Checkpointing and State Transfer for Industrial Controller Redundancy

Bjarne Johansson[a,b,*], Björn Leander[a,b], Olof Holmgren[a], Thomas Nolte[b], Alessandro V. Papadopoulos[b]

[a] *ABB Process Automation, Process Control Platform, Västerås, Sweden*
[b] *Mälardalen University, Västerås, Sweden*

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.

Keywords: Checkpointing, fault tolerance, communication, industrial control systems, redundancy, reliability, state transfer.

*Corresponding author.
Email address: bjarne.johansson@se.abb.com (Bjarne Johansson)

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 mecha-

nism, 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 1.

Step I presents a literature search and summary covering checkpointing/state-transfer work in industrial controller redundancy and in container/orchestration contexts.[0]

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.

[0]This is a targeted literature scan to identify relevant mechanisms and technologies for our use case; it is *not* a systematic literature review.

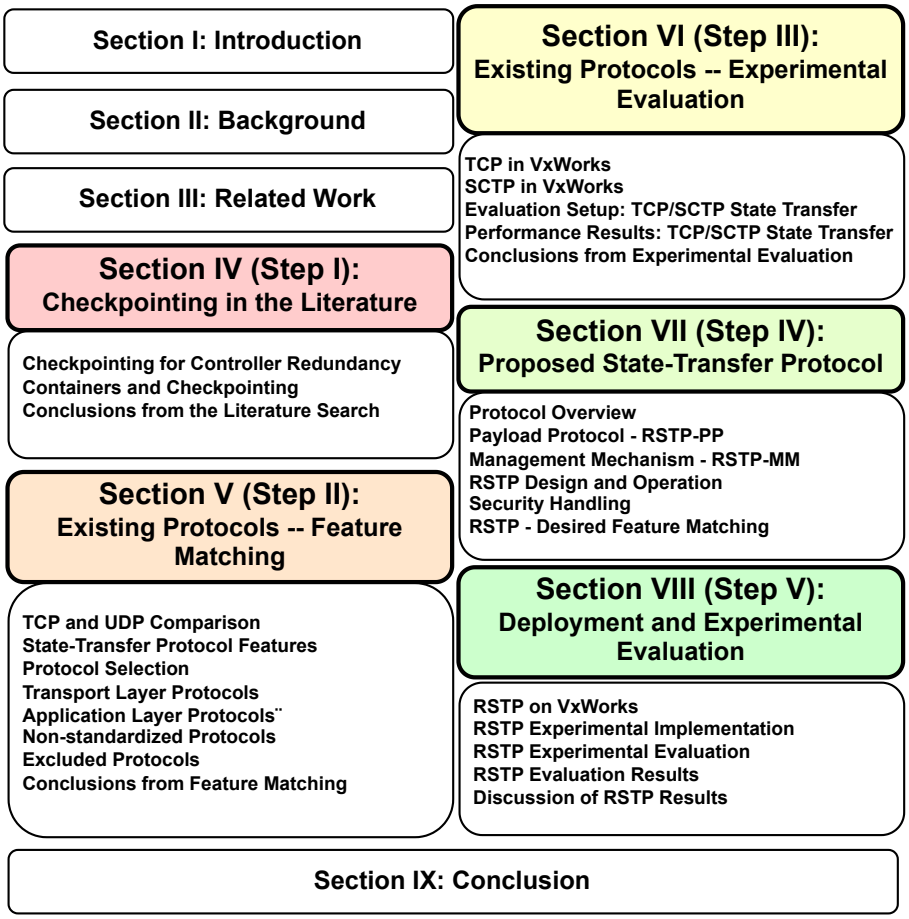


Figure 1: Overview of the paper's sections and the five-step workflow (Step I–V).

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.

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 2 introduces industrial controllers, the execution model, fault tolerance, and container orchestration, and Section 3 provide related work. Section 4 (Step I) details the checkpointing literature search and summarizes the results. Section 5 (Step II) defines desired features for state-transfer protocols, introduces candidate protocols, and assesses them against these features. Section 6 (Step III) experimentally evaluates the top candidates and reports results. Section 7 (Step IV) presents the proposed protocol, and Section 8 (Step V) evaluates it. Section 9 concludes. Figure 1 illustrates the paper structure.

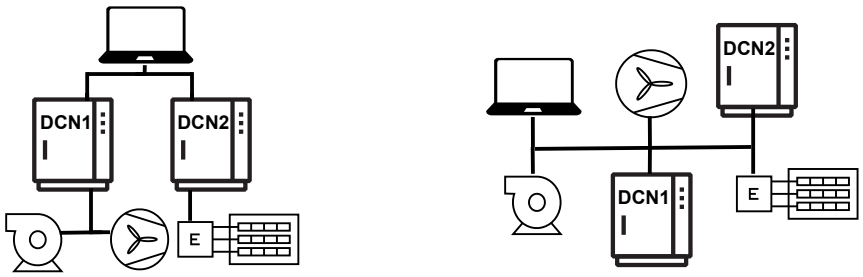
2. Background

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.

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.

Figure 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



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

Figure 2: Controllers, field devices, and upper layers of the automation pyramid in a controller-centric architecture and a network-centric architecture.

requirements and more reliance on IT systems. Figure 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^{TM} 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.

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 3 [19, 20].

As shown in Figure 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*.

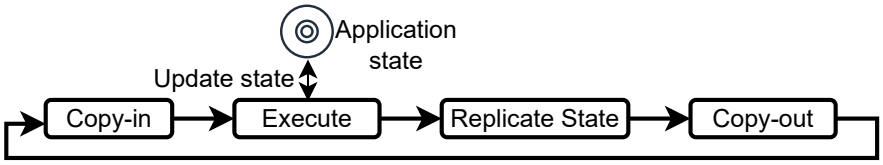


Figure 3: Controller application execution sequence.

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 4 illustrates the application phases and the phases' dependency on input data and internal state from earlier periods. Figure 4 also shows the output from the different phases.

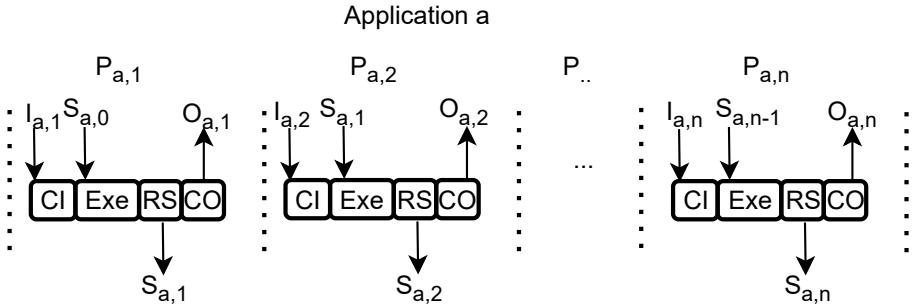


Figure 4: Internal state and application execution phases dependency and relation.

$I_{a,n}$ 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 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.

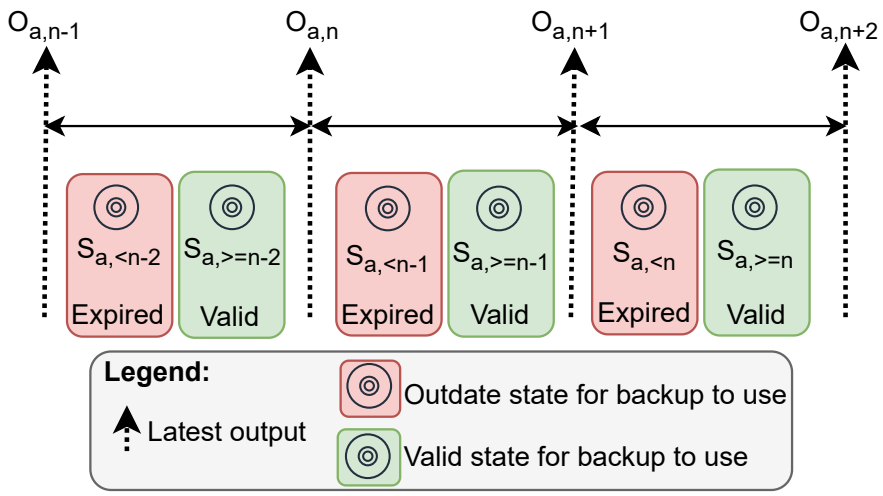


Figure 5: Output to field devices and state and aging on the backup.

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.

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].

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 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 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.

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.

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 1. We followed references to widen the search, and the relevant literature was added to the list in Table 2. Table 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 2, most publications do not discuss checkpointing or the transfer method. The ones that do are further summarized in Section 4.1.1 below.

