSH) flag is received. The PSH flag indicates that the data should be delivered to the application as soon as possible.

Fast recovery and fast retransmission are often described as two separate algorithms; however, they are two parts of a cooperative process aimed at reducing the time to retransmission in the event of lost segments. If the algorithms mentioned above do not detect the loss, RFC 6298 specifies that the minimum retransmission timeout should be one second [173]. In VxWorks, the minimum retransmission timeout is configurable. The basic principle is that when a receiver gets an out-of-order segment, it should immediately send an acknowledgment for the last in-order segment received rather than delaying the acknowledgment. A sender that receives three duplicate acknowledgments assumes that the likely cause is a segment loss and issues a retransmit. When a segment loss is detected, either by a retransmission timer timeout or by receiving the third duplicate acknowledgment, the $ssthresh$ is updated as shown in Equation 11.2.

$$ssthresh = max(\frac{bytesInFlight}{2}, mss \times 2) \tag{11.2}$$

The $cwnd$ is updated upon segment loss detection, and the new value depends on whether the loss was detected by duplicate acknowledgments (using fast retransmission and fast recovery) or by the expiration of the retransmission timer. The expiry of the retransmission timer sets $cwnd$ to mss, while detecting lost segments via duplicate acknowledgments sets $cwnd$ to the updated $ssthresh$ plus three times mss, reflecting the duplicate acknowledgment limit of three. RFC 5681 describes the details; where alternatives exist, this section describes the VxWorks variant [172]. Algorithm 5 summarizes the behavior.

RFC 6298 describes how TCP should derive the retransmission timer and timeout from round-trip time measurements [173]. However, if the retransmission timeout is lower than one second, RFC 6298 describes that the retransmission timeout should be rounded up to one second. In VxWorks, this minimum retransmission timeout is, by default, one second and configurable. The "round up to one-second" requirement has significance "SHOULD", which means that under some circumstances a deviation might be acceptable; however, the full implications of such deviation must be understood [174]. RFC 6298 also dic-

tates that the retransmission timer should be doubled for every retransmission due to a timeout, and VxWorks follows this rule.

Algorithm 5 TCP congestion control in VxWorks.

▷ Intial variable values.
1: $mss \leftarrow 1420$
2: $rwnd \leftarrow 60000$ ▷ Peer anounced $rwnd$ (60000 bytes)
3: $cwnd \leftarrow 2 * mss$
4: $ssthresh \leftarrow 65535$
▷ For every received acknowledgement of new data.
5: **if** $cwnd <= ssthresh$ **then** ▷ Slow start.
6: $cwnd \leftarrow cwnd + mss$
7: **else** ▷ Congestion avoidance.
8: $cwnd \leftarrow cwnd + ((mss^2)/cwnd)$
9: **end if**
▷ Packet loss congestion window handling.
10: **if** $SegmentLost$ **then** ▷ Segment loss detected.
11: $ssthresh \leftarrow max((bytesInFlight)/2, mss * 2)$
12: **if** $LossDetectedByAcknowledgement$ **then**
13: $cwnd \leftarrow ssthresh + 3 * mss$
14: **else** ▷ Retransmission timeout.
15: $cwnd \leftarrow mss$
16: **end if**
17: **end if**

11.6.2 SCTP in VxWorks

The VxWorks 24.03 SCTP implementation follows the second latest SCTP RFC, RFC 4960 [175, 132]. This section focuses on the aspects of the VxWorks implementation of RFC 4960 that affect transfer time. VxWorks also supports RFC 3873, which describes the management and information base of SCTP, as well as the draft RFC 6458 related to the SCTP socket API [176, 177].

SCTP, in contrast to TCP, is packet-oriented and supports multiple streams. Each packet can contain multiple chunks belonging to different streams. Like TCP, SCTP announces the receiver window in acknowledgments in a field called Advertised Receiver Window Credit (a_rwnd). VxWorks set a_rwnd to the socket receive buffer size, which is, by default, 60,000 bytes.

Like TCP, SCTP can delay acknowledgments. An acknowledgment is sent

for at least every other packet received and should not be delayed more than 200 milliseconds. These are also the default values for VxWorks. The delay time is changeable through a socket option. Like TCP, SCTP requires the receiver to send an acknowledgment immediately if duplicate packets are received. If a packet contains only duplicate chunks, the receiver must send an acknowledgment immediately. If a packet is received where some chunks are duplicates and some are not, the receiver may send an acknowledgment immediately, as VxWorks does. Whenever an acknowledgment is received, the sender updates the $rwnd$ to reflect any change in a_rwnd and the number of bytes acknowledged. In other words, $rwnd$ equals a_rwnd minus the number of bytes still unacknowledged.

SCTP uses a transmission timer called $T3\text{-}rtx$, the SCTP equivalent of the TCP retransmission timer. The $T3\text{-}rtx$ value is calculated based on the round-trip time, as it is for TCP. Before an initial value has been calculated, $T3\text{-}rtx$ is set to $RTO.Initial$ according to RFC 4960 [175]. If the calculated $T3\text{-}rtx$ is less than $RTO.Min$, it should be rounded up to $RTO.Min$. RFC 4960 recommends setting $RTO.Initial$ to three seconds and $RTO.Min$ to one second. These are also the default values in VxWorks.

Since SCTP is packet-oriented, a message might require fragmentation. RFC 4960 states that a sender may support fragmentation, while a receiver must [175]. VxWorks supports sender-side fragmentation, and a message must be fragmented so that smaller chunks fit in an SCTP packet over IP on Ethernet. The largest size packet allowed before fragmentation is needed is determined by the Maximum Transmission Unit (MTU) for the path, which we denote mtu.

SCTP congestion control, as mentioned, is based on TCP congestion control, i.e., RFC 5681 [175, 172]. SCTP includes the TCP optional SACK mechanism with gap acknowledgment blocks. Gap-acknowledged chunks are included in the in-flight data size until they are included in the total cumulative acknowledgment. Another difference is that SCTP supports multihoming; hence, in addition to the receiver-side window $rwnd$, SCTP maintains the congestion control–related variables $cwnd$ and $ssthresh$ for each destination address.

The initial value of $cwnd$ should be no larger than four times the mtu, and the initial value of $ssthresh$ is arbitrary according to RFC 4960 [175]. For VxWorks, $ssthresh$ is set to the peer's receiver window, which is 60,000 bytes, assuming the peer is also a VxWorks node.

The slow-start phase is active when $cwnd$ is less than or equal to $ssthresh$, and during slow start, the sender increases $cwnd$ by the smaller of the number of bytes in-flight that is acknowledged by the received acknowledgment or the

mtu. The $cwnd$ shall only be increased if the current $cwnd$ is fully utilized and if the received acknowledgment increases the cumulative acknowledged sequence number. In other words, acknowledgments of received chunks that are out of order do not increase $cwnd$. The VxWorks implementation does not verify that the $cwnd$ is fully utilized before increasing it.

The congestion avoidance algorithm is active when $cwnd$ is larger than $ssthresh$. A key difference compared to TCP is the use of an additional congestion control variable named $partial\_bytes\_acked$, which is used by the congestion avoidance algorithm. For each acknowledgment that increases the cumulative acknowledgment, $partial\_bytes\_acked$ is increased by the total number of bytes in all the chunks acknowledged by the received acknowledgment. When $partial\_bytes\_acked$ is equal to or greater than $cwnd$, $cwnd$ is increased by mtu and $partial\_bytes\_acked$ is reduced by mtu.

When a packet loss is detected, $ssthresh$ is set according to Equation 11.3.

$$ssthresh = max(\frac{cwnd}{2}, mtu \times 4) \tag{11.3}$$

Depending on how the loss is detected, $cwnd$ is updated slightly differently. If the loss is detected by acknowledgment information, $cwnd$ is set to $ssthresh$, and in case of expiration of $T3\text{-}rtx$, it is set to mtu. The above is how VxWorks handles the slow start and congestion avoidance to comply with RFC 4960, summarized in Algorithm 6 [175].

Similar to TCP, SCTP quickly acknowledges lost data. When a receiver detects lost data, it directly sends an acknowledgment to the sender, making the sender aware of the loss. Correspondingly, when an acknowledgment indicates loss, the sender waits for three acknowledgments that indicate loss before retransmitting the lost data.

11.6.3 Evaluation Setup: TCP/SCTP State Transfer

To test TCP and SCTP performance in the state transfer use case, we developed an evaluation application that transfers a configurable amount of state data and waits for the receiver to acknowledge its reception. Algorithm 7 summarizes the evaluation application. The *Sender* function runs on the sending node (representing the primary), while the *Receiver* function runs on the receiving node (representing the backup); see Figure 11.6.

We measure transfer time for various data sizes using either a Single Connection (SC) for all transfers or a new connection for each transfer, i.e., Multiple Connections (MC). The connection strategy affects the transfer times since $cwnd$ grows with each successful transfer.

Algorithm 6 VxWorks SCTP congestion control.

▷ Intial variable values
1: $mtu \leftarrow 1500$
2: $rwnd \leftarrow a\_rwnd$ ▷ Peer anounced $rwnd$ (60000 bytes)
3: $cwnd \leftarrow min(4 * mtu, max(2 * mtu, 4380))$
4: $ssthresh \leftarrow rwnd$
▷ *pb_acked* is short for $partial\_bytes\_acked$
5: $pb\_acked \leftarrow 0$
▷ For every received acknowledgement.
6: **if** $cwnd <= ssthresh$ **then** ▷ Slow start.
7: **if** $AckAdvancesCumulativeSeq$ **then**
8: $cwnd \leftarrow cwnd + min(ackBytes, mtu)$
9: **end if**
10: **else** ▷ Congestion avoidance.
11: **if** $AckAdvancesCumulativeSeq$ **then**
12: $pb\_acked \leftarrow pb\_acked + ackBytes$
13: **if** $pb\_acked >= cwnd$ **then**
14: $cwnd \leftarrow cwnd + mtu$
15: $pb\_acked \leftarrow pb\_acked - mtu$
16: **end if**
17: **end if**
18: **end if**
▷ Packet loss congestion window handling.
19: **if** $PacketLost$ **then** ▷ Packet loss detected.
20: $ssthresh \leftarrow max((cwnd)/2, mtu * 4)$
21: **if** $LossDetectedByAcknowledgement$ **then**
22: $cwnd \leftarrow ssthresh$
23: **else** ▷ Retransmission (T3-rtx) timeout.
24: $cwnd \leftarrow mtu$
25: **end if**
26: **end if**

Algorithm 7 State transfer benchmark application.

```
1:  function SENDER
2:      sndIterations ← 0
3:      while sndIterations < IterationsToRun do
4:          if connEachIt OR isFirstIt then
5:              socket ←CONNECTTORECEIVER( )
6:          end if
7:          startTime ← GETTIME( )
8:          SENDALLDATA(socket)
9:          WAITFORACK(socket)
10:         elapsedTime ← GETTIME( )−startTime
11:         sndIterations ← sndIterations + 1
12:     end while
13: end function
14: function RECEIVER
15:     rcvIterations ← 0
16:     while rcvIterations < IterationsToRun do
17:         if connEachIt OR isFirstIt then
18:             socket ←ACCEPTCONNECTIONFROMSENDER( )
19:         end if
20:         RECIEVEALLDATA(socket)
21:         SENDACK(socket)
22:         rcvIterations ← rcvIterations + 1
23:     end while
24: end function
```

Additionally, we measure transfer time under different loss conditions. Note that TCP and SCTP use different terminology: TCP refers to segments, while SCTP refers to packets. In the following, we use the term "packet" for both, with the understanding that when referring to TCP, a packet means a segment. We consider three loss scenarios: (i) loss of the first packet, (ii) loss of the last packet, and (iii) loss of middle packets. The rationale behind selecting these cases is explained below.

As described in Section 11.6.1 and Section 11.6.2, transfer times are likely higher when losses are not detected by fast retransmission. For example, if the first packet after establishing a connection is lost and no acknowledgment is received for that packet, the sender is forced to rely on a retransmission timeout. Similarly, if the last packet in a state transfer is lost, the sender must again rely on a retransmission timeout. Therefore, losses of the first and last packets represent two distinct cases. The third case involves the loss of a middle packet, where data is in flight, and fast retransmission and fast recovery mechanisms typically detect and handle the loss. We also simulate an increasing number of lost packets to demonstrate that the retransmission timeout will be triggered if too many packets are lost. Additionally, each time a packet is resent due to a retransmission timeout, the timeout doubles for both SCTP and TCP.

The loss of the first or last frame also serves as a test of edge cases, since there is only one first and one last frame per transfer, whereas there are many middle frames. Loss of several consecutive middle frames simulates a burst-loss. Additional losses may occur due to queue overflow on an overutilized path; however, we consider these as configuration faults that are outside the scope of this work.

Table 11.13 provides an overview of the evaluation cases. The Size column lists the data sizes used, and the Connection column indicates whether SC, MC, or both are evaluated. The First Drop and Last Drop columns indicate whether the test was run with the loss of the first and last packets, respectively; "Both" means tests were conducted both with and without such losses. The Middle Drop column specifies if middle packets, which are neither last nor first, were dropped and how many. Note that first, last, and middle drops are not combined in a single test iteration. Middle packet loss is simulated only for data sizes of 10 KB and above, as the packet size is approximately 1 KB for both SCTP and TCP; hence, larger sizes are needed for middle packets to exist. We run each test for 100 iterations, recording the minimum, maximum, and average transfer times.

As mentioned, VxWorks allows customization of SCTP and TCP-related parameters. Therefore, we evaluate using two configurations per protocol: default parameters and optimized for loss recovery performance, as shown

Table 11.13: TCP and SCTP state transfer evaluation cases.

Size	Connection	First Drop	Last Drop	Middle Drop
128B	Both	Both	Both	No
256B	Both	Both	Both	No
512B	Both	Both	Both	No
1KB	Both	Both	Both	No
2KB	Both	Both	Both	No
5KB	Both	Both	Both	No
10KB	Both	Both	Both	0,1,5,10
25KB	Both	Both	Both	0,1,5,10
50KB	Both	Both	Both	0,1,5,10
100KB	Both	Both	Both	0,1,5,10
250KB	Both	Both	Both	0,1,5,10
500KB	Both	Both	Both	0,1,5,10
1MB	Both	Both	Both	0,1,5,10

in Table 11.14. The minimum retransmission timeout in Table 11.14 corresponds to the minimum retransmission timeout according to RFC 6298 [173], which should be one second for TCP and similarly one second for SCTP by default [175]. The optimized version reduces the minimum timeout to one millisecond. The maximum delayed acknowledgment time defines how long a receiver is allowed to delay an acknowledgment. By default, this delay is 200 milliseconds for both SCTP and TCP; in the optimized configuration, we reduce it to one millisecond. The third parameter we adjust is the limit at which an acknowledgment is forced. By default, this limit is two packets for TCP and SCTP; in the optimized setting, we reduce it to one to ensure that acknowledgments are never delayed.

These optimizations may introduce system-level side effects, e.g., higher CPU utilization due to shorter timeouts. Assessing whether such effects occur and their consequences is left to future work, as is evaluating the generalizability of these settings across implementations and operating systems.

Figure 11.6 illustrates the evaluation setup, where the simulated redundant controller pair connects over a switched 1 Gbps Ethernet with one switch between. DCN 1 represents the primary and runs the sender part of the application, and DCN 2 is the receiver as described in Algorithm 7. The OS on the DCN is VxWorks 24.03, and each DCN is a mini-PC with 2 GHz Intel I7-9700T, with 16 GB RAM.

Table 11.14: VxWorks TCP and SCTP configurations.

Parameter	Default setting	Optimized setting
Min. retransmission tmo.	One second	One millisecond
Max delayed ack.	200 milliseconds	One millisecond
Force immediate ack.	Two packets	One packets

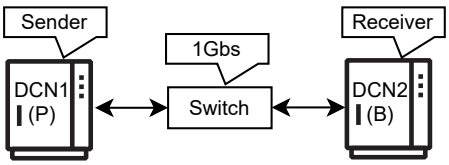


Figure 11.6: The evaluation setup used.

