Distributed P/T Net Simulation Prototypes Based on Event Streaming⋆

Laif-Oke Clasen[1,*], Sophie Bartelt[1], Yannik Stahl[1] and Daniel Moldt[1]

[1]University of Hamburg, Department Informatik, Vogt-Kölln-Str. 30, 22523 Hamburg, Germany

Abstract

Effective and goal oriented development of complex systems relies heavily on models, which serve as essential tools. Simulation and verification of these models enable in-depth analysis of system components and validation of complex system characteristics. This capability significantly influences the advancement of system development processes. Petri nets represent a technique for modeling such complex systems.

Traditionally, simulation and verification of complex systems occur on a single computing platform, which inherently imposes limitations of computation power. Recognizing the potential benefits of distributed computing, this paper explores the concept of distributed simulation of P/T nets. Methodologically, prototypes are employed and rigorously evaluated within this paper.

The contribution of this paper is a prototypical implementation of a compiler for simulating Distributed P/T Nets for the Petrinetzsimulator and -editor Renew. Furthermore, it explains the design of the distributed execution environment and demonstrates the feasibility of distributed simulation through a classical example in computer science.

This contribution represents a further step towards the distributed simulation of Petri nets. However, this is an exploratory prototype of a compiler, whereby the associated formalism does not yet exist. This realization distinguishes itself from other implementations through central unification and event-based communication.

Keywords

P/T-nets, Synchronous Channels, P/T-nets with Synchronous Channels, Event Streaming, Distributed Simulation

1. Introduction

The modeling and analysis of complex systems is an important part of modern computer science. A widely used formal modeling technique in this area are Petri nets, whose strengths lie in their flexibility, expressive power, concurrency and formal definition. The latter in particular ensures their ability to enable both the simulation and verification of real-world systems. This makes Petri nets a valuable tool in computer science and systems theory.

Despite their many possible applications and advantages, the simulation and verification of Petri nets often reach their limits. A reason for this is that most formalisms are designed to be executed locally on a computer. To circumvent this limitation, simulation and verification can be implemented in a distributed way. Distribution may offer better performance [1] by using multiple processes for one simulation that act asynchronously in parallel [2]. However, this requires a new formalism that addresses the particular challenges of distributed simulation and verification.

Explicit communication between Petri nets based on transitions has been explored and introduced in the form of synchronous channels [3, 4], but they are mostly used in high-level Petri net formalisms [4, 5, 6]. However, Place/Transition (P/T) nets in [7] were also equipped with synchronous channels. Compared to high-level Petri nets, such as colored Petri nets [8, 9], these are a simple variant of Petri nets.

This contribution focusses on the distributed simulation of P/T nets with the event-streaming platform Kafka. The synchronous channels within this contribution are used as the starting point by distributing the synchronous channels themselves. Thus, the central question of this paper concentrates on the

PNSE'24: International Workshop on Petri Nets and Software Engineering, June 24–25, 2024, Geneva, Switzerland

⋆Supported by several colleagues and students.

*Corresponding author.

laif-oke.clasen@uni-hamburg.de (L. Clasen)

© 2024 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

CEUR-WS.org/Vol-3730/paper12.pdf

distributed execution of synchronous channels in the context of P/T nets. This paper provides a prototypical implementation of a compiler for the simulation of DISTRIBUTED P/T NETS, which are not formalised now. This Formalisation is part of ongoing work.

To achieve this goal, the methodology of prototyping is applied. This practical approach allows the implementation of theoretical considerations directly in functioning software and thus evaluate the feasibility and efficiency of the proposed formalism. By combining theoretical foundation and practical application, this contribution represents an important step in the research on the simulation of distributed Petri nets.

For this purpose, the basics of place/transition nets, synchronous channels, place/transition nets with synchronous channels, workflow transitions, distributed simulations, RENEW, Kafka and AGENT INTERACTION PROTOCOL DIAGRAMS (AIPS) are covered in the section 2. This is followed by the objectives in the section 3. There is also a description of the structural elements of the distributed system in section 4. Three prototypes follow, building upon the previous sections. The first prototype in section 5 implements a first initially marked circuit of the distributed synchronous channel. An update of the globally known marking is only considered in the second prototype in section 6. The last prototype in section 7 deals with several up- and downlinks of a distributed synchronous channel, as well as several distributed synchronous channels and several net instances. Finally, the results are discussed (section 8), delimited in the same way as related work (section 9)) and a conclusion is drawn (section 10).

2. Foundations

2.1. Place/Transition Nets

Place/Transition (P/T) nets originate from Carl Adam Petri. [10] This kind of nets is sometimes also referred as Petri nets. [11]

Definition 1. *Based on [12, 13] a Place-/Transition Net, short P/T-Net is a 5-Tuple $N = (P, T, F, W, m_0)$ where*

- *P is a finite set of places,*
- *T is a finite set of transitions,*
- *$F \subseteq (P \times T) \cup (T \times P)$ is its flow relation,*
- *$W : F \to \mathbb{N}_0$ are its arc weights and*
- *$m_0 : P \to \mathbb{N}_0$ is its initial marking.*

To describe the state of the system, tokens are used that are situated on places. This state is described by marking m. The number of tokens situated on one place p is defined as $m(p)$. The pre-set of a net element $x \in P \cup T$ is according to [11]:

$$\bullet x = y \in P \cup T : (x, y) \in F.$$

Symmetrically to the pre-set every transition has a post-set that is defined by the flow relation.

A transition t is declared enabled and can fire with a marking m if all places in the pre-set of t contain at least the amount of tokens declared by the arc weight of the arcs that connect the two net elements. This is defined by [12] in the following way:

$$\forall p \in \bullet t : m(p) \geq W(p, t).$$

If a transition t fires, it removes the tokens that are necessary for the firing of t from its places in the pre-set. After that it generates tokens to the places in the post-set. The amount of tokens generated by

a transition is defined by the arc weights that can be found on the arcs connecting the two elements. The firing of a transition t_0 changes its marking, thus the state of the net.

In Figure 1 a P/T net with two transitions and two places is displayed. The transition on the right consumes one token and produces two, while the transition on the left does the opposite. The place on the left holds one token, thus enabling the transition on the right.

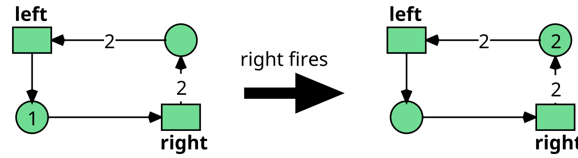


Figure 1: A Place/Transition Net

2.2. Synchronous Channels

A synchronous channel in the context of Petri nets ensures that Petri nets can communicate with each other by implementing rendezvous synchronization. Transitions are inscribed with signatures so that they can synchronize if they have the appropriate 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 *!?* and *?!* in [4] or as *uplinks* and *downlinks* in [6]. 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.