Table 1: Controller redundancy checkpointing literature query.

```
("controller" OR "PLC") AND "redundan*" AND ("state" OR
"checkpoint*")
```

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 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

Table 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

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 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.

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 3.

Table 3: Checkpointing in container and orchestration context literature query.

```
("checkpoint*" OR "replicat*" OR "restore") AND
("kubernetes" OR "k8s" OR "k3s" OR "orchestrat*" OR
"container*")
```

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 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 4.2.1, gives a summary of the found publications.

Table 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	✗
[79]	App. upgrade	Custom	OPC UA CS	ICS
[80]	Migration	CRIU	❍	✗
[81]	Migration	CRIU	rsync	✗
[82]	Contention	CRIU	No transfer	✗
[4]	Fault tolerance	CRIU	FTP	✗
[83]	Fault tolerance	CRIU	FTP	✗
[84]	Fault tolerance	CRIU	DRBD	✗
[85]	Fault tolerance	Custom	No transfer	✗
[86]	Fault tolerance	Custom	❍	✗
[87]	Fault tolerance	CRIU	❍	✗
[88]	Fault tolerance	CRIU	❍	✗
[89]	Fault tolerance	Custom	No transfer	✗
[90]	Fault tolerance	Custom	Raft	✗
[91]	Migration	Custom (CRIU)	❍	✗
[92]	Migration	CRIU	NFS	✗
[93]	Storage	Custom	❍	✗
[94]	DB persistence	CRIU	No transfer	✗
[95]	Migration	CRIU	NFS	✗
[96]	Migration	CRIU	❍	✗
[97]	Fault tolerance	Custom	❍	✗
[98]	Migration	CRIU	❍	✗
[99]	Migration	CRIU SR-IOV	NFS	✗
[100]	Migration	CRIU	rsync	✗
[101]	Migration	CRIU	❍	✗
[102]	Fault tolerance	CRIU	❍	✗
[103]	Migration	CRIU	❍	✗
[104]	Migration	File replication	❍	✗
[105]	Fault tolerance	CRIU	❍	✗

Legend: ❍ Mentioned (no technical details), ✗ Not mentioned

4.2.1. Summary of Identified Papers

As seen in Table 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 [83]. 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) [81]. MPTCP is a protocol that provides multihoming TCP transfer, that is, multiple paths between communication endpoints [106]. Guitart et al. also move the CRIU checkpoint file using rsync [100].

Widjajarto et al. measure the resource utilization when using CRIU for migration [92]. 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 [95]. 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 [99]. 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 [84, 102]. 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 [107].

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 [94].

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 [80, 96]. Bhardwaj et al. compare the checkpointing performance between containers and virtual machines [98]. Di et al. develop a tool for migrating containers using CRIU [101]. Oh et al. propose a CRIU-based application transparent migration [103]. Schmidt et al. introduce a Kubernetes operator for transparent checkpointing using CRIU in the Kubernetes context [105]. 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 [87, 88]. 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 [79]. 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 [82]. Yu et al. propose a CRIU optimization that checkpoints to memory instead of disk. The checkpointed data is transferred, but without providing details [91]. 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 [104]. Stavrinides et al. let each task checkpoint its data, but the data is not transferred [85]. Cai et al. replicate to a double buffer; one page of the buffer is replicated, while the other is updated by the application [86].

Choi et al. propose a checkpointing solution called iContainer, and Luati et al. optimize storage of checkpointed data using a distributed storage [89, 93]. Behera et al. propose a predictive checkpointing solution for High-Performance Computing (HPC) [97]. 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 [108, 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 [90].

Javed et al. use Apache Kafka to replicate the data produced amongst different processing nodes exemplified in a camera surveillance use case [71].

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 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 2 [79]. 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 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 6.

5. Existing Protocols – Feature Matching

The results from the literature search in Section 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 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.

5.1. TCP and UDP Comparison

Similarly to how we retrieved the checkpointing-related literature in Section 4, we turn to Scopus and Web of Science with the query in Table 5 to retrieve literature related to TCP and UDP performance in a real-time context.

Table 5: TCP and UDP performance comparison literature query.

```
"tcp" AND "udp" AND ("real-time" OR "real time") AND
"performance" AND "evaluation"
```

Table 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:

$$PDR = PacketsReceived \div PacketsSent \tag{1}$$

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 6: TCP and UDP performance comparison.

Ref.	**Topic**	**PDR (highest)**	**Tput (highest)**	**Latency (lowest)**	**Network**
[109]	Dist. RT systems	-	-	UDP	Wired
[110]	Video streaming	TCP	TCP	UDP	Simulated
[111]	Voice streaming	TCP	-	UDP	Wired
[112]	Video streaming	TCP	UDP	-	Simulated
[113]	Congestion ctrl.	-	-	UDP	Simulated
[114]	Voice streaming	TCP	-	UDP	Simulated
[115]	Vehicle comm.	TCP	-	UDP	Wireless
[116]	Microgrid ctrl.	TCP	UDP	UDP	Simulated
[117]	Long-dist.	TCP	TCP	-	Wired
[118]	Session init.	-	UDP	UDP	Simulated

As seen in Table 6, three studies did not measure PDR [109, 113, 118], 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 [119]. In contrast, all studies that measured latency found that UDP offers lower latency. Regarding throughput, three studies favor UDP [112, 116, 118], while two favor TCP [110, 117].

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 5.2.3. Hence, we further explore the desired features of a protocol for transferring state data.

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 [120]. 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 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 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.

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 [121], 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 [122]. 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 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 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 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

5.2.2. Real-time

As described in Section 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 2.2 explains the time span until state data invalidation.

Section 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 8 provides an overview of the real-time related features described above.

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 to obtain a false view of the state, which at failover can result in unexpected, faulty behavior of the new primary, including incorrect setting

Table 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

Table 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

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 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 [123] 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

Table 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

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 amendments. If so, it is described for each protocol and summarized, along with the security protocol feature fulfillment, in Table 10.

5.3. Protocol Selection

The protocols selected are identified from the literature referenced in earlier sections, i.e., in Section 4 and Section 5.1. In addition, we complemented the list by turning to Google with the query shown in Table 11 below.

Table 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 5.2.3, which are matched against security features in Table 9.

Table 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 typical protocols used. If a security protocol is listed under "Security Integration," it means that the use of that specific protocol is mandated.

5.4. Transport Layer Protocols

This section presents the desired feature fulfillment of the standardized transport protocol alternatives.

5.4.1. Transmission Control Protocol - TCP

The Transmission Control Protocol (TCP) was first standardized in the early 1980s [124]. It is a connection-based, reliable protocol that provides an ordered byte stream to its users.

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 [124], 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*,

[1]OPC UA CS (Client/Sever) utilizes OPC UA TCP.
[2]OPC UA PubSub utilizes OPC UA UDP.

Table 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 5.7.1							
COAP	See Section 5.7.2							
DCCP	See Section 5.7.3							
FASP	See Section 5.7.4							
MQTT	See Section 5.7.5							
RoCE	See Section 5.7.6							
Industrial protocols	PROFINET, EthernetIP, EtherCAT, and ModbusTCP. See Section 5.7.7							

as further detailed in Section 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 [125]. 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.

5.4.2. User Datagram Protocol - UDP

Like TCP, the User Datagram Protocol (UDP) was standardized in the early 1980s [126]. 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.

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 [127]. 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 [127]. Hence, it has to rely on post-protocol security but does not explicitly mention any other protocol besides IPsec.

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 [128]. Although RTP typically utilizes UDP, it is not limited to UDP. RTP uses the RTP Control Protocol (RTCP) for control [128]. 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 [129]. 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*.

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 [130]. 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 [131, 132, 133].

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 [134], and later RFC 6083 describes DTLS over SCTP [135]. 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.

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 [136]. RFC 9000 describes the core parts of the protocol, and IETF RFC 9002 defines the congestion control [137, 138]. 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.

5.5. Application Layer Protocols

This section presents the desired feature matching of standardized application layer protocols.

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 [139]. DDS utilizes a UDP-mappable abstract protocol called Real-Time Publish-Subscribe (RTPS) [140]. DDS also has a specification in beta state on how to map DDS onto TSN capable networks [141].

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 [142]. DDS does not mandate the use of the security specification; 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.

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 [143]. OPC UA Client/Server invokes remote procedures exposed by OPC UA servers [144]. 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 [145, 144, 146]. 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 Ta-

ble 10.

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 [147]. 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 [148].

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 [148]. 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 per-

spective. 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 [147]. 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 10.

5.6. Non-standardized Protocols

This section presents the desired feature matching on non-standardized protocols found in the literature.

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 [149]. 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.

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 [150]. 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.

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 [151]. 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.

