Investigations Towards Dynamic Scaling of Distributed P/T Nets[*]

Laif-Oke Clasen[1], Can Nayci[1], Efe Nayci[1], Justus Middendorf[1] and Till Mack[1]

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

Abstract

Simulating large P/T nets requires substantial computational resources to support the execution and exploration of complex systems. Efficient use of resources is crucial to simulate larger models and realize faster computations. Distribution to several computing devices is one option to support this kind of simulation.

Static assignment of P/T net structures to distributed simulators can lead to uneven load distribution, which affects the simulation's overall performance. A dynamic solution is needed to overcome these challenges and ensure optimal resource utilization.

A dynamic scaling model adapts the simulation to the current resource utilization. A self-regulating system controls the vertical and horizontal scaling of the simulation in real time based on the monitoring, ensuring better resource utilization. The research methodology is based on prototyping and focuses on constructivist principles.

The proposed system employs the Kubernetes Metrics Server to monitor current resource utilization. Based on these metrics, the Dynamic Resource Scaling system (DyReS) performs real-time vertical and horizontal scaling of simulators. DyReS comprises a decision-making component and dedicated modules for both scaling directions. Horizontal scaling necessitates the dynamic redistribution of P/T net fragments among simulators; to this end, the Renew simulator has been extended with the NetSplit and NetExchange plugins.

The results show that dynamic scaling improves resource utilization through flexible adaptation of computing resources, increasing the performance and efficiency of the distributed simulation of P/T nets.

Keywords

Dynamic Scaling, Distributed Simulation, P/T Nets, P/T Nets with Synchronous Channels

CEUR-WS.org/Vol-3998/paper08.pdf

1. Introduction

The increasing complexity of technical and organizational systems requires powerful simulation methods to ensure high accuracy and realistic mapping. The efficient use of available computing resources is a key challenge, especially in distributed simulation environments, which makes efficient simulation strategies indispensable. Current approaches are often based on static resource allocation, but in practice, this often leads to uneven load distribution and thus impairs the simulation's scalability and performance.

Observations show that static distribution methods often lead to uneven load distribution. This uneven load distribution not only impairs performance but also significantly limits scalability. Against this background, a dynamic, adaptive resource distribution approach is becoming increasingly important. This contribution investigates the potential and practical feasibility of a dynamic scaling model for the distributed simulation of place/transition nets (P/T nets). This topic is particularly relevant in the fields of software engineering and Petri net simulation, as it has a direct impact on the efficiency and ecological sustainability of distributed Petri net simulations.

The work specifically addresses how adaptive resource control can be realized to ensure load distribution during the simulation process. The study focuses on the hypothesis that continuously and automatically adapting the resource distribution can significantly increase the simulation's performance and energy efficiency.

Methodologically, the research is based on a constructivist approach that relies on developing and evaluating prototypes. [1, 2, 3] This approach involves developing a concept for dynamic resource

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

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

© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

management that enables the simulators to scale autonomously, vertically, and horizontally. For this purpose, a combination of monitoring and an adaptive decision component such as Dynamic Resource Scaler (DyReS) developed in this contribution is used. The used simulator is Renew[1] [4], which already implements a distributed P/T net simulation [5]. In addition, Renew is extended by the plugins NetSplit and NetExchange to enable a flexible and situation-dependent distribution of net segments between different simulator instances. The results of this work show that a dynamic scaling concept improves load balancing, increases simulation performance, and allows a more sustainable use of resources. We are aiming for scientific cloud computing with the overall system by providing it as scientific software as a service on our in-house cloud infrastructure.

Within the Foundations (Section 2), the topics of Renew (Section 2.1), Distributed P/T Nets (Section 2.2), Kubernetes (Section 2.3), Scalability and Elasticity (Section 2.4) and Monitoirng (Section 2.5) are addressed. Subsequently, the Problem Description (Section 3) and the design of the Distributed System (Section 4) are presented. The prototypes of this work follow this. These are the new Renew NetExchange plugin (Section 5), which can send P/T nets to other Renew simulators, the new Renew NetSplit plugin (Section 6), which can split P/T nets. In addition, further prototypes regarding the monitoring (Section 7) of the nodes and simulators and the dynamic scaling in the new Dynamic Resource Scaler (DyReS) (Section 8) are covered. A critical discussion (Section 9) of the proposed concept's advantages, disadvantages, and limitations follows. Finally, the article concludes with an overview of Related Work (Section 10) and the Conclusion (Section 11).

2. Foundations

2.1. Renew

Renew [4] is an open-source tool designed for the modeling, analysis, and simulation of various Petri net types, with a particular emphasis on distributed P/T nets (Section 2.2). The tool was developed by the Algorithms, Randomization, and Theory (ART) research group, previously known as Theoretical Foundations of Computer Science (TGI), at the University of Hamburg.

Renew is implemented in Java 17 [6] and built using Gradle 8.4 [7], ensuring robustness and platform independence. Its software architecture is based on a plugin system, as described by Duvigneau [8]. Its modularity and maintainability have recently been enhanced by adopting the Java Platform Module System (JPMS) [9, 10].

For each supported Petri net formalism, Renew offers a corresponding dedicated plugin, with the reference net formalism described by Kummer [11] being the most prominent. Additionally, the cloud-native plugin [12]—relevant to this contribution—enables the exposure of HTTP endpoints for Renew using Java Spring [13]. These endpoints allow for the remote initiation and control of Renew simulations.

2.2. Distributed P/T Nets

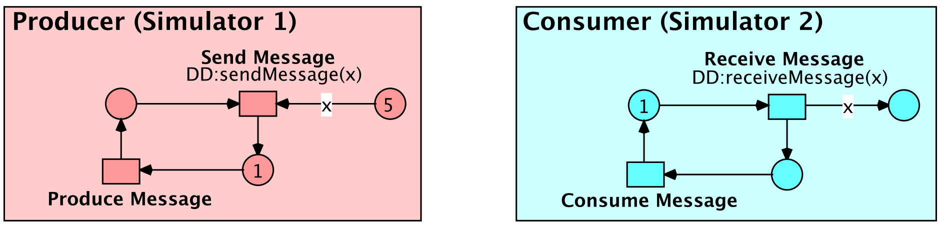
The overall system architecture for the distributed P/T nets [5] comprises multiple simulators, the event-based communication medium Kafka, and a synchronization service. The distributed P/T nets are statically allocated to the available simulators.

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 [14, 15]. 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

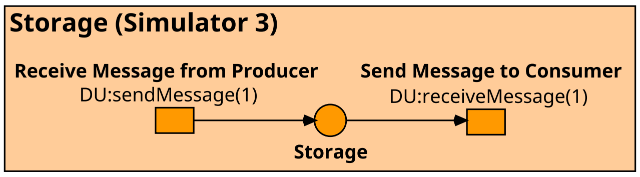
[1]**Re**ference **Ne**t **W**orkshop can be downloaded directly from its official website: http://renew.de.

and programming languages [16, 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. [5], who describe 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 [5]

2.3. Kubernetes

Modern distributed systems need an orchestrator unit to interconnect the different system parts and manage (distributed) applications. Kubernetes[2] is an open-source container orchestration software, developed by Google, which allows different machines to act as a computer cluster and to manage the lifecycle of containerized applications across them [17, p. 3]. Containers are lightweight, portable units that package an application together with its dependencies and runtime environment, ensuring consistency across environments. Docker[3] is the most widely used containerization platform, providing tools to build, distribute, and run containers efficiently on any system that supports it. Kubernetes-Cluster consist of a control and a worker node, where the controller is responsible for managing the cluster resources. For example, they delegate container creations to the worker nodes or configure a wide spectrum of network, security, storage, and other settings [17, p. 4].