2.3. Place/Transition Nets with synchronous Channels

One approach of Place/Transition Net with synchronous Channels (PTC net) is proposed by [7] and is formally defined as followed.

Definition 2. *A Place-/Transition net with synchronous channels, short PTC-net is described by a 8-Tuple* $N = (P, T, F, W_{sync}, m_0, Var, Ch, CE)$ *where*

- P *is a finite set of places,*
- T *is a finite set of transitions,*
- $F \subseteq (P \times T) \cup (T \times P)$ *is its flow relation,*
- $W_{sync} : F \to \mathbb{N}_0$ *are its arc weights and*
- $m_0 : P \to \mathbb{N}_0$ *is its initial marking,*
- Var *is a set of channel variables,*
- Ch *is a set of channels and*
- $CE : T \to (type, ch, X)$ *is a channel expression function containing*
  - $type \in uplink, downlink$ *a type,*
  - $ch \in Ch$ *a channel and*
  - X *a tuple of channel variables and/or positive integer that may be empty*

The channel expression of t_1 and t_2 can either have no parameter, one or two parameters, that can as well be integers. If a channel expression contains more than one parameter, the order of these variables gets important for the binding search between the transitions. A parameter in the i'th place of the Tuple $T(i)$ of transition t_1 will bind with the i'th element of the Tuple of transition t_2.

If there is a variable on an arc connected with a transition belonging to a distributed synchronous channel it needs to appear in a second place. This variable must be in the tuple defined by the channel expression function. A formal definition is given by [7]:

$$\forall t \in T : \forall v \in \{W_{sync}(p,t) | p \in P\} \cup \{W_{sync}(t,p) | p \in P\} : (v \in V \Rightarrow v \in X)$$

Definition 3. *Two Transitions t_1 and t_2 match, according to [7] meaning they can be synchronized iff*

- *one transition is an uplink and the other one is a downlink of the same channel,*
- *the number of channel expression parameters are the same and*
- *any parameter that is a variable will be connected with the appropriate parameter that is an integer from the channel expression of the transition that is to be synchronized.*

Definition 4. *As stated by [7] a downlink t_1 is enabled with a marking m if*

- *There exists an uplink t_2 with the same channel name and*
- *the marking of the cumulated pre-set is able to satisfy the needed tokens defined by $W^*_{sync}(p, t_1, t_2)$ and*
- *t_1 and t_2 match.*

The definition for the uplink being enabled follows trivially by replacing downlink with uplink and vice versa.

Definition 5. *The number of tokens on each place p needed for firing transitions t_1 and t_2 synchronously is described in [7] by*

$$W^*_{sync}(p,t_1,t_2) := \begin{cases} W_{sync}(p,t_1) + W_{sync}(p,t_2) & W_{sync}(p,t_1) \in \mathbb{N} \wedge W_{sync}(p,t_2) \in \mathbb{N} \\ X(i) + W_{sync}(p,t_1) & W_{sync}(p,t_1) \in \mathbb{N} \wedge W_{sync}(p,t_2) \in Var \\ & \wedge i \in \mathbb{N} : Y(i) = W_{sync}(p,t_2) \\ Y(i) + W_{sync}(p,t_2) & W_{sync}(p,t_1) \in Var \wedge W_{sync}(p,t_2) \in \mathbb{N} \\ & \wedge i \in \mathbb{N} : X(i) = W_{sync}(p,t_1) \\ X(i) + Y(j) & W_{sync}(p,t_1) \in Var \wedge W_{sync}(p,t_2) \in Var \\ & \wedge i \in \mathbb{N} : Y(i) = W_{sync}(p,t_2) \\ & \wedge j \in \mathbb{N} : X(j) = W_{sync}(p,t_1) \end{cases}$$

Analogously the amount of produced tokens by a synchronized transition on each place p is described by

$$W^*_{sync}(t_1,t_2,p) := \begin{cases} W_{sync}(t_1,p) + W_{sync}(t_2,p) & W_{sync}(t_1,p) \in \mathbb{N} \wedge W_{sync}(t_2,p) \in \mathbb{N} \\ X(i) + W_{sync}(t_1,p) & W_{sync}(t_1,p) \in \mathbb{N} \wedge W_{sync}(t_2,p) \in Var \\ & \wedge i \in \mathbb{N} : Y(i) = W_{sync}(t_2,p) \\ Y(i) + W_{sync}(t_2,p) & W_{sync}(t_1,p) \in Var \wedge W_{sync}(t_2,p) \in \mathbb{N} \\ & \wedge i \in \mathbb{N} : X(i) = W_{sync}(t_1,p) \\ X(i) + Y(j) & W_{sync}(t_1,p) \in Var \wedge W_{sync}(t_2,p) \in Var \\ & \wedge i \in \mathbb{N} : Y(i) = W_{sync}(t_2,p) \\ & \wedge j \in \mathbb{N} : X(j) = W_{sync}(t_1,p) \end{cases}$$

In Figure 2 a PTC net with one synchronous channel is presented. Thus the transition with the inscription *this:s()* is forced to fire with the transition having the inscription *:s()*. In this relationship the transition on the left is the downlink and the transition on the right is the uplink.

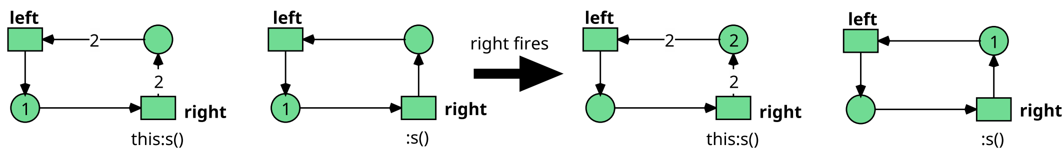


Figure 2: Firing of Place/Transition Net with synchronous Channel

2.4. Workflow Transitions

As referenced in [14], a workflow or process resembles a series of steps designed to achieve a company's objectives. It begins with at least one event and culminates in generating one or more specific outcomes. Workflows can be modelled with Petri nets. [15, 16, 17] Crucial for the modeling of a workflow is the possibility to undo operations as many workflow languages allow this kind of action. [15]

To allow the reset of actions in workflows resettable transitions are used in [17]. These transitions allow a reset by substituting one transition by multiple transitions that handle the reset to default of the marking.

In Figure 3 transition $T1$ is made resettable by its substitution in transition $T1.1$ that requests the firing, transition $T1.2$ that may reset the firing and transition $T1.3$ that may complete the firing. This concept originates from [17].