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 [152]. 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.

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 [153]. 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.

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 [154] SABUL, like RBUDP and PA-UDP, uses UDP for data exchange and TCP for acknowledgment and rate control; see Section 5.6.2 and Section 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.

5.6.7. Tsunami

Tsunami is an application protocol designed to achieve faster file transfer than FTP over TCP by FTP over UDP [155]. Tsunami is an application-layer protocol, and like SABUL and others (see Section 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 ex-

plicitly 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.

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.

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 [156].

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.

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 [157].

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.

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 [158]. The motivation behind DCCP is to spare applications using data-

gram protocols the need for congestion control implementation, as congestion control is highly recommended for Internet-bound traffic [159, 160].

DCCP is excluded due to its limited spread and support in operating systems' network stacks.

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 [161, 162], 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.

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 [163].

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.

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 [164, 165]. 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 [166].

RoCE and RDMA are excluded since they require OS Kernel and/or hardware support.

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[167, 168].

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 [169, 170].

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.

5.8. Conclusions from Feature Matching

From the desired protocol feature matching, summarized in Table 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 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 5.4.5.

As both OPC UA TCP and SCTP only partly fulfill RT_PT, Section 6 evaluates the transfer times and the predictability of those. OPC UA TCP uses TCP. Hence, the evaluation utilizes TCP, further elaborated in Section 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 6).

6. Existing Protocols – Experimental Evaluation

As shown and motivated in Section 5.8, OPC UA TCP and SCTP are the top candidates. OPC UA TCP runs on top of TCP, and as described in Section 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 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 6.1. In Section 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 6.3. This evaluation is followed by a discussion of the results and their implications for our redundancy use case.

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 [133]. 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 [124, 171]. RFC 2018 introduces Selective ACKnowledgment (SACK), which allows a receiver to inform a sender about segments it has received in the event of losses [172]. 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 [173]. 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 1 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 [173]. 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 [174]. 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 2.

$$ssthresh = max(\frac{bytesInFlight}{2}, mss \times 2) \tag{2}$$

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 [173]. Algorithm 1 summarizes the behavior.

RFC 6298 describes how TCP should derive the retransmission timer and timeout from round-trip time measurements [174]. 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 [175]. RFC 6298 also dictates that the retransmission timer should be doubled for every retransmission due to a timeout, and VxWorks follows this rule.

6.2. SCTP in VxWorks

The VxWorks 24.03 SCTP implementation follows the second latest SCTP RFC, RFC 4960 [176, 133]. 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 [177, 178].

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

Algorithm 1 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**

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 [176]. 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 [176]. 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 [176, 173]. 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 [176]. 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 3.

$$ssthresh = max(\frac{cwnd}{2}, mtu \times 4) \tag{3}$$

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 2 [176].

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.

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 3 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 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.