The quantity of extra features besides container orchestration makes Kubernetes a popular choice among other orchestration tools like Docker Swarm[4] and others. In the following, the basic features of popular container orchestrators are compared (Table 2):

In Table 2, we see that Kubernetes and Docker Swarm stand out with regard to covered features. However, despite the minimal efficiency overhead of Kubernetes due to the lightweight nature of Docker Swarm, the latter does not cover infrastructurally crucial functionalities [19, p. 7].

[2]https://kubernetes.io/docs
[3]https://docs.docker.com
[4]https://docs.docker.com/engine/swarm/

	Kubernetes / Docker Swarm	**Marathon**	**Cloudify**
Resource distribution	CPU / Memory	-	-
Scheduling	Multiple types	Multiple types	Multiple types
Loadbalancing	Round-Robin	Manual impl.	-
Pod-State (Health-Check)	Network controlled	Network controlled	Network controlled
Error tolerance	Replicas (or HA)	Replicas (or HA)	-
Auto-Scaling	For all kinds of metrics	Manual impl.	Manual impl.

Figure 2: Comparison of Kubernetes, Docker Swarm, Marathon and Cloudify with regards to core-features of container-orchestration [18, p. 225]

2.4. Scalability

The term scalability with regards to computer systems was defined many times in the past [20, p. 205]. Scalability of algorithms/simulations and systems are two distinct concepts: In 1998, Darren Law defined the scalability of simulations as following:

> *"A scalable simulation is one that exhibits improvements in simulation capability in direct pro- portion to improvements in system architectural capability.'* [21]

To elaborate on the definition: A simulation is scalable if it utilizes the changing resources a system offers to improve its performance during runtime. This system architecture change may be done automatically (Autoscaling) or manually and in a horizontal or vertical manner: Horizontal scaling, also known as scaling out, is changing the amount of running applications in a system, for example by running more OS threads or adding computer nodes to a cluster [22, p. 1]. Vertical scaling, also known as scaling up, describes the change of resources on a single node [23, p. 19], which could be processors, allocated memory, disk space and much more.

Since modern operating systems [24, p. 9-14] and virtual machines [25, p. 15f] manage the growth of local per-node resources automatically, even non-scalable simulations take directly advantage of vertical scaling. Implementing horizontal scaling in simulation is much more difficult, as it requires the simulation to distribute itself across multiple nodes and introduce network communication.

Dynamic scaling, as examined in this contribution, entails a hybrid approach that integrates both horizontal and vertical scaling strategies to respond adaptively to workload fluctuations. The primary emphasis lies on the additional allocation of resources.

2.5. Monitoring and Alerting

To automatically scale computational resources vertically and horizontally based on utilization, there needs to be a collection and processing of measurements for provided and utilized resources. This section provides the foundations for these approaches discussed in this paper.

Metrics[26, 27] are time-series numerical measurements in aggregated form. The aggregated form indicates that metrics are not logs containing events. Metrics could, for example, include logs of a specific kind within a designated timeframe. These measurements can be anything and can be used for various purposes such as evaluating a system's performance or cost-effectiveness. For the purpose of this paper, the broader term 'metrics' is restricted to hardware statistics such as CPU, memory, disk storage, disk I/O, and network usage.

Due to contextual ambiguity in the definitions of the terms monitoring and alerting [27], based on definitions provided, this paper defines them in the following way:

Definition 1. The process of monitoring is regarded as the production, management, and consumption of metrics.

Definition 2. Observation is regarded as the process of applying a threshold or other tiered decision logic to metrics to define states of alertness, upon which a triggered reaction can occur.

Definition 3. Alerting is regarded as observation triggering an alert to be sent and processed by an external alert consumer.

The Kubernetes Metrics-API is a standard for consuming metrics present in a Kubernetes cluster [28]. It can be used by any metric consumers as a standardized basis on which to analyze a Kubernetes cluster. It is often provided and implemented by a cluster service (e.g., Prometheus Adapter or Metrics Server). The Kubernetes Metrics-API allows consumers to only need to interact with Kubernetes API Server components. This removes the necessity for a metric consumer to interact with other application-specific interfaces or APIs, such as the PromQL endpoint of a Prometheus Server.

Prometheus [26] is an open-source general-purpose metric system that provides multiple standards and tools. Among these are the open scrape format OpenMetrics and the Prometheus Server for metric accumulation and storage, built primarily for cloud environments like Kubernetes. The broader Prometheus scope discussed in this paper includes:

- A central Prometheus Server for scraping, storing, and querying metrics via the Prometheus Query Language (PromQL).
- Prometheus exporters, which provide metrics primarily for a central Prometheus Server to scrape at a tunable interval using the Prometheus exposition format, OpenMetrics.
- The Alertmanager, which manages the process of alerting, such as notifying administrators via messaging platforms or forwarding alerts to other alert-receiver services if metrics satisfy configurable PromQL statements.
- Additionally, there is the PrometheusAdapter, providing aggregated Prometheus metrics to the Kubernetes Metrics-API.

The Kubernetes SIG Metrics Server [29] is an open-source CPU and memory metric producer, directly providing the most recently scraped data point of a metric to the Kubernetes Metrics-API in a cluster. The Metrics Server uses a predefined scraping interval to update the provided metrics. It is intended to provide live metrics for autoscaling applications. The Metrics Server does not accumulate metrics like Prometheus does, nor does it observe, alert, or otherwise consume them via an additional component like the Prometheus Alertmanager.

In this paper, these technologies are regarded as providing different fundamental approaches to metric consumption by a metric consumer. The pull approach to metric consumption is implementable via the Kubernetes Metrics-API or PromQL. Metrics are retrieved and consumed at intervals by the consumer. The push approach, on the other hand, is implementable via the Alertmanager. In this approach, metrics rising above a certain threshold trigger an alert to notify a metric consumer.

3. Problem Description

In the distributed simulation of Petri nets, the entire net is traditionally partitioned statically across a fixed number of simulators, as described in [5]. Such static partitioning, however, severely limits the scalability of the simulators: although additional simulators can be launched, migrating the ongoing net simulation to these simulator instances requires considerable effort. This would require splitting the Petri net, including its current marking, and transferring the resulting segments to the new simulators.

Moreover, the efficiency of the simulation is significantly influenced by the chosen distribution of the net. Suboptimal partitioning can lead to an imbalanced load distribution, causing individual simulators to become overloaded.

This contribution presents a concept for enabling the dynamic scaling of simulators — and thereby of the simulation itself. As a first step, the scalability of P/T net simulation is to be achieved by dynamically segmenting the net at runtime. In this process, net segments are split at transitions, and distributed synchronous communication channels are introduced for these transitions.