11.6.4 Performance Results: TCP/SCTP State Transfer

Tables 11.15 and 11.16 detail the measured transfer times for the evaluation scenarios summarized in Table 11.13. The theoretical limit for 1 Gbps Ethernet is 125 MB/second, or 8 milliseconds to transfer 1 MB. As shown in Figure 11.7 and Table 11.15, the default TCP transfer with a single connection approaches this theoretical maximum throughput, transferring 1 MB in less than 10 milliseconds, overhead excluded. However, the optimized version's throughput is lower, as depicted in Figure 11.7. Nevertheless, when packet losses occur, the optimized version significantly outperforms the default settings, as presented in Figure 11.9 and detailed in Table 11.15.

For SCTP, as shown in Figure 11.8 and detailed in Table 11.16, the op-

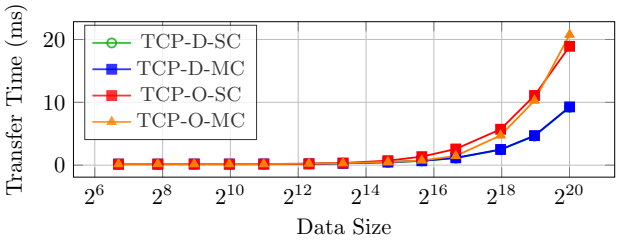


Figure 11.7: TCP - no losses, maximum transfer time, with default (D) or optimized (O) settings and either Single Connection (SC) or Multiple Connections (MC).

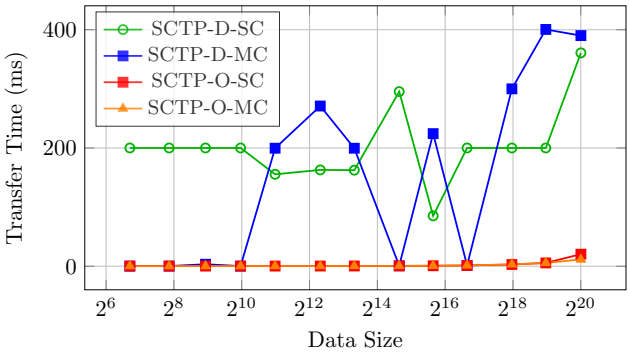


Figure 11.8: SCTP - no losses, maximum transfer time, with default (D) or optimized (O) settings and either Single Connection (SC) or Multiple Connections (MC).

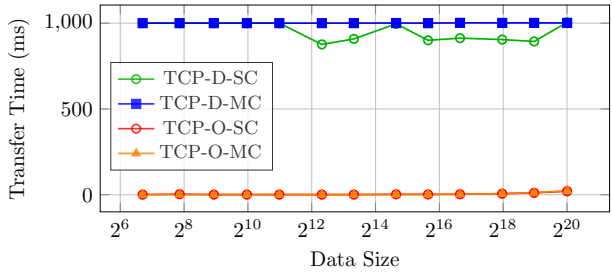


Figure 11.9: TCP - loss of first segment, maximum transfer time, with default (D) or optimized (O) settings using either Single Connection (SC) or Multiple Connections (MC).

timized version offers better performance even in scenarios without packet losses. Generally, SCTP performance is not as good as TCP. However, both the optimized SCTP and the optimized TCP exhibit recovery times exceeding one second; TCP only does so when losing ten packets for 10 kB of data. In other words, it is a relatively extreme loss situation. One potential reason for the lower SCTP performance is the hardcoded SCTP progression tick time of 200 milliseconds.

For TCP and SCTP, the longest recovery time occurs in scenarios with a single connection where ten consecutive packets are lost. This result is due to the cumulative reduction in the congestion window ($cwnd$) caused by repeated packet losses on the same connection and the increased retransmission timeout when resends are triggered by timeout. The results confirm that losing the first or last packets presents significant problems for TCP and SCTP. Al-

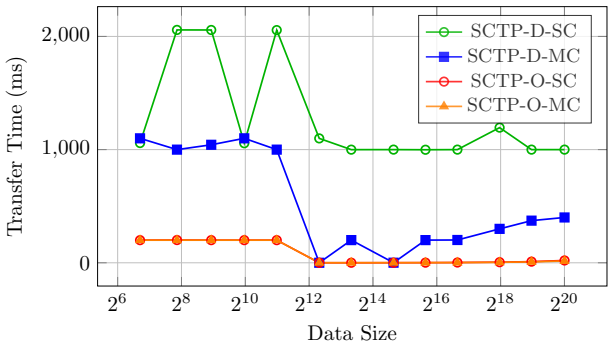


Figure 11.10: SCTP - loss of first segment, maximum transfer time, with default (D) or optimized (O) settings using either Single Connection (SC) or Multiple Connections (MC).

though optimization substantially improves results for both protocols, SCTP still shows a maximum transfer time of around 200 milliseconds when the first or last packet is lost, compared to approximately 20 milliseconds for TCP.

Additionally, the optimized TCP version shows transfer times exceeding 600 milliseconds during scenarios with multiple packet losses, as consecutive losses reduce the $cwnd$ and increase the retransmission timeout.

11.6.5 Conclusions from Experimental Evaluation

With default settings, internal TCP and SCTP recovery mechanisms can extend transfer times by several seconds in the presence of packet losses. However, the recovery times are reducible by applying the optimizations described in Table 11.14. The optimized TCP version's results indicate that it typically requires several consecutive packet losses to affect transfer time significantly.

The likelihood of consecutive packet losses might be acceptably low, especially when redundant networks are employed for state data exchanges. However, TCP does not offer prioritization, and if a single TCP connection is used for all state transfers, one application's data transfer might block another, especially under loss [124]. Additionally, TCP optimizations in VxWorks are implemented at the kernel configuration level, influencing all TCP connections. Preferably, it would be handled as for SCTP on the socket level, only impacting the connection for which the optimization is needed. As mentioned earlier, deviations from the standard one-second minimum timeout should be thoroughly understood, which is easier if only applied to certain connections [174].

Conclusions:

Table 11.15: TCP transfer times (ms) with minimum, average, and maximum values under packet loss scenarios.

Size	No Loss			First pkt. lost			Last pkt. lost			1 mid pkt. lost			5 mid pkts. lost			10 mid pkts. lost		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Default settings - Single connection																		
128 B	0.1	0.1	0.2	888.9	998.9	1E3	899.8	999.0	1E3	–	–	–	–	–	–	–	–	–
256 B	0.1	0.1	0.2	892.9	998.9	1E3	899.8	999.0	1E3	–	–	–	–	–	–	–	–	–
512 B	0.1	0.1	0.2	880.9	998.8	1E3	833.8	998.3	1E3	–	–	–	–	–	–	–	–	–
1 KB	0.1	0.1	0.2	888.9	998.9	1E3	825.8	998.2	1E3	–	–	–	–	–	–	–	–	–
2 KB	0.1	0.1	0.2	998.9	1E3	1E3	949.8	999.5	1E3	–	–	–	–	–	–	–	–	–
5 KB	0.1	0.1	0.2	0.3	9.0	876.0	982.8	999.8	1E3	–	–	–	–	–	–	–	–	–
10 KB	0.2	0.2	0.3	0.4	9.5	908.1	995.8	999.9	1E3	0.5	988.9	1E3	1E3	3E3	3E3	6.3E4	1.2E5	1.2E5
25 KB	0.3	0.3	0.4	0.4	49.5	995.8	917.8	999.1	1E3	0.6	0.7	0.8	994.3	999.9	1E3	1.5E4	6.2E4	6.3E4
50 KB	0.5	0.5	0.7	0.8	10.0	899.9	898.8	999.0	1E3	0.8	1.0	1.1	0.9	990.7	1E3	978.8	1.5E4	1.5E4
100 KB	0.9	1.0	1.1	1.0	10.9	912.5	898.8	999.0	1E3	1.2	1.5	2.4	1.2	1E3	7E3	1.3	3E3	3E3
250 KB	2.3	2.3	2.4	2.3	12.8	904.2	996.8	1E3	1E3	2.7	3.0	4.1	2.5	2.6	3.2	3.1	625.5	3E3
500 KB	4.5	4.5	4.6	4.5	16.0	893.4	893.8	998.9	1E3	5.0	6.4	9.8	4.8	64.2	989.0	4.6	94.3	7E3
1 MB	9.0	9.1	9.2	9.3	23.6	1E3	889.8	998.9	1E3	9.6	12.8	18.7	10.9	21.4	942.9	9.1	1.4E3	1.2E5
Default settings - Many connections																		
128 B	0.1	0.1	0.2	890.9	998.8	1E3	899.9	998.9	1E3	–	–	–	–	–	–	–	–	–
256 B	0.1	0.1	0.2	899.9	998.9	1E3	909.9	999.0	999.9	–	–	–	–	–	–	–	–	–
512 B	0.1	0.2	0.2	975.9	999.7	1E3	889.9	998.8	999.9	–	–	–	–	–	–	–	–	–
1 KB	0.1	0.2	0.2	973.9	999.7	1E3	899.9	998.9	999.9	–	–	–	–	–	–	–	–	–
2 KB	0.1	0.2	0.2	990.0	999.9	1E3	859.8	998.5	999.9	–	–	–	–	–	–	–	–	–
5 KB	0.2	0.2	0.2	987.0	999.9	1E3	899.9	998.9	999.9	–	–	–	–	–	–	–	–	–
10 KB	0.3	0.3	0.3	957.1	999.7	1E3	899.8	998.9	999.9	0.4	0.5	0.5	1E3	1E3	1E3	6.3E4	6.3E4	6.3E4
25 KB	0.4	0.4	0.5	890.5	999.4	1E3	934.9	999.2	999.9	0.6	0.6	0.6	901.3	999.4	1E3	1.5E4	1.5E4	1.5E4
50 KB	0.6	0.7	0.7	802.9	998.0	1E3	901.8	998.9	1E3	0.8	0.8	0.8	0.9	1.0	1.0	999.8	999.9	1E3
100 KB	1.1	1.1	1.2	920.6	999.8	1E3	897.8	998.9	1E3	1.2	1.2	1.3	1.2	1.2	1.4	1.3	1.4	1.6
250 KB	2.4	2.4	2.5	919.2	999.4	1E3	998.9	999.9	1E3	2.6	2.7	3.3	2.7	2.8	2.8	2.6	3.0	3.1
500 KB	4.6	4.6	4.7	915.5	999.6	1E3	999.9	999.9	999.9	5.0	5.0	6.2	5.5	5.7	7.4	5.7	5.7	5.8
1 MB	9.2	9.2	9.3	942.2	999.5	1E3	898.8	999.0	1E3	9.5	9.7	12.6	11.3	11.5	15.5	11.4	11.4	11.5
Optimized settings - Single connection																		
128 B	0.1	0.1	0.1	0.9	1.0	1.0	0.8	1.0	1.0	–	–	–	–	–	–	–	–	–
256 B	0.1	0.1	3.3	0.9	1.0	1.0	0.8	1.0	1.0	–	–	–	–	–	–	–	–	–
512 B	0.1	0.1	0.1	0.9	1.0	1.0	0.8	1.0	1.0	–	–	–	–	–	–	–	–	–
1 KB	0.1	0.1	0.2	0.9	1.0	1.0	0.9	1.0	1.1	–	–	–	–	–	–	–	–	–
2 KB	0.1	0.1	3.3	0.8	1.0	1.0	0.8	1.0	1.1	–	–	–	–	–	–	–	–	–
5 KB	0.1	0.1	0.2	0.2	0.3	1.1	0.8	1.0	1.0	–	–	–	–	–	–	–	–	–
10 KB	0.2	0.2	0.3	0.3	0.4	1.1	0.8	1.0	1.0	0.4	1.0	1.1	1.2	3.0	3.0	99.9	215.3	5.3E3
25 KB	0.3	0.4	0.7	0.4	0.6	1.7	0.8	1.0	1.1	0.6	0.7	0.7	0.9	1.0	1.3	15.4	62.2	63.1
50 KB	0.5	0.8	4.2	0.6	1.0	2.0	0.8	1.9	2.1	0.8	1.0	1.9	0.9	3.3	18.5	1.4	39.9	335.0
100 KB	0.9	1.6	2.5	1.4	1.9	2.7	1.1	1.9	4.1	1.2	1.8	4.0	1.4	2.8	12.0	1.4	57.9	497.1
250 KB	3.3	4.1	5.7	3.3	4.0	6.6	3.4	4.3	6.5	3.1	4.1	6.2	3.1	5.0	22.4	3.2	46.6	645.6
500 KB	6.4	8.0	11.0	6.5	7.8	12.0	6.1	8.4	12.2	6.5	8.0	11.7	6.4	8.5	18.6	6.6	23.9	455.7
1 MB	12.8	14.9	22.4	13.0	15.4	24.3	13.1	15.3	23.1	13.0	16.1	23.4	13.2	15.9	31.0	13.5	60.5	628.3
Optimized settings - Many connections																		
128 B	0.1	0.1	0.2	0.9	0.9	1.0	0.8	0.9	0.9	–	–	–	–	–	–	–	–	–
256 B	0.1	0.1	0.2	0.9	1.0	1.0	0.8	0.9	1.0	–	–	–	–	–	–	–	–	–
512 B	0.1	0.2	0.2	0.9	0.9	1.0	0.8	0.9	0.9	–	–	–	–	–	–	–	–	–
1 KB	0.1	0.2	0.2	0.9	1.0	1.0	0.8	0.9	0.9	–	–	–	–	–	–	–	–	–
2 KB	0.1	0.2	0.2	0.9	0.9	1.0	0.8	0.9	0.9	–	–	–	–	–	–	–	–	–
5 KB	0.2	0.2	0.2	1.0	1.1	1.1	0.9	0.9	0.9	–	–	–	–	–	–	–	–	–
10 KB	0.3	0.3	0.3	1.1	1.2	1.2	0.8	0.9	1.0	0.4	0.5	0.5	1.2	1.9	2.2	63.2	203.8	5.9E3
25 KB	0.4	0.5	0.5	1.6	1.6	1.7	0.8	0.9	1.0	0.5	0.6	0.6	1.3	1.4	1.4	15.4	15.5	18.7
50 KB	0.6	0.7	0.7	1.8	2.0	2.2	0.8	0.9	1.0	0.7	0.8	0.8	0.9	0.9	1.0	2.7	3.6	3.8
100 KB	1.2	1.3	1.4	2.6	2.9	3.3	1.2	1.3	1.4	1.2	1.2	1.3	1.2	1.2	1.3	1.2	1.3	1.4
250 KB	3.2	4.0	5.0	4.7	5.6	6.6	3.3	4.1	4.9	3.2	4.2	5.2	3.2	3.6	5.0	3.1	3.7	4.7
500 KB	6.4	7.7	10.2	7.8	10.2	12.2	6.8	8.2	10.4	6.8	7.7	17.3	6.4	8.1	22.4	6.9	67.3	316.6
1 MB	13.3	15.2	21.6	14.3	18.2	23.6	13.5	15.9	22.8	13.4	15.6	21.5	13.4	15.4	28.1	13.2	36.7	198.7

Table 11.16: SCTP transfer times (ms) with minimum, average, and maximum values under packet loss scenarios.

