Resilient Distributed P/T Net Simulators*

Laif-Oke Clasen[1], Patrick Leonhardt[1] and Leven Wichelmann[1]

[1]University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences, Department of Informatics,
http://www.paose.de

Abstract

A distributed simulation of P/T nets requires partitioning the overall model into modules that run on multiple simulators. Failures in distributed systems can compromise consistency, resulting in incorrect outcomes. Ensuring resilience in such simulations is essential for maintaining correctness, especially in long-running executions.

This research develops a concept for resilient simulators in distributed P/T net simulations. A prototyping approach grounded in constructivist principles enables the detection and recovery of failures in P/T net simulations. The evaluation follows a summative ex-post methodology, applying a criteria-based assessment to validate effectiveness.

Experimental validation demonstrates the feasibility of the proposed concept of resilience mechanisms for distributed P/T net simulations. The system maintains simulation consistency despite failures by integrating state-saving and recovery techniques. The results confirm improved fault tolerance and reliability in distributed P/T net simulations.

The introduced concept for resilient P/T net simulations enhances the robustness of distributed simulations. Preventing inconsistencies ensures accurate analysis and reliable execution over extended periods. The findings contribute to the development of fault-tolerant simulation frameworks, supporting more reliable distributed computing environments.

Keywords

Resilience, Failure Detection, Failure Recovery, Distributed Simulation, P/T Nets, P/T Nets with Synchronous Channels, Event Streaming, Container, Container Orchestration

1. Introduction

The distribution of complex simulation models across several independent computing nodes is becoming increasingly important in software engineering, particularly system simulation. [1, 2, 3] The ability to simulate models in a distributed manner offers significant advantages in terms of model simulation scalability; however, it also risks inconsistencies and data loss due to errors in individual components. Against this background, research into resilient mechanisms in distributed simulation environments is becoming increasingly important, as it directly addresses the reliability and accuracy of such simulation results. [4, 5, 6]

The present work lies at the intersection of distributed systems and the simulation of Petri nets, particularly in the context of resilience, i.e., the ability to recover system states after faults occur. Particularly in application areas such as process automation, workflow management, or complex, long-running simulations of critical systems, simulations must continue to run resiliently and consistently despite errors. Despite considerable progress in the distribution and scaling of P/T net simulations, there are still significant research gaps regarding suitable strategies for fault detection, state safety, and systematic recovery from faults. The development of resilient simulators is not only a technical challenge but also opens up new methodological perspectives for the reliability of distributed simulations. The state safety and recovery mechanisms developed and validated in this work are expected to enhance fault tolerance and provide a comprehensive understanding of the causes and effects of inconsistencies in distributed Petri net simulations.

Therefore, the central research question of this work is: How can distributed simulators of P/T nets be designed to guarantee consistent simulation results even in the event of system failures? This research question is based on the hypothesis that implementing structured statefulness procedures and

PNSE'25, International Workshop on Petri Nets and Software Engineering, 2025

*Supported by participants of our teaching project classes and student theses.

liveness-based monitoring strategies can significantly increase the resilience of distributed simulations of P/T nets, allowing simulators to continue operating consistently despite failures.

This contribution adopts a constructivist approach to addressing the research question, developing a prototype simulator grounded in resilient design principles. [7, 8, 9] As a starting point, we use the distributed P/T net simulation [10] in the Petri net editor, simulator and verifier Renew[1] [11]. In doing so, concepts for the state assurance procedure and recovery after failures are developed, and their effectiveness is experimentally tested. This evaluation involves a criteria-based assessment to objectively prove the effectiveness and robustness of the developed concepts.

Within the Foundations (Section 2), the topics of Renew (Section 2.1), Distributed P/T Nets (Section 2.2), Failures (Section 2.3), Resilience (Section 2.4), and Kubernetes (Section 2.5) are systematically introduced. Subsequently, the Problem Description (Section 3) and the design of the Distributed System (Section 4) are presented. The ensuing section details the prototypes developed during this work, specifically Detecting Failures of Simulators (Section 5) and Recovering Failures of Simulators – DPT-NResiliency (Section 6). The overall system is then evaluated using a classical case study in computer science, the Producer-Storage-Consumer scenario (Section 7). A critical discussion (Section 8) of the advantages, disadvantages, and limitations of the proposed concept follows. Finally, the article concludes with an overview of Related Work (Section 9) and the Conclusion (Section 10).

2. Foundations

This section introduces the relevant concepts and technologies that are the focus of this paper. Firstly, we present Renew as a Java-based multi-formalism editor and simulator for especially reference nets (2.1) and further introduce our set-up for distributed P/T nets (2.2). Then, we proceed to different possible failure types, including classification, detection, and recovery (2.3). We finish with a definition of resilience (2.4) and an overview of the relevant terms and concepts within the Kubernetes environment (2.5).

2.1. Renew

Renew [11] is an open-source software tool for modeling, analyzing, and simulating various types of Petri nets, with a particular focus on distributed P/T nets (Section 2.2). It was developed by the Algorithms, Randomization, and Theory (ART) research group, formerly Theoretical Foundations of Computer Science (TGI), at the University of Hamburg.

Renew is implemented in Java 17 [12] and built using Gradle 8.4 [13], ensuring robustness and platform independence. Its software architecture is based on a modular plugin system, as described by Duvigneau [14]. Its modularity and maintainability have recently been enhanced by adopting the Java Platform Module System (JPMS) [15, 16].

For each supported Petri net variant, Renew provides a dedicated formalism plugin, the most prominent being the reference net formalism according to Kummer [17]. Another plugin relevant to this contribution is the cloud-native plugin [18], which can expose HTTP endpoints for Renew via Java Spring [19]. Through these HTTP endpoints, Renew simulations can be initiated and controlled.

2.2. Distributed P/T Nets

The distributed P/T nets used in this contribution are the same as in [10]. In this context, they build on the formal definition of [20] and extend it with the informal extension of distributed synchronous channels. If we are considering simulation time, multiple instances of these distributed P/T nets are executed within a single simulator; however, there are also multiple simulators in place.

The informal extension of distributed synchronous channels ensures that P/T nets can communicate with each other by implementing rendezvous synchronization. Transitions are labeled with signatures

[1] **Re**ference **Ne**t **W**orkshop can be downloaded directly from its official website: http://renew.de.

so that they can synchronize if they have the corresponding signature. These labeled transitions with matching signatures can only fire together.

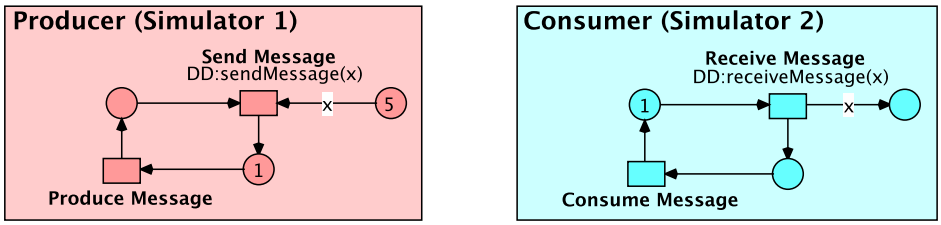
The signature of a synchronous channel, which is used here, consists of type, identifier, and parameter. There are two different types: the downlink and uplink, which can be denoted as DD and DU, respectively. [10] Downlinks are the active or calling part and uplinks are the passive or called part. The identifier usually describes the name of the channel or a relation. Whereas the parameters are used to exchange information between the synchronizing transitions.

Each of the distributed P/T nets can be considered as an individual module, whereby the consideration of a module based on [21] is applied. Modules on this basis have a left and a right interface. In the context of distributed P/T nets, the left and right interfaces only contain distributed synchronous channels.

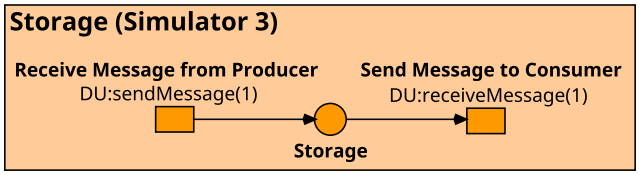
The overall system architecture for the distributed P/T nets [10] comprises multiple simulators, an event-based communication medium (Kafka), and a synchronization service. The distributed P/T nets are statically allocated to the available simulators.