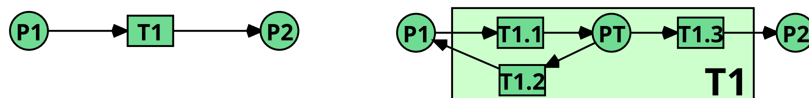


Figure 3: Turning a common transition into a resettable transition (based on [17])

2.5. Distributed Simulation

The simulation of systems is a tool used to analyze such systems. A simulation is discrete if the state of the simulation changes at discrete points in time [2]. A discrete event simulation is one where state changes are triggered by specific events [18, 2].

Distributed simulations are simulations that are executed with the help of multiple processes, which run asynchronously and in parallel. Additionally, these processes mostly communicate via messages [2]. The components of a distributed simulation are typically connected by networks [19].

There are multiple reasons to use distributed simulation. Firstly, it is possible to use information for one simulation that is geographically distributed [19]. Secondly, it is possible to integrate multiple different simulators, each with different foundations [19]. Lastly, distributed simulation may offer better performance for the simulation [1].

2.6. Renew

Renew[1] [20] is an extensible modeling, execution and analysis environment for Petri nets. For this purpose, the open source tool Renew offers various formalisms and modeling techniques [21]. Renew itself is written in the Java programming language. The architecture is based on a plugin system [22], which has been modularized with the current version based on the Java Platform Module System.

The best-known Petri net formalism in Renew is the reference net formalism according to Kummer [6]. This formalism combines the concepts of nets in nets according to Valk [23] with reference semantics and the expressive power of object-oriented programming in the form of Java as a labeling language. In addition to this, Renew offers a variety of formalisms, including a P/T net formalism (see section 2.1) and P/T nets with synchronous channels (see section 2.3).

2.7. Kafka

Apache Kafka is a platform developed for event streaming [24]. Event streaming is a method for processing, transferring and storing data. Great scalability, low latency, high availability as well as high data throughput count to its main advantages [24]. Contrary to conventional message queues [25], messages do not get deleted after being received in Kafka.

Kafka is usually running in form of a cluster of server nodes, the event brokers. These brokers are the central contact point for any communication running through Kafka. They receive and hold messages so that they can be later received.

These messages are called events. They contain a key value tuple which can be of different formats like Json [26].

Events are always assigned to a topic, which are a form of organizing and storing events. A topic can be divided in partitions to allow spreading out the topic over multiple brokers. Events are then distributed evenly among the partitions while still ensuring that events carrying the same key will always land in the same partition.

A producer can publish events on these topics. Consumers who are subscribed to the topic the event was produced on can then receive those messaged by requesting them at the broker.

2.8. Agent Interaction Protocol Diagrams

Agent Interaction Protocol Diagrams (Aips), allow to the control flow by showing the interactions between parts of the systems or different participants for specific use cases. [27, 28] Agent Interaction Protocol Diagrams are defined by the Foundation for Intelligent Physical Agents (Fipa). [29] As agents are predestined to be used in a distributed context [30] they evoke concurrency and parallelism.

Aips extend the sequence diagrams of UML [31] by additional alternatives and concurrency. [32] Agent Interaction Protocol Diagrams can be splited at various points vertically and horizontally and result in different system behavior models. It is possible to

- split with AND to get parallelism and
- to split with an inclusive OR, gaining parallelism as well or
- to split with an exclusive OR, gaining concurrency.

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

Splitting may need a merge or synchronization in the following execution. Figure 4 represents an abstracts Aip. Further examples of Aips are visualized, for instance, in Figure 6 and Figure 7.

Furthermore, a formal semantics for Aips based on Petri nets, was proposed in [33], by which an operational semantics based on partial order semantics is given. The partial order semantics support the design of complex distributed systems in a natural way.

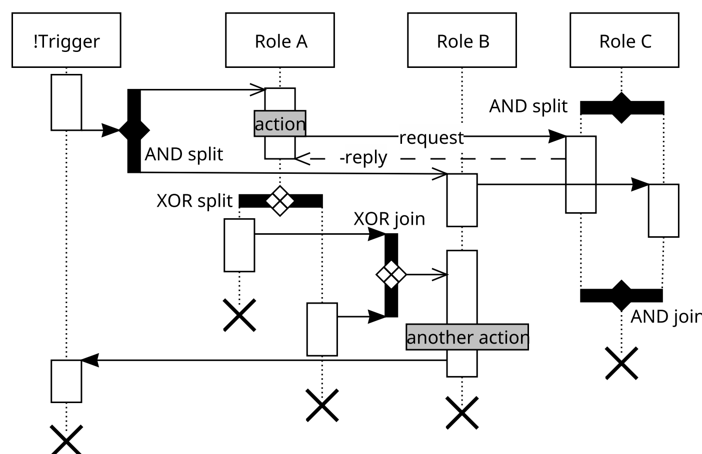


Figure 4: Example of an abstract Agent Interaction Protocol Diagram

3. Objectives

The main goal of the prototypes for Distributed P/T Nets is to provide modeling extensions to enable larger models and faster simulation and verification through the distribution. The models are based on the P/T nets with synchronous channels [7], whereby synchronization has now been implemented using event-based communication. A formalization of the models can be found at [7].

The results of this paper serve as a basis for further advanced concepts. These include the use of Kubernetes and its infrastructure as an execution environment. In addition, Renew will be expanded in the long term to include another important conceptual and technical option.

For the realization of the protoypes, synchronous channels should be able to be executed in a distributed manner. The following sub-goals can be derived from this:

- There should be a prototype implementation.
- Since this is a distributed execution across physical boundaries, a distributed execution environment is required.
- Furthermore, a classical example from computer science should be used as a proof of concept.

One of these sub-goals is to provide a prototype for Distributed P/T Nets. Another sub-goal is the provision of an execution environment, caused by the new distributed synchronous channels that are introduced into the formalism. These distributed synchronous channels should enable communication between P/T nets on different computers. Therefore a distributed execution environment with at least four computers is required: downlink simulator, uplink simulator, synchronization service and communication medium. The last sub-goal of this article addresses the proof of concept. Here a classic example from computer science should be used.

An evaluation against other methods and the application of the results to more complex use cases are planned for further work. The focus here is on investigating the feasibility in principle.

Related Concept	Brief Explanation
Distributed P/T Net	A distributed P/T net with synchronous and distributed synchronous channels
DSC	A distributed synchronous channel
Event Streaming	Method to process, transfer and store data
Workflow Transitions	Realization concept for distributed synchronous channels
Artificial Transition	A transition that is formed in the Synchronizations Service
Synchronization Service	Performs a centralized unification on the basis of aritificial transitions

Table 1
Brief explanations of related concepts

4. Distributed System

This section describes the distributed system resulting from the formalism presented in this paper. The distributed system consists of at least four computers, with at least two computers each running a simulator, one computer running the communication medium and one computer running the synchronization service. The computers are connected via a network.