Size	No Loss			First pkt. lost			Last pkt. lost			1 mid pkt. lost			5 mid pkts. lost			10 mid pkts. lost		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Default settings - Single connection																		
128 B	0.2	197.7	200.0	940.5	999.4	1.1E3	999.9	1.1E3	1.4E3	–	–	–	–	–	–	–	–	–
256 B	0.2	197.6	200.0	942.4	1.1E3	2.1E3	899.5	1.1E3	1.4E3	–	–	–	–	–	–	–	–	–
512 B	0.2	197.6	200.0	939.5	1E3	2.1E3	999.9	1.1E3	1.4E3	–	–	–	–	–	–	–	–	–
1 KB	0.2	197.6	200.0	943.6	999.4	1.1E3	899.6	1.1E3	1.4E3	–	–	–	–	–	–	–	–	–
2 KB	0.2	1.8	155.5	944.2	1.1E3	2.1E3	999.9	1E3	1.1E3	–	–	–	–	–	–	–	–	–
5 KB	0.2	1.9	162.9	0.3	491.0	1.1E3	999.9	1E3	1.1E3	–	–	–	–	–	–	–	–	–
10 KB	0.3	2.0	162.5	0.4	493.0	999.6	999.9	1E3	1.3E3	0.5	0.6	0.6	1.2E3	1.4E3	3.6E3	6.3E4	6.3E4	6.4E4
25 KB	0.6	199.0	295.4	0.7	591.0	999.9	913.6	1.1E3	1.4E3	100.0	199.0	200.0	1.0	1.5E3	1.6E3	3.1E4	6.3E4	6.4E4
50 KB	0.7	1.6	85.2	1.2	491.0	998.8	999.9	1E3	1.3E3	100.1	199.0	200.0	1.4	1.3E3	1.4E3	499.9	6.2E4	6.4E4
100 KB	6.0	196.9	200.0	2.3	540.0	999.9	999.9	1E3	1.2E3	199.9	203.0	400.0	200.0	1.5E3	1.6E3	1.5	6.2E4	6.4E4
250 KB	2.5	96.9	200.0	5.5	548.9	1.2E3	1E3	1.1E3	1.2E3	2.8	140.9	279.1	2.8	201.0	1E3	2.8	1.7E4	3.1E4
500 KB	4.8	100.9	200.0	9.2	535.0	999.9	897.7	1E3	1.4E3	5.3	118.9	380.5	5.2	289.0	1.8E3	6.5	4.9E4	1.8E5
1 MB	9.6	96.9	360.9	22.6	528.9	999.9	845.7	1E3	1.2E3	10.0	108.1	400.0	18.1	309.4	1.4E3	99.8	2.2E5	2.3E5
Default settings - Many connections																		
128 B	0.2	0.2	0.2	999.6	1E3	1.1E3	999.6	1E3	1.1E3	–	–	–	–	–	–	–	–	–
256 B	0.2	0.2	0.2	899.6	998.6	999.7	999.6	1E3	1.1E3	–	–	–	–	–	–	–	–	–
512 B	0.2	0.2	3.4	999.6	1E3	1E3	999.6	1E3	1.1E3	–	–	–	–	–	–	–	–	–
1 KB	0.2	0.2	0.2	999.7	1E3	1.1E3	999.6	1E3	1.1E3	–	–	–	–	–	–	–	–	–
2 KB	73.6	198.3	199.6	899.7	998.7	999.7	999.6	1.1E3	1.4E3	–	–	–	–	–	–	–	–	–
5 KB	199.9	200.7	271.0	0.2	0.3	0.3	999.6	1.1E3	1.4E3	–	–	–	–	–	–	–	–	–
10 KB	61.5	198.2	199.6	0.4	161.4	200.5	999.6	1.1E3	1.3E3	0.4	0.5	0.6	3.2E3	3.5E3	3.8E3	6.3E4	6.4E4	6.4E4
25 KB	0.6	0.6	0.7	0.6	0.7	0.8	899.7	998.7	999.7	200.0	201.0	300.0	0.9	1.0	1.1	1.5E4	3E4	3.1E4
50 KB	0.8	194.5	224.3	1.1	177.4	200.4	899.7	1.1E3	1.2E3	0.9	193.2	200.2	100.0	199.0	200.0	299.6	398.5	399.6
100 KB	1.3	1.4	1.4	2.0	4.2	201.4	899.7	998.7	999.7	1.4	157.1	300.1	1.4	57.3	389.8	1.5	1.5	1.6
250 KB	2.6	97.0	300.0	4.6	144.8	299.8	899.7	1.1E3	1.4E3	2.8	71.0	200.0	2.8	95.0	294.9	2.9	79.0	199.9
500 KB	4.9	103.2	400.5	8.9	111.3	372.3	999.6	1.1E3	1.4E3	5.1	99.3	200.4	5.1	117.2	395.3	5.2	111.3	370.4
1 MB	9.7	103.2	390.1	17.9	101.6	400.4	999.7	1.1E3	1.2E3	10.1	107.8	389.9	10.1	118.5	369.9	10.2	79.2	389.5
Optimized settings - Single connection																		
128 B	0.1	0.2	0.2	41.5	198.4	200.0	99.5	199.0	200.0	–	–	–	–	–	–	–	–	–
256 B	0.1	0.2	0.2	80.5	198.8	201.0	99.5	199.0	200.0	–	–	–	–	–	–	–	–	–
512 B	0.1	0.2	0.2	78.6	198.8	200.0	99.5	199.0	200.0	–	–	–	–	–	–	–	–	–
1 KB	0.2	0.2	0.2	78.5	198.8	200.0	99.5	199.0	200.0	–	–	–	–	–	–	–	–	–
2 KB	0.2	0.2	0.3	77.6	198.7	200.0	99.5	199.0	200.0	–	–	–	–	–	–	–	–	–
5 KB	0.2	0.2	0.3	0.3	0.3	0.4	39.5	198.4	200.0	–	–	–	–	–	–	–	–	–
10 KB	0.3	0.4	0.4	0.4	0.4	0.5	52.5	198.5	200.0	0.4	0.5	0.5	339.8	399.4	400.1	1.5E3	1.6E3	1.6E3
25 KB	0.5	0.5	0.6	0.6	0.7	0.8	123.5	199.2	200.0	0.6	0.7	0.8	0.8	393.7	400.0	300.0	1.6E3	1.6E3
50 KB	0.7	0.7	0.8	1.0	1.2	1.3	78.6	198.8	200.0	0.8	1.2	1.2	1.0	389.1	400.1	1.3	1.6E3	1.6E3
100 KB	1.2	1.3	1.3	1.8	2.2	2.4	175.5	199.7	200.0	1.4	2.0	2.2	1.5	385.6	400.1	1.7	1.6E3	1.6E3
250 KB	2.7	2.8	2.9	4.1	5.1	5.3	173.6	199.7	200.0	3.2	4.6	5.2	3.2	189.6	200.3	3.2	4.5E3	6.4E3
500 KB	5.2	5.4	5.6	7.9	10.0	10.2	192.5	199.9	200.0	5.8	8.7	9.9	5.7	175.8	201.2	6.1	991.9	1.2E3
1 MB	10.6	13.8	20.4	15.8	19.9	20.5	199.9	200.1	209.6	11.4	18.0	20.4	11.8	150.9	202.6	11.4	904.4	1.2E3
Optimized settings - Many connections																		
128 B	0.1	0.2	0.2	99.6	198.6	199.7	99.6	198.6	199.7	–	–	–	–	–	–	–	–	–
256 B	0.1	0.2	0.2	99.7	198.6	199.7	99.6	198.6	199.7	–	–	–	–	–	–	–	–	–
512 B	0.2	0.2	0.2	99.6	198.7	199.7	99.6	198.7	199.7	–	–	–	–	–	–	–	–	–
1 KB	0.2	0.2	0.2	99.7	198.7	199.7	99.6	198.7	199.7	–	–	–	–	–	–	–	–	–
2 KB	0.2	0.2	0.2	99.7	198.7	199.7	99.6	198.7	199.7	–	–	–	–	–	–	–	–	–
5 KB	0.2	0.3	0.3	0.2	0.3	0.3	99.7	198.7	199.7	–	–	–	–	–	–	–	–	–
10 KB	0.3	0.4	0.4	0.4	0.4	0.4	99.6	198.7	199.7	0.4	0.5	0.6	300.0	399.0	400.1	1.5E3	1.6E3	1.6E3
25 KB	0.5	0.5	0.6	0.6	0.6	0.7	99.7	198.7	199.7	0.6	0.6	0.6	0.8	0.8	0.8	300.1	399.2	400.2
50 KB	0.7	0.8	0.8	0.9	1.0	1.1	99.6	198.7	199.7	0.8	0.8	0.9	0.9	1.0	5.7	1.2	1.2	1.3
100 KB	1.2	1.3	1.4	1.7	1.8	1.9	99.7	198.7	199.7	1.3	1.4	1.4	1.4	1.5	4.8	1.6	1.7	1.8
250 KB	2.6	2.8	3.2	4.0	4.1	4.3	99.6	198.7	199.7	2.8	3.1	3.9	3.0	3.1	4.0	3.1	7.2	202.8
500 KB	5.2	5.4	5.6	7.8	8.0	8.2	99.7	198.7	199.8	5.5	5.7	5.9	5.6	5.8	7.6	5.7	6.0	7.0
1 MB	10.5	10.8	11.4	15.4	16.0	16.4	99.6	198.7	199.7	11.1	11.4	15.2	11.2	11.5	11.8	11.2	17.6	409.7

- Utilizing VxWorks TCP/SCTP stack parameters adjustment possibility significantly reduces retransmission latency versus defaults.
- TCP outperforms SCTP in transfer time across the tested scenarios.
- With default parameters, even a single lost packet can push retransmission time into the seconds range.
- With optimization, isolated losses improve, but burst losses still drive retransmission time back into the seconds range.
- The resulting loss-induced tail in transfer time damages predictability, making these protocols ill-suited for deadline-bound state transfer.

11.7 Proposed State-Transfer Protocol

As shown in Section 11.6, numerous protocols exist; however, none of the compared ones meet all the desired features of a state transfer protocol for industrial controller redundancy. This section aspires to design and describe a protocol that provides the desired features. The first subsections describe the protocol, and the final subsection performs a desired feature match of our proposed protocol, as we did for other protocols in Section 11.5.

11.7.1 Protocol Overview

The protocol we propose is named Reliable State Transfer Protocol (RSTP). Similar to RBUDP (see Section 11.5.6.2) and PA-UDP (see Section 11.5.6.3), RSTP comprises a communication protocol for the actual data exchange and an additional mechanism for exchanging related metadata. In the case of RSTP, the RSTP Payload Protocol (RSTP-PP) handles data exchange. The time-insensitive information used by RSTP-PP is managed by the RSTP Management Mechanism (RSTP-MM). It is described as a mechanism rather than a protocol, leaving it open for implementation that suits the specific deployment, as further discussed in Section 11.7.3. Figure 11.11 presents a high-level overview of RSTP, and further details are provided in the two subsections that follow.

11.7.2 Payload Protocol - RSTP-PP

RSTP-PP targets the transfer of internal state data for controller applications in a standby redundancy context. The protocol utilizes a channel concept to

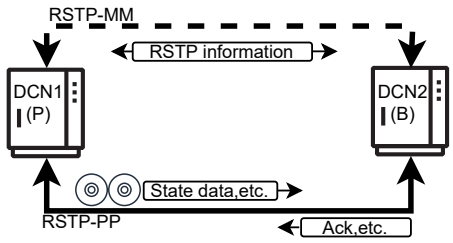


Figure 11.11: RSTP high-level overview.

allow the transfer of each application state to be scheduled and handled independently. The fact that a controller typically hosts multiple applications with different cycle times and state transfer requirements, as previously explained in Section 11.2, motivates the channel concept, further elaborated in Section 11.7.2.2. Typically, an RSTP-PP deployment consists of one sender and one receiver, where the sender is the primary controller sending to the receiver, the backup controller. However, mixing is allowed.

11.7.2.1 Packet Format

RSTP-PP is a packet-oriented protocol, and Table 11.17 shows the data packet header layout. $HType$ and $HVer$ specify the header type and the version of the packet header. $ChId$ is the identity of the channel; the concept of channels is further described in Section 11.7.2.2. Note that the $ChId$ must be unique within an RSTP-PP communication, but the protocol does not specify how to allocate the channel identities.

The next field in the data packet header, as shown in Table 11.17, is $TCycle$, the transfer cycle, a steadily incrementing number for each transfer cycle. A transfer cycle begins when the first packet for a channel is sent and ends when the receiver has acknowledged all packets or the deadline expires. The channel description in Section 11.7.2.2 introduces the concept of deadlines.

Following the $TCycle$ is $SeqNo$, the sequence number within a $TCycle$, and $ChId$. The sequence number starts at zero for every new cycle, and the receiver uses the sequence number to order received packets. The sequence number, combined with the $TtlSzCyc$ and the RSTP-PP payload size, allows the receiver to calculate the byte offset for each incoming packet and store the received data in a contiguous buffer. Section 11.7.3 explains the RSTP-PP payload size.

Table 11.17: RSTP-PP – data packet header.

Name	Bytes	Value	Description
$HType$	2	0x0001	Header and message type.
$HVer$	2	1	Header version.
$ChId$	2	$[1, 2^{16} - 1]$	Channel identity.
$TCycle$	2	$[0, 2^{16} - 1]$	Transfer cycle.
$SeqNo$	2	$[0, 2^{16} - 1]$	Sequence number.
$TtlSzCyc$	4	$[0, 2^{32} - 1]$	Total number of payload bytes in this cycle.
$ExpInMs$	2	$[0, 2^{16} - 1]$	Expiration time, in milliseconds.
$FAckCh$	2	$[1, 2^{16} - 1]$	Force acknowledgement from specified channel.

$TtlSzCyc$ is the total number of payload bytes to transfer in this channel during this $TCycle$; header size excluded. It remains constant throughout the $TCycle$. $TtlSzCyc$ is included in each packet to allow the size to change without additional communication to the receiver. Even if the first message is lost, the receiver can allocate a reception buffer for the whole message.

$ExpInMs$ is the expiration time in milliseconds. The data in the packet is valid for the specified number of milliseconds. The sender updates $ExpInMs$ immediately before passing the packet to the network stack. The receiver should use the shortest expiration time received for the $ChId$ and $TCycle$. Although the time-stamping occurs at the application level, it is deemed accurate enough for the purpose, which is to prevent the backup from using expired data, as described in Section 11.2. If needed, more sophisticated mechanisms can be designed and incorporated, which is deemed future work.

The last field in the header is $FAckCh$, which forces the receiver to acknowledge the specified channel. When a receiver receives a packet where $FAckCh \neq 0$ and $FAckCh$ is a valid channel identity known by the receiver, the receiver must send the current packet reception status for the specified channel. After $FAckCh$, the last field in the header, is the payload data.

Table 11.18 shows the acknowledgment packet layout, and the following describes RSTP-PP acknowledgment fields.

$HType$ and $HVer$ specify the type of acknowledgment and version, similar to the data packet. $TCycle$ and $ChId$ indicate which transfer cycle and channel the acknowledgment is for. Next is $AckBlocks$, which specifies the number of acknowledgment blocks in the acknowledgment. An acknowledgment acknowledging a complete reception of all channel packets for one trans-

Table 11.18: RSTP-PP – acknowledgement header.

Name	Bytes	Value	Description
$HType$	2	0x1001	Header and message type.
$HVer$	2	1	Header version.
$ChId$	2	$[1, 2^{16} - 1]$	Channel identity.
$TCycle$	2	$[0, 2^{16} - 1]$	Transfer cycle.
$AckBlocks$	1	$[0, 150]$	Number (i) of acknowledgment ranges that follow.
$LowAck_i$	2	$[0, 2^{16} - 1]$	Lowest received sequence number for $ChId$ and $TCycle$ in acknowledgement range i.
$HighAck_i$	2	$[0, 2^{16} - 1]$	Highest received sequence number for $ChId$ and $TCycle$ in acknowledgement range i.

fer cycle has an $AckBlocks$ value of one. For each non-consecutive missing sequence number, $AckBlocks$ will increase by one since there will be one more $LowAck_i$,$HighAck_i$ pair in the header. Defined as follows: N is the total number of packets in a transfer cycle, with sequence numbers $\{0, 1, \ldots, N-1\}$ and $R \subseteq \{0, 1, \ldots, N-1\}$ is the set of sequence numbers of packets received, arranged in increasing order:

$r_0 < r_1 < \cdots < r_{m-1}$, where $m = |R|$.

The the sequence numbers in R partitioned into maximal contiguous acknowledgment blocks ab (i.e., intervals) where AB is the set of all ab and in one ab consecutive numbers differ by 1, $\forall ab_i \in AB$ and any two successive elements $r_j, r_{j+1} \in ab_i$ where $r_{j+1} = r_j + 1$ and if $r_j \in ab_i$ $r_{j+1} \notin ab_i$ then r_{j+1} starts a new acknowledgement block. Hence, for each acknowledgment block ab_i:

$LowAck_i = \min\{ab_i\}$ and $HighAck_i = \max\{ab_i\}$ The number of acknowledgement blocks, i.e, is $AckBlocks = |AB|$.

11.7.2.2 Channels and Scheduling

The purpose of the channels is to serve as schedulable entities for state data transfers between various applications running on a controller. We utilize Earliest Deadline First (EDF) scheduling, which was originally developed for task scheduling and later extended to communication channel scheduling by Zhen et al.[178, 179]. The RSTP channel concept is based on the work of Zheng et

al., as summarized below [179].

A channel is described by the following tuple $\langle C, T, d \rangle$, where C is the transmission time, i.e., the time it takes to transfer the data. T is the minimum (shortest) interarrival time of new data to be transmitted, and d is the deadline, denoting the time by which the transfer must be completed.