The communication between P/T nets across simulator boundaries is facilitated through distributed synchronous channels. The event-based communication medium Apache Kafka is employed for this purpose. Apache Kafka is an open-source, distributed event-streaming platform designed to deliver scalability and high performance [22, 23]. It is extensively utilized in distributed systems for real-time data processing and transmission. Event streaming refers to the continuous processing of data as discrete, immutable events, each annotated with a timestamp and sequence number. Such events can be persistently stored and subsequently reused, enabling efficient analysis and processing. Kafka provides persistence, high throughput, real-time processing capabilities, and support for diverse architectures and programming languages [24, p. 6f]. The decoupling of producers and consumers promotes the development of loosely coupled system architectures, establishing Kafka as a scalable and robust solution for modern distributed systems, particularly when deployed with high-availability configurations.

An illustrative example is provided by Clasen et al. [10], who describes a classic IT scenario - the producer-consumer storage model, which is visualized in Figure 1. In this example, the producer, consumer, and storage components are distributed across different simulators. The producers and consumers act as active components, whereas the storage operates as a reactive component, featuring only distributed uplinks and lacking downlinks.



(a) Producer and Consumer Net



(b) Storage Net

Figure 1: Components of the producer storage consumer example [10]

2.3. Failures

In every computer system, it is unavoidable that, at some point, some failure can occur. [25] This applies specifically to distributed simulation systems. These errors may have different causes and origins.

Internal or in-process errors occur within an application and can be classified into different types. Logical errors [26] occur when incorrect instructions are present in the application's source code, resulting in issues such as erroneous calculations or infinite loops. These represent implementation faults caused by flawed algorithms or incorrect control flows.

Closely related are semantic errors, which are often used synonymously. Semantic errors [26] affect the interpretation of data or interactions: although the code functions correctly on a technical level, the meaning of the results or processes deviates from the specification.

In addition to the mentioned types of errors that result in faulty program behavior, runtime errors may also occur despite a "correct" program implementation. For example, excessive nested calls of a recursive function typically lead to a stack overflow, where the call stack of a program exceeds the allocated memory space. If a program continuously consumes memory without releasing it correctly, this leads to a memory leak. Both stack overflows and memory leaks fall into the category of memory-related errors. [27]

Problems in the parallel execution of (sub-)processes can cause synchronization errors, such as deadlocks or race conditions. A deadlock occurs when a cyclic waiting situation arises between the involved processes, with each waiting for the release of a system resource that is exclusively held by another. A race condition, on the other hand, describes an unintended fault where multiple operations influence the final result due to their timing behavior. [28]

We can group logical and semantic errors, as well as synchronization and memory errors, into a broader category of software errors to fit into the categorization of Schroeder et al. [29]. They also mention other types of errors that are not directly software-related, which we look at next.

When disruptions occur during data transmission or exchange between distributed components, they are referred to as network or communication errors. These can result from issues such as packet loss or connection failures, leading to inconsistencies in the state of distributed systems. Such inconsistencies can adversely affect the synchronization and correctness of the simulation.

System and hardware errors originate from physical defects or failures in the components of the distributed system, such as the CPU or memory. These errors are often difficult to predict and can cause abrupt system crashes or faulty data processing.

In addition to internal errors, there are also external error types that depend on external factors. Undesired user inputs or environmental influences can impair the correctness of the simulation or even disrupt the intended functionality of the entire system.

In the context of our contribution, a relevant structure of error types emerges, as shown in Figure 2. We distinguish between internal and external errors that may affect our distributed simulation. Among internal errors, we further differentiate between software-based errors (such as logical, semantic, synchronization-specific, and memory-related errors) and infrastructure-based errors, which depend on the system architecture. The latter includes communication and hardware errors.

To ensure the robustness and reliability of a distributed simulation, the early detection of occurring failures is essential. These errors can occur in both deterministic and non-deterministic ways, which complicates their identification and reproducibility. Typical deterministic errors include logical flaws or faulty algorithms, whereas synchronization errors, such as deadlocks and race conditions, are considered non-deterministic.

The first method for detecting failures is through static analysis. Although they primarily serve compile-time checking, formal analysis techniques can detect potential runtime errors before execution. Static analysis tools such as SonarGraph [30] or FindBugs [31] can flag unused variables, incorrect assignments, or potential null references. Many modern IDEs already include basic static analysis tools with various features as standard.

On the other hand, there are dynamic methods to detect failures at runtime. This involves monitoring the system during execution. Techniques such as assertion checking, instrumentation, or the use

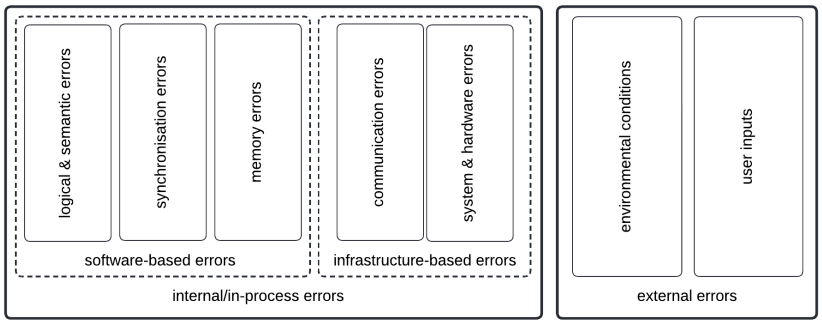


Figure 2: Classification of Failure Types

of debugging tools like Valgrind [32], the GNU Debugger [33], or AddressSanitizer [34] enable the detection of memory errors, invalid memory accesses, or synchronization issues. Particularly in productive, complex, and automated systems or simulations, continuous and comprehensive monitoring plays a crucial role. It is the foundation and, therefore, essential for ensuring the availability, reliability, and performance of a system.

Monitoring technologies such as Prometheus [35] offer suitable solutions by collecting a wide range of metrics and visualizing them. In most cases, an alert manager is included for triggering alerts in the event of anomalies. The open-source monitoring framework Kieker is also worth mentioning as a valuable tool for the runtime monitoring of software-based systems. It incorporates the aforementioned capabilities and is particularly useful for analyzing performance, architectural behavior, and failures in distributed applications.[36]

Since we cannot entirely prevent failures, we must at least design our systems to tolerate or recover from them when they occur. It is worth noting that, in some situations, it can be better to tolerate small failures by not addressing them if an automated recovery system might do more harm than good [25].

In general, we can categorize fault tolerance methods into two main groups: reactive and proactive methods. Static analysis methods act as a proactive method. Proactive methods take action preemptively to try and limit failures as much as possible, while reactive methods rely on failure detection and start acting when a failure has occurred. [37] For this reason, to make a system truly failure-tolerant, one needs to implement at least one reactive method.

Since proactive methods cannot entirely prevent failures, we do not elaborate on them further in this paper. More information can be found, e.g., in [38].

An obvious reactive method is checkpointing, sometimes also referred to as checkpoint-restart. [39] Here, each system component regularly stores its state on some form of persistent, highly available storage. If a component fails (e.g., due to a crash), it can be restarted by a failure detection system and automatically load the latest checkpoint to continue execution from that point.

Another reactive method, which can also be considered a hybrid method, is replication. Here, individual components can be replicated, potentially even on different physical machines, resulting in multiple instances of each component. If one instance fails, a replica can take its place to ensure smooth execution, and the original instance gets restarted to serve as another replica. [40]

2.4. Resilience

According to Laprie et al. [41], the term *resilient* has been used mainly as a synonym of *fault-tolerant* for many years in the field of computer science. A resilient, fault-tolerant, or robust system should be able to deliver its service, even in some circumstances that are not part of its typical mode of operation. We use the term "some" here since we have already learned in section 2.3 that it is impossible to completely prevent failures, including those that are not tolerable by any software system (like a simultaneous hardware failure in all machines).

For this paper, we use the definition from Pradhan et al. [42], which defines a resilient system as

a system that "includes efficient techniques for [...] ensuring its correct operation [...], even in the presence of faults and failures [...].". Additionally, it requires the resilient mechanism of a system to be autonomous, as human interaction is slow and introduces an additional opportunity for errors.

2.5. Kubernetes

Kubernetes [43] is an open-source software for container orchestration developed by Google. It is commonly used to manage, harden and scale each individual component of deployed applications. It allows for easy creation of clusters made of multiple physical computers (nodes).

On each node, a service called kubelet acts as the "primary node agent" for Kubernetes, manages everything that Kubernetes runs on that node. If some application crashes on a specific node for example, the set of kubelets of all nodes would be responsible for restarting it.

Additionally, various resource types allow applications to scale automatically according to demand, making it suitable for all kinds of applications and workloads.

Containerisation can be defined as the act of bundling a software application and all necessary dependencies and system libraries into a single container. [44] There are various technologies for containerising applications like Docker [45] or Podman [46]. Under the hood, they all implement the specifications of the Open Container Initiative [47] (OCI), making them mostly interchangeable at runtime.

A key property of containers is that they are stateless by default. That means every time a container is started, it has no recollection of past instances of itself or other state.