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

Algorithm 2 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 3 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
```

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 6.1 and Section 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 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 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 14. The minimum retransmission timeout in Table 14 corresponds to the minimum retransmission timeout according to RFC 6298 [174], which should be one second for TCP and similarly one second for SCTP by default [176]. 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 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 3. The OS on the DCN is VxWorks 24.03, and each DCN is a mini-PC with 2 GHz Intel

Table 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

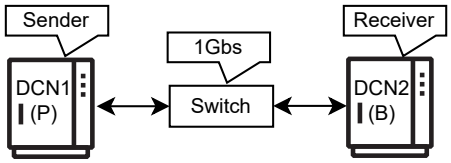


Figure 6: The evaluation setup used.

I7-9700T, with 16 GB RAM.

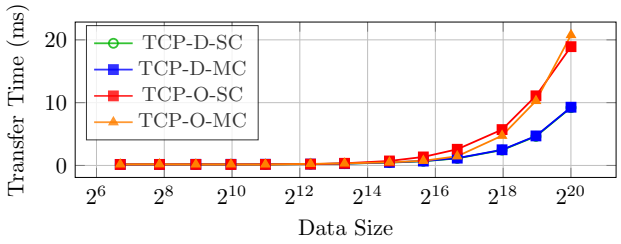


Figure 7: TCP - no losses, maximum transfer time, with default (D) or optimized (O) settings and either Single Connection (SC) or Multiple Connections (MC).

6.4. Performance Results: TCP/SCTP State Transfer

Tables 15 and 16 detail the measured transfer times for the evaluation scenarios summarized in Table 13. The theoretical limit for 1 Gbps Ethernet is 125 MB/second, or 8 milliseconds to transfer 1 MB. As shown in Figure 7 and Table 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 7. Nevertheless, when packet losses occur, the optimized version significantly outperforms the default settings, as presented in Figure 9 and detailed in Table 15.

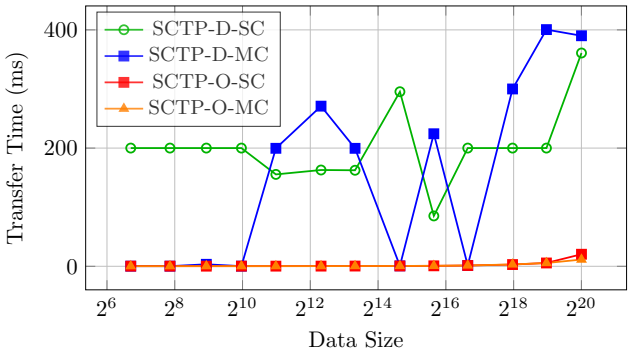


Figure 8: SCTP - no losses, maximum transfer time, with default (D) or optimized (O) settings and either Single Connection (SC) or Multiple Connections (MC).

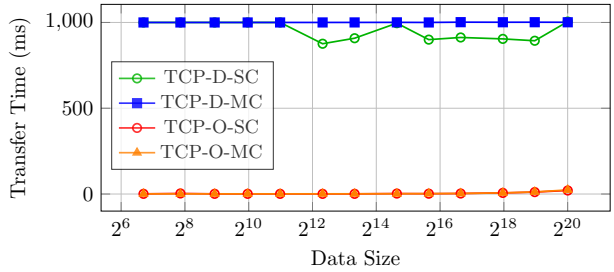


Figure 9: TCP - loss of first segment, maximum transfer time, with default (D) or optimized (O) settings using either Single Connection (SC) or Multiple Connections (MC).

For SCTP, as shown in Figure 8 and detailed in Table 16, the optimized 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. Although optimization substantially improves results for both

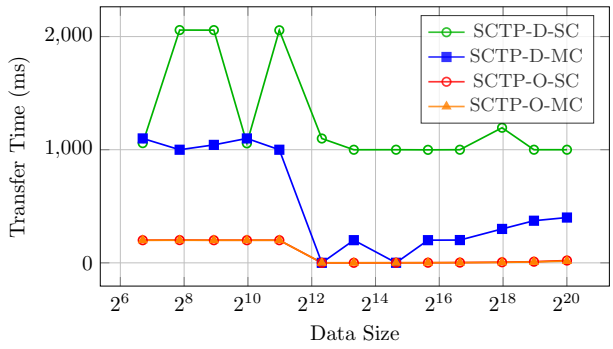


Figure 10: SCTP - loss of first segment, maximum transfer time, with default (D) or optimized (O) settings using either Single Connection (SC) or Multiple Connections (MC).

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.

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 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 [125]. 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 [175].

Table 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 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

Conclusions:

- 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.

7. Proposed State-Transfer Protocol

As shown in Section 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 5.

7.1. Protocol Overview

The protocol we propose is named Reliable State Transfer Protocol (RSTP). Similar to RBUDP (see Section 5.6.2) and PA-UDP (see Section 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 7.3. Figure 11 presents a high-level overview of RSTP, and further details are provided in the two subsections that follow.

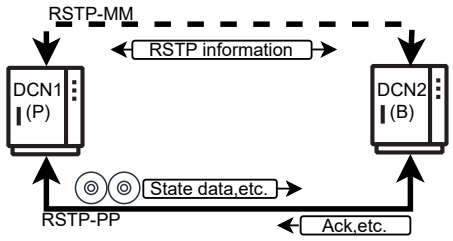


Figure 11: RSTP high-level overview.

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 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 2, motivates the channel concept, further elaborated in Section 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.

7.2.1. Packet Format

RSTP-PP is a packet-oriented protocol, and Table 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 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 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 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

Table 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.

the receiver to calculate the byte offset for each incoming packet and store the received data in a contiguous buffer. Section 7.3 explains the RSTP-PP payload size.

$TtlSzCyc$ 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 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 18 shows the acknowledgment packet layout, and the following describes RSTP-PP acknowledgment fields.

Table 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.

$HType$ 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 transfer 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 $AckBlocks = |AB|$.

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.[179, 180]. The RSTP channel concept is based on the work of Zheng et al., as summarized below [180].

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 [180]. 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 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.

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 mech-

anism; 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 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 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 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 $TCycle$.

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, 181]. 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 $p_s = 1 - q_s$ [182]. 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.

$$P_{\text{msg success}}(R) = \sum_{k=0}^{R} \binom{N+k-1}{k} p^k \, q^N. \tag{4}$$

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 12.

Figure 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.

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.

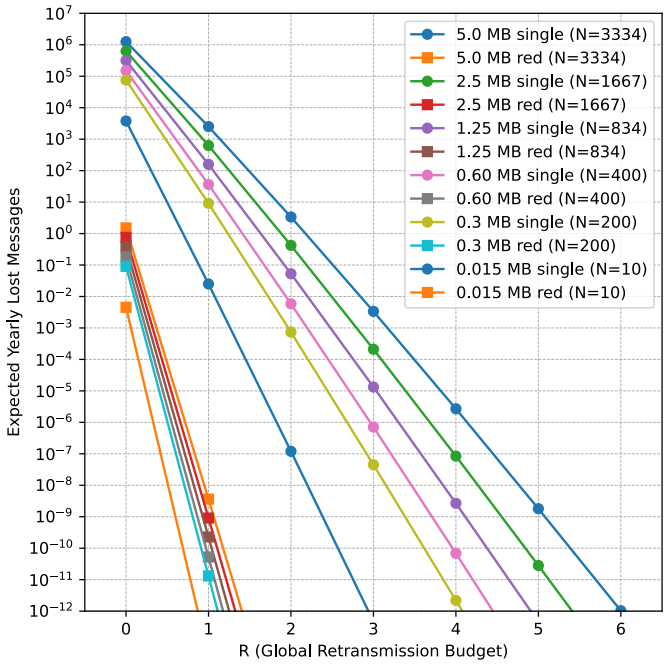


Figure 12: Expected yearly message loss for messages with varying sizes and retransmission budgets, sent every hundred milliseconds, using single (single) and redundant (red) links (network paths).

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).

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.

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:
$sz_{lp} = TtlSzCyc - ((N-1)PayloadSz_{ht1})$.

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.

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.

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 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 14, which details the RSTP-PP receiver flow in the backup.

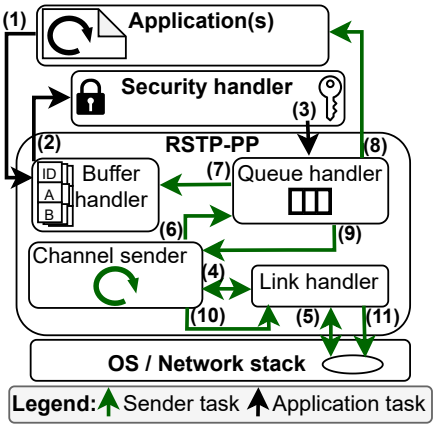


Figure 13: RSTP-PP sender design, internal components interaction, and data flow during normal operation.

As described in Section 2, an application periodically sends its updated internal state to the backup during normal operation; step (1) in Figure 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, 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 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 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 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).

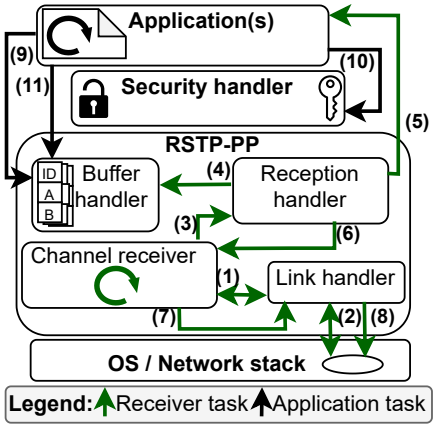


Figure 14: RSTP-PP receiver design, internal components interaction, and data flow during normal operation.

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 degree of parallelism is achieved by having two or more tasks, where one handles acknowledgment sending and the other handles packet reception.

7.4.2. RSTP - Configuration

Figure 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 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 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.

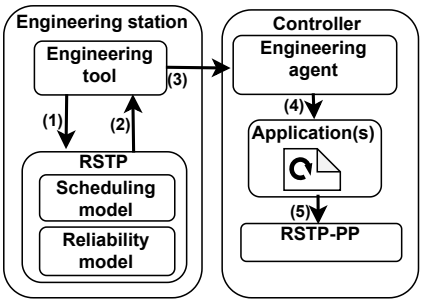


Figure 15: Configuration example.

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.

7.5. Security Handling

The function of the *Security handler* is to add security features as defined in section 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 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 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

Table 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

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 [147] works, as shown in Fig. 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.

7.6. RSTP - Desired Feature Matching

We conclude the design section by showing how RSTP fulfills the desirable features presented in Section 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 pre-