To determine whether a channel is schedulable, we use the necessary condition check for non-preemptive scheduling in a bounded time frame provided by Zheng et al. and summarized below [179]. It consists of three steps, summarized below: (i) utilization check, (ii) determination of the time intervals for utilization check, and (iii) check that deadlines are met for all time intervals from step (ii).

For n channels, as mentioned, the first check is to verify that the utilization does not exceed the physical link capacity. We refer to the entire physical or logical path across the network connecting the primary and backup controllers as a link.

$$\sum_{i=1}^{n} \frac{C_j}{T_j} \leq 1$$

Next is to deduce the time intervals, S is the set of time intervals to check, defined as:

$$t_{\max} \;=\; \max\Big\{\, d_1, \ldots, d_n,\; \frac{C_p \;+\; \sum_{i=1}^{n} \Big(1 - \frac{d_i}{T_i}\Big)\, C_i}{1 \;-\; \sum_{i=1}^{n} \frac{C_i}{T_i}} \Big\}.$$

$$S \;=\; \bigcup_{i=1}^{n} S_i, \quad S_i \;=\; \Big\{\, d_i + n\,T_i :\; n = 0, 1, \ldots, \Big\lfloor \frac{t_{\max} - d_i}{T_i} \Big\rfloor \Big\}$$

Third, verify that we transfer all channels before the required deadlines, i.e., the deadlines can be met for all intervals in S and all channels n.

$$\forall\, t \in S, \quad \sum_{i=1}^{n} \Big(\Big\lceil \frac{t - d_i}{T_i} \Big\rceil^{+} C_i \Big) \;+\; C_p \;\leq\; t$$

C_p is the preemption blocking time faced by the higher priority transfer induced by, for example, the operating system or by utilizing the link for unscheduled traffic of lower priority.

The schedulability check can be performed either by an engineering tool or by functionality provided by the protocol implementation. Its purpose is to prevent overcommitment, ensuring that the system does not promise to transfer more data than can be realistically handled. While the ideal transfer time can

be estimated by dividing the data size by the available bandwidth, this is a simplification. In practice, protocol processing introduces overhead that increases transfer times, and this overhead is likely dependent on the specific hardware used. Developing a realistic, yet not overly pessimistic, model for estimating transfer times is left as future work.

The data transferred by RSTP-PP is typically fragmented into multiple packets, as state data is often larger than the MTU of the underlying link, usually the size of a standard Ethernet frame. Since RSTP-PP is designed to be reliable and tolerate packet loss, the transmission time C depends not only on the link capacity and the efficiency of the protocol implementation, C also needs to account for potential retransmissions. These aspects are further discussed in Section 11.7.2.4.

Since each channel consumes buffer memory, the receiver must reserve enough for incoming packets. The engineering tool can verify that the required buffer memory does not exceed the available memory (with a suitable margin). Memory-management optimizations are left to future work.

11.7.2.3 RSTP-PP Interaction

This section describes the RSTP-PP protocol interaction between the sender and receiver. The sending process begins with the sender transmitting a packet belonging to the channel with the earliest deadline, i.e., the packet designated for transmission according to the EDF scheduling scheme. Before sending the first packet for a channel in a channel period T_i, $TCycle$ is incremented. During the cycle, $TtlSzCyc$ must remain constant, allowing the receiver to reserve memory as needed upon reception of the first message. The sender can switch between sending packets for different channels if a switch is deemed suitable by the scheduler. The transmission of $ChId$ for $TCycle$ is complete when an acknowledgment is received, confirming the reception of all packets, i.e., the reception of all payload bytes, or if the deadline can't be met.

To prevent exhausting the receiver, RSTP-PP utilizes a flow control mechanism; the number of packets in flight, $packetsInFlight$ (unacknowledged packets), is limited to be no higher than $packetsInFlightMax$. Each packet sent that is not a retransmission increments $packetsInFlightMax$. Each received acknowledgment confirms that the reception of previously unacknowledged packets yields a decrement of $packetsInFlight$, with the amount of newly acknowledged packets deduced from the received acknowledgment. Alternatively, an additional field can be added to the acknowledgment, specifically stating the number of received packets, to further simplify handling, at the cost of extra bytes in the acknowledgment. RSTP-MM provide the initial

$packetsInFlightMax$ value, discussed in Section 11.7.3

Acknowledgments are sent from the receiver to the sender for a channel under four conditions: (i) upon reception of the last packet for a $TCycle$, (ii) when the number of $AckBlocks$ increases, (iii) upon explicit request from the sender using $FAckCh$, and (iv) when the limit ($packetsRcvdNoAckCntMax$) of unacknowledged packets ($packetsRcvdNoAckCnt$) received is reached. The initial value of $packetsRcvdNoAckCntMax$ is discussed in Section 11.7.3.

Receiving the final packet, i.e., the one with the highest sequence number, completes the transfer if no losses have occurred. Regardless of packet loss, the receiver always sends an acknowledgment upon receiving the last packet. The second condition is triggered by an increment in $AckBlocks$, which indicates reception after a loss (i.e., a newly detected gap in contiguous packet reception). The third condition occurs when the sender explicitly requests an acknowledgment using $FAckCh$, typically when it has sent all packets for a channel but has not yet received confirmation of receipt from the receiver. The fourth reason acknowledgments are sent is for flow control. Since the sender is only allowed to send up to $packetsInFlightMax$ packets without receiving an acknowledgment, the receiver must issue acknowledgments at regular intervals if none of the above-mentioned conditions are met. This interval is governed by $packetsRcvdNoAckCnt$ and $packetsRcvdNoAckCntMax$, which are further detailed in Section 11.7.3.

As mentioned, the receiver sends an acknowledgment when receiving the last packet. However, acknowledgments, as well as data packets, can be susceptible to loss. Hence, the sender typically uses $FAckCh$ to request acknowledgment for the channel with the earliest deadline, where all packets have been sent but receipt remains unacknowledged, if such a channel exists.

When the sender has transmitted all packets for a channel, it starts with the next one, as appointed by EDF. If there is no next channel to transmit, the last packet is resent until acknowledgment information is received or the deadline expires. When the acknowledgment is received, it either confirms the reception of all packets or provides the information required to determine which packets to retransmit. Consequently, while awaiting acknowledgments, unfinished channels are served in deadline order with the $FAckCh$ set to the identity of the channel with the earliest deadline waiting for an acknowledgment.

A sender informed about missing sequence numbers tags those packets (or adds them to a resend queue). If the total time left to send all packets known not to have been received exceeds the channel's deadline, the sending of that channel aborts for the current $TCycle$.

11.7.2.4 Fault-tolerance and Network Redundancy

Redundant controllers are commonly deployed with network redundancy to avoid the network being a single point of failure. Gigabit Ethernet IEEE 802.3ab specifies a Bit Error Rate (BER) smaller than 10^{-10}, and in controlled environments, a BER as low as 10^{-12} is plausible [1, 180]. As an example, the above BER span yields, for a channel utilizing 100 Mbps of a 1 Gbps link, an hourly frame loss between 0.36 and 36 frames of total $30 * 10^6$ frames per hour.

As mentioned, a state transfer typically fragments into several packets and Ethernet frames; without retransmission, the loss of one frame would invalidate the entire transfer. A standard Ethernet full-size frame is 1518 bytes, or 12144 bits large. If F is the number of bits in a full-size frame, then the probability that such a frame is transmitted correctly is $q_s = (1 - BER)^F$, and the probability of failure is $p_s = 1 - q_s$ [181]. N is the total number of frames (or packets) constituting a fragmented message. In the case of redundant links with parallel transmission, as for PRP, the probability that at least one of the frames is successfully received is $q_r = 1 - p_s^2$ and $p_r = 1 - q_r$.

As mentioned, all packets that constitute a message must be received for the message to be successfully delivered. We assume that the packet transmissions are independent and that we can make R individual packet retransmissions within the deadline. R is the retransmission budget. Hence, a message is successfully delivered if all packets are delivered with or without consuming the whole retransmission budget, R. To illustrate the impact of retransmission we assume each packet to be independent, the successful transfer of a message can be modeled with the sum of negative binominal distribution from zero to R retransmissions, where q and p are either q_s and p_s or q_r and p_r depending on if redundant links are used or not. Note that even though RSTP uses the retransmission budget R, it is agnostic to how R is chosen. The model below is just one example of illustrating the impact of R on the transfer reliability; future work can explore deployment-specific models.

$$P_{\text{msg success}}(R) = \sum_{k=0}^{R} \binom{N+k-1}{k} p^k \, q^N. \tag{11.4}$$

To exemplify how retransmission can improve reliability, we use variable message sizes sent every hundred milliseconds, ranging from 5 MB to 0.015 MB. The 5 MB message corresponds to over forty percent utilization of a Gigabit Ethernet link, fragmented into more than 3000 packets, embedded in an equal number of Ethernet frames. The 0.015 MB message consumes less than one percent of the Gigabit link and fragments into ten packets. We use a

conservative BER of 10^{-10} and calculate the expected number of lost messages during a year of operation for different retransmission budgets, with a cutoff at 10^{-12} messages lost per year, plotted in Figure 11.12.

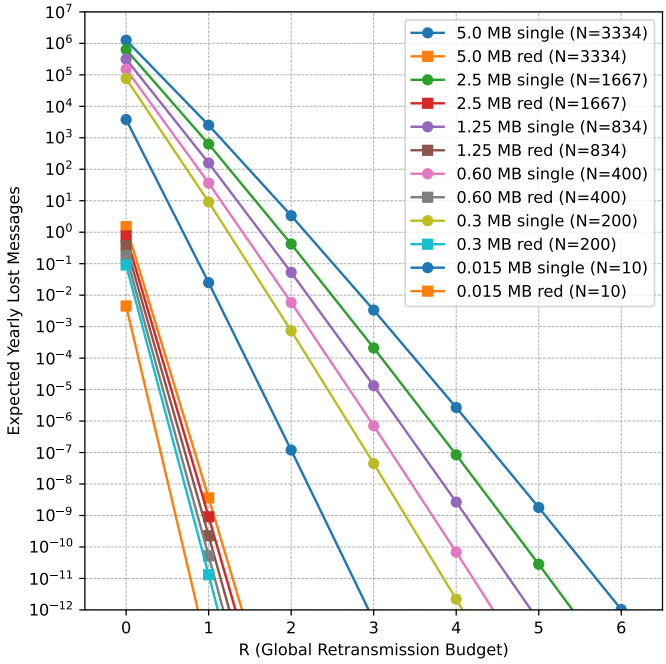


Figure 11.12: Expected yearly message loss for messages with varying sizes and retransmission budgets, sent every hundred milliseconds, using single (single) and redundant (red) links (network paths).

Figure 11.12 shows that the yearly message loss drops rapidly for each extra packet retransmission in the retransmission budget, and for $R \geq 5$, the annual loss is less than 10^{-6} for all message sizes. Hence, it may be sufficient for many applications to utilize a single network for state transfer, as the retransmission mechanism can provide sufficient messaging reliability. However, this would mean that if one link (network path) fails, the backup is no longer ready until the link is repaired. Whether this is acceptable depends on the application and the domain; RSTP supports both. An RSTP channel can be scheduled on all networks or a single network.

Redundant links that utilize a redundancy protocol, such as PRP, are seen as one link from RSTP. If the deployment requires redundancy to function when one of the redundant links has failed, the retransmission budget should be set accordingly.

11.7.3 Management Mechanism - RSTP-MM

The purpose of the management mechanism is to provide configuration and management capabilities to RSTP. It is a denoted mechanism, as it does not necessitate communication but rather details the information and data that RSTP requires, as elaborated below.

We categorized the required RSTP information provided by RSTP-MM into three profiles: (i) node profile, (ii) sender profile, and (iii) receiver profile. The sender profile gives the receiver information about the sender and vice versa. Currently, all the information is mandatory; future work could detail additional optional information that might be beneficial.

Do note that exchanging this information over a communication channel is not mandated, even if that offers better flexibility. An alternative approach would be to provide the information during the configuration phase and download it to the respective controllers, i.e., the sender (primary) and receiver (backup).

11.7.3.1 Node Profile

The node profile provides RSTP with the necessary node information, which includes details about the node it is running on.

Mandatory: LI is a set of tuples with link information li, where $\forall li \in LI$ are designated to RSTP-PP and li is the link information tuple $\langle BW, id \rangle$ where BW is the link bandwidth and id is the link identity.

11.7.3.2 Sender Profile

The receiver uses the information provided by the sender's profile, i.e., information about the sender that must be made available to the receiver.

Mandatory: $PayloadSz_{ht1}$ is the default number of payload bytes in an RSTP-PP data frame for $HType$ one. All data packets sent have this payload size, except the last one for each $TCycle$. The last frame carries at most $PayloadSz_{ht1}$, or less. The exact size is determined using number of packets N where $N = \lceil \frac{TtlSzCyc}{PayloadSz_{ht1}} \rceil$. Hence,the payload size of the last packet sz_{lp} is:
$sz_{lp} = TtlSzCyc - ((N-1)PayloadSz_{ht1})$.

11.7.3.3 Receiver Profile

The receiver profile contains information the sender needs to know about the receiver.

Mandatory: Receiver link information, RLI, is a set of tuples, rli, with link information. The receiver's LI is made available to the sender in the receiver's profile. The sender uses this to determine the end-to-end connection capacity. We do not consider potential reducing factors in the network in between; that is, future work. Typically, the id is the link's IP address, and a node has only one link per subnet. Hence, pairing links can be done based on subnet belonging. The bandwidth capacity is the lowest of $li_i.BW$ and $rli_i.BW$ for the specific link, i.

As mentioned, the sender uses $packetsInFlightMax$ for flow control to prevent overwhelming the receiver. Hence, the receiver is responsible for providing $packetsInFlightMax$. Queue sizes on the receiver must be large enough to allow $packetsInFlightMax$ packets to be transmitted without any losses due to full queues. The value of $packetsInFlightMax$ affects $packetsRcvdNoAckCntMax$, as acknowledgment reception at the sender reduces the number of packets in flight ($packetsInFlight$). Preferably, $packetsInFlight$ should not exceed $packetsInFlightMax$ due to lost acknowledgments, as reaching $packetsInFlightMax$ pauses sending. Hence, $packetsRcvdNoAckCntMax$ is set to a fifth of $packetsInFlightMax$ to tolerate some acknowledgment losses without pausing sending.

11.7.4 RSTP Design and Operation

This section presents an RSTP design. With the design as a foundation, we describe RSTP interaction in two use cases: (i) normal operation and (ii) (re-)configuration.

11.7.4.1 RSTP - Normal Operation

By "normal operation," we refer to an operational controller pair configured for redundancy. The primary controller manages the process by executing controller applications and continuously transferring the latest application state to the backup. Figure 11.13 illustrates the internal component interactions during the normal operation of a primary controller that utilizes RSTP (i.e., RSTP-PP as a sender), detailed below. This is followed by Figure 11.14, which details the RSTP-PP receiver flow in the backup.

As described in Section 11.2, an application periodically sends its updated internal state to the backup during normal operation; step (1) in Figure 11.13 begins at that point. The example application passes its updated state to the RSTP-PP along with the channel ID ($ChId$). The RSTP-PP *Buffer handler* ensures that a buffer capable of holding all the application's state data is reserved,

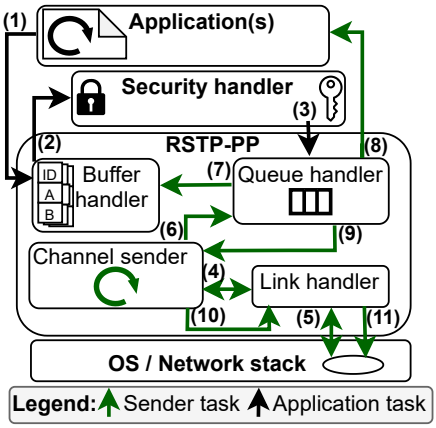


Figure 11.13: RSTP-PP sender design, internal components interaction, and data flow during normal operation.

for example, through double-buffer handling. Once a buffer has been reserved for the data, the application state is copied to that area using the *Security handler* (step 2). The *Security handler* applies the configured security measures to the state data, including additional security-related metadata if needed. See Section 11.7.5. Once the appropriate security measures have been applied and the state data has been copied to the allocated buffer, the updated channel data is ready to be queued for transmission (step 3). If data already exists for the same channel, for which the transfer has not yet started, two alternatives exist: replace the old data or keep both. By default, the interpretation is that there is no need to retain the old data for state data; therefore, the old buffer is released for reuse, and the updated data takes its place. The deadline for the replaced data is preserved, but the expiration time is updated. The updated channel data is enqueued, and the application execution context has completed its part. It is now up to the RSTP-PP sender task to transfer the updated channel data.