The core idea is that based on the events of the distributed channel, an artificial net is created in the synchronization service when a possible unification shall be tested. This net contains an artificial transition representing one up- and downlink. For both transitions the pre-set is built exactly as transmitted in the respective events.

The synchronization service then checks whether the distributed synchronous channel can fire and informs the up- and downlink. The exact process within the respective prototype is specified in the form of a protocol, which is modeled as Aip (section 2.8).

Each simulator contains a Distributed P/T Net, whereby at least one of the nets contains a uplink and at least one contains a downlink. There can also be several downlinks and/or uplinks in one network. A corresponding visualization can be found in Figure 5. Additionally, an overview of the associated concepts can be found in table 1.

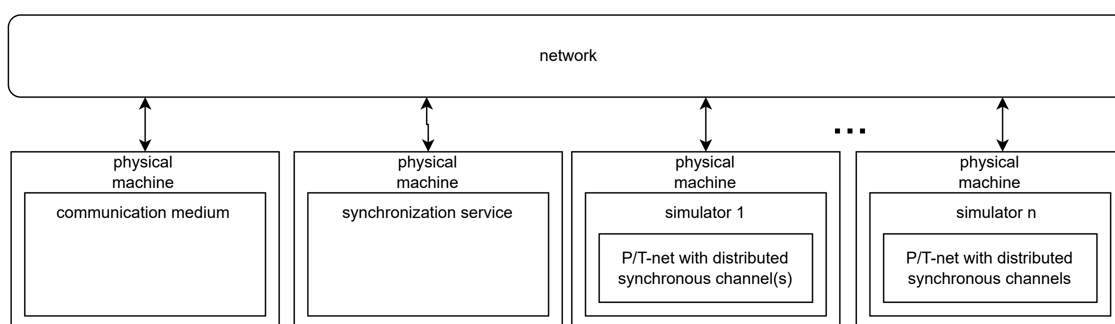


Figure 5: Model of distributed Simulation

A distributed synchronous channel (DSC) consists of at least one up- and downlink and is the main concept of the prototypes presented here. Here, DSCs are given a new inscription syntax by denoting a downlink with “DD” and an uplink with “DU”. This is intended to ensure differentiation from local synchronous channels. An example of a DSC is visualized in Figure 9. Distributed synchronous channels or synchronous channels can be understood as interfaces of (distributed) P/T nets.

With distributed synchronous channels (DSCs), the distribution of the whole simulated net follows. These DSCs can be understood as interfaces of the sub nets within the different simulators. Therefore its marking becomes obscur. In the following sections we refer to a local marking as the marking that exists in one net of one simulator. In contrast, we refer to the global marking as the union of all markings of all nets that belong to one simulation. An overview of the associated concepts can be found in the table 1.

5. Firing of one Distributed Synchronous Channel

5.1. Requirements

This is the first prototype describes a variant of Distributed P/T Nets. According to the prototyping methodology used here, a first simple version is realized, which is also the basis for the requirements. To initate a simulation of a Distributed P/T Net we need at least two simulators, where each simulator is responsible for a part of the whole Distributed P/T Net. For the first prototype we want exactly two simulators for the sake of simplicity.

Req. 1 There should be exactly two simulators.

The communication between these net parts is established through synchonrous channels, more specifically DSCs. To enable a communication connection between these two simulators we need at least one DSC. For the sake of simplicity we assume exactly one DSC. Whereby one of the simulators contains a downlink and the other simulator contains an uplink. In a P/T net, synchronous channels can have several down- and uplinks. Nevertheless, we further restrict the DSCs by only allowing a 1:1 mapping from downlink to uplink for this prototype.

Req. 2 There should be exactly one DSC.

Req. 3 The DSC should consist of exactly one downlink and one uplink.

Therefore the synchronization service handles the distributed unification of variables and the orchestration of a firing. The synchronization service is situated according to Figure 8 outside of the two simulators. For this first prototype, we establish that the synchronization service exists exactly once. This means that the synchronization service in this prototype is not scalable.

Req. 4 There should be exactly one instance of the synchronization service.

Besides the simulators and the synchronization service we need as well a communication medium and the possibility to synchronize the downlink and uplink belonging to a DSC. The event broker Kafka (section 2.7) is used as the communication medium and the network address is well-known to the simulators and the synchronization service. Therefore, all components communicate via a Kafka topic, which must be known to everyone. Following the prototyping methodology, we initially only have one instance of the communication medium. The main disadvantage of this is that the events are not highly available. This means that the previous events are no longer available if the communication medium fails.

Req. 5 There should be exactly one instance of the communication medium.

Req. 6 The network address of the communication medium should be well-known to the simulators and to the synchonization service.

Req. 7 The topic used for the communication between the participants should be well-known.

For the transitions of a DSC to fire in this prototype, they must be activated by the initial marking. The reason for this is that this prototype only realizes the firing. Therefore, changes to the pre-set of a DSC would not be recognized by this prototype. For these changes to be recognized, an update mechanism is required, which is addressed in the next prototype (see section 6).

Req. 8 The initial marking enables the transitions of the distributed channel at least once.

Req. 9 The transitions of the DSC should be able to fire at least once.

To satisfy the objectives 3, a classic example of computer science is chosen as a proof of concept. For the sake of simplicity and to match the prototype, we assume that this proof of concept is of minimal scale, therefore there should only be one synchronous channel.

Req. 10 A classic example of computer science should be chosen as a proof of concept.

Req. 11 There should be exactly one DSC in the proof of concept.

5.2. Design

5.2.1. Distributed Simulation

The simulation of the first prototype consists out of three phases, that are the initialization, the actual simulation and the end of the simulation. This structure is common for discrete event simulation. [18]

The first phase includes the setup of all components and the registration of the simulators at the synchronization service. The second phase is defined by the simulation of the execution of the DSC, while the thrid phase will deregister the simulators from the synchronization service and shut down the simulation.

The simulation is a distributed discrete event simulation (see section 2.5) as the state of the simulation changes whenever a common transition in a net fires or events are sent at a discrete point od time. Additionally, the simulation is distributed as the nets lay in different simulators that each stand for an own logical process of the simulation.

5.2.2. Distributed Synchronous Channel

The DSC is the main concept of the prototypes presented here. According to the requirement *Req. 2* and *Req. 3*, this prototype consists of exactly one DSC featuring exactly one downlink as well as its counterpart the uplink. Other mappings are not possible in this prototype.

5.2.3. Distributed Components

In this section, we look at the design of the specific components. The requirements *Req. 1*, *Req. 4* and *Req. 5* state that there should be a communication medium, two simulators and a synchronization service. Each of these are independent components that can implemented with different technologies.