A Pod is the smallest deployable entity in Kubernetes and consists of one or more containers [48]. Just like containers, Pods are not persistent by default; if a Pod dies and is restarted, a new replica of the Pod is created, without any memory of previous instances of the Pod.

Container lifecycle hooks are part of the OCI runtime specification [49] and widely used when working with containers. They are used to intercept specific events in the container lifecycle, for example the container starting or stopping.

Kubernetes offers a few hooks (more commonly called probes) related to Pod lifecycles, most importantly liveness probes. As the name suggests, liveness probes allow Kubernetes to check that our Pods are still active and have not failed or crashed [50]. If a liveness probe fails too often, Kubernetes will treat the corresponding Pod as failed, and automatically restart it. This makes them an important tool when developing any kind of failure-resistant application.

In Kubernetes, one usually does not directly create Pods themselves. Instead, there are a number of resources types that create and manage a set of Pods, each having unique use cases, advantages and disadvantages.

A StatefulSet owns a set of Pods and maintains a unique identity for each one, as well as an ordering over all its Pods. The Pod identity it provides includes a network address, (if configured) a dedicated storage mount, a name and and index label. If any Pod that belongs to a StatefulSet fails, for example by crashing or because the associated Liveness Probe fails, a new Pod will be created with the same identity of the failed Pod.

Some applications do require Pods to have some form of persistent storage, or memory, even between restarts. For this purpose, Kubernetes implements the concept of Persistent Volumes (PVs) and Persistent Volume Claims (PVCs).

A PV is a resource in a cluster that provides a piece of storage. Their lifecycle is independent from Pods. If a Pod needs some persistent storage, it can request some via a PVC. In this case, the Pod can only start when a fitting PV binds to its PVC. The volume will then mount in the corresponding container(s) filesystem.

PVs can either be created manually by cluster administrators, or by a StorageClass that is configured to automatically provision PVs when a PVC requests storage from it.

Ceph is a mature distributed file system that is developed for performance, scalability and reliability [51]. Rook [52] is a cloud native Kubernetes deployment of Ceph [53] that allows developers to focus on configuring Kubernetes resources, while silently managing the Ceph file system on all nodes in the

background. It does so by providing StorageClasses for various purporses that automatically provision PVs as needed.

The Cloud Native Computing Foundation [54], an offshoot of the Linux Foundation [55], lists Rook as one of only two graduated technologies in the context of Cloud Native Storage. [56] Graduated projects are "[...] considered stable, widely adopted, and production ready, attracting thousands of constributors". This makes Rook a great choice for managing storage in a Kubernetes cluster.

3. Problem Description

All simulators collaboratively execute a single simulation. To this end, each simulator is capable of concurrently executing multiple distributed P/T nets. The global state of the simulation comprises the complete set of distributed P/T nets along with their respective markings. The local state of a simulator is defined as the subset of distributed P/T nets it executes, including the corresponding markings. Thus, each simulator is responsible for a partition of the overall system and manages the associated segment of the simulation state.

In the context of complex systems or processes, simulations often run for extended periods of time. Prolonged runtimes increase the probability of individual component failures. In the worst-case scenario, such failures can result in inconsistencies that necessitate a complete restart of the simulation. Since failures may stem from a variety of causes—most notably hardware faults—they cannot be entirely precluded.

It is, therefore, imperative to develop robust mechanisms for detecting and mitigating failures in distributed simulations. Failures are detected through continuous monitoring of the simulators' operational status. Once a failure is identified, the affected simulator is reinitiated on a functional computing node.

To ensure correct recovery, appropriate techniques must be employed to reconstruct the simulator's state, as each simulator retains a distinct portion of the global state. This is achieved by periodically persisting the simulator's state under predefined conditions to highly available and durable storage. In the event of a failure, the simulator can resume execution from the most recent consistent state, ensuring continuity of the simulation.

The concept introduced in this work is validated in the context of distributed simulation of P/T nets. Given the distributed nature of the simulation, the overall system is inherently decentralized. The first objective of this work is the conceptual design and technical realization of the distributed system (Section 4).

A subsequent objective involves developing mechanisms for fault detection in simulators (Section 5). Since simulators may be inaccessible in the event of failure, direct fault detection within the simulators is infeasible. Consequently, fault detection must be implemented as part of the surrounding distributed system infrastructure.

Recovery techniques must be tailored to the specific architectural design of the simulators, which are themselves responsible for ensuring their resilience (Section 6). Simulators must be capable of persistently storing their state at semantically meaningful intervals and resuming execution from that state following a failure.

Finally, the proposed concept is evaluated within a representative scenario (Section 7), followed by a critical discussion of its advantages, limitations, and potential drawbacks (Section 8).

4. Distributed System

The distributed system comprises simulation components, a highly available and distributed memory architecture, and the overarching distributed environment. A core requirement of this environment is the ability to detect failures in the participating simulation components and initiate appropriate recovery measures, such as restarting a failed simulator instance on an alternative computational node.

The simulation components operate within the context of distributed simulations of P/T nets. This setup necessitates the presence of the simulation components themselves and a reliable communication medium through which simulators can interact and coordinate during distributed execution.

Given the distributed nature of the simulation of these P/T nets, the system requires at least two simulators operating concurrently. By the proposed recovery mechanism, a highly distributed and persistent memory system is essential for maintaining the simulator states. As long as a simulator's state is preserved within this memory, it remains accessible even after a failure, enabling effective recovery and continuation of the simulation process.

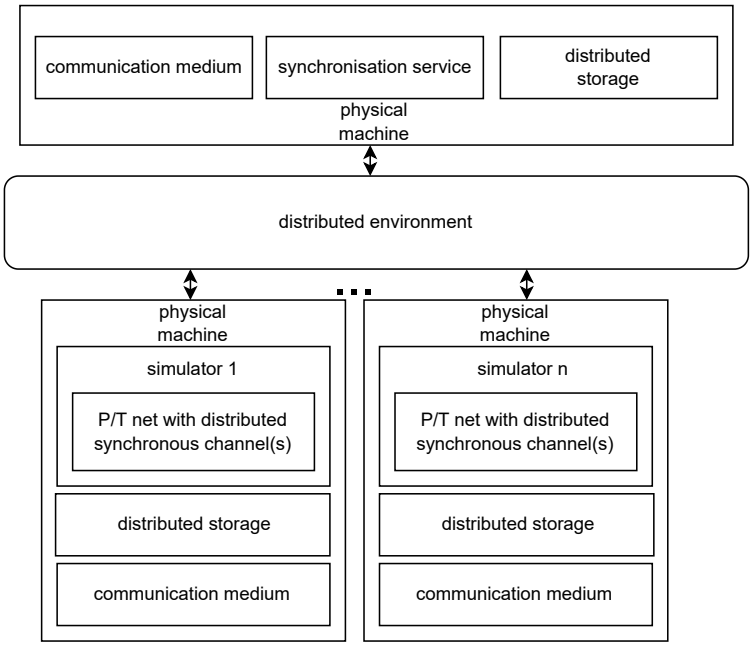


Figure 3: System for resilient distributed P/T net simulations

The container orchestration platform Kubernetes [43] (Section 2.5) is employed to implement the distributed environment. This choice necessitates that all system components be encapsulated as containers. At the same time, Kubernetes' built-in fault detection mechanisms, such as liveness probes, can be leveraged to monitor and maintain system integrity.

The distributed architecture (Section 2.5) further necessitates a storage system that is not only distributed and persistent but also highly available to maintain the simulators' states reliably. To this end, the open-source software Rook [52] is utilized as a cloud-native storage orchestrator within the Kubernetes environment. Rook builds upon the mature and widely adopted storage solution Ceph [57], which is extensively used in production environments.

While there is no predefined upper limit on the total number of nodes, a minimum number of nodes is essential for the distributed storage system to function correctly. Specifically, a minimum of three nodes is required to ensure high availability and consistency. Operating with only two nodes introduces the risk of a split-brain scenario, whereas a single-node configuration constitutes a single point of failure.

The simulation components are developed in the context of distributed P/T nets (Section 2.2). The simulators must incorporate effective failure recovery mechanisms to support fault tolerance, necessitating a design emphasizing extensibility. The simulator Renew (Section 2.1) is particularly well suited for this purpose, as it not only facilitates the distributed execution of P/T nets but also features a modular plugin architecture that supports straightforward extension.

Moreover, Renew requires a dedicated synchronization service to coordinate the distributed simulation of P/T nets (Section 2.2). This service is responsible for determining which distributed synchronous

transitions may fire together. For this coordination, the synchronization service employs Renew's unification algorithm.