Subsequently, the dynamic scaling of simulators is addressed, incorporating autonomous, vertical, and horizontal scaling mechanisms. To this end, the utilization of the simulators is continuously monitored during operation. A dynamic resource management system adjusts the number of simulators based on the monitoring data collected. This dynamic scaling mechanism increases overall efficiency by automatically compensating for any initially unfavorable load distribution.

The proposed concept of dynamic scaling of simulators will be evaluated through the simulation of distributed P/T nets. The simulation components involved require a distributed execution environment.

In order to enable dynamic scaling, simulators must be capable of partitioning the simulated model — the P/T net — at runtime. They must also be able to transfer parts of it to other simulators.

Monitoring the utilization of the simulation components is essential for enabling automatic scaling based on workload. The Dynamic Resource Scaler (DyReS) autonomously performs vertical and horizontal scaling of the simulators based on the monitored metrics.

4. Distributed System

The distributed system consists of simulation components, a monitoring system, and the Dynamic Resource Scaler (DyReS). A fundamental requirement for the environment is the capability to scale the distributed simulation of P/T nets horizontally and vertically, driven by the current utilization of the simulators. Horizontal scalability requires dynamically instantiating additional simulators or nodes as needed and redistributing the P/T nets accordingly.

Continuous simulator and node utilization monitoring is essential to enable utilization-driven scaling decisions. The monitoring data shows that the DyReS autonomously manages horizontal and vertical scaling operations.

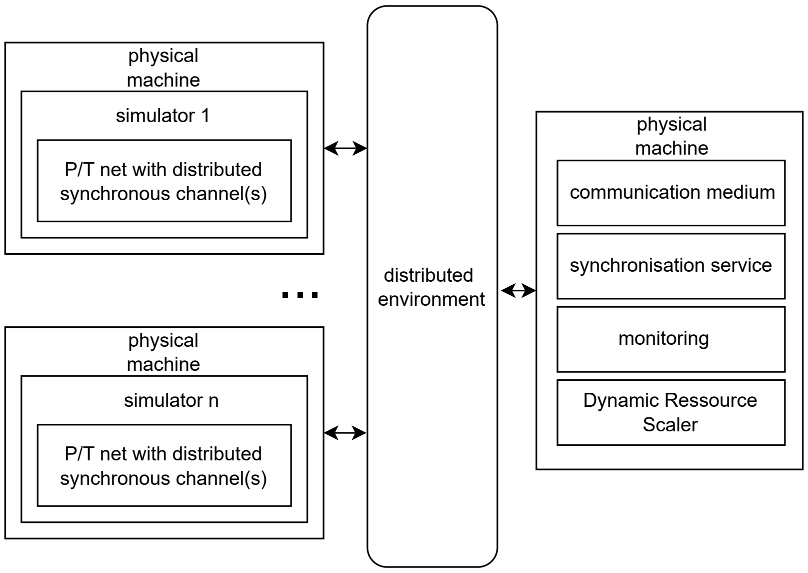


Figure 3: System for distributed P/T net simulations with dynamic scaling

The distributed environment is realized using the container orchestration platform Kubernetes [30] (Section 2.3). This approach requires that all system components be encapsulated as containers.

The simulation components are implemented based on the concept of distributed P/T nets (Section 2.2). The simulator Renew (Section 2.1) is particularly well suited for this task, as it not only enables the distributed execution of P/T nets but also provides a modular plugin architecture that facilitates straightforward extensibility.

Furthermore, Renew necessitates a dedicated synchronization service to coordinate the distributed simulation of P/T nets (Section 2.2). This service determines which distributed synchronous transitions may fire simultaneously and leverages Renew's unification algorithm for this coordination.

In addition, a new plugin is introduced to split a P/T net into multiple interconnected P/T nets. To this end, the NetSplit plugin (Section 6) is developed and presented in this work.

Following the partitioning, some resulting nets must be transferred to other simulators for execution. For this purpose, the NetExchange plugin (Section 5) facilitates the exchange of P/T nets on the simulator level.

An event-driven communication infrastructure (Section 2.2) is indispensable for the messaging between distributed simulators. Kafka is integrated into the Renew simulation framework and serves as the communication backbone to fulfill this requirement.

Effective monitoring in the distributed system must ensure that relevant metrics are available within seconds to enable dynamic and efficient scaling by the DyReS (Section 8).

5. Renew: NetExchange

In this section, the NetExchange plugin will be presented. It fulfills the feature we need within our simulators to send and receive running simulations of nets interchangeably.

5.1. Requirements

Fundamentally, this plugin exchanges running net simulations. This is needed to allow the NetSplit plugin to send away locally divided net parts and let Renew resume the simulation there. The act of loadbalancing nets across the cluster is not part of this plugin, as it mainly serves the purpose of being used by NetSplit.

It is required, that the network communication runs over KafkaRegistry since introducing an alternative way would result in an heterogenous setup. Also, the split itself should be initiated both by code and by the user, especially to verify the correctness of the exchange algorithm.

With regards to latency, it needs to be kept as low as possible. This is crucial since resource utilization spikes on a machine, that result in a net split and exchange action by a Renew simulator, can be short-timed, the split itself could become unnecessary with a high latency. With high latency, the cost of exchanging nets over the network would surpass momentary system overloads, thus making them bad. All in all, NetExchange needs to be implemented as a Renew plugin, to access loaded net objects and the KafkaRegistry plugin.

5.2. Specification

Sending nets across the network is done by serializing and putting them into special Kafka events designed for KafkaRegistry. To address certain Renew simulators that are part of our network, the KafkaRegistry identification mechanism can be used.

The exchange initiation by the user should be integrated by a GUI-Button and a command, to also allow Renew to receive direct commands through other network protocols with Renew's CloudNative plugin. The only computationally expensive operation when exchanging nets would be the serialization, since nets are complex objects with deeply structured dependencies. This needs to be done by modifying Renew's basic net type and cutting down the serialization depth.

5.3. Design

Sending nets through KafkaRegistry with its identification management was done by sharing handles to each other's machine in a short frequency. The handles are kept minimal to reduce the network overhead. Through the handles, one sends his net and the simulated net object. Those belong together and the running simulation builds upon the bare net object with regards to the on-screen rendering, which results in a two-stepped sending mechanism.

Sending the net and the simulated net object together does not cause any big network lag at all since objects use Java's *transitive* feature to exclude them from serialization. For example, nets in Renew keep references to the Renew simulator core objects, which reference a huge amount of Renew's ecosystem, so such simulator objects were made transitive and after receive, the own simulator would be injected.

The Renew plugin was designed to encapsulate code, that is only accessed by the plugin itself, within a custom module. This plugin infrastructure of two modules is coherent with the other plugins introduced in this paper.

5.4. Implementation

The NetExchange plugin was developed using Java 17, which is the Java version used in Renew. Gradle 8.4 was utilized for build and dependency management, which is also part of Renew's build eco system.

Git and GitLab were used, aswell as Renew's CI/CD pipeline to automate compilation, testing, packaging and other tasks belonging to Renew. With such steps, we assured code quality and correct functionalities.

5.5. Evaluation

Evaluation is simply done by triggering the net exchange with the button provided by Renew's GUI and the command. This step verified the correctness of the feature and initiated the integration of the NetExchange plugin into the NetSplit plugin.