Step (4) and the remaining steps are executed by the RSTP-PP sender task(s), as shown in Figure 11.13. The example design only shows one RSTP-PP sender task; dividing it into two or more can increase the parallelism, one for acknowledgment handling and one for sending. The first action is to retrieve any incoming acknowledgments. The *Channel sender* asks the *Link handler* to check all links for received acknowledgments (steps 4 and 5). The

received acknowledgments determine whether any channels have been completely transferred (step 6). If a channel is completed, the buffer it uses is released (step 7), and the application is notified (step 8). The same applies if the deadline expires without any acknowledgment; in this case, the buffer is released, and the application is notified that the state transfer failed.

The *Queue handler* returns the next packet to be sent, which can be either a retransmission of a previously sent packet (determined to be lost) or the next channel to be sent (step 9). The *Queue handler* uses the same scheduling model as used to determine if the channels are schedulable; for this work, it is assumed to be EDF. Future work could dig deeper to investigate if there are more suitable scheduling alternatives. The *Queue handler* finds and returns the next packet to send according to the scheduling used, i.e., EDF. The packet is then passed to the *Link handler* (step 10) and sent to all links on which the channel is scheduled (step 11).

Figure 11.14 shows the receiver flow that begins with the *Channel receiver* asking the *Link handler* for new packages. The *Link handler* checks all links and provides the new packages to the *Channel receiver* (steps 1 and 2). The received packages are then passed to the *Reception handler*, which checks if the newly received package is the first in $TCycle$ or the first for a channel; in such cases, buffers are allocated and reserved accordingly (step 4). As with the sender, a double buffer can be a suitable implementation alternative, preserving consistency by allowing the application to read from the inactive buffer while the *Channel receiver* updates the active buffer until all packages are received; at this point, the active and inactive buffers are swapped.

The application is informed when the *Reception handler* determines that a complete message has been received or that no new messages have been received within the expiration time (step 5).

The Reception handler informs the *Channel receiver* if any acknowledgments need to be sent. If so, the *Channel receiver* uses the *Link handler* to send the acknowledgments (steps 6, 7, and 8). When the application is notified of new data (step 5), it retrieves the latest data for the specific channel ID ($ChId$) from RSTP-PP (step 9). The receiving application and its sending counterpart must share the same channel ID; hence, the channel identity can be part of the application configuration. RSTP-PP returns a handle to a buffer that the application uses to access the data. This handle is passed to the *Security handler* (step 10), which applies the configured security measures to the received data while copying it to application-managed memory. Once this is done, the application informs RSTP-PP that the data has been processed, and the buffer is freed for reuse (step 11).

We use one RSTP-PP *Channel receiver* task in the example; a higher de-

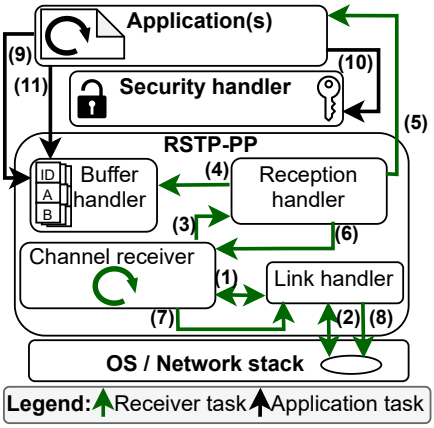


Figure 11.14: RSTP-PP receiver design, internal components interaction, and data flow during normal operation.

gree of parallelism is achieved by having two or more tasks, where one handles acknowledgment sending and the other handles packet reception.

11.7.4.2 RSTP - Configuration

Figure 11.15 shows a high-level configuration flow and exemplifies a configuration of a redundant controller pair utilizing RSTP. An engineer uses an engineering tool to modify the application or create an initial version. The tool utilizes the scheduling and reliability models presented in Section 11.7 to determine if the configuration is schedulable given the amount of state data, cycle time, deadline, and desired reliability (step 1).

Given reliability-related parameters, the reliability model provides a retransmission budget (see Section 11.7.2.4). With preconditions that include the retransmission budget, data size, deadline, cycle time, and links, the scheduling model determines if it is possible to meet these requirements. The scheduling model can also choose the order in which to apply the changes during a reconfiguration. To avoid overutilizing links during a configuration change, it is essential to apply changes that reduce link utilization before those that increase utilization, or perform the switch atomically across all applications. The scheduling model helps identify changes that reduce link load and those that increase it, thereby determining the proper order in which to apply these

changes.

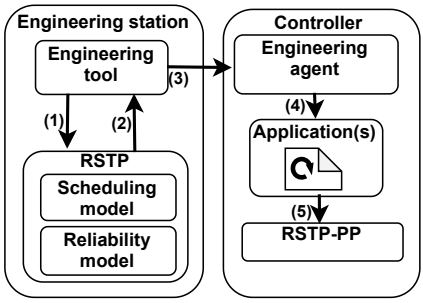


Figure 11.15: Configuration example.

When the engineer completes the configuration changes, the changes are downloaded to the affected controllers (step 3). The engineering tool sends the new configuration to the engineering agent, which applies the change (step 4). The change can be applied stepwise by first applying changes that reduce link load before applying changes that increase load, or atomically for all applications at a specific time. Once applied, the applications begin using the new settings (step 5).

It is worth noting that if an application is deleted, it should inform RSTP-PP in some manner so that the channel resources, in terms of buffers, can be completely deallocated. This is implicit in step 5.

11.7.5 Security Handling

The function of the *Security handler* is to add security features as defined in section 11.5.2.3. Different use cases may require various features. Security features will also add to the payload size and execution time needed.

The *Security handler* in the sending and receiving entities is separated from the RSTP-PP protocol, allowing the application to handle any overhead related to security measures instead of RSTP tasks. This ensures that the time consumption for tasks related to receiving and sending data can be kept down while providing high flexibility in selecting the security mechanisms to include.

It is also possible to combine the RSTP-PP protocol with IP-sec, if the level of security provided by IPSec is deemed sufficient. In this case, the security handler will do nothing, as the operating system provides the security

Table 11.19: Security configurations for the RSTP Security handler. R/O column: R=Required, O=Optional.

Property	**R/O**	**Value**
Integrity mechanism	R	None, Checksum, Symmetric, Certificate
Encryption mechanism	R	None, Symmetric
Replay Protection	R	None, Counter, Session, Timestamp
Key Exchange	O	None, KeyServer, IKE
Key Server	O	URL, etc to trusted key server
Partner Certificate	O	(public) certificate of partner

functionality. As noted, it may be challenging for the application to verify that protection is actually in place when using this approach, as the security mechanism is implicitly added outside the protocol.

To secure the application layer, the state data is wrapped within a security header that contains security-related fragments.

Security mechanisms needed must be indicated as part of the RSTP *Security handler* configuration. Table 11.19 outlines the options required to fulfill the previously described security requirements. The list of values is, however, not exhaustive; more potential methods exist, and several variations of each technique are also available. Quantification of security induced latency is future work.

Options for *Integrity mechanism* indicate how integrity and/or authenticity of data is assured. The checksum option will only give a basic integrity protection of the data (*Sec_Int*), the symmetric signature will be done using a symmetric key exchanged either by secure key server or using a peer-to-peer key establishment protocol such as Internet Key Exchange (IKE), which will give some degree of authenticity (*Sec_Auth*), while a certificate-based signature will provide highest level of authenticity protection.

The *Encryption mechanism* option describes whether data encryption will be used to fulfill the requirement *Sec_Conf*, with the options None and Symmetric. In reality, this option must be complemented with a list of supported symmetric encryption algorithms.

Replay protection outlines how the freshness of data is ensured (*Sec_Fresh*), providing options with static counters, session identities, and timestamps.

The *Key exchange* option defines how the backup and partner exchange the keys needed for encryption or integrity protection, if based on symmetric keys. If the communicating entities are capable of using public key cryptography and

the communication is peer-to-peer, a certificate-based key exchange protocol is advised to be used.

If public-key cryptography is not a viable option, or if there are multiple backup entities that should receive state data, the suggested scheme is to use a key distribution server to provide symmetric keys to the communicating parties. This scheme is inspired by how secure OPC UA PubSub [146] works, as shown in Fig. 11.16. The key distribution server can either push keys or the communicating party can fetch them. However, communication with the key distribution service needs to be encrypted as well as access-controlled, as the keys would otherwise be accessible to anyone. The implementation of this protocol is outside the scope of the current work, but the suggestion is to follow the approach outlined in the OPC UA specification.

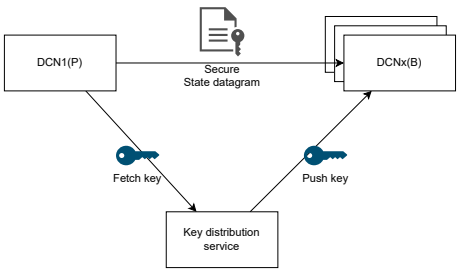


Figure 11.16: Key distribution according to pattern from OPC UA Secure pub-sub.

11.7.6 RSTP - Desired Feature Matching

We conclude the design section by showing how RSTP fulfills the desirable features presented in Section 11.5.2.

Reliability features: RSTP has a mechanism for retransmission, with a configurable retransmission budget to provide the desired reliability; hence, RSTP fully fulfills *Rel_RD*. RSTP has a mechanism for flow control to prevent exhausting receiver buffers; hence, RSTP fully fulfills *Rel_RC*. Finally, since no channel should be deployed on RSTP before confirming that it can be scheduled, RSTP avoids overutilization of the network; hence, RSTP fully fulfills *Rel_NC*.

Real-time features: RSTP guarantees that channel messages are delivered before their deadline, provided that the channels are successfully scheduled; hence, RSTP fully fulfills *RT_PT*. The protocol also sets an expiration date for the passed data, meaning that new data is expected to arrive before that deadline. If new data is not received in time, the consuming application

can be informed by RSTP; hence, RSTP fully fulfills *RT_UE*. Lastly, RSTP uses channel concepts and transmits packets from the channel closest to the deadline. All channels have preallocated buffers at the receiver side, and when a packet is received, its payload is stored at the correct offset in the receive buffer. In other words, RSTP uses channels and an earliest-deadline-first approach to prioritize the channels; hence, RSTP fully fulfills *RT_PR*.

Security features: The *Security Handler* allows RSTP to support a configurable security level. Thus, it can be configured to fulfill any combination of the desired security features, from none to all.

11.8 Deployment and Experimental Evaluation

This section describes RSTP in a VxWorks deployment, examining how an operating system can be configured to align with RSTP. After that, we describe an actual implementation of RSTP on VxWorks, which we use for experimental evaluation.

11.8.1 RSTP on VxWorks

VxWorks's default network stack configuration consists of a single network stack instance with one network task that serves all outgoing and incoming traffic [132]. Figure 11.17 shows RSTP and other applications on a VxWorks system with the default settings. Using the default settings means that one network task handles both time-sensitive and time-insensitive communications.

Regarding RSTP and outgoing traffic, RSTP will post a packet on the socket for the channel with the highest priority, as described in Section 11.7.4. However, since a single network stack and task serve all sockets, that packet might not be handled immediately under the default VxWorks settings. The socket-related network job is placed in a queue, and the network task processes that queue in a first-in-first-out fashion without any prioritization. The single queue and network task may cause a potential latency increase for time-sensitive packets due to the processing of time-insensitive traffic ahead in the queue, as illustrated in Figure 11.17.

The number of network stacks (and tasks) in VxWorks can be configured [132]. A feature utilized by Johansson et al. as a foundation for processing time-sensitive traffic with a higher-priority network task [182]. An application can direct outbound traffic to different network stack instances and, consequently, different network tasks by assigning the socket to a specific stack instance using socket options.

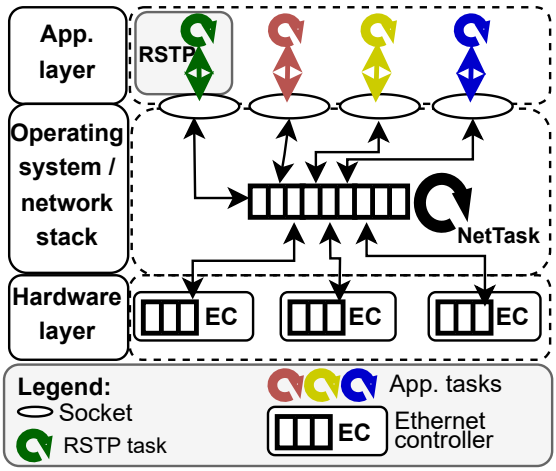


Figure 11.17: RSTP and other network-dependent applications using the default configured VxWorks network stack. RSTP, multiple applications, and Ethernet Controllers (EC) share one network stack and Network Task (NetTask).

Figure 11.18 shows two stack instances—one for high-priority time-sensitive traffic and one for best-effort traffic. In this example, the high-priority network stack serves RSTP, including the Ethernet Controller for the RSTP link. The low-priority network task handles time-insensitive data. Note that this configuration can be scaled to include additional tasks and priority levels. Figure 11.18 serves as an example.

11.8.2 RSTP Experimental Implementation

To experimentally evaluate RSTP, we implemented an RSTP sender, an RSTP receiver, and a prototype RSTP engineering tool that checks schedule feasibility.

RSTP Engineering: The RSTP Engineering prototype is a Python script that, given the periodicity, available bandwidth, BER, and state size of the applications and the acceptable annual loss, provides the channel parameters and checks if the channels are schedulable. It provides output used by the RSTP sender prototype, such as the utilization of the links.

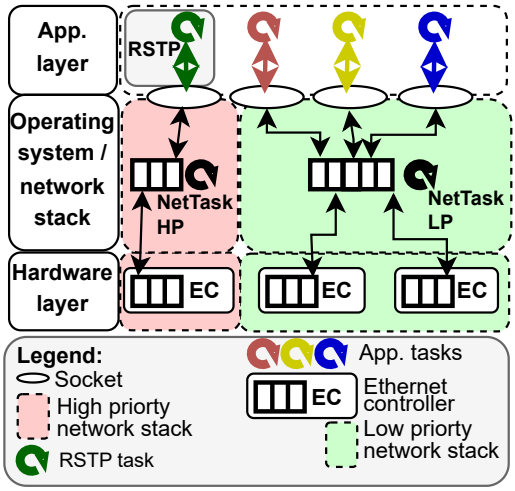


Figure 11.18: Network-dependent applications using a customized VxWorks network stack configuration with a High Priority (HP) network stack instance for handling RSTP traffic and a Low Priority (LP) network stack instance for best-effort time-insensitive traffic.

RSTP Sender: The RSTP Sender is a simplified version of what is depicted in Figure 11.13, with the main differences being in the buffer and security handling. The prototype does not implement double buffering or a security handler. The applications in the evaluation prototype provide a state data buffer to be transmitted every period and measure the time until RSTP indicates that the transfer is completed. The RSTP-PP data is sent over UDP, with packets transmitted on one UDP port and acknowledgments on another. VxWorks is configured according to Figure 11.18 to prioritize the RSTP traffic.

RSTP Receiver: The RSTP Receiver prototype is a simplified version of the receiver depicted in Figure 11.14. The prototype omits the security handler and buffer management. The application is a placeholder that registers the reception of new state data through a callback call upon state reception completion from RSTP.

11.8.3 RSTP Experimental Evaluation

We experimentally evaluate RSTP using two state transfer arrangements. The first one uses the same state data size selection as we used to assess TCP and SCTP in Section 11.6.4. The second arrangement is a multi-application scenario that mimics a controller running multiple applications utilizing channels of varying sizes, periods, and deadlines, corresponding to the combinations of applications with distinct periodicity and state sizes, as shown in Table 11.20.

Table 11.20: RSTP evaluation configuration for multiple applications (and channels). Each application runs in its own task, enabling concurrent RSTP use.