Finally, an event-driven communication infrastructure (Section 2.2) is essential for inter-simulator messaging. Kafka is employed as a communication medium by integrating within the Renew simulation framework to meet this requirement. In addition, Kafka itself is provided as a highly available communication medium in order to create resilience for the communication medium as well.

5. Detecting Failures of Simulators

Reliable detection of simulator failures is a fundamental prerequisite for ensuring the overall resilience and correctness of the simulation framework. This prototype outlines the systematic approach adopted to address this challenge, beginning with a detailed analysis of the requirements (Section 5.1) that such a failure detection mechanism must satisfy. Based on these requirements, we then provide a specification (Section 5.2) of the detection logic, followed by a discussion of the design (Section 5.3) that guided the development of the solution. The corresponding implementation (Section 5.4) is subsequently described. Finally, the effectiveness of the proposed approach is assessed through a comprehensive evaluation (Section 5.5).

5.1. Requirements

To achieve a simulation of a distributed P/T net that is as error-free and correct as possible, it is all the more important to efficiently detect as many types of occurring errors during the simulation as possible. This enables their subsequent elimination using appropriate recovery mechanisms in the following prototype (Section 6), thereby ensuring the resilience of the simulators.

For this purpose, various error detection methods described in Section 2.3 are employed, with dynamic runtime analysis, as well as monitoring and logging, playing a key role in identifying potential runtime errors within and between individual simulators. The focus of this prototype is the requirement that it should be possible to detect failures within a simulator.

5.2. Specification

Besides possible logical and semantic errors—which we already try to detect and resolve within our quality assurance process—the most critical errors to identify are fail-stop errors. These are the kinds of errors that can halt the entire simulation and potentially compromise its results.

A mechanism is therefore required that can identify the fail-stop. For this purpose, the liveliness, i.e., the availability, of the simulator is to be checked at regular short intervals.

Particular attention is given to infrastructure-related faults. Thus, if communication errors occur during the simulation—i.e., errors in data transmission between the distributed components caused by, for example, connection losses or packet loss—they must be detected and reported. Similarly, if system or hardware faults arise due to processor or memory failures or malfunctions, these errors must also be identified for recovery. Furthermore, all external errors, including those due to Force Majeure or specific user inputs, are also known but not relevant to our context of distributed simulations.

5.3. Design

In order to detect errors in a simulator, we need an infrastructure that can check the availability of the simulators and, if necessary, start new simulators on other nodes. For this purpose, a container-based infrastructure is built that can recognize failing nodes and control a container's lifecycle.

In addition, a corresponding HTTP endpoint is required in Renew to check the availability of the simulator. The CloudNative plugin is used to provide this endpoint.

5.4. Implementation

We implement our containers with Docker [45] and the container orchestration system using Kubernetes [43] (Section 2.5). For this reason, we can utilize Kubernetes liveness probes to detect the liveness of our simulators.

5.5. Evaluation

With this implementation, we can detect failures of physical machines, as Kubernetes detects node failures and reschedules Pods on healthy nodes. Additionally, we can detect failures of the simulator Pods directly if they disturb the availability of the HTTP endpoint of the CloudNative plugin.

This means we can detect fail-stop errors using this method, including crashes, node failures, network errors, and other similar issues. However, Silent errors, like deadlocks in the internal Renew-internal simulation thread pool, would remain undetected.

6. Recovering Failures of Simulators - DPTNResiliency

This section presents the DPTNResiliency prototype, which addresses the challenge of recovering from simulator failures within distributed simulation environments. We begin by outlining the requirements (Section 6.1) that guide the development of a resilient recovery mechanism. Based on these requirements, we then provide a specification (Section 6.2) of the failure recovery behavior. This is followed by a detailed description of the design (Section 6.3) that shape the structure and coordination logic of DPTNResiliency. Subsequently, we describe the concrete implementation (Section 6.4) of the proposed mechanism within our simulation framework. Finally, the section concludes with a thorough evaluation (Section 6.5) of this prototype.

6.1. Requirements

Our overarching objective is to develop a fully resilient DPTN simulation. However, the present contribution explicitly addresses the resilience of the simulators themselves.

In this context, it is imperative that fail-stop failures affecting individual simulators do not compromise the overall functionality or correctness of the distributed system. To this end, resilience must be ensured through a reactive failure recovery mechanism, as proactive strategies alone are insufficient to eliminate the occurrence of fail-stop failures.

The prototype developed in this work is designed to facilitate reactive recovery from such simulator failures. Specifically, when a simulator process crashes or experiences a fail-stop event, it must be automatically restarted without impairing the operational integrity of the distributed simulation. While a minimal delay associated with the recovery process is unavoidable, it remains functionally inconsequential to the system as a whole.

6.2. Specification

In accordance with the requirements outlined in Section 6.1, a reactive recovery mechanism is necessary to mitigate the effects of fail-stop events. Given that the distributed P/T nets simulated within the Renew framework are serializable, we adopt a checkpoint-based recovery strategy. This approach entails periodically saving and, if required, reloading the simulation state within Renew.

Each checkpoint must encapsulate the complete state of the simulation, including both the internal markings of all distributed P/T Net (DPTN) instances and the communication state with the synchronization service. This includes metadata on distributed transitions—such as whether a firing request has been issued—and a consistent record of which communication events have been processed up to the checkpoint.

To ensure the durability of checkpoints beyond the occurrence of a fail-stop event, these must be stored in a fault-tolerant, highly available storage system. This storage must not reside within a single

container or be bound to a single physical node. Reliance on a single container is inherently fragile, as container failure leads to complete data loss. Similarly, tying checkpoint persistence to a single node is inadequate since a node-level failure would result in the irrevocable loss of all checkpoint data. Consequently, a resilient, distributed storage solution is imperative—one that can withstand partial system failures without compromising checkpoint integrity.

Our distributed simulation system can be viewed as a distributed database system that executes distributed transactions, in the sense that each simulator processes events coming in from the event broker. We can design our system in a way similar to how database systems manage transactions [58], making sure to uphold the ACID properties that serve as foundational guarantees [59, 60]. The distributed nature of our system also means we are subject to the constraints of the CAP theorem [61]. By implementing ACID, we make sure to guarantee Consistency and Partition Tolerance at the cost of Availability, since it's never possible to guarantee all three properties at once. Sacrificing availability means we may get stuck in a recovery loop for a while, in order to make sure the other properties are upheld.

Upon initialization, a simulator will check if it finds an existing checkpoint, in which case it must assume a previous failure and recover from that checkpoint. Using the list of events from the event broker as a log, it can recover from the failure by replaying uncommited events on top of the latest checkpoint. In order for this to work, we must only commit events (i.e., mark them as processed) once a checkpoint has been created that includes the implications of the events.

6.3. Design

To facilitate resilient simulations, we developed a dedicated Renew plugin named DPTNResiliency. This plugin relies on other essential Renew components—specifically, the Simulator and GUI plugins—to enable functionalities such as saving and loading simulation states. It is employed by the DPTNFormalism plugin, introduced in [10], to execute distributed P/T net simulations.

The DPTNResiliency plugin introduces the console command `startResilientSimulation`, which initiates the simulation of a specified distributed P/T net. If a checkpoint is available, the simulation automatically resumes from the most recent one. This command serves as the entry point for determining the current simulation state when launching a simulator instance.

Furthermore, the event-streaming mechanism within the DPTNFormalism plugin has been extended to notify the DPTNResiliency plugin after the successful processing of each event. This notification mechanism is essential for persisting and tracking the accurate communication state throughout the simulation. Additionally, the DPTNResiliency plugin manages Kafka commits to make sure events are only marked as read once a checkpoint for them has been created.

The responsibility for checkpoint creation lies with the DPTNResiliency plugin, which utilizes the current marking and communication status. To ensure fault tolerance, especially in the event of a simulator crash, each checkpoint is initially written to a temporary location. Only upon successful creation is it copied to its final destination. Subsequently, the corresponding event is marked as consumed, ensuring that it cannot be processed again.

Checkpoints generated by the DPTNResiliency plugin must be stored in a resilient, highly available storage system. This storage operation is triggered after each Kafka event has been successfully processed. For this purpose, the distributed storage system Ceph [57] has been selected. Ceph ensures high availability and fault tolerance by requiring a minimum of three participating nodes, thereby eliminating single points of failure and mitigating split-brain scenarios.

The creation of checkpoints can also be regarded as a transaction in a (multi-)database system, making it effectively a sub-transaction in the context of the triggering of a distributed synchronous channel. As its write operations conform to ACID principles, Ceph ensures the correctness and durability of stored simulation checkpoints. Additionally, being a distributed system itself, Ceph also underlies the constraints of the CAP theorem. Being designed to prioritize Consistency and Partition Tolerance, in the event of network partitioning, it may temporarily compromise the availability of specific components to uphold global consistency guarantees, matching the specification of our system.