6. Renew: NetSplit

This section provides a detailed description of the developed NetSplit plugin. Initially, the requirements that form the basis of the plugin's development are outlined in section 6.1. Following this, a comprehensive specification of its functionalities is provided in section 6.2. The design considerations and architectural choices that informed the development process are then outlined in section 6.3 and 6.4. In conclusion, the developed plugin is evaluated in section 6.5 according to the requirements established at the outset.

6.1. Requirements

In the case that a simulator reaches its load limits, it is necessary to relieve the simulator by scaling, which is elaborated in section 2.4. In this context, the ability to divide distributed P/T nets is paramount. The process of dividing a distributed P/T net enables the simulation of one simulator to be transferred to a new simulator through the NetExchange plugin describe in section 5. The function of the NetSplit plugin is to divide distributed P/T net of the simulator's simulations into the desired number of parts. The option to select an unlimited number of splits is important, as it introduces greater variability and flexibility to the scaling of the simulations.

It is imperative that the plugin ensures the resulting split distributed P/T nets continue to exhibit the same behaviour as the original. Failing this criterion would lead to a modification of the simulation and, consequently, the production of erroneous results, an outcome that must be prevented.

In addition, the plugin should operate in a highly efficient manner, ensuring that no unnecessary resources are consumed. This is to ensure that the simulation process is accelerated by the scaling, rather than hindered. To illustrate this point, it should be noted that slow splitting could render the scaling process redundant, as the process would have been completed in an equivalent timeframe or faster under these conditions. The selection of the algorithm for the division of distributed P/T nets is, therefore, of paramount importance. Nevertheless, it is imperative to emphasise the significance of practical feasibility in this context.

The integration of the NetSplit plugin within Renew (Section 2.1) is therefore pivotal in this regard. This inclusion allows for improved communication between its components, such as the NetExchange plugin (Section 5). It is essential to adhere to the existing specifications and standards of Renew. This approach facilitates seamless integration into the existing plugin landscape of Renew and ensures optimal maintainability.

6.2. Specification

In order to ensure the successful division of distributed P/T nets by the NetSplit plugin to scale a simulation, there are several factors that must be taken into consideration. Firstly, it is important that distributed P/T net can be splitted into a freely selectable number of parts. To this end, it is essential to ensure the availability of an integer parameter that can be utilised for this purpose.

Secondly, the reconnection of splitted distributed P/T nets via synchronous channels is imperative. This is to ensure that the new distributed P/T nets exhibit the same behaviour as the original, as described in the requirements (Section 6.1).

To facilitate this process, it is necessary to transfer existing tokens from the original distributed P/T net to the new distributed P/T nets. This process ensures the continuity of the simulation, enabling the resumption of the simulation at the precise point at which it was previously interrupted for the purpose of the split. This is a pivotal consideration to ensure the integrity of the simulation and the preservation of data.

The fourth point pertains to the transfer of inscriptions. All inscriptions that are supported by distributed P/T nets [5] must be transferred. These are weighted arcs, but above all synchronous channels which are only located at transitions in distributed P/T nets, which is why we split transitions. The transfer of inscriptions is of paramount importance, as otherwise the behaviour of the shared distributed P/T nets no longer corresponds to that of the original.

Finally, it is imperative to adopt the connected text of places and transitions. While this does not directly influence the behaviour of the distributed P/T nets, it does facilitate understanding and ensure the sustainability of the division. This approach ensures that, even following multiple splits, the precise identification of transitions or places within each new distributed P/T net remains attainable.

6.3. Design

The new NetSplit plugin is being developed as a Renew plugin. As a consequence of the modular structure of the former [9, 10], the plugin is developed in a separate module. This is necessary to fulfil the current development standards of Renew and thus enables direct integration.

The Contraction algorithm, which forms part of the Karger-Stein algorithm [31], was utilised as the fundamental algorithm for the division of distributed P/T nets by the NetSplit plugin. The algorithm functions on the premise of Randomised Contraction Algorithms, which entail the random selection of an edge and the subsequent merging of the two end nodes, accompanied by the requisite update of the edges. This process is repeated iteratively until only two nodes remain. The edges that persist at this stage are designated as the intersecting edges.

It is imperative to acknowledge that the contraction algorithm only ever splits an undirected graph into two parts. This challenge can be circumvented by running the algorithm several times on the graph with the most elements and more than one node representing a place, as illustrated in Algorithm 1. This approach guarantees that the graph is divided into any number of parts. Nevertheless, it is important to note the limitation that this approach does not guarantee that the resulting subgraphs will have equivalent sizes. This constitutes a further rationale for opting for the Contractions algorithm of the Karger-Stein algorithm. Consequently, in instances where the inequality of the components is unduly pronounced in practice, the Karger-Stein algorithm can be implemented with greater facility in retrospect.

Furthermore, the algorithm exhibits a favourable runtime complexity of $O(n^2)$[31]. In order to establish the runtime, it is also necessary to run the algorithm k times, where k is the number of divisions. This results in an increased runtime of $O(kn^2)$. While the efficacy of the algorithm is not optimal, given the NP-hard nature of the minimum cut problem, the randomised algorithm provides highly satisfactory results in practice and meets the requirements (Section 6.1).

However, it is important to note that the Contraction algorithm always cuts edges, not nodes that represent places and transitions. This leads to synchronization errors, as the graph is split between a place node and a transition node. The root cause of this issue can be attributed to the incompatibility of places and transitions with synchronization in distributed P/T nets. Consequently, the implementation of split distributed P/T nets necessitates the incorporation of transitions subsequent to the execution of the algorithm. The strategic positioning of these transitions, either before or after places, is pivotal in ensuring the creation of a transitions-bordered distributed P/T net. This is achieved with a runtime of $O(m \cdot n \cdot p)$, where m is the number of edges splited, n is the number of nets created and p is the number of places. This approach facilitates the subsequent synchronisation of the distributed P/T nets. This is achieved by the artificial division of only transitions and not the edges themselves.

Given that the input of the Contractions algorithm is an undirected graph, it follows that an existing distributed P/T net must also be regarded as an undirected graph. In this case, transitions and places are regarded as nodes. Existing arcs are regarded as edges, but without direction. This straightforward approach facilitates the preprocessing and postprocessing of distributed P/T nets. This approach also facilitates the adoption of existing labelling and tokens. The inverse transformation is equally straightforward. Nodes and edges remain linked to their original objects. It can thus be concluded that the runtime of these two processes is O(p).

It can thus be demonstrated that the entirety of the algorithm presented in Algorithm 1 achieves a runtime of $O(k \cdot n^2) + O(m \cdot n \cdot p)$. It is important to note that the number of cuts, k, is not significant, nor are the number of resulting nets, n, or the number of split edges, m. The only factor that can be moderately large is the number of places p.

Algorithm 1: Splitting Algorithm

```
   Input: distributed P/T net, number of parts
   Output: distributed P/T nets
1  graph = distributedP/TNetToGraph(petriNet);
2  result = List();
3  while result.size() < numberOfParts do
4      biggest = result.getBiggestGraph();
5      splittedGraph = contractionAlgorithm(biggest);
6      result.add(splittedGraph);
7      result.remove(biggest);
8  addBoundaryTransitions(result);
9  addSynchronousChannels(result);
10 return result;
```

6.4. Implementation

The NetSplit plugin was developed using Java 17 [6], which corresponds to the Java version of Renew (Section 2.1). Gradle 8.4 [7] is utilised for the management of dependencies, the process of modularisation, and the reproducibility of the build.