App. (ChId)	Period (deadline)	Retransmission budget	Size	Utilization (of 1 Gbps)
1	10 ms	2	800 B	0.06%
2	20 ms	2	800 B	0.03%
3	50 ms	3	40 KB	0.64%
4	100 ms	3	400 KB	3.2%
5	100 ms	4	800 KB	6.4%
6	200 ms	4	1.6 MB	6.4%
7	500 ms	5	8.0 MB	12.8%
8	500 ms	5	8.0 MB	12.8%
9	500 ms	5	8.0 MB	12.8%
10	500 ms	5	8.0 MB	12.8%
11	1000 ms	5	8.0 MB	6.4%
		Total size and utilization:	42.8 MB	74.3%

An application that consists of 100,000 variables, corresponding to 0.4 MB of state data, and a period of 100 milliseconds, is considered a reasonably large application with a relatively short cycle time for the process control domain [21, 183]. Moreover, controllers today offer varying amounts of memory for application usage; for example, a PM 891 from ABB offers approximately 200 MB of memory available for applications [184], meaning a controller can host many applications of the size mentioned above, a limit likely to increase with newer generation controllers.

Table 11.20 summarizes the concurrent multi-application simulation configuration, where each application runs as a separate task. Applications 1 and 2 have small data sizes (800 B) and short periods, representing a small application or a heartbeat-based failure detection utilizing RSTP. Application 3

has a data size of 40 KB and a period of 50 milliseconds. Given, as stated above, that an 400 KB application with a cycle time of 100 milliseconds is considered large and fast, application 3 serves as smaller, but even faster application [21, 183]. Application 4 and 5 have period of 100 milliseconds and a size of 400 KB and 800 KB respectively, to serve as examples of fast and large applications. Applications 6-11 are even larger, but with longer periods. We believe that this selection serves as an example that pushes beyond typical utilization, given the combination of fast and large concurrent applications. Table 11.20 lists the complete configuration, which yields a payload utilization of 74.3%.

We deliberately avoid going higher to spare some capacity for best-effort traffic as well as the protocol processing-induced transfer time overhead. In a real deployment, time-sensitive RSTP traffic would likely have precedence over best-effort traffic by using a priority mechanism, such as Priority Code Point (PCP) [182].

We ran the evaluation for one hour and collected the minimum, maximum, and average application state transfer times. In addition, every second, a packet is dropped to simulate a very lossy link and utilize the recovery mechanism. The following section, Section 11.8.4, presents the result.

11.8.4 RSTP Evaluation Results

Figure 11.19 shows the RSTP prototype performance under the same arrangement previously used to evaluate TCP and SCTP, described in Section 11.6.3. It provides an overview and displays the longest measured transfer times, while Table 11.21 provides the more detailed measurements.

For the scenario without packet loss, transferring 1 MB (2^{20} bytes), the longest RSTP transfer time is 11 milliseconds, with an average transfer time effectively at 10 milliseconds, compared to 9.2 milliseconds for TCP. In the scenario with ten packet losses, the RSTP transfer time peaks at 16 milliseconds, whereas TCP's worst-case transfer time occurs for 10 KB and measures over five seconds.

Therefore, while RSTP's overall throughput without packet losses is somewhat lower than TCP, RSTP recovers significantly faster when losses occur. Additionally, RSTP was evaluated on a prototype implementation, whereas the TCP implementation is a mature, production-level implementation that is likely to be highly optimized. Hence, the throughput of RSTP can most likely be improved with optimization efforts.

Table 11.22 shows the results from the concurrent multiple-application RSTP evaluation arrangement described in Table 11.20. Transfer times in-

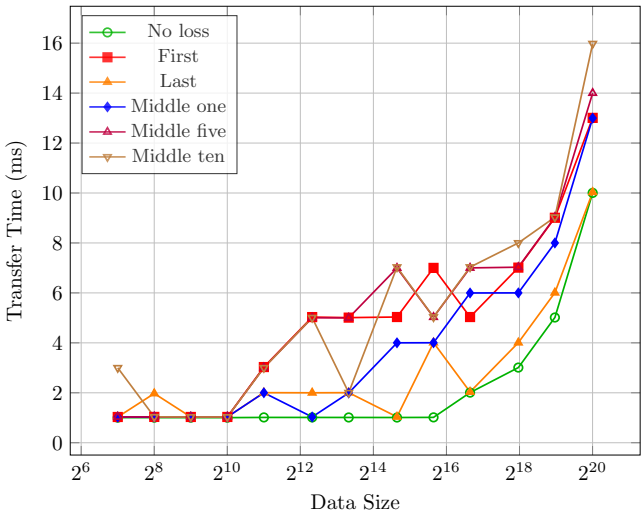


Figure 11.19: RSTP max transfer times. First, last, and middle denote if the first, last, or middle packets are lost.

crease with higher $ChId$, likely because channels are processed in ascending order of $ChId$ when deadlines are identical. Using $ChId$ as a tiebreaker was deemed sufficient for this proof-of-concept implementation; however, it is a potential area for future work to investigate whether there are more suitable alternatives. Alternatives that would reduce the increase in transfer time for higher $ChId$.

Additionally, note that deadlines are met even under conditions of frequent packet loss, as one packet per second was dropped during the one-hour experimental run, causing all channels to experience packet loss.

11.8.5 Discussion of RSTP Results

Under no-loss scenarios, RSTP is not as performant as TCP with default settings in terms of throughput; however, RSTP is more performant than the recovery-optimized TCP. When comparing SCTP and RSTP under no-loss conditions, the optimized version of SCTP outperforms the non-optimized SCTP configuration. However, the optimized SCTP is still less performant than RSTP for larger data sizes. For smaller data sizes without losses, both SCTP and TCP are slightly faster than RSTP. Again, we believe this is due to specific implementation details in the prototype, which could be mitigated with further optimizations. Such optimizations are reserved for future work. The protocol is demonstrated to be performant, achieving 75%

Table 11.21: Transfer times (ms) with minimum, average, and maximum values under packet loss scenarios.

Size	No Loss			First pkt. lost			Last pkt. lost			1 mid pkt. lost			5 mid pkts. lost			10 mid pkts. lost		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
128 B	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	3.0
256 B	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
512 B	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1 KB	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2 KB	1.0	1.0	2.0	3.0	3.0	3.0	1.0	1.0	2.0	1.0	1.0	2.0	1.0	1.0	3.0	1.0	1.0	3.0
5 KB	1.0	1.0	2.0	3.0	4.1	5.0	1.0	1.0	2.0	1.0	1.0	1.0	1.0	1.0	5.0	1.0	1.0	5.0
10 KB	1.0	1.0	2.0	3.0	3.5	5.0	1.0	1.0	2.0	1.0	1.0	2.0	1.0	1.0	5.0	1.0	1.0	2.0
25 KB	1.0	1.0	1.0	3.0	3.4	5.0	1.0	1.0	1.0	2.0	2.4	4.0	3.0	3.9	7.0	2.0	4.0	7.0
50 KB	1.0	1.0	4.0	3.0	3.9	7.0	1.0	1.0	4.0	2.0	2.2	4.0	3.0	3.9	5.0	2.0	3.7	5.0
100 KB	2.0	2.0	4.0	4.0	5.0	5.0	2.0	2.0	2.0	4.0	4.0	6.0	5.0	5.0	7.0	2.0	5.1	7.0
250 KB	3.0	3.0	4.0	5.0	5.1	7.0	3.0	3.0	4.0	4.0	4.1	6.0	5.0	5.1	7.0	4.0	4.6	8.0
500 KB	5.0	5.0	6.0	7.0	7.1	9.0	5.0	5.0	6.0	6.0	6.0	8.0	7.0	7.1	9.0	6.0	7.4	9.0
1 MB	10.0	10.0	11.0	11.0	11.1	13.0	10.0	10.0	10.0	11.0	11.0	13.0	11.0	11.9	14.0	11.0	12.2	16.0

utilization across multiple channels (and applications) under significant loss conditions, as shown in the measurements in Table 11.22. Additionally, RSTP significantly reduces the maximum transfer time under loss compared to TCP and SCTP.

Regarding multiple application evaluations, all deadlines were successfully met, as shown in Table 11.22. However, we conducted the evaluation using only one set of concurrent applications, which utilized roughly 75% of the available bandwidth. For comparison, as shown in Table 11.15, TCP transfers 1 MB in 9.2 milliseconds under lossless conditions, corresponding to roughly 86% utilization of the 1 GB/s link.

The retransmission cost is twofold, since a retransmitted packet consumes bandwidth and processing time. To push the RSTP limit even closer to the theoretical maximum, the processing of transmission and retransmission needs to be analyzed in greater depth, and optimizations applied to push the utilization boundaries, thereby enabling tighter deadlines.

A transfer-reliability model (Section 11.7.2.4) estimates the probability of state-transfer failure. The retransmission budget balances the trade-off between failure probability and worst-case transfer time. With RSTP schedulability checks plus the reliability model, engineers can verify at design time whether deadline and reliability targets are achievable. Future work is to integrate this analysis into existing toolchains and present actionable and guiding outputs to support engineering decisions.

In addition to the above-mentioned future work, future evaluations could include additional application configurations under loss and no-loss conditions. Comparative analyses against other protocols under multi-application conditions, including security measures and measuring the associated overhead in both lossless and lossy situations, are examples of relevant future

Table 11.22: RSTP multiple applications evaluation result.

App. (ChId)	Period (deadline)	Size	Min	Avg	Max
1	10 ms	800 B	0.96 ms	1.16 ms	5.01 ms
2	20 ms	800 B	0.96 ms	2.02 ms	6.00 ms
3	50 ms	40 KB	0.96 ms	2.80 ms	15.99 ms
4	100 ms	400 KB	4.00 ms	10.59 ms	32.00 ms
5	100 ms	800 KB	7.97 ms	13.48 ms	34.03 ms
6	200 ms	1.6 MB	23.00 ms	32.37 ms	104.00 ms
7	500 ms	8.0 MB	77.98 ms	106.16 ms	157.01 ms
8	500 ms	8.0 MB	154.01 ms	213.69 ms	257.01 ms
9	500 ms	8.0 MB	216.98 ms	309.76 ms	363.01 ms
10	500 ms	8.0 MB	282.05 ms	403.81 ms	459.01 ms
11	1000 ms	8.0 MB	456.04 ms	922.46 ms	981.00 ms

work.

11.9 Conclusion

In this work, we have explored checkpointing and state replication solutions within both OT and IT contexts. In OT, we investigated checkpointing solutions used in industrial controllers and Programmable Logic Controllers (PLCs). In the IT context, we examined checkpointing solutions used within container and orchestration environments. CRIU was identified as a commonly used solution for retrieving state data; however, we also observed that none of the reviewed works specifically focused on transferring the retrieved state data. The literature search and the outcome of that constitute the first contribution.

The lack of literature detailing state transfer for redundancy purposes motivated us to investigate suitable alternatives further. We defined a set of desired features for a state transfer protocol that we used to evaluate existing protocols against, identifying OPC UA Client/Server, running on top of TCP, and SCTP as relevant candidates. The identification of features desired from protocols used for state transfer, as well as the matching of existing protocol properties against these desired features, constitutes the second contribution.

Considering that OPC UA Client/Server utilizes TCP as its underlying transport protocol, we compared TCP and SCTP on VxWorks, a commonly

used real-time operating system, using state transfer simulation scenarios. VxWorks allows for customization of TCP and SCTP internals, a feature we used to optimize TCP and SCTP settings beyond the default settings. The TCP configuration optimized for quick recovery in the event of losses demonstrated strong performance but still suffered from high transfer times under specific loss scenarios. Additionally, optimization impacts all TCP connections globally on the node. This evaluation, under lossy and no-loss scenarios, using optimized and default settings of TCP and SCTP, constitutes the third contribution.

To the best of our knowledge, deduced from the findings mentioned above, there exists no publicly available protocol targeting state transfer for industrial controller redundancy. That finding motivated us to design a new protocol for that specific purpose, which we named the Reliable State Transfer Protocol (RSTP), explicitly tailored to fulfill all desired features, including security. RSTP incorporates a retransmission budget, a channel-based concept, and a security handler. Each channel is assigned its own period, deadline, and retransmission budget.

A scheduler manages packet transfers based on deadlines, prioritizing packets with the earliest deadlines. Our evaluation demonstrated that RSTP handles packet loss scenarios more efficiently than TCP and SCTP, although its throughput is somewhat lower than that of TCP. Moreover, we evaluated RSTP under a multi-application scenario with high utilization (75%), experiencing significant loss to stress recovery handling.

We attribute RSTP's lower throughput compared to TCP to the current lack of performance optimizations. Addressing these optimizations remains part of our future work. Future work also includes more extensive evaluations involving multiple application scenarios and the comprehensive integration of security mechanisms, among other examples. RSTP and the experimental evaluation constitute the fourth contribution.

In summary, research on state transfer for spatial redundancy is limited, and existing protocols cover only subsets of the desired features. To close these gaps, we introduced RSTP, a schedule-aware state-transfer protocol that fulfills the desired features and, on VxWorks, shows lower worst-case transfer times under loss than TCP/SCTP.

Bibliography

[1] Björn Leander, Bjarne Johansson, Tomas Lindström, Olof Holmgren, Thomas Nolte, and Alessandro V. Papadopoulos. Dependability and

security aspects of network-centric control. In *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2023.

[2] Dietmar Bruckner, Richard Blair, M Stanica, A Ademaj, W Skeffington, D Kutscher, S Schriegel, R Wilmes, K Wachswender, L Leurs, et al. Opc ua tsn a new solution for industrial communication. *Whitepaper. Shaper Group*, 168:1–10, 2018.

[3] T. Hegazy and M. Hefeeda. Industrial automation as a cloud service. *IEEE Transactions on Parallel and Distributed Systems*, 26(10):2750–2763, Oct 2015.

[4] Yuke Jia, Tiejun Wang, Tianbo Qiu, Xiaohan Zhang, Rui Wang, and Tianyu Wo. Fault tolerance of stateful microservices for industrial edge scenarios. In *2023 IEEE International Conference on Joint Cloud Computing (JCC)*, pages 50–56. IEEE, 2023.

[5] Bjarne Johansson, Mats Rågberger, Thomas Nolte, and Alessandro V Papadopoulos. Kubernetes orchestration of high availability distributed control systems. In *Proc. ICIT*, 2022.

[6] Thomas Goldschmidt, Stefan Hauck-Stattelmann, Somayeh Malakuti, and Sten Grüner. Container-based architecture for flexible industrial control applications. *J. Syst. Arch.*, 84, 2018.

[7] Heiko Koziolek, Andreas Burger, PP Abdulla, Julius Rückert, Shardul Sonar, and Pablo Rodriguez. Dynamic updates of virtual plcs deployed as kubernetes microservices. In *European Conference on Software Architecture*, pages 3–19. Springer, 2021.

[8] Alexandru Moga, Thanikesavan Sivanthi, and Carsten Franke. Os-level virtualization for industrial automation systems: Are we there yet? In *SAC*, 2016.

[9] Elena Dubrova. *Fault-tolerant design*. Springer, 2013.

[10] Jacek Stój. Cost-effective hot-standby redundancy with synchronization using ethercat and real-time ethernet protocols. *IEEE Trans. on Autom. Science and Eng.*, 18(4):2035–2047, 2020.

[11] Andrei Simion and Calin Bira. A review of redundancy in plc-based systems. *Advanced Topics in Optoelectronics, Microelectronics, and Nanotechnologies XI*, 12493:269–276, 2023.

[12] Algirdas Avižienis. Design of fault-tolerant computers. In *Proceedings of the November 14-16, 1967, fall joint computer conference*, pages 733–743, 1967.

[13] Jacques Losq. *Influence of fault detection and switching mechanisms on the reliability of stand-by systems*, volume 75. Digital Systems Laboratory, Stanford Electronics Laboratories, Stanford Univ., 1975.

[14] Thomas Kampa, Amer El-Ankah, and Daniel Grossmann. High availability for virtualized programmable logic controllers with hard real-time requirements on cloud infrastructures. In *2023 IEEE 21st International Conference on Industrial Informatics (INDIN)*, pages 1–8. IEEE, 2023.

[15] Björn Leander, Bjarne Johansson, Saad Mubeen, Mohammad Ashjaei, and Tomas Lindström. Redundancy link security analysis: An automation industry perspective. In *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2024.

[16] Vxworks - real-time operating system. `https://www.windriver.com/products/vxworks`. Accessed: 2025-05-15.

[17] Dietmar Bruckner, Marius-Petru Stănică, Richard Blair, Sebastian Schriegel, Stephan Kehrer, Maik Seewald, and Thilo Sauter. An introduction to OPC UA TSN for industrial communication systems. *Proc. IEEE*, 107(6):1121–1131, 2019.