6.4. Implementation

The DPTNResiliency plugin is implemented as an additional modular Renew plugin. The simulators, as well as the synchronization service, run in Docker containers within Kubernetes (Section 2.5), deployed via StatefulSets that include liveness probes. Kafka (Section 2.2), our event broker, is deployed with high availability on Kubernetes. Attached to each Renew container is a Persistent Volume provided by Rook, the Kubernetes Deployment of Ceph, on which the checkpoints are stored.

6.5. Evaluation

Our recovery mechanism is backed by the Kafka event history that acts as a highly available and distributed log. Additionally, our checkpoints are stored on a highly available distributed storage system as well. After processing an event, a simulator creates a checkpoint, using an atomic copy operation when putting it into the right place to prevent checkpoint corruption. Only after the checkpoint is created, the simulator commits its Kafka offset, marking the event as consumed in the log and making sure it's not consumed again. Furthermore, any simulator that fails is automatically restarted, which triggers the recovery mechanism that upholds the simulations integrity.

If a failure occurs before a checkpoint for an event is written, the simulator restores from the previous checkpoint and executes the event again, thus recovering successfully. If a failure occurs after a checkpoint is written, but before the Kafka commit has been completed, the simulator restores from the new checkpoint but is unable to execute the event again. This will lead to a global event timeout and the event will be attempted again, in which case it will now succeed, thus completing the recovery. Failures during the recovery process also fall into one of these two categories.

When drawing a parallel to database transactions, it becomes evident that our solution adheres to the ACID properties. If we view the processing of each Kafka event as a single transaction, it becomes clear that our methods guarantee atomicity and durability, in a similar way to how databases implement transaction logic. This is because we either process a single event fully or not at all (in which case we later recover and do process it), and we make the changes durable by writing them to our storage medium. The non-resilient Distributed P/T Net implementation already ensures consistency and isolation and remains unaffected by our enhancements.

Altogether, these guarantees fulfill all functional requirements for our prototype, thereby confirming the resilience of the simulators. Nonetheless, one limitation persists: unresolved race conditions exist within the simulation thread pool of Renew. Although a delay mechanism has been introduced to temporarily mitigate this issue—and occurrences are exceedingly rare—it nonetheless imposes a slowdown on the distributed simulation.

7. Producer-Storage-Consumer Scenario

To validate the functionality of our prototypes, as outlined in the preceding sections, we experimented within our Kubernetes cluster to substantiate this claim. The structure of the experiment is presented as follows: Section 7.1 details the experimental setup, Section 7.2 describes the execution process, and Section 7.3 discusses our observations.

The experimental scenario is based on a classical problem in computer science: the Producer-Consumer–Storage example. In this context, we employ a modeling approach based on distributed P/T nets (Section 2.2), featuring distributed up- and downlinks in place of standard synchronous communication channels, as depicted in Figure 1.

The scenario comprises three components: a producer, a consumer, and a storage unit. The producer operates within a cyclic process in which a message is first generated and subsequently transmitted to the storage via a distributed downlink. Conversely, the consumer follows a cyclic process in which messages are actively retrieved from storage—again via a distributed downlink—and subsequently consumed. This configuration implies the presence of two active entities: the producer and the consumer. In contrast, the storage component, which exclusively features distributed uplinks, remains entirely passive.

7.1. Setup

As the foundation for this experiment, we employ the distributed system described previously in Section 4, comprising three physical machines. Within our Kubernetes cluster, we deploy a highly available Kafka cluster to serve as the communication medium. Furthermore, we deploy four components—each a Renew instance—using Kubernetes StatefulSets: the synchronization service, Producer, Storage, and Consumer. Each of these components consists of exactly one Container in a Kubernetes Pod with a replica size of one. All StatefulSets are configured with liveness probes that target endpoints exposed by the CloudNative Renew plugin. Moreover, each component is provisioned with 5 GiB of highly available persistent storage via Rook/Ceph.

Prior to initiating the experiment, we ensure that all system components are returned to their default state. To this end, we delete the four StatefulSets, if present, and erase the data stored in their associated volumes. Additionally, we remove all Kafka topics and consumer offsets to prevent residual data from affecting communication in the upcoming simulation. This reset procedure is critical, as remnants of prior experiments could otherwise influence the outcomes.

```
ScriptCommand: Try to load file startscript_storage.txt
Opening gui...
Passing args to gui...
Initialising CheckpointStorageServiceImpl with no previous checkpoint
Starting Simulation...
...
Simulation initialized.
INFO: Consumed event UpdateUplink from topic receiveMessage.
INFO: Consumed event UpdateDownlink from topic sendMessage.
INFO: Consumed event RequestFiring from topic sendMessage.
INFO: Record ConfirmUplink sent successfully on topic sendMessage.
INFO: Consumed event ConfirmUplink from topic sendMessage.
INFO: Consumed event ConfirmDownlink from topic sendMessage.
INFO: Consumed event ConfirmFiring from topic sendMessage.
INFO: Fired Confirm Transition of channel: sendMessage in Storage.
INFO: Record UpdateDownlink sent succcessfully on topic receiveMessage.
INFO: Consumed event UpdateDownlink from topic sendMessage.
INFO: Consumed event UpdateDownlink from topic receiveMessage.
INFO: Consumed event UpdateDownlink from topic sendMessage.
INFO: Consumed event RequestFiring from topic sendMessage.
INFO: Record ConfirmUplink sent successfully from topic sendMessage.
INFO: Consumed event ConfirmDownlink from topic sendMessage.
INFO: Consumed event RequestFiring from topic receiveMessage.
INFO: Record ConfirmDownlink sent successfully on topic receiveMessage.
INFO: Fired Request Transition of channel: receiveMessage in Storage.
INFO: Record UpdateDownlink sent successfully on topic receiveMessage.
INFO: Consumed event ConfirmUplink from topic receiveMessage.
INFO: Consumed event ConfirmUplink from topic sendMessage.
INFO: Consumed event UpdateDownlink from topic sendMessage.
INFO: Consumed event ConfirmFiring from topic sendMessage.
...
```

Figure 4: The logs of the Storage Pod after the simulation is started

7.2. Execution

Once the setup has been completed, the experiment can be initiated. The simulation commences with the deployment of the four system components. To verify correct execution, the logs of each Pod are inspected to ensure that the simulation is actively running.

Following confirmation of execution, a brief waiting period is introduced. This period is sufficiently long to allow the simulation to make measurable progress, yet not so extensive that it reaches completion during this interval.

Subsequently, a fail-stop fault is emulated by deliberately terminating one of the simulator Pods. Upon automatic restart, the process resumes, and the simulation continues until it reaches completion.

7.3. Observations

```
k8user@artpc17:~$ kubectl get pods -n dptn
NAME                        READY   STATUS  RESTARTS    AGE
consumer-statefulset-0      1/1     Running 0           60s
producer-statefulset-0      1/1     Running 0           60s
storage-statefulset-0       1/1     Running 0           60s
syncservice-statefulset-0   1/1     Running 0           60s
k8user@artpc17:~$ kubectl delete pod storage-statefulset-0 -n dptn
pod "storage-statefulset-0" deleted
k8user@artpc17:~$ kubectl get pods -n dptn
NAME                        READY   STATUS  RESTARTS    AGE
consumer-statefulset-0      1/1     Running 0           115s
producer-statefulset-0      1/1     Running 0           115s
storage-statefulset-0       1/1     Running 0           21s
syncservice-statefulset-0   1/1     Running 0           115s
```

Figure 5: Deleting the Storage Pod

```
ScriptCommand: Try to load file startscript_storage.txt
Opening gui...
Passung args to gui...
Initialising CheckpointStorageServiceImpl with checkpoint Storage-232.rst
Starting Simulation...
...
Simulation initialised.
INFO: Record RegisterDownlink sent successfully on topic RegisterTopic.
INFO: Subscribed to topic: receiveMessage.
INFO: Subscribed to topic: sendMessage.
INFO: Consumed event ConfirmDownlink from topic receiveMessage.
INFO: Consumed event ConfirmFiring from topic receiveMessage.
INFO: Fired Confirm Transition of channel: receiveMessage Storage.
INFO: Consumed event UpdateDownlink from topic receiveMessage.
INFO: Consumed event UpdateUplink from topic receiveMessage.
INFO: Consumed event UpdateDownlink from topic receiveMessage.
INFO: Consumed event RequestFiring from topic receiveMessage.
INFO: Record ConfirmDownlink sent successfully on topic receiveMessage.
INFO: Fired Request Transition of channel: receiveMessage in Storage.
INFO: Record UpdateDownlink sent successfully on topic receiveMessage.
...
```

Figure 6: The logs of the Storage Pod after the restart

Upon initiating the simulation, the logs of all simulator instances consistently indicate that the simulation is actively running. As illustrated in Figure 4, Kafka events are successfully transmitted and received, confirming the correct operation of the simulation. In this context, the UpdateUplink and UpdateDownlink events represent communication with the synchronization service regarding updates

to the up- and downlinks of the registered distributed synchronous channels. The RequestFiring event is employed to coordinate a distributed firing across all relevant simulators. Upon successful execution, the ConfirmUplink, ConfirmDownlink, and ConfirmFiring events confirm the resulting state changes. Detailed specifications of these event types are provided by Clasen et al. [10].