The integration of the NetSplit plugin into Renew facilitates the utilisation of existing plugins and their associated functions. This integration enables direct communication via defined interfaces and using existing data types. To illustrate this point, consider the data type employed by the NetSplit plugin for distributed P/T nets, which is consistent with that utilised by the NetExchange plugin

(Section 5).

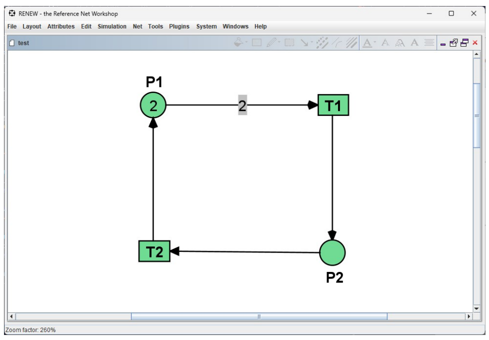
The behaviour of the plugin was tested using unit tests. JUnit [32] version 5.9.0 was utilised as the foundational framework, with Mockito version 4.8.0 [33] employed for the purpose of mocking classes.

The administration and management of versions was facilitated by the utilisation of Git [34] and GitLab [35]. These tools have been utilised in the field of software development for an extended period, and their efficacy in fulfilling the requisite tasks has been well-documented.

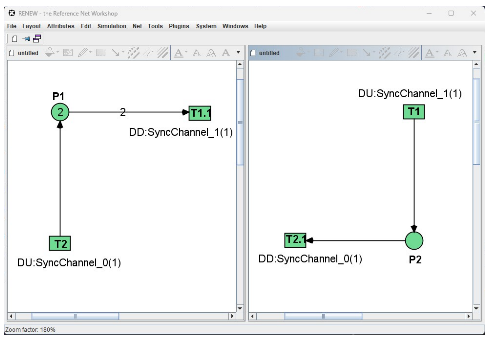
In this context, a particular emphasis was placed on the utilisation of branches and merge requests. Subsequent to this, a peer review process is undertaken, after which the merge requests are approved by other developers. Additionally, a continuous integration/continuous deployment (CI/CD) pipeline was implemented to automate the construction of the project and the execution of unit tests, including integration tests. This ensures the functionality of both the NetSplit plugin and Renew.

6.5. Evaluation

The integration of the NetSplit plugin into Renew has been accomplished with success, as demonstrated in Figure 4. This integration was achieved by employing the same principles as the Java versions, or by leveraging existing concepts such as modularisation. However, the focus on practical feasibility also contributed to this.


RENEW - the Reference Net Workshop
File Layout Attributes Edit Simulation Net Tools Plugins System Windows Help
test
P1
2
2
T1
T2
P2
Zoom factor: 260%

(a) Undivided distributed P/T net



(b) Distributed P/T nets splitted into two parts

Figure 4: Distributed P/T nets in Renew. Once in the original and once in two parts

The plugin has been demonstrated to meet the requirement of dividing a distributed P/T net into multiple sections. The implementation of Karger's algorithm, in conjunction with the creation of custom data types for undirected graphs, facilitates a process of transformation and decomposition.

The developed, customised version of the contraction algorithm of the Karger-Stein algorithm achieves a runtime of $O(kn^2)$. It is demonstrated that this efficient runtime can be achieved by repeatedly executing the contraction algorithm, which has a runtime of $O(n^2)$, when splitting a net. The procedure known as 'splitting' is executed with a runtime of $O(k \cdot n^2) + O(m \cdot n \cdot p)$. This runtime fulfils the requirements for efficient division, as the factors k, m and n are very small on average and only p may be larger.

The added transitions and synchronous channels allow the new parts of the distributed P/T nets to communicate. This is a crucial aspect for ensuring that the new transitions exhibit equivalent behaviour to the original distributed P/T net. Furthermore, the possibility to directly take over inscriptions and tokens, facilitated by references in the objects of the nodes and edges, is a crucial aspect in ensuring the same behaviour.

The developed NetSplit plugin has been shown to fulfil all the requirements (Section 6.1) and specifications (Section 6.2) that were defined at the beginning of this section. As demonstrated in Figure 4, a distributed P/T net is depicted before and after splitting, illustrating the effectiveness of the process. This assertion is not limited to small distributed P/T nets but extends to more complex ones as well.

7. Monitoring: Utilization

This section provides a comprehensive report on the monitoring and observation strategies considered for the Dynamic Resource Scaler (DyReS) and the system implemented.

First, it needs to be established why there is a need for monitoring and observation. This and the requirements regarding the monitoring and observation system are outlined in the Requirements (Section 7.1). In the following Specification (Section 7.2), there is a description of the choices made to fulfill the requirements. After that, a description of how the specified system is structured is presented in the Design Section (Section 7.3). The Implementation (Section 7.4) details how the metrics are retrieved and how states of alertness are generated for the prototype. The chosen implementation is then assessed in the Evaluation (Section 7.5).

7.1. Requirements

For the DyReS to achieve accuracy in making complex autoscaling decisions based on arbitrary simulations with changing resources (Section 2.4), there needs to be precise monitoring and observation regarding resource utilization. Therefore, the implementation must monitor cluster-wide computational resource availability and utilization associated with horizontally or vertically scalable components.

Delays between a simulation or simulator eligible for scaling and DyReS being able to make a decision should be minimized to further enhance the accuracy of scaling decisions. Hence, the monitoring should provide up-to-date metrics that are mostly concurrent with actual utilization, and the observation should occur with as little delay as possible. For this requirement, a worst-case time for a monitoring implementation can be determined both theoretically and experimentally.

Monitoring, alerting, and observation components are not part of the computation that yields results; therefore, the consumption of computational resources by monitoring components should be minimized. This is to ensure that monitoring, observation, alerting, and subsequent scaling allow the simulation to use more computational resources for acceleration, thereby improving the effectiveness of scaling decisions.

To enhance longevity, the implementation should utilize standard technologies and be open to changes. This ensures that the system can be used in future prototyping work and eventually beyond the prototyping stage.

7.2. Specification

To ensure monitoring and observation regarding resource utilization by horizontally and vertically scalable components, the system needs to be deployed in close communication with the Kubernetes environment used for scaling the simulation. Therefore, monitoring, observation, and alerting system components are deployed in the cluster used for scaling, and metrics are collectable about the Nodes and Pods components of the cluster. This also adheres to the established requirement to minimize delays, as it is generally assumed that delays increase with more communicative hops in the system.

To avoid the problem of providing data points that may not accurately depict resource utilization outside the measurement interval, the monitoring implementation provides summed and averaged data. This approach is assumed to offer a more accurate assessment of utilization by acknowledging potential computational spikes between measurements.

The system components are chosen to be lightweight, providing low additional resource utilization for the monitored components. This fulfills the requirement for low additional resource consumption.