[18] O-pas standard, version 2.1. `https://publications.opengroup.org/c230`. Accessed: 2025-05-06.

[19] Martin A Sehr, Marten Lohstroh, Matthew Weber, Ines Ugalde, Martin Witte, Joerg Neidig, Stephan Hoeme, Mehrdad Niknami, and Edward A Lee. Programmable logic controllers in the context of industry 4.0. *IEEE Transactions on Industrial Informatics*, 17(5):3523–3533, 2020.

[20] Stefan Stattelmann, Stephan Sehestedt, and Thomas Gamer. Optimized incremental state replication for automation controllers. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–8. IEEE, 2014.

[21] Heiko Koziolek, Andreas Burger, and Abdulla Puthan Peedikayil. Fast state transfer for updates and live migration of industrial controller runtimes in container orchestration systems. *Journal of Systems and Software*, 211:112004, 2024.

[22] David Powell, Gottfried Bonn, Douglas T Seaton, Paulo Verissimo, and François Waeselynck. The delta-4 approach to dependability in open distributed computing systems. In *FTCS*, volume 18. Citeseer, 1988.

[23] B. Johansson, M. Rågberger, A. V. Papadopoulos, and T. Nolte. Heartbeat bully: Failure detection and redundancy role selection for network-centric controller. In *IECON*, 2020.

[24] Bjarne Johansson, Mats Rågberger, Alessandro V Papadopoulos, and Thomas Nolte. Consistency before availability: Network reference point based failure detection for controller redundancy. In *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2023.

[25] Abb webpage - ac 800m high integrity controllers. `https://new.abb.com/control-systems/safety-systems/system-800xa-high-integrity/ac-800m-hi-controller`. Accessed: 2025-08-27.

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

[27] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993.

[28] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

[29] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pages 305–319, 2014.

[30] Václav Struhár, Moris Behnam, Mohammad Ashjaei, and Alessandro V. Papadopoulos. Real-Time Containers: A Survey. In *Fog-IoT*, 2020.

[31] Docker webpage. `https://www.docker.com/`. Accessed: 2025-04-02.

[32] Docker criu webpage. https://docs.docker.com/reference/cli/docker/checkpoint/. Accessed: 2025-04-04.

[33] Criu webpage. https://criu.org/Main_Page. Accessed: 2025-07-17.

[34] Emiliano Casalicchio. Container orchestration: A survey. *Systems Modeling: Methodologies and Tools*, pages 221–235, 2019.

[35] Kubernetes webpage. https://kubernetes.io/. Accessed: 2025-04-02.

[36] Jacek Stój. State machine of a redundant computing unit operating as a cyber-physical system control node with hot-standby redundancy. In *Information Systems Architecture and Technology: Proceedings of 40th Anniversary International Conference on Information Systems Architecture and Technology–ISAT 2019: Part II*, pages 74–85. Springer, 2020.

[37] Yixin Zhao and Feng Liu. The implementation of a dual-redundant control system. *Control engineering practice*, 12(4):445–453, 2004.

[38] Rongkuan Ma, Peng Cheng, Zhenyong Zhang, Wenwen Liu, Qingxian Wang, and Qiang Wei. Stealthy attack against redundant controller architecture of industrial cyber-physical system. *IEEE Internet of Things Journal*, 6(6):9783–9793, 2019.

[39] Bjarne Johansson, Olof Holmgren, Thomas Nolte, and Alessandro V Papadopoulos. Partible state replication for industrial controller redundancy. In *2024 IEEE International Conference on Industrial Technology (ICIT)*, pages 1–8. IEEE, 2024.

[40] Zeinab Bakhshi, Guillermo Rodriguez-Navas, and Hans Hansson. Fault-tolerant permanent storage for container-based fog architectures. In *2021 22nd IEEE International Conference on Industrial Technology (ICIT)*, volume 1, pages 722–729. IEEE, 2021.

[41] Jan Nouruzi-Pur, Jens Lambrecht, The Duy Nguyen, Axel Vick, and Jörg Krüger. Redundancy concepts for real-time cloud-and edge-based control of autonomous mobile robots. In *2022 IEEE 18th International Conference on Factory Communication Systems (WFCS)*, pages 1–8. IEEE, 2022.

[42] Juraj Ždánsky and Karol Rástočnỳ. Influence of redundancy on safety integrity of srcs with safety plc. In *2014 ELEKTRO*, pages 508–512. IEEE, 2014.

[43] Gnana Swathika OV, Aayush Karthikeyan, K Karthikeyan, P Sanjeevikumar, Sajju Karapparambil Thomas, and Amin Babu. Critical review of scada and plc in smart buildings and energy sector. *Energy Reports*, 12:1518–1530, 2024.

[44] Raghavendra Barkur and S Shriram. An expeditious method for implementing a full redundant drive in hils with the use of plc and user interface panel. In *2017 IEEE Transportation Electrification Conference and Expo, Asia-Pacific (ITEC Asia-Pacific)*, pages 1–5. IEEE, 2017.

[45] Mary Nankya, Robin Chataut, and Robert Akl. Securing industrial control systems: components, cyber threats, and machine learning-driven defense strategies. *Sensors*, 23(21):8840, 2023.

[46] Soonwoo Lee, Jimyung Kang, Sung Soo Choi, and Myo Taeg Lim. Design of ptp tc/slave over seamless redundancy network for power utility automation. *IEEE Transactions on Instrumentation and Measurement*, 67(7):1617–1625, 2018.

[47] Pei Zhou, Wei He, and Zhanglong Zhang. The reheating furnace control system design based on siemens pcs7. In *2008 World Automation Congress*, pages 1–4. IEEE, 2008.

[48] L Rajesh and P Satyanarayana. Dual channel scanning in communication protocol in industrial control systems for high availability of the system. *International Journal on Technical and Physical Problems of Engineering*, 11(4):22–27, 2019.

[49] Jim Luo, Myong Kang, Emmanuel Bisse, Mike Veldink, Dmitriy Okunev, Scott Kolb, Joseph G Tylka, and Arquimedes Canedo. A quad-redundant plc architecture for cyber-resilient industrial control systems. *IEEE Embedded Systems Letters*, 13(4):218–221, 2020.

[50] Michael Wahler and Manuel Oriol. Disruption-free software updates in automation systems. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–8. IEEE, 2014.

[51] Yu Kaneko and Toshio Ito. A reliable cloud-based feedback control system. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 880–883. IEEE, 2016.

[52] Michael Gundall, Julius Stegmann, Mike Reichardt, and Hans D Schotten. Downtime optimized live migration of industrial real-time control services. In *2022 IEEE 31st International Symposium on Industrial Electronics (ISIE)*, pages 253–260. IEEE, 2022.

[53] Sten Grüner, Somayeh Malakuti, Johannes Schmitt, Tarik Terzimehic, Monika Wenger, and Haitham Elfaham. Alternatives for flexible deployment architectures in industrial automation systems. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 35–42. IEEE, 2018.

[54] Lucas Vogt, Anselm Klose, Valentin Khaydarov, Christian Vockeroth, Christian Endres, and Leon Urbas. Towards cloud-based control-as-a-service for modular process plants. In *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4. IEEE, 2023.

[55] Marco Barletta, Luigi De Simone, Raffaele Della Corte, and Catello Di Martino. Failover timing analysis in orchestrating container-based critical applications. In *2024 19th European Dependable Computing Conference (EDCC)*, pages 81–84. IEEE, 2024.

[56] Keerthana Govindaraj and Alexander Artemenko. Container live migration for latency critical industrial applications on edge computing. In *2018 IEEE 23rd international conference on emerging technologies and factory automation (ETFA)*, volume 1, pages 83–90. IEEE, 2018.

[57] Codesys redundancy - product data sheet. `https://store.codesys.com/media/n98_media_assets/files/2302000040-F/4/CODESYS%20Redundancy%20SL_en.pdf`. Accessed: 2025-05-22.

[58] Zahid Khan, Fakhar Abbas, et al. A conceptual framework of virtualization and live-migration for vehicle to infrastructure (v2i) communications. In *2019 IEEE 11th International Conference on Communication Software and Networks (ICCSN)*, pages 590–594. IEEE, 2019.

[59] Aditya Bhardwaj and C Rama Krishna. A container-based technique to improve virtual machine migration in cloud computing. *IETE Journal of Research*, 68(1):401–416, 2022.

[60] Shuo Zhang, Ningjiang Chen, Hanlin Zhang, Yijun Xue, and Ruwei Huang. A high-performance adaptive strategy of container checkpoint

based on pre-replication. In *Security, Privacy, and Anonymity in Computation, Communication, and Storage: 11th International Conference and Satellite Workshops, SpaCCS 2018, Melbourne, NSW, Australia, December 11-13, 2018, Proceedings 11*, pages 240–250. Springer, 2018.

[61] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. A kubernetes controller for managing the availability of elastic microservice based stateful applications. *J. Syst. and Soft.*, 175:110924–, 2021.

[62] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. Microservice based architecture: Towards high-availability for stateful applications with kubernetes. In *2019 IEEE 19th international conference on software quality, reliability and security (QRS)*, pages 176–185. IEEE, 2019.

[63] Reza Afshari, Rimba Frida Pusparini, Muhammad Helmi Utomo, Favian Dewanta, and Ridha Muldina Negara. A method for microservices handover in a local area network. In *2020 3rd International Conference on Computer and Informatics Engineering (IC2IE)*, pages 427–431. IEEE, 2020.

[64] Dani Adhipta, Selo Sulistyo, and Widyawan Widyawan. A process checkpoint evaluation at user space of docker framework on distributed computing infrastructure. In *2020 12th International Conference on Information Technology and Electrical Engineering (ICITEE)*, pages 141–145. IEEE, 2020.

[65] Yejin Han, Myunghoon Oh, Jaedong Lee, Seehwan Yoo, Bryan S Kim, and Jongmoo Choi. Achieving performance isolation in docker environments with zns ssds. In *2023 IEEE 12th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 25–31. IEEE, 2023.

[66] Rodrigo H Müller, Cristina Meinhardt, and Odorico M Mendizabal. An architecture proposal for checkpoint/restore on stateful containers. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 267–270, 2022.

[67] Chengxi Pu, Huiming Xu, Hua Jiang, Daigang Chen, and Pengfei Han. An environment-aware and dynamic compression-based approach for

edge computing service migration. In *2022 2nd International Conference on Consumer Electronics and Computer Engineering (ICCECE)*, pages 292–297. IEEE, 2022.

[68] Hein Htet, Nobuo Funabiki, Ariel Kamoyedji, Xudong Zhou, and Minoru Kuribayashi. An implementation of job migration function using criu and podman in docker-based user-pc computing system. In *Proceedings of the 9th International Conference on Computer and Communications Management*, pages 92–97, 2021.

[69] Zeinab Bakhshi, Guillermo Rodriguez-Navas, and Hans Hansson. Analyzing the performance of persistent storage for fault-tolerant stateful fog applications. *Journal of systems architecture*, 144:103004, 2023.

[70] Moiz Arif, Kevin Assogba, and M Mustafa Rafique. Canary: fault-tolerant faas for stateful time-sensitive applications. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2022.

[71] Asad Javed, Keijo Heljanko, Andrea Buda, and Kary Främling. Cefiot: A fault-tolerant iot architecture for edge and cloud. In *2018 IEEE 4th world forum on internet of things (WF-IoT)*, pages 813–818. IEEE, 2018.

[72] Pekka Karhula, Jan Janak, and Henning Schulzrinne. Checkpointing and migration of iot edge functions. In *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*, pages 60–65, 2019.

[73] Patrick Denzler, Daniel Ramsauer, Thomas Preindl, Wolfgang Kastner, and Alexander Gschnitzer. Comparing different persistent storage approaches for containerized stateful applications. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2022.

[74] Shunmugapriya Ramanathan, Koteswararao Kondepu, Marco Tacca, Luca Valcarenghi, Miguel Razo, and Andrea Fumagalli. Container migration of core network component in cloud-native radio access network. In *2020 22nd International Conference on Transparent Optical Networks (ICTON)*, pages 1–5. IEEE, 2020.

[75] Ahmed Chebaane, Simon Spornraft, and Abdelmajid Khelil. Container-based task offloading for time-critical fog computing. In *2020 IEEE 3rd 5G World Forum (5GWF)*, pages 205–211. IEEE, 2020.

[76] Mao V Ngo, Tie Luo, Hieu T Hoang, and Tony QS Ouek. Coordinated container migration and base station handover in mobile edge computing. In *GLOBECOM 2020-2020 IEEE Global Communications Conference*, pages 1–6. IEEE, 2020.

[77] Shunmugapriya Ramanathan, Abhishek Bhattacharyya, Koteswararao Kondepu, Miguel Razo, Marco Tacca, Luca Valcarenghi, and Andrea Fumagalli. Demonstration of containerized central unit live migration in 5g radio access network. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, pages 225–227. IEEE, 2022.

[78] Taha Gharaibeh, Steven Seiden, Mohamed Abouelsaoud, Elias Bou-Harb, and Ibrahim Baggili. Don't, stop, drop, pause: Forensics of container checkpoints (conpoint). In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, pages 1–11, 2024.

[79] Radostin Stoyanov and Martin J Kollingbaum. Efficient live migration of linux containers. In *High Performance Computing: ISC High Performance 2018 International Workshops, Frankfurt/Main, Germany, June 28, 2018, Revised Selected Papers 33*, pages 184–193. Springer, 2018.

[80] Yuqing Qiu, Chung-Horng Lung, Samuel Ajila, and Pradeep Srivastava. Experimental evaluation of lxc container migration for cloudlets using multipath tcp. *Computer Networks*, 164:106900, 2019.

[81] Ranjan Sarpangala Venkatesh, Till Smejkal, Dejan S Milojicic, and Ada Gavrilovska. Fast in-memory criu for docker containers. In *Proceedings of the International Symposium on Memory Systems*, pages 53–65, 2019.

[82] Alexander Droob, Daniel Morratz, Frederik Langkilde Jakobsen, Jacob Carstensen, Magnus Mathiesen, Rune Bohnstedt, Michele Albano, Sergio Moreschini, and Davide Taibi. Fault tolerant horizontal computation offloading. In *2023 IEEE International Conference on Edge Computing and Communications (EDGE)*, pages 177–182. IEEE, 2023.

[83] Diyu Zhou and Yuval Tamir. Fault-tolerant containers using nilicon. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1082–1091. IEEE, 2020.

[84] Georgios L Stavrinides and Helen D Karatza. Fault-tolerant orchestration of bags-of-tasks with application-directed checkpointing in a distributed environment. In *2021 International Conference on Communi-*

cations, Computing, Cybersecurity, and Informatics (CCCI), pages 1–6. IEEE, 2021.

[85] Weilin Cai, Heng Chen, Zhimin Zhuo, Ziheng Wang, and Ninggang An. Flexible supervision system: A fast fault-tolerance strategy for cloud applications in cloud-edge collaborative environments. In *IFIP International Conference on Network and Parallel Computing*, pages 108–113. Springer, 2022.

[86] Giovanni Venâncio and Elias P Duarte Junior. Highly available virtual network functions and services based on checkpointing/restore. *International Journal of Critical Computer-Based Systems*, 11(1-2):115–142, 2024.

[87] Giovanni Venâncio and Elias P Duarte Jr. Nham: an nfv high availability architecture for building fault-tolerant stateful virtual functions and services. In *Proceedings of the 11th Latin-American Symposium on Dependable Computing*, pages 35–44, 2022.

[88] Sang-Hoon Choi and Ki-Woong Park. icontainer: Consecutive checkpointing with rapid resilience for immortal container-based services. *Journal of Network and Computer Applications*, 208:103494, 2022.

[89] Hylson Vescovi Netto, Aldelir Fernando Luiz, Miguel Correia, Luciana de Oliveira Rech, and Caio Pereira Oliveira. Koordinator: A service approach for replicating docker containers in kubernetes. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 00058–00063. IEEE, 2018.

[90] Zhixing Yu, Kejing He, Chao Chen, and Jian Wang. Live container migration via pre-restore and random access memory. In *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 102–109. IEEE, 2020.

[91] Adityas Widjajarto, Deden Witarsyah Jacob, and Muharman Lubis. Live migration using checkpoint and restore in userspace (criu): Usage analysis of network, memory and cpu. *Bulletin of Electrical Engineering and Informatics*, 10(2):837–847, 2021.