Following the deletion of a simulator Pod, Kubernetes automatically initiates a replacement Pod to restore the simulation topology. As depicted in Figure 5, one simulator Pod exhibits a delayed startup relative to the others, corresponding to the previously deleted instance.

Subsequently, once the newly instantiated Pod begins to receive Kafka events, Figure 6 demonstrates that it resumes participation in distributed transitions, including up- and downlink operations. Moreover, an examination of the logs from the remaining simulator Pods verifies that interaction with the restarted Pod is functioning as expected.

This experiment can be replicated with either the Consumer or the Producer Pod, yielding analogous results. These observations collectively substantiate the resilience of our simulator architecture.

8. Discussion

A key advantage of the proposed concept lies in the inherent resilience of the simulators. This resilience ensures that the failure of individual simulators during runtime does not compromise the integrity or continuity of the overall simulation process.

Furthermore, the system enables the execution of resilient distributed simulations over extended durations—potentially spanning several weeks or even months. This capability markedly enhances the system's usability, as it alleviates user concerns regarding potential simulator failures during long-term simulations.

A notable drawback of the proposed system concerns the complexity of the underlying execution infrastructure. Deployment necessitates the orchestration of multiple physical nodes within containerized environments such as Kubernetes. These infrastructural requirements impose substantial demands on the system's architectural and operational design.

Another disadvantage, albeit with a very low probability of occurrence, is the potential for deadlocks arising from the interaction between the DPTNResiliency plugin and the internal Renew simulation thread pool. Specifically, to utilize Renew's existing functionality for saving simulation states, the simulation must be paused. However, there is no guarantee that all queued events will be processed before the pause takes effect. Given the highly asynchronous architecture of Renew, a rapid succession of pause and resume operations—especially in conjunction with checkpoint creation—can, under rare circumstances, cause the internal thread pool to enter a deadlock state. If such a deadlock occurs, it halts the simulator and remains undetectable. This issue is currently the subject of ongoing investigation.

One limitation of the current system design is the non-resilient nature of the synchronization service. Ensuring system stability under failure conditions necessitates the development and integration of supplementary recovery mechanisms for this component.

The checkpointing-based recovery strategy, while critical for fault tolerance, introduces additional computational overhead. This overhead negatively impacts the performance of individual simulators and the overall simulation, resulting in increased execution times. In particular, delays may occur in the processing of events transmitted via Kafka, further contributing to reduced simulation efficiency.

A further limitation stems from the current checkpointing policy, which creates a checkpoint for every Kafka event consumed. This leads to the generation of a substantial number of checkpoints during the simulation. Future work will focus on optimizing checkpoint frequency to minimize this overhead while maintaining resilience.

9. Related Work

Research conducted by Moldt et al.[62] and Röwekamp et al.[63, 64, 65, 66, 67, 18, 68] focuses on distributed Reference Net simulations, with a particular emphasis on platform management. Their

work incorporates Mulan agent concepts [69] and employs Spring Boot to enable initial experimental implementations. While these contributions establish essential foundations, they do not fully exploit the potential of distributed systems, thereby limiting their applicability to complex, real-world application scenarios.

In contrast, the studies presented in [70, 71, 72, 73] address the distributed simulation of timed Petri nets, employing various strategies to ensure accurate simulation of time-dependent behaviors. This work differentiates itself from these approaches by focusing on a P/T net class that does not incorporate a notion of time.

A related yet distinct approach is proposed in [6], which addresses resilient simulation by replicating entire simulators to tolerate failures. The primary distinction from the concept presented in this paper lies in the recovery mechanism: rather than replication, checkpointing is employed as the means of fault tolerance.

This contribution builds on the article by Clasen et al. [10] and extends the simulators developed there by the property of resilience through a checkpoint-based recovery technique of the simulators. Whereas Clasen et al. [74] does not focus on resilience but on scalability. The idea there is that the number of simulators can be adapted dynamically.

10. Conclusion

The following conclusion begins with a concise summary (Section 10.1) of the main findings and contributions of this work. It then outlines potential directions for future research and development (Section 10.2).

10.1. Summary

After introducing the foundational concepts—Renew, distributed P/T nets, failure semantics, resilience strategies, and Kubernetes (Section 2)—this work delineates the central research problem (Section 3): the design of a resilient simulation framework for P/T nets within a Kubernetes-based cloud environment. The distributed system described in Section 4 comprises a communication medium, highly available storage, and simulation components for distributed P/T nets.

Subsequently, we present the developed prototypes. In Section 5, Detecting Failures of Simulators, we examine strategies for reliably identifying faults within simulation components. Building upon these insights, Section 6, Recovering Failures of Simulators, introduces the novel Renew plugin DPTNResiliency, which facilitates the automated recovery of failed simulation instances based on checkpointing.

To evaluate the proposed system, we employ the well-established Producer-Storage-Consumer scenario (Section 7). During simulation, a simulator instance is intentionally terminated. Recovery is demonstrated as a new instance seamlessly resumes execution using state checkpoints generated by its predecessor. Finally, Section 8 offers a critical assessment of the approach's strengths, limitations, and potential trade-offs, followed by a contextualization within the scope of related research (Section 9).

10.2. Future Work

The immediate next steps involve systematically resolving existing workarounds and eliminating current race conditions. This effort is expected to reduce the overhead associated with the current approach significantly. Furthermore, we intend to eliminate the need for checkpoint creation on every consumed event, thereby further enhancing performance.

Concurrently, a more advanced and fully automated testing infrastructure is required—one that can realistically emulate a broad range of failure scenarios, including those studied in the field of chaos engineering. Such a framework is essential to bolster the reliability and credibility of the proposed solution.

The synchronization service must also be extended to ensure fault tolerance not only at the individual simulator level but across the entire simulation system. Using the aforementioned improved testing infrastructure, the proposed concept should be rigorously validated against real-world models to substantiate its empirical robustness.

Additional refinements are conceivable concerning the health metrics reported by Renew. Increasing the granularity and precision of these metrics could facilitate more nuanced operational responses and may enable the early detection of system-level anomalies such as deadlocks.

A further avenue for research pertains to the scalability of the simulation framework. To date, a fixed number of simulators has been employed, resulting in uneven load distribution during simulations. Exploring dynamic scaling strategies could mitigate this inefficiency and unlock new levels of performance.

Declaration on Generative AI

During the preparation of this work, the authors used …

- …Bing Translate in order to: Translate Text.
- …DeepL in order to: Translate Text.
- …ChatGPT in order to: Rephrasing.
- …Grammarly in order to: Grammar and spelling check, Repharsing.

After using these tool(s)/service(s), the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] R. Fujimoto, Parallel and distributed simulation, in: Proceedings of the 2015 Winter Simulation Conference, Huntington Beach, CA, USA, December 6-9, 2015, IEEE/ACM, 2015, pp. 45–59. URL: https://doi.org/10.1109/WSC.2015.7408152. doi:10.1109/WSC.2015.7408152.

[2] R. M. Fujimoto, Research challenges in parallel and distributed simulation, ACM Transactions on Modeling and Computer Simulation (TOMACS) 26 (2016) 1–29.

[3] R. M. Fujimoto, Development of the parallel and distributed simulation field, Simul. 100 (2024) 1197–1223. URL: https://doi.org/10.1177/00375497241261407. doi:10.1177/00375497241261407.

[4] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE transactions on dependable and secure computing 1 (2004) 11–33.

[5] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, D. B. Johnson, A survey of rollback-recovery protocols in message-passing systems, ACM Computing Surveys (CSUR) 34 (2002) 375–408.

[6] G. D'Angelo, S. Ferretti, M. Marzolla, Fault tolerant adaptive parallel and distributed simulation through functional replication, Simulation Modelling Practice and Theory 93 (2019) 192–207.

[7] R. Budde, K. Kautz, K. Kuhlenkamp, H. Züllighoven, What is prototyping?, Information Technology & People 6 (1990) 89–95.