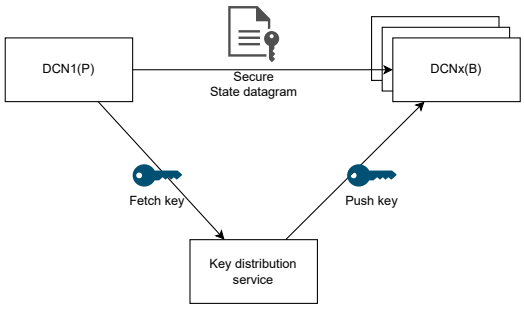


Figure 16: Key distribution according to pattern from OPC UA Secure pub-sub.

vent 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.

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.

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 [133]. Figure 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 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 17.

The number of network stacks (and tasks) in VxWorks can be configured [133]. A feature utilized by Johansson et al. as a foundation for processing time-sensitive traffic with a higher-priority network task [183]. 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 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 18 serves as an example.

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.

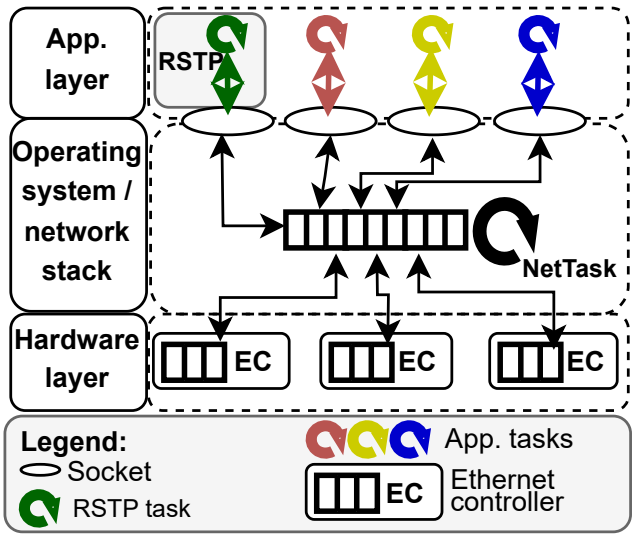


Figure 17: RSTP and other network-dependent applications using the default configured VxWorks network stack. RSTP, multiple applications, and Ethernet Controllers (EC) share one network stack and Network Task (NetTask).

RSTP Sender: The RSTP Sender is a simplified version of what is depicted in Figure 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 18 to prioritize the RSTP traffic.

RSTP Receiver: The RSTP Receiver prototype is a simplified version of the receiver depicted in Figure 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.

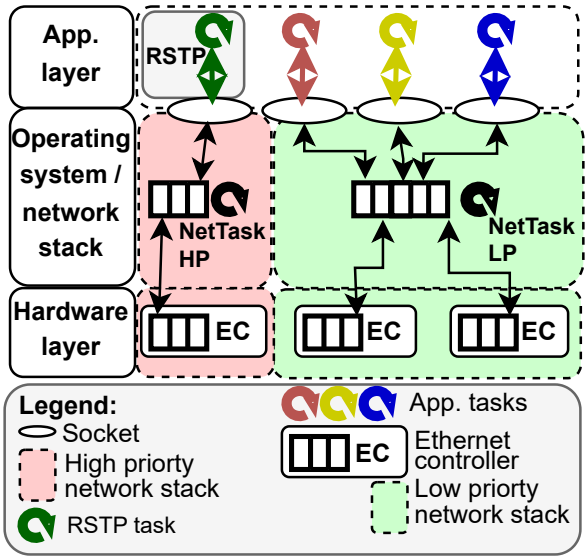


Figure 18: Network-dependent applications using a customized VxWorks network stack configuration with a High Priority (HP) network stack instance for handling RSTP traffic and a Low Priority (LP) network stack instance for best-effort time-insensitive traffic.

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 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 20.

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, 184]. 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 [185], meaning a controller can host many applications of the size mentioned above, a limit

Table 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%

likely to increase with newer generation controllers.

Table 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, 184]. 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 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) [183].

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 8.4, presents the result.

8.4. RSTP Evaluation Results

Figure 19 shows the RSTP prototype performance under the same arrangement previously used to evaluate TCP and SCTP, described in Section 6.3. It provides an overview and displays the longest measured transfer times, while Table 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 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

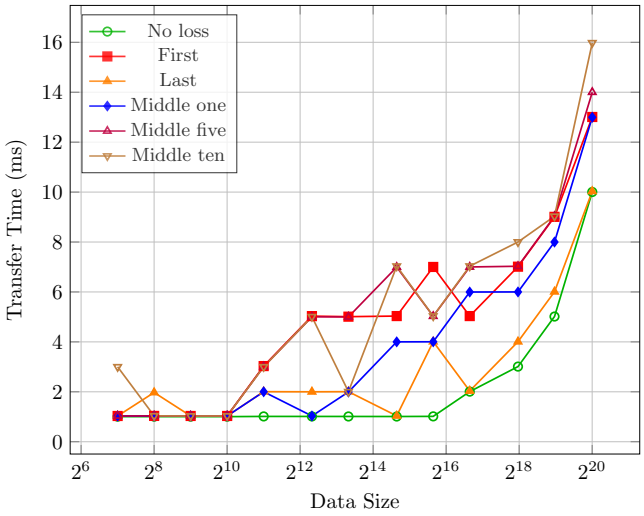


Figure 19: RSTP max transfer times. First, last, and middle denote if the first, last, or middle packets are lost.

Table 22 shows the results from the concurrent multiple-application RSTP evaluation arrangement described in Table 20. Transfer times increase 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.

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

Table 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

with further optimizations. Such optimizations are reserved for future work. The protocol is demonstrated to be performant, achieving 75% utilization across multiple channels (and applications) under significant loss conditions, as shown in the measurements in Table 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 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 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 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 work.

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.

References

[1] B. Leander, B. Johansson, T. Lindström, O. Holmgren, T. Nolte, A. V. Papadopoulos, Dependability and security aspects of network-centric control, in: 2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA), 2023, pp. 1–8. doi:10.1109/ETFA54631.2023.10275344.

[2] D. Bruckner, R. 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 (2018) 1–10.

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

[4] Y. Jia, T. Wang, T. Qiu, X. Zhang, R. Wang, T. Wo, Fault tolerance of stateful microservices for industrial edge scenarios, in: 2023 IEEE International Conference on Joint Cloud Computing (JCC), IEEE, 2023, pp. 50–56.

[5] B. Johansson, M. Rågberger, T. Nolte, A. V. Papadopoulos, Kubernetes orchestration of high availability distributed control systems, in: IEEE Int. Conf. on Ind. Tech. (ICIT), 2022.

[6] T. Goldschmidt, S. Hauck-Stattelmann, S. Malakuti, S. Grüner, Container-based architecture for flexible industrial control applications, Journal of Systems Architecture 84 (2018) 28–36.

[7] H. Koziolek, A. Burger, P. Abdulla, J. Rückert, S. Sonar, P. Rodriguez, Dynamic updates of virtual plcs deployed as kubernetes microservices, in: European Conference on Software Architecture, Springer, 2021, pp. 3–19.

[8] A. Moga, T. Sivanthi, C. Franke, OS-Level Virtualization for Industrial Automation Systems: Are We There Yet?, Association for Computing Machinery, New York, NY, USA, 2016, p. 1838–1843.
URL `https://doi.org/10.1145/2851613.2851737`

[9] E. Dubrova, Fault-tolerant design, Vol. 8, Springer, 2013.

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

[11] A. Simion, C. Bira, A review of redundancy in plc-based systems, Advanced Topics in Optoelectronics, Microelectronics, and Nanotechnologies XI 12493 (2023) 269–276.

[12] A. Avižienis, Design of fault-tolerant computers, in: Proceedings of the November 14-16, 1967, fall joint computer conference, 1967, pp. 733–743.

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

[14] T. Kampa, A. El-Ankah, D. 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), IEEE, 2023, pp. 1–8.

[15] B. Leander, B. Johansson, S. Mubeen, M. Ashjaei, T. Lindström, Redundancy link security analysis: An automation industry perspective, in: 2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA), IEEE, 2024, pp. 1–8.

[16] Vxworks - real-time operating system, `https://www.windriver.com/products/vxworks`, accessed: 2025-08-25.

[17] D. Bruckner, M.-P. Stănică, R. Blair, S. Schriegel, S. Kehrer, M. Seewald, T. Sauter, An introduction to opc ua tsn for industrial communication systems, Proceedings of the IEEE 107 (6) (2019) 1121–1131.

[18] O-pas standard, version 2.1, `https://publications.opengroup.org/c230`, accessed: 2025-04-30.

[19] M. A. Sehr, M. Lohstroh, M. Weber, I. Ugalde, M. Witte, J. Neidig, S. Hoeme, M. Niknami, E. A. Lee, Programmable logic controllers in the context of industry 4.0, IEEE Transactions on Industrial Informatics 17 (5) (2020) 3523–3533.

[20] S. Stattelmann, S. Sehestedt, T. Gamer, Optimized incremental state replication for automation controllers, in: Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA), IEEE, 2014, pp. 1–8.

[21] H. Koziolek, A. Burger, A. P. Peedikayil, Fast state transfer for updates and live migration of industrial controller runtimes in container orchestration systems, Journal of Systems and Software 211 (2024) 112004.

[22] D. Powell, G. Bonn, D. T. Seaton, P. Verissimo, F. Waeselynck, The delta-4 approach to dependability in open distributed computing systems., in: FTCS, Vol. 18, Citeseer, 1988.

[23] B. Johansson, M. Rågberger, A. V. Papadopoulos, T. Nolte, Heartbeat bully: failure detection and redundancy role selection for network-centric controller, in: IECON 2020 The 46th Annual Conference of the IEEE Industrial Electronics Society, IEEE, 2020, pp. 2126–2133.

[24] B. Johansson, M. Rågberger, A. V. Papadopoulos, T. 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), IEEE, 2023, pp. 1–8.

[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] L. LAMPORT, R. SHOSTAK, M. PEASE, The byzantine generals problem, ACM Transactions on Programming Languages and Systems 4 (3) (1982) 382–401.

[27] N. Budhiraja, K. Marzullo, F. B. Schneider, S. Toueg, The primary-backup approach, Distributed systems 2 (1993) 199–216.

[28] L. Lamport, The part-time parliament, ACM Trans. Comput. Syst. 16 (2) (1998) 133–169. doi:10.1145/279227.279229.
URL `https://doi.org/10.1145/279227.279229`

[29] D. Ongaro, J. Ousterhout, In search of an understandable consensus algorithm, in: 2014 USENIX annual technical conference (USENIX ATC 14), 2014, pp. 305–319.