The communication medium is realized by using Apache Kafka (see section 2.7). This is due to the key benefits of Kafka, as it is a widely used and supported open-source technology designed for distribution. It offers a very high scalability as well as options to ensure high availability. It is also easy to setup and highly configureable. Therefore it suits the requirements of this implementation very well. In addition, there is the option for Kafka to be made highly available. This ensures that no messages are lost, which is essential for maintaining a consistent state in the simulation.

In contrast, the simulators and the synchronization service are implemented as plugins in Renew (see section 2.6), which is easily possible thanks to its modular plugin architecture. This allows us to build on the existing editor and simulator, which is one of the reasons why its implementation in Renew is a natural choice. The decision to implement this feature with multiple plugins helps to capsulate the feature from other features. This ensures, for example, that you can draw without simulating and simulate without drawing.

No special network implementation is selected for the first prototype. However, the physical machines hosting the different components need to be in the same network to enable communication between them without additional effort. The network address of the communication medium will be well-known to the simulators as *Req. 6* clarifies. The minimal setup of the network allows us to focus on the main functionalities and concepts of the prototype. In future work, the distributed components are to be delivered within a cluster, which will then take over the technical realization of the network.

5.2.4. Protocols

This section discusses the protocols between the components for firing a DSC and for registration of up- and downlinks. These protocols define distributed contracts between the roles involved, which all must comply with. The registration protocol is presented in Figure 6 and the firing protocol in Figure 7.

Since this protocol is distributed, it is modeled as an Aip (see section 2.8). The encircled numbers in the two mentioned Figures have no modeling significance, but help to visualize certain steps of the protocol in more detail below.

Before the protocol for firing can start, the uplink and downlink must be registered, which is achieved by the registration protocol. The registration phase is important as it structures the simulation and enables the central synhcronization service to gain an overview of the whole simulation. The registration protocol is triggered by the start of the simulation.

For the first prototype we assume exactly one up- and downlink, that are located in different simulators. Both of which need to be started independently which is represented by the arrows shown in the registration protocol in Figure 6 in the first step. In the second step, both the up- as well as the downlink forming the DSC send their respective registration event via Kafka.

In the third and final step the synchronization service fetches all new events, receiving the two registration events sent by the up- and downlink. The synchronizations service uses these events to extract structural information about the DSC, these are needed as prerequisites for the firing protocol.

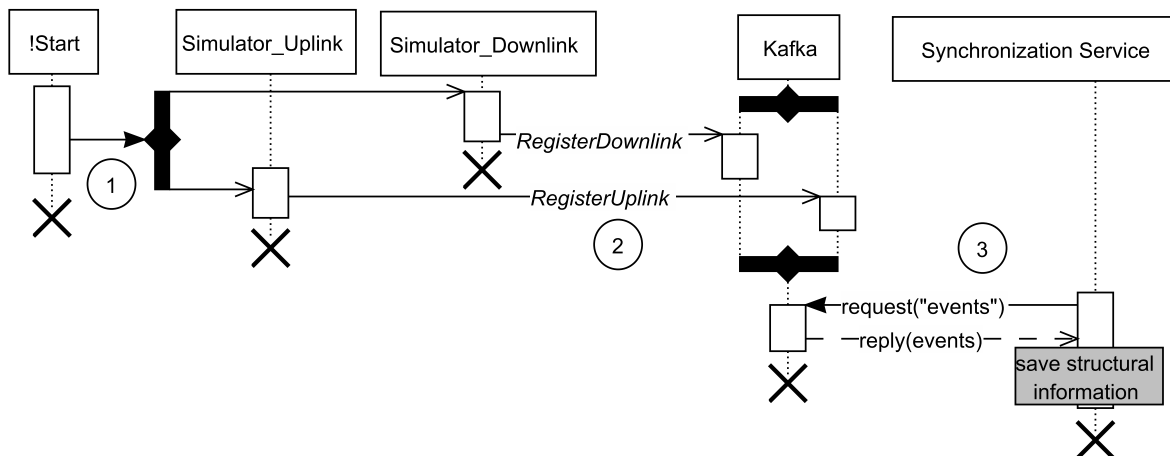


Figure 6: Protocol for registration of up- and downlink

In the first Prototype the registration protocol immediately starts the firing protocol, that is visualized in Figure 7. The instant start of the firing protocol is necessary as we want to keep the first protocol as simple as possible. The design of another event would consider multiple additional design decisions and more event traffic that is not necessarily needed. In the further prototypes, the trigger for the firing process is therefore the storage of the structural information of the uplink and downlink, which can be seen in step one.

In the second step, the synchronization service uses the structural information of the up- and downlink to generate an artificial transition locally. The pre-set of the artificial transition is the union of the pre-sets of the up- and downlink.

Following, an attempt is made to unify the artificial transition within the synchronization service. A corresponding unification is required for this. If the artificial transition is unified successfully and if the transition is enabled, the synchronization service sends the event “ConfirmFireable” with the corresponding bindings via Kafka to the Simulators.

After this, the up- as well as the downlink can successfully fetch this new event in step three. If the up- and downlink have been informed about the possible binding of the circuit, it must be evaluated locally whether this binding is still possible in step four. The reason for this again lies in the distributed

system, as the local markings are constantly changing due to firing of local transitions which happens significantly faster than the firing of a distributed channel. This is why the needed tokens may already be consumed by a local transition. The respective responses of the up- and downlink are then transmitted via Kafka as an event, whereby the markings are blocked locally if the response is positive.

In the fifth step, the previous step of the respective counterpart is evaluated. If both provide a positive result, firing of the channel is confirmed and finalized in step six. On the other hand, if there is at least one negative response from the up- or downlink, a rollback occurs at the transition that sent the confirm event. The rollback releases the reserved tokens into the net by putting them back on their initial places. The process is completed with a firing or a rollback.