For future relevance, open-source projects are utilized. These publicly version-controlled and community-maintained projects can be modified and specialized if necessary. Furthermore, these projects are mostly already readily used, and therefore it is assumed that past work in the domain of this paper can be better utilized. Even though a fully self-maintained monitoring project may fulfill other requirements more closely, it is not a priority of this work to create and maintain such monitoring components from the ground up. However, the system does contain a self-maintained observation project as part of the DyReS implementation, ensuring the ability to establish new observation strategies. Further closed-source projects may be relevant but lack the ability to be modified for potential specialization. This aims to fulfill the requirement for the longevity of the implementation.

7.3. Design

For the prototype, two prominent designs were primarily considered: the Metrics Server approach and the Prometheus approach. The first and implemented Metrics Server design for the monitoring and observation system is comprised of two components. First, the Kubernetes cluster-deployed Metrics Server provides the Kubernetes Metrics-API. Second, the Java-implemented metric handler, as part of the DyReS project, collects metrics from the endpoint of the Kubernetes Metrics-API and additionally implements basic threshold observation on these metrics. The process of observation is therefore delegated to be implemented by the DyReS project, and the need for alerting is circumvented.

This is a pull-based approach that merges the complexity of observation with the process of making a scaling decision in DyReS. All relevant metrics need to be retrieved and processed by the metric handler in a definable scrape interval, after which some decision logic needs to be applied. Initially, it was assumed that such a design would not suffice for achieving a low resource footprint in the DyReS, and a push-based approach was considered. This second design approach includes more than two components:

- Kubernetes exposing pod metrics for a Prometheus system to integrate.
- Node Exporters for exposing node metrics (Node CPU usage, RAM, Network, and Disk load) for a Prometheus system to integrate.
- Prometheus Server to scrape and store exposed metrics at specified intervals.
- Alertmanager to apply alerting rules to collected metrics and push alerts to DyReS.
- DyReS to receive alerts defined by the Alertmanager and make scaling decisions based on them.

It was assumed that the advantage of such a design would lie in leaving the decision process to DyReS and the observation process to the Alertmanager. Therefore, the load of both processes could be independent. However, experimentation showed that the Alertmanager did not provide the needed speed for the use case. In combination with the multiple components that needed to cooperate and communicate, there were delays of up to 5 minutes between the load rising and DyReS being able to react. This contravenes the requirement to minimize delay, which is why this approach was abandoned in favor of the Metrics Server system and a pull-based approach.

Additionally, this design does not allow for purely value-based decisions based on metrics, as alerts are quantized by the Alertmanager. An example of a DyReS behavior that would not be possible with this design but would be possible with the Metrics Server design is an up-to-date dynamic determination of the number of simulators based on the total CPU consumed.

7.4. Implementation

The implementation encompasses the provisioning and access control of the Kubernetes Metrics-API within the cluster context for scaling, as well as the Java implementation of the metric handler, including the associated observation as part of the decision-making process.

Metrics Server is deployed in the cluster via the helm chart provided by the project's GitHub repository [29] and provides metrics through the Kubernetes Metrics-API. Additionally, access to these metrics via the Kubernetes Metrics-API hosted by the Kubernetes API Server must be granted to the DyReS containerized deployment. This is achieved through a Kubernetes ServiceAccount and Role-Based Access Control (RBAC).

The information to access the ServiceAccount is injected as a file into the DyReS deployment. It is used for communicating with the Kubernetes API Server to create new pods, detect nodes, and retrieve metrics about node usage. Extraction of metrics from the Kubernetes Metrics-API is performed by the metric handler, which is implemented as part of DyReS using Java 17 with Gradle 8.4, as specified in the DyReS Implementation (Section 8.4). The API communication is facilitated by the Kubernetes API Client for Java [36].

For both implementations, Git [34] and GitLab [35] were utilized to version control the progress. A shared and virtualized Kubernetes cluster was used for testing and iterating over the necessary deployments.

7.5. Evaluation

The implementation provides access to the following metrics:

- Per Node and Pod CPU usage in milliCPU and RAM usage in bytes.
- Per Node CPU and RAM availability in percent.

However, this does not allow for more complex decisions based on other metrics, such as network and disk load. This system is therefore unable to scale simulations that primarily use such resources in an observed manner.

The Metrics Server implementation is able to provide metrics averaged over 15-second intervals [29]. DyReS is therefore enabled to start the process of making scaling decisions every 15 seconds and is guaranteed to work with data that is, in the worst case, 15 seconds old.

It has been observed that the Metrics Server is a lightweight monitoring application. The DyReS has been noted to be a heavier application due to the constant observation process that is running. Further experimentation needs to be conducted on the prototype to gather meaningful precise timing and resource usage data over longer periods of time.

8. Dynamic Resource Scaler

8.1. Requirements

The DyReS (Dynamic Resource Scaler)[5] enables dynamic resource scaling in distributed P/T-net simulations while adhering to strict functional and architectural constraints. The system's design is governed by seven core principles to ensure seamless interoperability with Renew while preserving simulation behavior integrity.

Central to this approach is the continuous monitoring of cluster-wide resource utilization metrics (Section 7), including CPU, memory, and disk usage in real time. These metrics are aggregated through a distributed monitoring framework and algorithmically analyzed to detect threshold breaches, which autonomously trigger scaling operations. The Decider module prioritizes vertical scaling when local Pod/Node resources are available, employing a configurable aggressiveness factor (AF) ranging 0–100% to determine scaling intensity. Through the Kubernetes Java Client API, this module dynamically adjusts CPU and memory allocations for simulator instances, proportionally scaling resources to minimize network overhead from horizontal scaling.

Network partitioning is achieved via the NetSplit plugin, which divides simulations into $n + 1$ logically isolated subnets ($n > 0$) without compromising structural integrity (Section 6.3). Synchronized communication between these subnets is facilitated by the NetExchange plugin using distributed channels, implementing Renew's architectural specifications for inter-subnet coordination (Section 5). This dual mechanism enables transparent migration of partitioned nets across heterogeneous pods/nodes during horizontal scaling operations, preserving both original network behavior and temporal

[5]The acronym DyReS was chosen to maintain consistency with established naming conventions in distributed systems literature, where "dynamic" typically precedes the controlled entity in scaling components.

consistency. Concurrently, vertical scaling adjusts computational resources within individual pods while maintaining strict real-time simulation constraints.

To balance autonomic control with operational flexibility, the system exposes Kubernetes API-compliant endpoints for manual intervention. While adhering to cloud-native versioning standards and temporarily overriding autonomic decisions, enabling controlled simulation redistribution during development and production phases.

8.2. Specification

The Dynamic Scaler is implemented as a standalone Java application interfacing with Kubernetes via the official Java client library (Section 2.3). It is composed of three modules: the Decider, the Horizontal Scaler, and the Vertical Scaler. The Decider uses a rule-based algorithm with a configurable AF (0–100%) to determine the scaling intensity. It ingests data from the Kubernetes Metrics server to minimize architectural complexity and reduce latency.

When utilization thresholds are breached, the Decider prioritizes pod-local vertical scaling, minimizing network overhead. If vertical scaling is not possible, horizontal scaling provides new pods or nodes by invoking NetSplit to split the simulation into $n + 1$ subnets. The Decider coordinates the migration of the subnet to new instances by invoking NetExchange after the Horizontal Scaler created the pods. Manual scaling is enabled through DyReS endpoints.