[8] G. Pomberger, W. Pree, A. Stritzinger, Methoden und Werkzeuge für das Prototyping und ihre Integration, Inform., Forsch. Entwickl. 7 (1992) 49–61.

[9] T. Wilde, T. Hess, Forschungsmethoden der Wirtschaftsinformatik, Wirtschaftsinformatik 4 (2007) 280–287.

[10] L. Clasen, S. Bartelt, Y. Stahl, D. Moldt, Distributed P/T Net Simulation Prototypes Based on Event Streaming, in: M. Köhler-Bußmeier, D. Moldt, H. Rölke (Eds.), Proceedings of the International Workshop on Petri Nets and Software Engineering 2024 co-located with the 45th International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2024), June 24 - 25, 2024, Geneva, Switzerland, volume 3730 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024, pp. 192–216. URL: https://ceur-ws.org/Vol-3730.

[11] O. Kummer, F. Wienberg, M. Duvigneau, L. Cabac, M. Haustermann, D. Mosteller, Renew – the Reference Net Workshop, 2023. URL: http://www.renew.de/, release 4.1.
[12] Oracle, Java Documentation, 2025. URL: https://docs.oracle.com/en/java/, accessed: 2025-04-25.
[13] Gradle, Gradle Documentation, 2025. URL: https://docs.gradle.org/, accessed: 2025-04-25.
[14] M. Duvigneau, Konzeptionelle Modellierung von Plugin-Systemen mit Petrinetzen, volume 4 of *Agent Technology – Theory and Applications*, Logos Verlag, Berlin, 2010. URL: http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=2561&lng=eng&id=.
[15] L. Clasen, D. Moldt, M. Hansson, S. Willrodt, L. Voß, Enhancement of Renew to Version 4.0 using JPMS, in: M. Köhler-Bußmeier, D. Moldt, H. Rölke (Eds.), Proceedings of the International Workshop on Petri Nets and Software Engineering 2022 co-located with the 43rd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2022), Bergen, Norway, June 20th, 2022, volume 3170 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022, pp. 165–176. URL: https://ceur-ws.org/Vol-3170.
[16] D. Moldt, J. Johnsen, R. Streckenbach, L. Clasen, M. Haustermann, A. Heinze, M. Hansson, M. Feldmann, K. Ihlenfeldt, RENEW: Modularized Architecture and New Features, in: L. Gomes, R. Lorenz (Eds.), Application and Theory of Petri Nets and Concurrency - 44th International Conference, PETRI NETS 2023, Lisbon, Portugal, June 25-30, 2023, Proceedings, volume 13929 of *Lecture Notes in Computer Science*, Springer Nature Switzerland AG, Cham, Switzerland, 2023, pp. 217–228. URL: https://doi.org/10.1007/978-3-031-33620-1_12.
[17] O. Kummer, Referenznetze, Logos Verlag, Berlin, 2002. URL: http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=0035&lng=eng&id=.
[18] J. H. Röwekamp, M. Taube, P. Mohr, D. Moldt, Cloud Native Simulation of Reference Nets, in: M. Köhler-Bußmeier, E. Kindler, H. Rölke (Eds.), Proceedings of the International Workshop on Petri Nets and Software Engineering 2021 co-located with the 42nd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2021), Paris, France, June 25th, 2021 (due to COVID-19: virtual conference), volume 2907 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021, pp. 85–104. URL: http://ceur-ws.org/Vol-2907.
[19] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, T. Risberg, A. Arendsen, D. Davison, D. Kopylenko, M. Pollack, T. Templier, E. Vervaet, P. Tung, B. Hale, A. Colyer, J. Lewis, C. Leau, M. Fisher, S. Brannen, R. Laddad, A. Poutsma, C. Beams, T. Abedrabbo, A. Clement, D. Syer, O. Gierke, R. Stoyanchev, P. Webb, R. Winch, B. Clozel, S. Nicoll, S. Deleuze, J. Bryant, M. Paluch, Spring Framework Reference Documentation, https://docs.spring.io/spring-framework/reference/index.html, 2025. Version 6.2.6, abgerufen am 25. April 2025.
[20] L. Voß, S. Willrodt, D. Moldt, M. Haustermann, Between Expressiveness and Verifiability: P/T-nets with Synchronous Channels and Modular Structure, in: M. Köhler-Bußmeier, D. Moldt, H. Rölke (Eds.), Proceedings of the International Workshop on Petri Nets and Software Engineering 2022 co-located with the 43rd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2022), Bergen, Norway, June 20th, 2022, volume 3170 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022, pp. 40–59. URL: https://ceur-ws.org/Vol-3170.
[21] P. Fettke, W. Reisig, Once and for all: how to compose modules – The composition calculus, 2024. URL: https://arxiv.org/abs/2408.15031. arXiv:2408.15031.
[22] J. Kreps, N. Narkhede, J. Rao, et al., Kafka: A distributed messaging system for log processing, in: NetDB 2011: 6th Workshop on Networking meets Databases, volume 11, Athens, Greece, 2011, pp. 1–7.
[23] A. S. Foundation, Apache Kafka Documentation, 2025. URL: https://kafka.apache.org/documentation/, accessed: 2025-01-21.
[24] N. Garg, Apache Kafka, Packt Publishing Birmingham, UK, 2013.
[25] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, P. Bodík, M. Musuvathi, Z. Zhang, L. Zhou, Failure Recovery: When the Cure Is Worse Than the Disease, in: HotOS, USENIX, 2013, pp. 1–6. URL: https://www.microsoft.com/en-us/research/publication/failure-recovery-when-the-cure-is-worse-than-the-disease/.
[26] Calvanese, Diego, Types of program errors, 2006. URL: https://www.inf.unibz.it/~calvanese/

teaching/06-07-ip/lecture-notes/uni10/node2.html.
[27] M. Ghanavati, D. Costa, A. Andrzejak, J. Seboek, Memory and resource leak defects in java projects: an empirical study, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 410–411. URL: https://doi.org/10.1145/3183440.3195032. doi:10.1145/3183440.3195032.
[28] D. Giebas, R. Wojszczyk, Deadlocks Detection in Multithreaded Applications Based on Source Code Analysis, Applied Sciences 10 (2020). URL: https://www.mdpi.com/2076-3417/10/2/532. doi:10.3390/app10020532.
[29] B. Schroeder, G. A. Gibson, A Large-Scale Study of Failures in High-Performance Computing Systems, IEEE Transactions on Dependable and Secure Computing 7 (2010) 337–350. doi:10.1109/TDSC.2009.4.
[30] hello2morrow GmbH, SonarGraph – Static Analysis and Architecture Validation Tool, https://www.hello2morrow.com/, 2024. Accessed on June 1, 2025.
[31] B. Pugh, D. Hovemeyer, FindBugs – Static Bug Detector for Java, https://findbugs.sourceforge.net/, 2015. Accessed on June 1, 2025.
[32] J. Seward, V. Developers, Valgrind – Debugging and Profiling Tools, https://valgrind.org/, 2024. Accessed on June 1, 2025.
[33] F. S. Foundation, GDB: The GNU Project Debugger, https://sourceware.org/gdb/, 2024. Accessed on June 1, 2025.
[34] G. LLC, AddressSanitizer – A Fast Memory Error Detector, https://github.com/google/sanitizers/wiki/AddressSanitizer, 2024. Accessed on June 1, 2025.
[35] P. Authors, Prometheus – Monitoring System & Time Series Database, https://prometheus.io/, 2024. Accessed on June 1, 2025.
[36] W. Hasselbring, A. van Hoorn, Kieker: A monitoring framework for software engineering research, Software Impacts 5 (2020) 100019. URL: https://www.sciencedirect.com/science/article/pii/S2665963820300063. doi:https://doi.org/10.1016/j.simpa.2020.100019.
[37] A. Ledmi, H. Bendjenna, S. M. Hemam, Fault Tolerance in Distributed Systems: A Survey, in: 2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS), 2018, pp. 1–5. doi:10.1109/PAIS.2018.8598484.
[38] A. Kumar, D. Malhotra, Study of Various Proactive Fault Tolerance Techniques in Cloud Computing, International Journal of Computer Sciences and Engineering 06 (2018) 81–87. doi:10.26438/ijcse/v6si3.8187.
[39] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, J. Dongarra, Post-failure recovery of MPI communication capability: Design and rationale, The International Journal of High Performance Computing Applications 27 (2013) 244–254. URL: https://doi.org/10.1177/1094342013488238. doi:10.1177/1094342013488238.
[40] A. Dagur, R. Yadav, R. Ranvijay, Fault Tolerance in Real Time Distributed System, International Journal on Computer Science and Engineering 3 (2011).
[41] J.-C. Laprie, et al., From dependability to resilience, in: 38th IEEE/IFIP Int. Conf. On dependable systems and networks, 2008, pp. G8–G9. URL: https://2008.dsn.org/fastabs/dsn08fastabs_laprie.pdf.
[42] S. Pradhan, A. Dubey, T. Levendovszky, P. S. Kumar, W. A. Emfinger, D. Balasubramanian, W. Otte, G. Karsai, Achieving resilience in distributed software systems via self-reconfiguration, Journal of Systems and Software 122 (2016) 344–363. URL: https://www.sciencedirect.com/science/article/pii/S0164121216300590. doi:https://doi.org/10.1016/j.jss.2016.05.038.
[43] The Kubernetes Authors, Kubernetes, 2025. URL: https://kubernetes.io/docs.
[44] A. M. Potdar, N. D G, S. Kengond, M. M. Mulla, Performance Evaluation of Docker Container and Virtual Machine, Procedia Computer Science 171 (2020) 1419–1428. URL: https://www.sciencedirect.com/science/article/pii/S1877050920311315. doi:https://doi.org/10.1016/j.procs.2020.04.152, third International Conference on Computing and Network Communications (CoCoNet'19).
[45] D. Inc., Docker – Empowering App Development for Developers, https://www.docker.com/, 2024.