[30] V. Struhár, M. Behnam, M. Ashjaei, A. V. Papadopoulos, Real-Time Containers: A Survey, in: 2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020), Vol. 80 of OpenAccess Series in Informatics (OASIcs), 2020, pp. 7:1–7:9. doi:10.4230/OASIcs.Fog-IoT.2020.7.

[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: 2025-04-04.

[34] E. Casalicchio, Container orchestration: A survey, Systems Modeling: Methodologies and Tools (2019) 221–235.

[35] Kubernetes webpage, https://kubernetes.io/, accessed: 2025-04-02.

[36] J. 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, Springer, 2020, pp. 74–85.

[37] Y. Zhao, F. Liu, The implementation of a dual-redundant control system, Control engineering practice 12 (4) (2004) 445–453.

[38] R. Ma, P. Cheng, Z. Zhang, W. Liu, Q. Wang, Q. Wei, Stealthy attack against redundant controller architecture of industrial cyber-physical system, IEEE Internet of Things Journal 6 (6) (2019) 9783–9793.

[39] B. Johansson, O. Holmgren, T. Nolte, A. V. Papadopoulos, Partible state replication for industrial controller redundancy, in: 2024 IEEE International Conference on Industrial Technology (ICIT), IEEE, 2024, pp. 1–8.

[40] Z. Bakhshi, G. Rodriguez-Navas, H. Hansson, Fault-tolerant permanent storage for container-based fog architectures, in: 2021 22nd IEEE International Conference on Industrial Technology (ICIT), Vol. 1, IEEE, 2021, pp. 722–729.

[41] J. Nouruzi-Pur, J. Lambrecht, T. D. Nguyen, A. Vick, J. 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), IEEE, 2022, pp. 1–8.

[42] J. Ždánsky, K. Rástočnỳ, Influence of redundancy on safety integrity of srcs with safety plc, in: 2014 ELEKTRO, IEEE, 2014, pp. 508–512.

[43] G. S. OV, A. Karthikeyan, K. Karthikeyan, P. Sanjeevikumar, S. K. Thomas, A. Babu, Critical review of scada and plc in smart buildings and energy sector, Energy Reports 12 (2024) 1518–1530.

[44] R. Barkur, 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), IEEE, 2017, pp. 1–5.

[45] M. Nankya, R. Chataut, R. Akl, Securing industrial control systems: components, cyber threats, and machine learning-driven defense strategies, Sensors 23 (21) (2023) 8840.

[46] S. Lee, J. Kang, S. S. Choi, M. T. Lim, Design of ptp tc/slave over seamless redundancy network for power utility automation, IEEE Transactions on Instrumentation and Measurement 67 (7) (2018) 1617–1625.

[47] P. Zhou, W. He, Z. Zhang, The reheating furnace control system design based on siemens pcs7, in: 2008 World Automation Congress, IEEE, 2008, pp. 1–4.

[48] L. Rajesh, 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) (2019) 22–27.

[49] J. Luo, M. Kang, E. Bisse, M. Veldink, D. Okunev, S. Kolb, J. G. Tylka, A. Canedo, A quad-redundant plc architecture for cyber-resilient industrial control systems, IEEE Embedded Systems Letters 13 (4) (2020) 218–221.

[50] M. Wahler, M. Oriol, Disruption-free software updates in automation systems, in: Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA), IEEE, 2014, pp. 1–8.

[51] Y. Kaneko, T. Ito, A reliable cloud-based feedback control system, in: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), IEEE, 2016, pp. 880–883.

[52] M. Gundall, J. Stegmann, M. Reichardt, H. D. Schotten, Downtime optimized live migration of industrial real-time control services, in: 2022

IEEE 31st International Symposium on Industrial Electronics (ISIE), IEEE, 2022, pp. 253–260.

[53] S. Grüner, S. Malakuti, J. Schmitt, T. Terzimehic, M. Wenger, H. Elfaham, Alternatives for flexible deployment architectures in industrial automation systems, in: 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA), Vol. 1, IEEE, 2018, pp. 35–42.

[54] L. Vogt, A. Klose, V. Khaydarov, C. Vockeroth, C. Endres, L. 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), IEEE, 2023, pp. 1–4.

[55] M. Barletta, L. De Simone, R. Della Corte, C. Di Martino, Failover timing analysis in orchestrating container-based critical applications, in: 2024 19th European Dependable Computing Conference (EDCC), IEEE, 2024, pp. 81–84.

[56] K. Govindaraj, A. 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), Vol. 1, IEEE, 2018, pp. 83–90.

[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] Z. Khan, F. 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), IEEE, 2019, pp. 590–594.

[59] A. Bhardwaj, C. Rama Krishna, A container-based technique to improve virtual machine migration in cloud computing, IETE Journal of Research 68 (1) (2022) 401–416.

[60] S. Zhang, N. Chen, H. Zhang, Y. Xue, R. 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, Springer, 2018, pp. 240–250.

[61] L. A. Vayghan, M. A. Saied, M. Toeroe, F. Khendek, A kubernetes controller for managing the availability of elastic microservice based stateful applications, J. Syst. and Soft. 175 (2021) 110924–.

[62] L. A. Vayghan, M. A. Saied, M. Toeroe, F. 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), IEEE, 2019, pp. 176–185.

[63] R. Afshari, R. F. Pusparini, M. H. Utomo, F. Dewanta, R. M. Negara, A method for microservices handover in a local area network, in: 2020 3rd International Conference on Computer and Informatics Engineering (IC2IE), IEEE, 2020, pp. 427–431.

[64] D. Adhipta, S. Sulistyo, W. 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), IEEE, 2020, pp. 141–145.

[65] Y. Han, M. Oh, J. Lee, S. Yoo, B. S. Kim, J. Choi, Achieving performance isolation in docker environments with zns ssds, in: 2023 IEEE 12th Non-Volatile Memory Systems and Applications Symposium (NVMSA), IEEE, 2023, pp. 25–31.

[66] R. H. Müller, C. Meinhardt, O. M. Mendizabal, An architecture proposal for checkpoint/restore on stateful containers, in: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, 2022, pp. 267–270.

[67] C. Pu, H. Xu, H. Jiang, D. Chen, P. 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), IEEE, 2022, pp. 292–297.

[68] H. Htet, N. Funabiki, A. Kamoyedji, X. Zhou, M. 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, 2021, pp. 92–97.

[69] Z. Bakhshi, G. Rodriguez-Navas, H. Hansson, Analyzing the performance of persistent storage for fault-tolerant stateful fog applications, Journal of systems architecture 144 (2023) 103004.

[70] M. Arif, K. Assogba, M. M. Rafique, Canary: fault-tolerant faas for stateful time-sensitive applications, in: SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2022, pp. 1–16.

[71] A. Javed, K. Heljanko, A. Buda, K. Främling, Cefiot: A fault-tolerant iot architecture for edge and cloud, in: 2018 IEEE 4th world forum on internet of things (WF-IoT), IEEE, 2018, pp. 813–818.

[72] P. Karhula, J. Janak, H. Schulzrinne, Checkpointing and migration of iot edge functions, in: Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking, 2019, pp. 60–65.

[73] P. Denzler, D. Ramsauer, T. Preindl, W. Kastner, A. Gschnitzer, Comparing different persistent storage approaches for containerized stateful applications, in: 2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA), IEEE, 2022, pp. 1–8.

[74] S. Ramanathan, K. Kondepu, M. Tacca, L. Valcarenghi, M. Razo, A. Fumagalli, Container migration of core network component in cloud-native radio access network, in: 2020 22nd International Conference on Transparent Optical Networks (ICTON), IEEE, 2020, pp. 1–5.

[75] A. Chebaane, S. Spornraft, A. Khelil, Container-based task offloading for time-critical fog computing, in: 2020 IEEE 3rd 5G World Forum (5GWF), IEEE, 2020, pp. 205–211.

[76] M. V. Ngo, T. Luo, H. T. Hoang, T. Q. Ouek, Coordinated container migration and base station handover in mobile edge computing, in: GLOBECOM 2020-2020 IEEE Global Communications Conference, IEEE, 2020, pp. 1–6.

[77] S. Ramanathan, A. Bhattacharyya, K. Kondepu, M. Razo, M. Tacca, L. Valcarenghi, A. Fumagalli, Demonstration of containerized central unit live migration in 5g radio access network, in: 2022 IEEE 8th International Conference on Network Softwarization (NetSoft), IEEE, 2022, pp. 225–227.

[78] T. Gharaibeh, S. Seiden, M. Abouelsaoud, E. Bou-Harb, I. Baggili, Don't, stop, drop, pause: Forensics of container checkpoints (conpoint), in: Proceedings of the 19th International Conference on Availability, Reliability and Security, 2024, pp. 1–11.

[79] H. Koziolek, A. Burger, P. Abdulla, J. Rückert, S. Sonar, P. Rodriguez, Dynamic updates of virtual plcs deployed as kubernetes microservices, in: European Conference on Software Architecture, Springer, 2021, pp. 3–19.