8.3. Design

The system is integrated into a GitLab CI pipeline and containerized using a multi-stage Docker build. Containers provide isolated runtime environments, while Docker is a widely used platform for building, packaging, and distributing containerized applications. The Decider's architecture follows three phases:

1. **Observing**: The Decider module observes threshold breaches in monitored metrics.
2. **Deciding**: The Aggressiveness Factor (AF) modulates scaling intensity through weighted resource utilization. The Decider prioritizes node-local vertical scaling when resources are available; horizontal scaling splits simulations into $n + 1$ subnets via NetSplit when cluster capacity requires expansion.
3. **Executing**: The Horizontal Scaler provisions new nodes by waking them through LAN and new pods through Kubernetes orchestration, while the Vertical Scaler adjusts CPU/memory allocations via API calls. NetSplit partitions simulations into subnets while maintaining net behavior, with NetExchange handling state migration through synchronized channels.

The DyReS operates as a stand-alone Java application decoupled from Renew, and does not impose any dependencies on external components unless otherwise configured. The decoupling from Renew will enable a general-purpose solution. The Java implementation will use the Kubernetes Client API for orchestration tasks. The DyReS avoids Kubernetes-native autoscaling tools (e.g., HPA/VPA) because they rely on generic resource metrics and fixed thresholds, which are insufficient for the dynamic demands of parametric simulations. These tools lack the contextual awareness to distinguish transient spikes from sustained load. Their rigid policies do not allow fine-tuned adjustments needed in simulation environments.

8.4. Implementation

The DyReS implementation builds upon the monitoring infrastructure using Java 17 with Gradle 7.6 and Kubernetes Java Client 22.0.0. The Vertical Scaler module dynamically adjusts CPU and memory allocations for existing simulator pods through Kubernetes resource quota updates, constrained by Java's static heap allocation limitations. This implementation choice necessitated careful memory profiling to avoid out-of-memory errors during vertical scaling operations. For horizontal scaling, the

Horizontal Scaler interacts directly with the Kubernetes API to provision new nodes or deploy new pods.

Manual scaling functionality is built in. All components are packaged as a multi-stage Docker image through GitLab CI, with dependency injection used to maintain decoupling from Renew's core architecture. Notable challenges included mitigating Java's garbage collection pauses during metric aggregation cycles. This was resolved through off-heap memory caching and ensuring same subnet behavior during concurrent migrations, which was addressed through version-stamped transition synchronization.

8.5. Evaluation

The evaluation of DyReS demonstrates its effectiveness in dynamically scaling distributed P/T-net simulations while maintaining simulation integrity and minimizing resource overhead. The system was tested under varying workload scenarios to assess its scalability, efficiency, and adaptability.

Scalability tests revealed that DyReS efficiently handled both vertical and horizontal scaling. Vertical scaling dynamically adjusted CPU and memory allocations, reducing resource contention without requiring additional pods. Horizontal scaling, enabled by NetSplit and NetExchange, ensured seamless migration of subnets to new nodes while preserving temporal consistency. These mechanisms allowed the system to adapt to workload changes with minimal disruption.

The Kubernetes Metrics Server reduced monitoring overhead by filtering irrelevant data. This optimization ensured that DyReS operated effectively even under high workloads, maintaining a low computational footprint. In comparison to Kubernetes-native autoscaling tools, DyReS demonstrated superior contextual awareness, enabling precise scaling decisions tailored to the dynamic demands of parametric simulations. This capability highlights its suitability for environments requiring fine-grained control over resource allocation.

In conclusion, DyReS proved to be a robust and efficient solution for dynamic resource scaling in distributed P/T-net simulations. Its modular design and reliable scaling mechanisms make it well-suited for scientific and cloud computing applications. Future work will focus on optimizing scaling algorithms, scaling in the other direction, scaling other nets, creating a test environment and addressing identified challenges to further enhance the system's performance and applicability.

9. Discussion

The concept introduced herein exhibits a significant advantage in that the initial distribution of P/T nets can be specifically tailored to the characteristics and capabilities of the target simulators. This adaptability enables the compensation of non-optimal initial allocations, enhancing the overall simulation efficiency.

Furthermore, the system supports an automated adjustment of the distribution at runtime, which dynamically responds to the prevailing computational load on the individual simulators and the underlying hardware nodes. This contributes to improved resource utilization and scalable performance.

The presented concept mitigates internal system constraints typically imposed by monolithic simulator architectures by enabling a distributed and scalable simulation of P/T nets. These constraints include, but are not limited to, the maximum number of tokens, concurrently active transitions, or threads.

In addition, simulator nodes' horizontal and vertical scaling facilitates circumventing the physical limitations of single-machine environments, such as restricted processor availability or limited main memory capacity. This architectural flexibility allows for the simulation of significantly larger and more complex models than would otherwise be feasible.

One disadvantage of the current implementation concerns the NetSplit plugin, which, in its current form, exclusively performs a feasibility analysis of the net partitioning. Consequently, neither the runtime behavior is subject to optimization nor is a balanced output of subnets guaranteed. Both aspects—runtime optimization and load-balanced output generation constitute active areas of ongoing research and development and are essential for achieving high efficiency in large-scale deployments.

A further disadvantage of the presented system lies in the considerable complexity of the required execution infrastructure. The implementation necessitates deploying and managing multiple physical nodes orchestrated via containerized environments such as Kubernetes. Moreover, additional components such as monitoring frameworks, the DyReS, and an advanced, modular simulation system are indispensable to support the described scaling strategies. These dependencies impose substantial requirements on the system's technical setup and operational maintenance.

A fundamental limitation of the current approach is the absence of mechanisms for scaling in (i.e., reduction in parallelism) or scaling down (i.e., reallocation to less powerful hardware) during simulation runtime. At present, the presented concept exclusively supports scale-up and scale-out strategies. The conceptual and technical foundations required to enable dynamic scale-in and scale-down operations are currently under investigation and represent a core objective of future research.

10. Related Work

Our work focuses on the Petri net simulator and editor known as Renew (Section 2.1). This tool provides a ready-to-use simulation environment alongside a modular plugin system that facilitates extensibility. Consequently, there is no need to develop a new simulator from scratch. It offers a comprehensive feature set to support diverse workflows in Petri net research and applications [9, 10].

The platform accommodates various Petri net formalisms and is designed for straightforward expansion through its modular plugin architecture [8]. Its main net formalism, the Reference Net, extends conventional colored Petri nets by incorporating nets-within-nets concepts combined with reference semantics, thereby enabling the integration and execution of Java code [11]. Additional supported formalisms include P/T-nets with channels, as discussed by [37], and the recently introduced Distributed P/T nets (DPTN) [5].

Research by Moldt et al. [38] and Röwekamp et al. [39, 40, 41, 42, 43, 12, 44] concentrates on distributed Reference Net simulations, with an emphasis on platform management. Their work integrates Mulan agent concepts [45] and utilizes Spring Boot to facilitate initial experimental implementations. Although these contributions provide essential groundwork, they fall short of fully exploiting distributed systems' potential, thereby constraining their suitability for addressing complex, real-world application scenarios.

As part of our project responsibilities, we oversee the complete deployment and orchestration of the Renew simulator. This enables comprehensive testing of newly developed functionalities within a cluster environment composed of multiple interconnected nodes, ensuring system reliability before full-scale implementation.