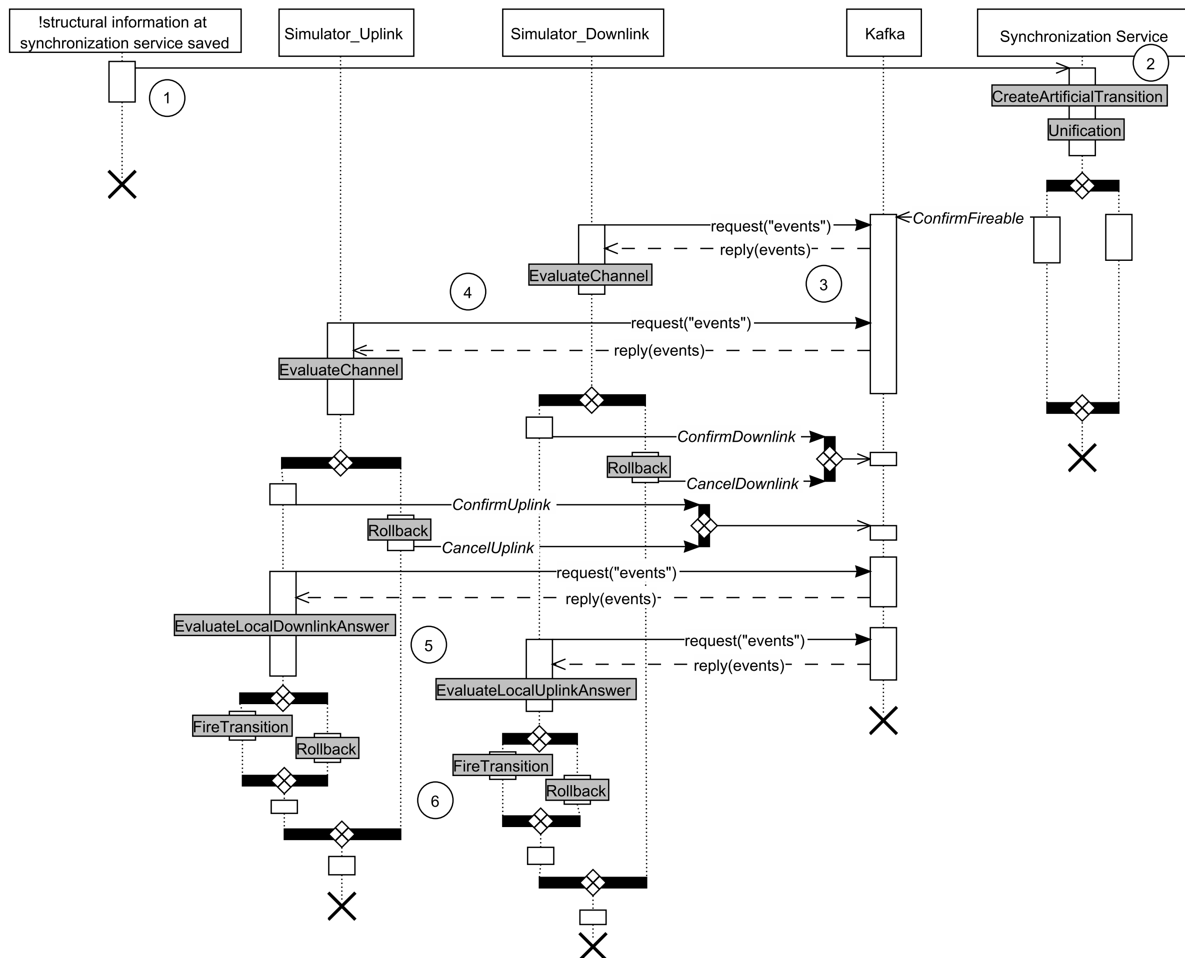


Figure 7: Protocol for firing a distributed synchronous channel

5.2.5. Events

This section addresses the design of the Kafka events exchanged between the components. Based on the registration protocol and firing protocol (see section 5.2.4), the required events for this prototype can be identified. A complete overview of the corresponding events can be found in table 2.

For global identification, the net name is required for each event. No further parameters are required for the events *DeregisterDownlink*, *DeregisterUplink*, *ConfirmDownlink*, *CancelDownlink*, *ConfirmUplink* and *CancelUplink*, as these only exchange boolean values and this is already encoded in the keys of the events.

Based on the two events *RegisterUplink* and *RegisterDownlink*, the synchronization service should be able to save all the structural information that are necesssary to build the artificial transition. For

Event	Parameters
RegisterUplink / *RegisterDownlink*	Parameter, Pre-set, Post-set, Parameter/Arc-Mapping, Marking of Pre-set
DeregisterUplink / *DeregisterDownlink*	
ConfirmFireable	Netname of downlink, Netname of uplink, Bound Variables of uplink, Bound Variables of downlink
ConfirmDownlink / *CancelDownlink*	
ConfirmUplink / *CancelUplink*	

Table 2
Design of Events for Registration and Firing

this, the synchronization service must know the complete pre-set and of the DSC. This requires both structural information, i.e. which places are connected to the transition and how exactly they are connected described by arcs and their respective inscriptions, as well as information on the current status, i.e. how the places are marked in the pre-set.

The *ConfirmFireable* event has the task of transmitting the possible binding derived by the synchronization service to the downlink and uplink. The values that were assigned to the variables of the up- and downlink are required for this.

5.2.6. Proof of Concept

For the proof of concept we chose the producer consumer example. This example is a classic example of computer science. There are numerous variants of the example, e.g. with or without storage. The example is intended to serve as a proof of concept within this work, following requirement *Req. 10*.

The variant used is a simple variant with one producer and one consumer and without storage. According to the requirement *Req. 2*, there is exactly one DSC connecting the producer and the consumer in this example. In addition, this synchronous channel consists of exactly one downlink and one uplink, as required in request *Req. 3*. For this purpose, the variant of the example used in this paper is modeled in Figure 9 as Distributed P/T Nets.

5.3. Implementation

5.3.1. Simulator

The implementation of the simulator was realized in the form of multiple Renew plugins. Namely the DPTNFormalism plugin, the DPTNDrawing plugin, the DPTNOntology plugin, the KafkaOntology plugin and the KafkaClient plugin.

The DPTNDrawing plugin provides a new file type with the ending ".drnw" to separate normal nets from distributed nets. Additionally it allows the already mentioned new inscription for transitions that belongs to a DSC.

The KafkaClient plugin is a specific interface for Renew to interact with a Kafka broker to communicate. Functionalities like the creation of topics, registration to topics and consuming of messages are implemented here.

To enable the provision of common types, the KafkaOntology plugin and the DPTNOntology plugin accumulate this common information. The KafkaOntology plugin provides information how to be a subscriber of a specific topic and what a KafkaEvent is. In contrast the DPTNOntology plugin provides functionalities to retrieve information that shall be encoded in the events and to create events specific to the distributed simulation.

The DPTNFormalism plugin defines a new compiler for nets in Renew that include transitions belonging to a DSC. The name of the plugin includes "Formalism" as compilers in Renew normally build up on Formalisms. Most importantly for the implemented compiler is that its formalization is still in the process of creation in ongoing work. The DPTNFormalism plugin has two major functionalities. At first it compiles given nets and their new special inscriptions that symbolize DSCs. Additionally, the plugin handles the reaction to incoming events. The DPTNFormalism Plugin uses the KafkaClient

plugin for distributed synchronization and delegates local synchronizations to the PTC Formalism plugin that is already implemented in Renew. This plugin as well uses both ontology plugins. The new compiler is used whenever new inscriptions are made, the simulation is started, or the syntax is checked.

The decision to implement the Simulator with multiple plugins serves the goal of forming encapsulated and easily extensible software with good quality. Especially the encapsulation of the KafkaClient is important as this plugin can stay the only one with dependencies to external software.