[80] R. Stoyanov, M. 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, Springer, 2018, pp. 184–193.

[81] Y. Qiu, C.-H. Lung, S. Ajila, P. Srivastava, Experimental evaluation of lxc container migration for cloudlets using multipath tcp, Computer Networks 164 (2019) 106900.

[82] R. S. Venkatesh, T. Smejkal, D. S. Milojicic, A. Gavrilovska, Fast in-memory criu for docker containers, in: Proceedings of the International Symposium on Memory Systems, 2019, pp. 53–65.

[83] A. Droob, D. Morratz, F. L. Jakobsen, J. Carstensen, M. Mathiesen, R. Bohnstedt, M. Albano, S. Moreschini, D. Taibi, Fault tolerant horizontal computation offloading, in: 2023 IEEE International Conference on Edge Computing and Communications (EDGE), IEEE, 2023, pp. 177–182.

[84] D. Zhou, Y. Tamir, Fault-tolerant containers using nilicon, in: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2020, pp. 1082–1091.

[85] G. L. Stavrinides, H. D. Karatza, Fault-tolerant orchestration of bags-of-tasks with application-directed checkpointing in a distributed environment, in: 2021 International Conference on Communications, Computing, Cybersecurity, and Informatics (CCCI), IEEE, 2021, pp. 1–6.

[86] W. Cai, H. Chen, Z. Zhuo, Z. Wang, N. 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, Springer, 2022, pp. 108–113.

[87] G. Venâncio, E. P. D. Junior, Highly available virtual network functions and services based on checkpointing/restore, International Journal of Critical Computer-Based Systems 11 (1-2) (2024) 115–142.

[88] G. Venâncio, E. 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, 2022, pp. 35–44.

[89] S.-H. Choi, K.-W. Park, icontainer: Consecutive checkpointing with rapid resilience for immortal container-based services, Journal of Network and Computer Applications 208 (2022) 103494.

[90] H. V. Netto, A. F. Luiz, M. Correia, L. de Oliveira Rech, C. P. Oliveira, Koordinator: A service approach for replicating docker containers in kubernetes, in: 2018 IEEE Symposium on Computers and Communications (ISCC), IEEE, 2018, pp. 00058–00063.

[91] Z. Yu, K. He, C. Chen, J. Wang, Live container migration via prerestore 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), IEEE, 2020, pp. 102–109.

[92] A. Widjajarto, D. W. Jacob, M. 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) (2021) 837–847.

[93] T. Louati, H. Abbes, C. Cérin, M. Jemni, Lxcloud-cr: towards linux containers distributed hash table based checkpoint-restart, Journal of Parallel and Distributed Computing 111 (2018) 187–205.

[94] J. Lee, H. Kang, H.-j. Yu, J.-H. Na, J. Kim, J.-h. Shin, S.-Y. Noh, Mdb-kcp: persistence framework of in-memory database with criu-based container checkpoint in kubernetes, Journal of Cloud Computing 13 (1) (2024) 124.

[95] S. Mangkhangcharoen, J. Haga, P. 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), IEEE, 2021, pp. 2004–2010.

[96] C. Li, J. Tao, G. Shi, J. Zeng, X. Bu, C. Zhu, Y. Zhang, Migration for cloud-native vnf in open source mano, in: 2023 International Conference on Future Communications and Networks (FCN), IEEE, 2023, pp. 1–6.

[97] S. Behera, L. Wan, F. Mueller, M. Wolf, S. Klasky, Orchestrating fault prediction with live migration and checkpointing, in: Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, 2020, pp. 167–171.

[98] A. Bhardwaj, A. P. Singh, P. Sharma, K. Abid, U. Gupta, Performance evaluation of virtual machine and container-based migration technique, in: International Conference on Data Analytics & Management, Springer, 2023, pp. 551–558.

[99] C. Prakash, D. Mishra, P. Kulkarni, U. Bellur, Portkey: Hypervisor-assisted container migration in nested cloud environments, in: Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, 2022, pp. 3–17.

[100] J. Guitart, Practicable live container migrations in high performance computing clouds: Diskless, iterative, and connection-persistent, Journal of Systems Architecture 152 (2024) 103157.

[101] Z. Di, E. Shao, M. He, Reducing the time of live container migration in a workflow, in: IFIP International Conference on Network and Parallel Computing, Springer, 2020, pp. 263–275.

[102] D. Zhou, Y. Tamir, {RRC}: Responsive replicated containers, in: 2022 USENIX Annual Technical Conference (USENIX ATC 22), 2022, pp. 85–100.

[103] S. Oh, J. Kim, Stateful container migration employing checkpoint-based restoration for orchestrated container clusters, in: 2018 International Conference on Information and Communication Technology Convergence (ICTC), IEEE, 2018, pp. 25–30.

[104] P. S. Junior, D. Miorandi, G. Pierre, Stateful container migration in geo-distributed environments, in: 2020 IEEE international conference on cloud computing technology and science (CloudCom), IEEE, 2020, pp. 49–56.

[105] H. Schmidt, Z. Rejiba, R. Eidenbenz, K.-T. Förster, Transparent fault tolerance for stateful applications in kubernetes with checkpoint/restore, in: 2023 42nd International Symposium on Reliable Distributed Systems (SRDS), IEEE, 2023, pp. 129–139.

[106] A. Ford, C. Raiciu, M. J. Handley, O. Bonaventure, C. Paasch, TCP Extensions for Multipath Operation with Multiple Addresses, RFC 8684 (2020). doi:10.17487/RFC8684.
URL `https://www.rfc-editor.org/info/rfc8684`

[107] A. Barak, O. La'adan, The mosix multicomputer operating system for high performance cluster computing, Future Generation Computer Systems 13 (4-5) (1998) 361–372.

[108] Z. Bakhshi, G. Rodriguez-Navas, H. 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), IEEE, 2021, pp. 1–6.

[109] P. W. Li, H. L. Zhao, H. T. Yang, S. Sun, Performance evaluation of transport protocol in data distribution service middleware, Advanced Materials Research 926 (2014) 1984–1987.

[110] M. A. Azad, R. Mahmood, T. 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, IEEE, 2009, pp. 40–45.

[111] P. Papadimitriou, V. Tsaoussidis, Assessment of internet voice transport with tcp, International Journal of Communication Systems 19 (4) (2006) 381–405.

[112] Y. Sahraoui, A. Ghanam, S. Zaidi, S. Bitam, A. 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), IEEE, 2018, pp. 67–72.

[113] Y. Wang, A. 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), IEEE, 2021, pp. 1–9.

[114] S. Asodi, S. V. Ganesh, E. Seshadri, P. 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, IEEE, 2009, pp. 238–242.

[115] M. N. Tahir, M. 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, Springer, 2021, pp. 257–279.

[116] O. Ali, A. Aghmadi, O. A. Mohammed, Performance evaluation of communication networks for networked microgrids, e-Prime-Advances in Electrical Engineering, Electronics and Energy 8 (2024) 100521.

[117] L. Cottrell, Characterization and evaluation of tcp and udp-based transport on real networks, Tech. rep., SLAC National Accelerator Lab., Menlo Park, CA (United States) (2005).

[118] M. Lulling, J. Vaughan, A simulation-based comparative evaluation of transport protocols for sip, Computer Communications 29 (4) (2006) 525–537.

[119] M. Allman, V. Paxson, E. Blanton, Tcp congestion control, Tech. rep. (2009).

[120] D. Hallmans, M. Ashjaei, T. Nolte, Analysis of the tsn standards for utilization in long-life industrial distributed control systems, in: 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Vol. 1, IEEE, 2020, pp. 190–197.

[121] D. Bertsekas, R. Gallager, Data networks, Athena Scientific, 2021.

[122] B. Turkovic, F. A. Kuipers, S. Uhlig, Fifty shades of congestion control: A performance and interactions evaluation, arXiv preprint arXiv:1903.03852 (2019).

[123] 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).

[124] W. Eddy, Transmission Control Protocol (TCP), RFC 9293 (2022). doi:10.17487/RFC9293.
URL https://www.rfc-editor.org/info/rfc9293

[125] M. Scharf, S. Kiesel, Nxg03-5: Head-of-line blocking in tcp and sctp: Analysis and measurements, in: IEEE Globecom 2006, IEEE, 2006, pp. 1–5.

[126] User Datagram Protocol, RFC 768 (1980). doi:10.17487/RFC0768.
URL https://www.rfc-editor.org/info/rfc768

[127] J. P. Macker, C. Bormann, M. J. Handley, B. Adamson, NACK-Oriented Reliable Multicast (NORM) Transport Protocol, RFC 5740 (2009). doi:10.17487/RFC5740.
URL https://www.rfc-editor.org/info/rfc5740

[128] H. Schulzrinne, S. L. Casner, R. Frederick, V. Jacobson, RTP: A Transport Protocol for Real-Time Applications, RFC 3550 (2003). doi:10.17487/RFC3550.
URL https://www.rfc-editor.org/info/rfc3550