Accessed on June 1, 2025.
[46] I. Red Hat, Podman – A Daemonless Container Engine for Developers, https://docs.podman.io/en/latest/, 2024. Accessed on June 1, 2025.
[47] O. C. Initiative, OCI – Open Container Initiative Specifications, https://opencontainers.org/, 2024. Accessed on June 1, 2025.
[48] M. Pace, Zero Trust Networks with Istio, Master's thesis, Politecnico Di Torino, 2021. URL: https://webthesis.biblio.polito.it/21170/.
[49] Open Container Initiative, OCI Runtime Spec, 2025. URL: https://github.com/opencontainers/runtime-spec/blob/main/runtime.md.
[50] C. Albuquerque, K. Relvas, F. F. Correia, K. Brown, Proactive monitoring design patterns for cloud-native applications, in: Proceedings of the 27th European Conference on Pattern Languages of Programs, EuroPLop '22, Association for Computing Machinery, New York, NY, USA, 2023, pp. 1–13. URL: https://doi.org/10.1145/3551902.3551961. doi:10.1145/3551902.3551961.
[51] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, C. Maltzahn, Ceph: A Scalable, High-Performance Distributed File System, in: 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06), USENIX Association, Seattle, WA, 2006, pp. 307–320. URL: https://www.usenix.org/conference/osdi-06/ceph-scalable-high-performance-distributed-file-system.
[52] Rook, Authors, Rook Documentation, 2025. URL: https://rook.io/docs/rook/latest-release, accessed: 2025-04-25.
[53] L. Mercl, J. Pavlik, Public Cloud Kubernetes Storage Performance Analysis, in: N. T. Nguyen, R. Chbeir, E. Exposito, P. Aniorté, B. Trawiński (Eds.), Computational Collective Intelligence, Springer International Publishing, Cham, 2019, pp. 649–660. URL: https://doi.org/10.1007/978-3-030-28374-2_56. doi:10.1007/978-3-030-28374-2_56.
[54] C. N. C. Foundation, CNCF – Cloud Native Computing Foundation, https://www.cncf.io/, 2024. Accessed on June 1, 2025.
[55] T. L. Foundation, The Linux Foundation – Supporting Open Source Innovation, https://www.linuxfoundation.org/, 2024. Accessed on June 1, 2025.
[56] The Linux Foundation, Cloud Native Landscape, 2025. URL: https://landscape.cncf.io/.
[57] Ceph, Authors, Ceph Documentation: Reef Release, 2025. URL: https://docs.ceph.com/en/reef/, accessed: 2025-04-25.
[58] Y. Breitbart, H. Garcia-Molina, A. Silberschatz, Overview of multidatabase transaction management, in: CASCON First Decade High Impact Papers, CASCON '10, IBM Corp., USA, 2010, p. 93–126. URL: https://doi.org/10.1145/1925805.1925811.
[59] T. Haerder, A. Reuter, Principles of transaction-oriented database recovery, ACM computing surveys (CSUR) 15 (1983) 287–317.
[60] A. Kemper, A. Eickler, Datenbanksysteme – Eine Einführung, 8 ed., Oldenbourg Verlag, 2011.
[61] E. A. Brewer, Towards robust distributed systems, in: PODC, volume 7, Portland, OR, 2000, pp. 343–477.
[62] D. Moldt, J. H. Röwekamp, M. Simon, A Simple Prototype of Distributed Execution of Reference Nets Based on Virtual Machines, in: R. Bergenthum, E. Kindler (Eds.), Algorithms and Tools for Petri Nets Proceedings of the Workshop AWPN 2017, Kgs. Lyngby, Denmark October 19-20, 2017, DTU Compute Technical Report 2017-06, 2017, pp. 51–57.
[63] J. H. Röwekamp, D. Moldt, M. Feldmann, Investigation of Containerizing Distributed Petri Net Simulations, in: D. Moldt, E. Kindler, H. Rölke (Eds.), Petri Nets and Software Engineering. International Workshop, PNSE'18, Bratislava, Slovakia, June 25-26, 2018. Proceedings, volume 2138 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2018, pp. 133–142. URL: http://ceur-ws.org/Vol-2138/.
[64] J. H. Röwekamp, Investigating the Java Spring Framework to Simulate Reference Nets with Renew, in: R. Lorenz, J. Metzger (Eds.), Algorithms and Tools for Petri Nets, number 2018-02 in Reports / Technische Berichte der Fakultät für Angewandte Informatik der Universität Augsburg, 2018, pp. 41–46. URL: https://opus.bibliothek.uni-augsburg.de/opus4/41861.
[65] J. H. Röwekamp, D. Moldt, RenewKube: Reference Net Simulation Scaling with Renew and

Kubernetes, in: S. Donatelli, S. Haar (Eds.), Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings, volume 11522 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 69–79. URL: https://doi.org/10.1007/978-3-030-21571-2_4.

[66] J. H. Röwekamp, M. Feldmann, D. Moldt, M. Simon, Simulating Place / Transition Nets by a Distributed, Web Based, Stateless Service, in: D. Moldt, E. Kindler, M. Wimmer (Eds.), Petri Nets and Software Engineering. International Workshop, PNSE'19, Aachen, Germany, June 24, 2019. Proceedings, volume 2424 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2019, pp. 163–164. URL: http://CEUR-WS.org/Vol-2424.

[67] J. H. Röwekamp, M. Buchholz, D. Moldt, Petri Net Sagas, in: M. Köhler-Bußmeier, E. Kindler, H. Rölke (Eds.), Proceedings of the International Workshop on Petri Nets and Software Engineering 2021 co-located with the 42nd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2021), Paris, France, June 25th, 2021 (due to COVID-19: virtual conference), volume 2907 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021, pp. 65–84. URL: http://ceur-ws.org/Vol-2907.

[68] J. H. Röwekamp, Skalierung von nebenläufigen und verteilten Simulationssystemen für interagierende Agenten, Ph.D. thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2023. URL: https://ediss.sub.uni-hamburg.de/handle/ediss/10040.

[69] H. Rölke, Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen, volume 2 of *Agent Technology – Theory and Applications*, Logos Verlag, Berlin, 2004. URL: http://logos-verlag.de/cgi-bin/engbuchmid?isbn=0768&lng=eng&id=.

[70] G. S. Thomas, J. Zahorjan, Parallel simulation of performance petri nets: Extending the domain of parallel simulation, Technical Report, Institute of Electrical and Electronics Engineers (IEEE), 1991.

[71] H. H. Ammar, S. Deng, Time warp simulation of stochastic Petri nets, in: Proceedings of the Fourth International Workshop on Petri Nets and Performance Models PNPM91, IEEE, 1991, pp. 186–195.

[72] G. Chiola, A. Ferscha, Distributed simulation of petri nets, IEEE Parallel and Distributed Technology 1 (1993) 33–50.

[73] A. Ferscha, Adaptive time warp simulation of timed petri nets, IEEE Transactions on Software Engineering 25 (1999) 237–257.

[74] L. Clasen, C. Nayci, E. Nacyi, J. Middendorf, T. Mack, Investigations Towards Dynamic Scaling of Distributed P/T Nets, in: M. Köhler-Bußmeier, D. Moldt, H. Rölke (Eds.), Proceedings of the International Workshop on Petri Nets and Software Engineering 2025 co-located with the 46th International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2025), June 22 - 27, 2025, Paris, France, CEUR Workshop Proceedings, CEUR-WS.org, 2025.