In the step of compilation, every distributed synchronous transition is translated into three transitions. These transitions are a “request transition”, “confirm transition”, and a “cancel transition”. In this case we use the concept of workflow transitions as explained in section 2.4. In contrast to its purpose to simulate a workflow our approach concentrates on the possiblity to reserve tokens.

The translation of one transition into three transitions inscribed with an uplink allows reserving tokens while allowing rollback in case of failed synchronization. Each of these uplinks is called whenever the matching event is consumed assuming its further conditions are also fulfilled. The translation is graphically displayed in Figure 8.

The concept of three transitions to construct a resettability is crucial as transitions are not commonly resettable. The usage of three transitions instead of one to create resettability uses the concept of workflow transitions (see section 2.4). It make the approach understandable and easier to verify in future work.

When starting a simulation the compilation of the transitions in a net is triggered. The compilation of a Distributed P/T Net results in the creation of the respective register event for the transition belonging to a DSC. This event is then produced using the functionalities of the KafkaClient plugin.

In the following paragraph the behaviour of a transition belonging to a DSC is explained, refering to Figure 8. If a simulator consumes the event “ConfirmFireable” the uplink of transition $U1$ is called with the specified bindings from the event and the transition tries to fire.

If this is successful the simulator sends the event “ConfirmUplink” if the transition is an uplink or “ConfirmDownlink” if the transition is a downlink. After this step, the place in between $U1$ and $U2$ holds x tokens. If the firing was not possible, the simulator produces the event “CancelUplink” or “CancelDownlink” depending on its type and consequently there would be no token on the place in the middle.

If a transition of a DSC receives a cancel event, $U3$ must fire. The firing of the DSC is no longer possible as at least one partner could not fire its $U1$ transition. On the contrary, after receiving a confirm event, $U2$ is fired and the distributed firing is completed.

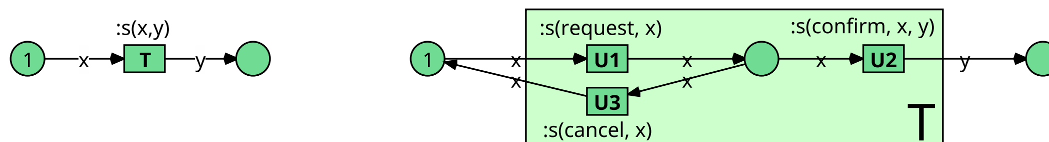


Figure 8: Model of translation of one DSC into three uplinks (based on [17])

5.3.2. Communication Medium

Our chosen communication medium Kafka offers some configuration options. Relevant for this prototype is that exactly one instance of the broker exists, a 1:1 mapping between partitions and topics, and that the address of the broker is well-known among all other components. The 1:1 mapping between partitions and topics ensures that the events are not split between different partitions and participants can receive all events. Because this prototype focuses on the firing of DSCs, only one instance of the broker is required. This means that the Kafka Broker is not highly available and therefore data loss may

occur during events. The high availability and robustness of the system is an integral part of future work.

5.3.3. Synchronization Service

The synchronization service, implemented through the new Renew plugin DPTNSyncService, handles the creation and execution of the before described artificial transition. Analog to the simulators this plugin also uses the KafkaClient Plugin, KafkaOntology Plugin as well as the DPTNOntology Plugin to send, receive and process events.

Firstly, it is important to give transitions belonging to a DSC the possiblity of registering to this service so that possible bindings among running simulators can be found. This is done by evaluating the events “RegisterUplink” and “RegisterDownlink” and storing their information on their respective transitions like the structure of their pre-set an in Figure 6.

This information is then further processed in a unification process where a compound net is build, that contains exactly one up- and downlink. Such a compound net represents a potential firing that shall be examined. An example for this process can be seen below, where the relevant information of the registered nets in 9 is used to create the artificial net in Figure 10.

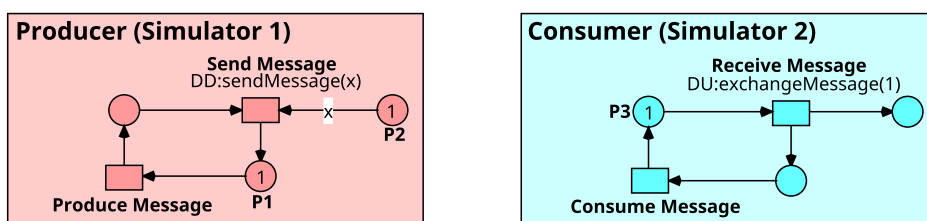


Figure 9: Producer Consumer Example of this Prototype (proof of concept)

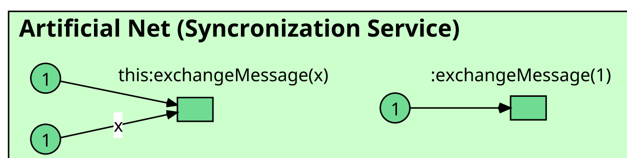


Figure 10: The artificial net that would be unified at the synchronization service

This net is then evaluated for a possible firing. If this is the case the synchronization service produces the event “ConfirmFirable”. This event carries the information the assigned values of their parameters.

5.4. Evaluation

It is trivial to see that the requirements set out in section 5.1 are met by the prototype presented here. However, there are various limitations that limit the proposed prototype. One main limitations regarding the size are given by *Req. 2* and *Req. 3*. The 1:1 mapping between up- and downlinks determine fixed synchronization partners, thus it is not possible to choose non-deterministically between uplinks, as it is the case with common P/T nets. Additionally, this prototype only allows for one DSC to exist in the simulation which prevents the simulation of any complex net.

Another main limitation is the necessity of a specific initial marking. (*Req. 8*) With this limitation the prototype can not be used as a general prototype for all Distributed P/T Nets.

The scalability is as well limited as there should be just one communication medium (*Req. 5*), one synchronization service (*Req. 4*) and two simulators (*Req. 1*). The limitation of the synchronization service additionally impedes multiple concurrently firings of one DSC. The implementation of the

prorotype as well limits the possible amount of firing processes to one, visible in the protocol. This leads to a minimal second phase (see section 5.2.1) of the distributed simulation. To address this issue is a big deal regarding the realization of a proper distributed discrete event simulation. [18]

The last point of criticism is that the network address of the communication medium must be well-known. This means that the prototype collapses as soon as the network address of the communication medium changes, which is not robust. Similar to this the channel topic needs to be well-known, defining an additional limitation.

In the next prototype, one of the most important restrictions is to be removed. This means that the transition of a DSC must initially be activated. To remove this main restriction, an update protocol will be implemented in the next prototype, with which changes in the local marking can be communicated to the global marking on the event stream.

6. Update global marking of a Distributed P/T Net

6.1. Requirements

The requirements for the second prototype are discussed here. This prototype expands on the implementation of the first prototype. The requirements of the first prototype 5.1 still apply to this prototype except for the following changes and additions.