[129] E. Carrara, K. Norrman, D. McGrew, M. Naslund, M. Baugher, The Secure Real-time Transport Protocol (SRTP), RFC 3711 (Mar. 2004). doi:10.17487/RFC3711.
URL https://www.rfc-editor.org/info/rfc3711

[130] R. R. Stewart, M. Tüxen, karen Nielsen, Stream Control Transmission Protocol, RFC 9260 (2022). doi:10.17487/RFC9260.
URL https://www.rfc-editor.org/info/rfc9260

[131] sctp - linux manual page, https://man7.org/linux/man-pages/man7/sctp.7.html, accessed: 2025-08-25.

[132] sctplib - windows sctp driver, https://github.com/dreibh/sctplib/blob/master/README.win32, accessed: 2025-08-25.

[133] 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: 2025-08-25.

[134] A. Jungmaier, E. Rescorla, M. Tüxen, Transport Layer Security over Stream Control Transmission Protocol, RFC 3436 (Dec. 2002). doi:10.17487/RFC3436.
URL https://www.rfc-editor.org/info/rfc3436

[135] M. Tüxen, E. Rescorla, R. Seggelmann, Datagram Transport Layer Security (DTLS) for Stream Control Transmission Protocol (SCTP), RFC 6083 (Jan. 2011). doi:10.17487/RFC6083.
URL https://www.rfc-editor.org/info/rfc6083

[136] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. 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, 2017, pp. 183–196.

[137] J. Iyengar, M. Thomson, QUIC: A UDP-Based Multiplexed and Secure Transport, RFC 9000 (2021). doi:10.17487/RFC9000.
URL https://www.rfc-editor.org/info/rfc9000

[138] J. Iyengar, I. Swett, QUIC Loss Detection and Congestion Control, RFC 9002 (2021). doi:10.17487/RFC9002.
URL https://www.rfc-editor.org/info/rfc9002

[139] Omg data distribution service version: 1.4, https://www.omg.org/spec/DDS/1.4/PDF, accessed: 2024-11-19.

[140] 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.

[141] 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.

[142] Dds security version 1.2, https://www.omg.org/spec/DDS-SECURITY/1.2/PDF, accessed: 2025-05-13.

[143] Opc 10000-1 - ua specification part 1: Overview and concepts, https://reference.opcfoundation.org/Core/Part1/v105/docs/, accessed: 2024-11-05.

[144] Opc 10000-4 - ua specification part 4: Services 1.05.03, https://reference.opcfoundation.org/Core/Part4/v105/docs/, accessed: 2024-11-05.

[145] Opc 10000-2 - ua specification part 2: Security 1.05.04, https://reference.opcfoundation.org/Core/Part2/v105/docs/, accessed: 2025-05-14.

[146] Opc 10000-6 - ua specification part 6: Mappings 1.05.03, https://reference.opcfoundation.org/Core/Part6/v105/docs/, accessed: 2024-11-05.

[147] Opc 10000-14 - ua specification part 14: Pubsub 1.05.03, https://reference.opcfoundation.org/Core/Part14/v105/docs/, accessed: 2024-03-26.

[148] Opc 10000-22 - ua specification part 22: Base network model 1.05.03, https://reference.opcfoundation.org/Core/Part22/v105/docs/, accessed: 2024-11-05.

[149] T. Bova, T. Krivoruchka, RELIABLE UDP PROTOCOL, Internet-Draft draft-ietf-sigtran-reliable-udp-00, Internet Engineering Task Force, work in Progress (1999).
URL https://datatracker.ietf.org/doc/draft-ietf-sigtran-reliable-udp/00/

[150] E. He, J. Leigh, O. Yu, T. A. DeFanti, Reliable blast udp: Predictable high performance bulk data transfer, in: Proceedings. IEEE International Conference on Cluster Computing, IEEE, 2002, pp. 317–324.

[151] B. Eckart, X. He, Q. Wu, Performance adaptive udp for high-speed bulk data transfer over dedicated links, in: 2008 IEEE International Symposium on Parallel and Distributed Processing, IEEE, 2008, pp. 1–10.

[152] Y. Gu, R. L. Grossman, Udt: Udp-based data transfer for high-speed wide area networks, Computer Networks 51 (7) (2007) 1777–1799.

[153] A. O. F. Atya, J. Kuang, Rufc: A flexible framework for reliable udp with flow control, in: 8th International Conference for Internet Technology and Secured Transactions (ICITST-2013), IEEE, 2013, pp. 276–281.

[154] R. L. Grossman, M. Mazzucco, H. Sivakumar, Y. Pan, Q. Zhang, Simple available bandwidth utilization library for high-speed wide area networks, The Journal of Supercomputing 34 (2005) 231–242.

[155] M. R. Meiss, et al., Tsunami: A high-speed rate-controlled protocol for file transfer, Indiana University (2004).

[156] Ampq: Advanced message queuing protocol, https://www.amqp.org/, accessed: 2024-11-04.

[157] Z. Shelby, The constrained application protocol (coap), Tech. rep., IETF RFC 7252 (2014).

[158] S. Floyd, M. J. Handley, E. Kohler, Datagram Congestion Control Protocol (DCCP), RFC 4340 (2006). doi:10.17487/RFC4340.
URL https://www.rfc-editor.org/info/rfc4340

[159] L. Eggert, G. Fairhurst, G. Shepherd, UDP Usage Guidelines, RFC 8085 (2017). doi:10.17487/RFC8085.
URL https://www.rfc-editor.org/info/rfc8085

[160] M. J. Handley, J. Padhye, S. Floyd, J. Widmer, TCP Friendly Rate Control (TFRC): Protocol Specification, RFC 5348 (2008). doi:10.17487/RFC5348.
URL https://www.rfc-editor.org/info/rfc5348

[161] A. Elghazi, M. Berrezzouq, Z. Abdelali, New version of iscsi protocol to secure cloud data storage, in: 2016 2nd International Conference on Cloud Computing Technologies and Applications (CloudTech), IEEE, 2016, pp. 141–145.

[162] Ibm aspera - product webpage, https://www.ibm.com/products/aspera, accessed: 2024-12-05.

[163] Mqtt: Message queue telemetry transport, https://mqtt.org/, accessed: 2024-11-04.

[164] J. Hu, H. Shen, X. Liu, J. Wang, Rdma transports in datacenter networks: Survey, IEEE Network (2024).

[165] The roce initiative, https://www.roceinitiative.org/, accessed: 2024-11-04.

[166] U. Abbasi, E. H. Bourhim, M. Dieye, H. Elbiaze, A performance comparison of container networking alternatives, IEEE Network 33 (4) (2019) 178–185.

[167] M. Shaikh, P. Shah, R. Sekhar, Communication protocols in industry 4.0, in: 2023 International Conference on Sustainable Emerging Innovations in Engineering and Technology (ICSEIET), IEEE, 2023, pp. 709–714.

[168] Market shares 2024 according to hms networks, https://www.hms-networks.com/news/news-details/17-06-2024-annual-analysis-reveals-steady-growth-in-industrial-network-market, accessed: 2025-01-08.

[169] R. Zurawski, Industrial communication technology handbook, CRC press, 2014.

[170] G. Prytz, A performance analysis of ethercat and profinet irt, in: 2008 IEEE International Conference on Emerging Technologies and Factory Automation, IEEE, 2008, pp. 408–415.

[171] Transmission Control Protocol, RFC 793 (1981). doi:10.17487/RFC0793.
URL https://www.rfc-editor.org/info/rfc793

[172] S. Floyd, J. Mahdavi, M. Mathis, D. A. Romanow, TCP Selective Acknowledgment Options, RFC 2018 (1996). doi:10.17487/RFC2018.
URL https://www.rfc-editor.org/info/rfc2018

[173] E. Blanton, D. V. Paxson, M. Allman, TCP Congestion Control, RFC 5681 (2009). doi:10.17487/RFC5681.
URL https://www.rfc-editor.org/info/rfc5681

[174] M. Sargent, J. Chu, D. V. Paxson, M. Allman, Computing TCP's Retransmission Timer, RFC 6298 (2011). doi:10.17487/RFC6298.
URL https://www.rfc-editor.org/info/rfc6298

[175] S. O. Bradner, Key words for use in RFCs to Indicate Requirement Levels, RFC 2119 (1997). doi:10.17487/RFC2119.
URL https://www.rfc-editor.org/info/rfc2119

[176] R. R. Stewart, Stream Control Transmission Protocol, RFC 4960 (2007). doi:10.17487/RFC4960.
URL https://www.rfc-editor.org/info/rfc4960

[177] J. Pastor, M.-C. Belinchon, Stream Control Transmission Protocol (SCTP) Management Information Base (MIB), RFC 3873 (2004). doi:10.17487/RFC3873.
URL https://www.rfc-editor.org/info/rfc3873

[178] M. Tüxen, V. Yasevich, P. Lei, R. R. Stewart, K. Poon, Sockets API Extensions for the Stream Control Transmission Protocol (SCTP), RFC 6458 (2011). doi:10.17487/RFC6458.
URL https://www.rfc-editor.org/info/rfc6458

[179] C. L. Liu, J. W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, Journal of the ACM (JACM) 20 (1) (1973) 46–61.

[180] Q. Zheng, K. G. Shin, On the ability of establishing real-time channels in point-to-point packet-switched networks, IEEE Transactions on Communications 42 (234) (1994) 1096–1105.

[181] C. E. Spurgeon, Ethernet: the definitive guide, " O'Reilly Media, Inc.", 2000.

[182] A. S. Tanenbaum, Computer networks, Pearson Education India, 2003.

[183] B. Johansson, M. Rågberger, T. Nolte, A. 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, IEEE, 2022, pp. 1–6.

[184] H. 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.

[185] Abb webpage - pm891 product specification, `https://compacthardwareselector.automation.abb.com/product/pm891k01`, accessed: 2025-08-27.