[92] Thouraya Louati, Heithem Abbes, Christophe Cérin, and Mohamed Jemni. Lxcloud-cr: towards linux containers distributed hash table

based checkpoint-restart. *Journal of Parallel and Distributed Computing*, 111:187–205, 2018.

[93] Jeongmin Lee, Hyeongbin Kang, Hyeon-jin Yu, Ji-Hyun Na, Jungbin Kim, Jae-hyuck Shin, and Seo-Young Noh. Mdb-kcp: persistence framework of in-memory database with criu-based container checkpoint in kubernetes. *Journal of Cloud Computing*, 13(1):124, 2024.

[94] Suchanat Mangkhangcharoen, Jason Haga, and Prapaporn Rattanatamrong. Migrating deep learning data and applications among kubernetes edge nodes. In *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pages 2004–2010. IEEE, 2021.

[95] Chengcheng Li, Jikun Tao, Guanju Shi, Jie Zeng, Xiangyuan Bu, Chao Zhu, and Yasheng Zhang. Migration for cloud-native vnf in open source mano. In *2023 International Conference on Future Communications and Networks (FCN)*, pages 1–6. IEEE, 2023.

[96] Subhendu Behera, Lipeng Wan, Frank Mueller, Matthew Wolf, and Scott Klasky. Orchestrating fault prediction with live migration and checkpointing. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, pages 167–171, 2020.

[97] Aditya Bhardwaj, Amit Pratap Singh, Priya Sharma, Konika Abid, and Umesh Gupta. Performance evaluation of virtual machine and container-based migration technique. In *International Conference on Data Analytics & Management*, pages 551–558. Springer, 2023.

[98] Chandra Prakash, Debadatta Mishra, Purushottam Kulkarni, and Umesh Bellur. Portkey: Hypervisor-assisted container migration in nested cloud environments. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 3–17, 2022.

[99] Jordi Guitart. Practicable live container migrations in high performance computing clouds: Diskless, iterative, and connection-persistent. *Journal of Systems Architecture*, 152:103157, 2024.

[100] Zhanyuan Di, En Shao, and Mujun He. Reducing the time of live container migration in a workflow. In *IFIP International Conference on Network and Parallel Computing*, pages 263–275. Springer, 2020.

[101] Diyu Zhou and Yuval Tamir. {RRC}: Responsive replicated containers. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 85–100, 2022.

[102] SeungYong Oh and JongWon Kim. Stateful container migration employing checkpoint-based restoration for orchestrated container clusters. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 25–30. IEEE, 2018.

[103] Paulo Souza Junior, Daniele Miorandi, and Guillaume Pierre. Stateful container migration in geo-distributed environments. In *2020 IEEE international conference on cloud computing technology and science (CloudCom)*, pages 49–56. IEEE, 2020.

[104] Henri Schmidt, Zeineb Rejiba, Raphael Eidenbenz, and Klaus-Tycho Förster. Transparent fault tolerance for stateful applications in kubernetes with checkpoint/restore. In *2023 42nd International Symposium on Reliable Distributed Systems (SRDS)*, pages 129–139. IEEE, 2023.

[105] Alan Ford, Costin Raiciu, Mark J. Handley, Olivier Bonaventure, and Christoph Paasch. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 8684, 2020.

[106] Amnon Barak and Oren La'adan. The mosix multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4-5):361–372, 1998.

[107] Zeinab Bakhshi, Guillermo Rodriguez-Navas, and Hans Hansson. Using uppaal to verify recovery in a fault-tolerant mechanism providing persistent state at the edge. In *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–6. IEEE, 2021.

[108] Peng Wei Li, Hong Li Zhao, Hai Tao Yang, and Shu Sun. Performance evaluation of transport protocol in data distribution service middleware. *Advanced Materials Research*, 926:1984–1987, 2014.

[109] Muhammad Ajmal Azad, Rashid Mahmood, and Tahir Mehmood. A comparative analysis of dccp variants (ccid2, ccid3), tcp and udp for

mpeg4 video applications. In *2009 International Conference on Information and Communication Technologies*, pages 40–45. IEEE, 2009.

[110] Panagiotis Papadimitriou and Vassilis Tsaoussidis. Assessment of internet voice transport with tcp. *International Journal of Communication Systems*, 19(4):381–405, 2006.

[111] Yesin Sahraoui, Atef Ghanam, Sofiane Zaidi, Salim Bitam, and Abdelhamid Mellouk. Performance evaluation of tcp and udp based video streaming in vehicular ad-hoc networks. In *2018 International Conference on Smart Communications in Network Technologies (SaCoNeT)*, pages 67–72. IEEE, 2018.

[112] Yu Wang and Alhussein A Abouzeid. Rcp: A reinforcement learning-based retransmission control protocol for delivery and latency sensitive applications. In *2021 International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9. IEEE, 2021.

[113] Sreekanth Asodi, S Vijay Ganesh, E Seshadri, and PK Singh. Evaluation of transport layer protocols for voice transmission in various network scenarios. In *2009 Second International Conference on the Applications of Digital Information and Web Technologies*, pages 238–242. IEEE, 2009.

[114] Muhammad Naeem Tahir and Marcos Katz. Its performance evaluation in direct short-range communication (ieee 802.11 p) and cellular network (5g)(tcp vs udp). In *Towards Connected and Autonomous Vehicle Highways: Technical, Security and Social Challenges*, pages 257–279. Springer, 2021.

[115] Ola Ali, Ahmed Aghmadi, and Osama A Mohammed. Performance evaluation of communication networks for networked microgrids. *e-Prime-Advances in Electrical Engineering, Electronics and Energy*, 8:100521, 2024.

[116] Les Cottrell. Characterization and evaluation of tcp and udp-based transport on real networks. Technical report, SLAC National Accelerator Lab., Menlo Park, CA (United States), 2005.

[117] Meredith Lulling and John Vaughan. A simulation-based comparative evaluation of transport protocols for sip. *Computer Communications*, 29(4):525–537, 2006.

[118] Mark Allman, Vern Paxson, and Ethan Blanton. Tcp congestion control. Technical report, 2009.

[119] Daniel Hallmans, Mohammad Ashjaei, and Thomas Nolte. Analysis of the TSN standards for utilization in long-life industrial distributed control systems. In *IEEE Int. Conf. Emerg. Tech. & Fact. Autom. (ETFA)*, pages 190–197, 2020.

[120] Dimitri Bertsekas and Robert Gallager. *Data networks*. Athena Scientific, 2021.

[121] Belma Turkovic, Fernando A Kuipers, and Steve Uhlig. Fifty shades of congestion control: A performance and interactions evaluation. *arXiv preprint arXiv:1903.03852*, 2019.

[122] IEC 62443-4-2 Security for Industrial Automation and Control Systems Part 4-2: Technical security requirements for IACS components. Standard, International Electrotechnical Commission, Geneva, CH, 2009-2018.

[123] Wesley Eddy. Transmission Control Protocol (TCP). RFC 9293, 2022.

[124] Michael Scharf and Sebastian Kiesel. Nxg03-5: Head-of-line blocking in tcp and sctp: Analysis and measurements. In *IEEE Globecom 2006*, pages 1–5. IEEE, 2006.

[125] User Datagram Protocol. RFC 768, 1980.

[126] Joseph P. Macker, Carsten Bormann, Mark J. Handley, and Brian Adamson. NACK-Oriented Reliable Multicast (NORM) Transport Protocol. RFC 5740, 2009.

[127] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, and Van Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, 2003.

[128] Elisabetta Carrara, Karl Norrman, David McGrew, Mats Naslund, and Mark Baugher. The Secure Real-time Transport Protocol (SRTP). RFC 3711, March 2004.

[129] Randall R. Stewart, Michael Tüxen, and karen Nielsen. Stream Control Transmission Protocol. RFC 9260, 2022.

[130] sctp - linux manual page. `https://man7.org/linux/man-pages/man7/sctp.7.html`. Accessed: 2024-11-20.

[131] sctplib - windows sctp driver. https://github.com/dreibh/sctplib/blob/master/README.win32. Accessed: 2024-11-20.

[132] Vxworks - network stack programmer's guide. supported rfcs. https://docs.windriver.com/r/bundle/Network_Stack_Programmers_Guide_Edition_17_1/page/1591988.html. Accessed: 2024-11-20.

[133] Andreas Jungmaier, Eric Rescorla, and Michael Tüxen. Transport Layer Security over Stream Control Transmission Protocol. RFC 3436, December 2002.

[134] Michael Tüxen, Eric Rescorla, and Robin Seggelmann. Datagram Transport Layer Security (DTLS) for Stream Control Transmission Protocol (SCTP). RFC 6083, January 2011.

[135] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 183–196, 2017.

[136] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, 2021.

[137] Jana Iyengar and Ian Swett. QUIC Loss Detection and Congestion Control. RFC 9002, 2021.

[138] Omg data distribution service version: 1.4. https://www.omg.org/spec/DDS/1.4/PDF. Accessed: 2024-11-19.

[139] The real-time publish-subscribe protocol dds interoperability wire protocol (ddsi-rtpstm) specification version: 2.5. https://www.omg.org/spec/DDSI-RTPS/2.5/PDF. Accessed: 2024-11-19.

[140] Dds extensions for time sensitive networking version: 1.0 - beta 1. https://www.omg.org/spec/DDS-TSN/1.0/Beta1/PDF. Accessed: 2024-11-20.

[141] Dds security version 1.2. https://www.omg.org/spec/DDS-SECURITY/1.2/PDF. Accessed: 2025-05-13.

[142] Opc 10000-1 - ua specification part 1: Overview and concepts. https://reference.opcfoundation.org/Core/Part1/v105/docs/. Accessed: 2025-05-05.

[143] Opc 10000-4 - ua specification part 4: Services 1.05.03. https://reference.opcfoundation.org/Core/Part4/v105/docs/. Accessed: 2025-05-05.

[144] Opc 10000-2 - ua specification part 2: Security 1.05.04. https://reference.opcfoundation.org/Core/Part2/v105/docs/. Accessed: 2025-05-14.

[145] Opc 10000-6 - ua specification part 6: Mappings 1.05.03. https://reference.opcfoundation.org/Core/Part6/v105/docs/. Accessed: 2025-05-05.

[146] Opc 10000-14 - ua specification part 14: Pubsub 1.05.03. https://reference.opcfoundation.org/Core/Part14/v105/docs/. Accessed: 2025-05-05.

[147] Opc 10000-22 - ua specification part 22: Base network model 1.05.03. https://reference.opcfoundation.org/Core/Part22/v105/docs/. Accessed: 2025-05-05.

[148] Tom Bova and Ted Krivoruchka. RELIABLE UDP PROTOCOL. Internet-Draft draft-ietf-sigtran-reliable-udp-00, Internet Engineering Task Force, 1999. Work in Progress.

[149] Eric He, Jason Leigh, Oliver Yu, and Thomas A DeFanti. Reliable blast udp: Predictable high performance bulk data transfer. In *Proceedings. IEEE International Conference on Cluster Computing*, pages 317–324. IEEE, 2002.

[150] Ben Eckart, Xubin He, and Qishi Wu. Performance adaptive udp for high-speed bulk data transfer over dedicated links. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10. IEEE, 2008.

[151] Yunhong Gu and Robert L Grossman. Udt: Udp-based data transfer for high-speed wide area networks. *Computer Networks*, 51(7):1777–1799, 2007.

[152] Ahmed Osama Fathy Atya and Jilong Kuang. Rufc: A flexible framework for reliable udp with flow control. In *8th International Conference*

for Internet Technology and Secured Transactions (ICITST-2013), pages 276–281. IEEE, 2013.

[153] Robert L Grossman, Marco Mazzucco, Harimath Sivakumar, Yiting Pan, and Q Zhang. Simple available bandwidth utilization library for high-speed wide area networks. *The Journal of Supercomputing*, 34:231–242, 2005.

[154] Mark R Meiss et al. Tsunami: A high-speed rate-controlled protocol for file transfer. *Indiana University*, 2004.

[155] Ampq: Advanced message queuing protocol. `https://www.amqp.org/`. Accessed: 2024-11-04.

[156] Zach Shelby. The constrained application protocol (coap). Technical report, IETF RFC 7252, 2014.

[157] Sally Floyd, Mark J. Handley, and Eddie Kohler. Datagram Congestion Control Protocol (DCCP). RFC 4340, 2006.

[158] Lars Eggert, Gorry Fairhurst, and Greg Shepherd. UDP Usage Guidelines. RFC 8085, 2017.

[159] Mark J. Handley, Jitendra Padhye, Sally Floyd, and Joerg Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 5348, 2008.

[160] Abdellatif Elghazi, Maryem Berrezzouq, and Zineelabidine Abdelali. New version of iscsi protocol to secure cloud data storage. In *2016 2nd International Conference on Cloud Computing Technologies and Applications (CloudTech)*, pages 141–145. IEEE, 2016.

[161] Ibm aspera - product webpage. `https://www.ibm.com/products/aspera`. Accessed: 2024-12-05.

[162] Mqtt: Message queue telemetry transport. `https://mqtt.org/`. Accessed: 2024-11-04.

[163] Jinbin Hu, Houqiang Shen, Xuchong Liu, and Jin Wang. Rdma transports in datacenter networks: Survey. *IEEE Network*, 2024.

[164] The roce initiative. `https://www.roceinitiative.org/`. Accessed: 2024-11-04.

[165] Ubaid Abbasi, El Houssine Bourhim, Mouhamad Dieye, and Halima Elbiaze. A performance comparison of container networking alternatives. *IEEE Network*, 33(4):178–185, 2019.

[166] Muskan Shaikh, Pritesh Shah, and Ravi Sekhar. Communication protocols in industry 4.0. In *2023 International Conference on Sustainable Emerging Innovations in Engineering and Technology (ICSEIET)*, pages 709–714. IEEE, 2023.

[167] HMS. Industrial network market shares 2025 according to hms networks. `https://www.hms-networks.com/news/news-details/27-05-2025-hms-networks-report-industrial-trends-2025`. Accessed: 2025-06-04.

[168] Richard Zurawski. *Industrial communication technology handbook*. CRC press, 2014.

[169] Gunnar Prytz. A performance analysis of ethercat and profinet irt. In *2008 IEEE International Conference on Emerging Technologies and Factory Automation*, pages 408–415. IEEE, 2008.

[170] Transmission Control Protocol. RFC 793, 1981.

[171] Sally Floyd, Jamshid Mahdavi, Matt Mathis, and Dr. Allyn Romanow. TCP Selective Acknowledgment Options. RFC 2018, 1996.

[172] Ethan Blanton, Dr. Vern Paxson, and Mark Allman. TCP Congestion Control. RFC 5681, 2009.

[173] Matt Sargent, Jerry Chu, Dr. Vern Paxson, and Mark Allman. Computing TCP's Retransmission Timer. RFC 6298, 2011.

[174] Scott O. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, 1997.

[175] Randall R. Stewart. Stream Control Transmission Protocol. RFC 4960, 2007.

[176] Javier Pastor and Maria-Carmen Belinchon. Stream Control Transmission Protocol (SCTP) Management Information Base (MIB). RFC 3873, 2004.

[177] Michael Tüxen, Vladislav Yasevich, Peter Lei, Randall R. Stewart, and Kacheong Poon. Sockets API Extensions for the Stream Control Transmission Protocol (SCTP). RFC 6458, 2011.

[178] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[179] Qin Zheng and Kang G Shin. On the ability of establishing real-time channels in point-to-point packet-switched networks. *IEEE Transactions on Communications*, 42(234):1096–1105, 1994.

[180] Charles E Spurgeon. *Ethernet: the definitive guide*. " O'Reilly Media, Inc.", 2000.

[181] Andrew S Tanenbaum. *Computer networks*. Pearson Education India, 2003.

[182] Bjarne Johansson, Mats Rågberger, Thomas Nolte, and Alessandro V Papadopoulos. Priority based ethernet handling in real-time end system with ethernet controller filtering. In *IECON 2022–48th Annual Conference of the IEEE Industrial Electronics Society*, pages 1–6. IEEE, 2022.

[183] Herbert Krause. Virtual commissioning of a large lng plant with the dcs 800xa by abb. In *6th EUROSIM Congress on Modelling and Simulation, Ljubljana, Slovénie*, 2007.

[184] Abb webpage - pm891 product specification. `https://compacthardwareselector.automation.abb.com/product/pm891k01`. Accessed: 2025-04-15.