This contribution builds upon the work of Clasen et al. [5], extending the simulators developed therein by introducing a form of dynamic scalability through the proposed conceptual enhancements. In contrast, Clasen et al. [46] concentrate not on scalability but rather on resilience. Their approach is based on the principle of periodic state preservation, enabling simulators to maintain fallback states in the event of failures. This mechanism allows a simulator to be redeployed on an alternative node following, for instance, a node failure and subsequently restored to its prior operational state.

11. Conclusion

11.1. Summary

After outlining the foundational concepts — including Renew, Distributed P/T Nets, Kubernetes, scalability, as well as monitoring and alerting (Section 2) — we define the research problem addressed in this work (Section 3). Specifically, we propose a dynamic scaling approach for the distributed simulation of P/T nets within a Kubernetes-based cloud environment.

The distributed system, detailed in Section 4, comprises a communication medium, monitoring infrastructure, DyReS, and simulation components for distributed P/T nets. Subsequently, we present the developed prototypes. One of these is the newly introduced Renew plugin NetExchange (Section 5), which enables the exchange of P/T nets between simulators at runtime. Additionally, the new NetSplit plugin (Section 6) facilitates the partitioning of P/T nets during simulation.

Two further prototypes presented in this article are the utilization monitoring system (Section 7) and the DyReS component (Section 8). The Kubernetes Metrics Server enables real-time node and pod utilization monitoring by providing up-to-date metrics within seconds. DyReS leverages these metrics to perform dynamic scaling of the simulators.

Finally, in Section 9, we revisit the research objectives and questions, evaluate the strengths, weaknesses, and limitations of the proposed concept, and situate it within the context of related work (Section 10).

11.2. Future Work

Further research includes the implementation of dynamic downscaling and scaling mechanisms. This aims to enable simulators or individual nodes to be shut down during simulation runtime when their resources are no longer required.

In this context, optimizing the NetSplit plugin is also a key objective. Specifically, improvements in runtime efficiency and load-balanced output generation must be addressed.

Moreover, the primary focus of this article was on demonstrating practical feasibility. Future research should include comprehensive benchmarking of the system's performance. Additionally, it is possible to integrate further heuristics into DyReS.

This enhancement would allow DyReS to be more demand-driven, optimizing resource utilization or system performance as needed.

Finally, integration with the Petri Net Registry is planned. The Petri Net Registry provides highly available cloud storage for network templates. This integration would enable the traceable storage of network templates generated during scaling operations.

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.
- …DeeplWrite in oder to: Rephrasing.
- …ChatGPT in order to: Rephrasing, Sentence Polishing.
- …Mistals Le Chat in order to Sentence Polishing.
- …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. Budde, K. Kautz, K. Kuhlenkamp, H. Züllighoven, What is prototyping?, Information Technology & People 6 (1990) 89–95.

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

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

[4] 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.

[5] 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.

[6] Oracle, Java SE 17 Documentation, 2025. URL: https://docs.oracle.com/en/java/javase/17/, accessed: Mach 24, 2025.

[7] Gradle, Gradle User Guide Version 8.4, 2023. URL: https://docs.gradle.org/8.4/userguide/userguide.html, accessed: March 24, 2025.

[8] 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=.

[9] 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.

[10] 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.

[11] O. Kummer, Referenznetze, Logos Verlag, Berlin, 2002. URL: http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=0035&lng=eng&id=.

[12] 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.

[13] 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.

[14] 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.

[15] A. S. Foundation, Apache Kafka Documentation, 2025. URL: https://kafka.apache.org/documentation/, accessed: 2025-01-21.

[16] N. Garg, Apache Kafka, Packt Publishing Birmingham, UK, 2013.

[17] A. Baur, Packaging of kubernetes applications, in: Proceedings of the 2020 OMI Seminars (PROMIS 2020), volume 1, Universität Ulm, 2021, pp. 1–1.

[18] E. Casalicchio, Container orchestration: A survey, Systems Modeling: Methodologies and Tools

(2019) 221–235.
[19] Y. Pan, I. Chen, F. Brasileiro, G. Jayaputera, R. Sinnott, A performance comparison of cloud-based container orchestration tools, in: 2019 IEEE International Conference on Big Knowledge (ICBK), IEEE, 2019, pp. 191–198.
[20] A. De Cerqueira Leite Duboc, A framework for the characterization and analysis of software systems scalability, Ph.D. thesis, UCL (University College London), 2010.
[21] D. R. Law, Scalable means more than more: a unifying definition of simulation scalability, in: 1998 Winter Simulation Conference. Proceedings (Cat. No. 98CH36274), volume 1, IEEE, 1998, pp. 781–788.
[22] M. R. López, J. Spillner, Towards quantifiable boundaries for elastic horizontal scaling of microservices, in: Companion Proceedings of the10th International Conference on Utility and Cloud Computing, 2017, pp. 35–40.
[23] Q. Jiang, Y. C. Lee, A. Y. Zomaya, The limit of horizontal scaling in public clouds, ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS) 5 (2020) 1–22.
[24] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, N. Zeldovich, An analysis of linux scalability to many cores, in: 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10), 2010, pp. 1–16.
[25] B. Ahmad, Coordinating vertical and horizontal scaling for achieving differentiated QoS, Master's thesis, University of Oslo (UiO), 2016.
[26] P. Authors, Prometheus Documentation, 2025. URL: https://prometheus.io/docs/introduction/overview/, accessed: March 26, 2025.
[27] J. Bastos, P. Araújo, Hands-on infrastructure monitoring with prometheus: implement and scale queries, dashboards, and alerting across machines and containers, Packt Publishing, 2019. URL: https://portal.igpublish.com/iglibrary/search/PACKT0005309.html.
[28] Kubernetes Community, metrics, https://github.com/kubernetes/metrics, 2025. Accessed: March 26, 2025.
[29] Kubernetes Instrumentation Special Interest Group, metrics-server, https://kubernetes-sigs.github.io/metrics-server/, 2025. Accessed: March 26, 2025.
[30] The Kubernetes Authors, Kubernetes, 2025. URL: https://kubernetes.io/docs.
[31] D. R. Karger, C. Stein, A new approach to the minimum cut problem, Journal of the ACM (JACM) 43 (1996) 601–640.
[32] JUnit, JUnit 5 User Guide, 2025. URL: https://junit.org/junit5/docs/current/user-guide/, accessed: March 24, 2025.
[33] Mockito, Mockito Core 4.8.0 Javadoc, 2022. URL: https://javadoc.io/doc/org.mockito/mockito-core/4.8.0/index.html, accessed: March 24, 2025.
[34] G. Community, Git Documentation, 2025. URL: https://git-scm.com/doc, accessed: March 24, 2025.
[35] G. Inc., GitLab Documentation, 2025. URL: https://docs.gitlab.com/, accessed: March 24, 2025.
[36] Kubernetes Community, client-java-api, https://mvnrepository.com/artifact/io.kubernetes/client-java-api/22.0.0, 2023. Version 22.0.0.
[37] 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.
[38] 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.
[39] 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/.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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=.

[46] L. Clasen, P. Leonhardt, L. Wichelmann, Resilient Distributed P/T Net Simulators, 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.