The main difference to the first prototype is the trigger for firing a transition that belongs to a synchronous channel. In the first prototype it was only possible to fire a synchronous channel once. In contrast the second prototype shall inform the synchronization service about new markings of presets belonging to transitions of a DSC. This information as well needs to be processed accordingly and enable the firing of a DSC. These requirements combined should make multiple firing of one DSC possible. This is the reason for adding the two following requirements:

Req. 12 A transition that belongs to a DSC is able to communicate changes in its pre-set.

Req. 13 The synchronization service is able to react to changes in local presets of transition that belong to a DSC

Additionally, we need to take a closer look at the protocol. As the transitions with distributed channels now shall fire multiple times we need an update protocol with which changes in the local marking can be communicated to the global marking on the event stream.

Req. 14 There should be a protocol addressing the update process.

To make a distributed channel fire more than once the marking needs to allow this scenario. This is why *Req. 8* needs to be changed accordingly:

Req. 8 (V2) The transitions shall be able to fire multiple times in the simulation of the proof of concept.

6.2. Design

6.2.1. Simulation

The kind of simulation for this prototype does not change. But as the firing of a distributed synchronous channel may be triggered multiple times by now the second phase of the simulation can be extended with multiple firings. Following the second phase is no longer minimal.

6.2.2. Protocol

Following *Req. 14*, an update protocol is implemented. It is also necessary for the update protocol that a registration has been carried out beforehand. This is because the update event will not communicate the structural information of the transition that is sending the event. Without the structural information the synchronization service cannot check whether a unification is possible. Furthermore, a firing protocol is started by completing an update protocol if this is possible due to the new marking. The update process is shown in Figure 11.

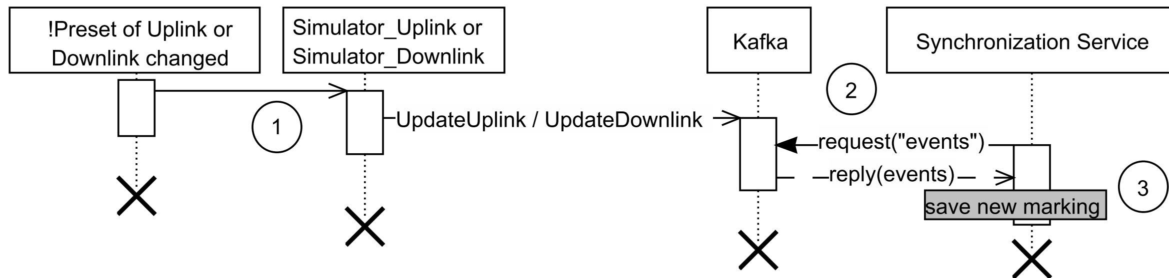


Figure 11: Protocol for the update process

The trigger for the process of sending an update event is the change of the pre-set for any up- or downlink belonging to a DSC as shown in step one in Figure 11. The event is similar to the register event sent via Kafka and will be fetched in step two by the synchronization service. In the third step the synchronization service saves the new received marking and may start a new unification if the artificial transition is enabled. Thus, the process is successfully completed if the new marking is just saved.

This addition results in the synchronization service constantly attempting to unify, resulting in the possibility of multiple "ConfirmFirable" events being sent if the default uplink or downlink setting changes during the simulation of the network. As a result, the networks running in the simulators can potentially be triggered repeatedly.

6.2.3. Events

For this prototype two new events are introduced. The events *UpdateUplink* and *UpdateDownlink* require the new marking of the pre-sets in addition to the corresponding net name as parameters. There is no need to include the structural information at this point, as this has already been transmitted with the events *RegisterUplink* and *RegisterDownlink* during the registration process as stated in Figure 6 and the structural information can not change during a running simulation. Additionally, the size of the events would be greater if every Update Event sends the structural information. Besides, the concept for the update events is different from the registration events. While the registration events belong to the first phase of a distributed simulation, the update events belong to the second phase.

6.2.4. Proof of Concept

Removing the limitation for the proof of concept for the first prototype the second prototype allows the distributed channel to fire multiple times by changing the initial marking. The graphical representation is nearly similar to Figure 9. It differs from the first proof of concept by the amount of tokens. For the second proof of concept there are multiple tokens instead of one token laying on place $p1$, $p2$ and $p3$

6.3. Implementation

6.3.1. Simulator

To enable continuous firing of the distributed channel it is necessary to inform the synchronization service about changes in the marking of the pre-set of a transition that belongs to a DSC. This is done by

sending the respective update events “UpdateDownlink” and “UpdateUplink” for the transition. These events are triggered when the pre-set of the transitions that are part of a DSC is changed.

6.3.2. Synchronization Service

Whenever an update event of a registered transition is received, the marking of the pre-set of that transition is updated. Additionally, any received update event will result in an attempted unification as soon as the marking is updated, so that no marking is missed that could have led to a successful unification.

6.4. Evaluation

The presented prototype additionally implemented two further requirements to the already mentioned requirements. *Req. 12* expects the simulators of the whole simulation to communicate the changes of a pre-set belonging to a transition of a DSC. This requirement was successfully implemented by the described events “UpdateDownlink” and “UpdateUplink”.

The second requirement that was added *Req. 13* concerned the synchronization service that has to react to the above mentioned events. This requirement is fulfilled as well by the given implementation explained in 6.3. To achieve the changed *Req. 8 (V2)* an example is given as depicted in section 6.4 of the proof of concept. In addition, the requirement *Req. 14* was fulfilled by implementing an update protocol in section 6.2.

The second prototype still suffers from various limitations. Nevertheless, some limitations have been addressed and successfully resolved, including the main limitation that the transition belonging to a distributed synchronous channel can only fire if it is initially marked. This was achieved by implementing an update protocol, whereby changes to local markings can be announced globally.

Another main limitation regarding the size are given by*Req. 1*, *Req. 2*, *Req. 7* and *Req. 3* will be addressed in the next prototype. Several Simulators, DSCs and up- and downlinks are to be used for this.

First, the amount of firing processes is no longer limited to one. Second, the marking of the net may enable the transition belonging to a DSC during the simulation instead of initally. Lastly, the second phase of the distributed discrete event simulation is no longer minimal and thus a proper simulation may be reached.

7. Handling several Simulators, Distributed Synchronous Channels and Uplinks and Downlinks

7.1. Requirements

The requirements for the third prototoype are discussed here. This prototype is designed to handle several up- and downlinks for one DSC, several simulators as well as several DSCs in one distributed simulation. This leads to a few changes in the requirements.

The first change applies to *Req. 1*. Allowing multiple instances of the same net requires more than two simulators, thus changing the first requirement. Secondly, we now want to allow multiple synchronous channels, therfore we need to adapt *Req. 2*. Thirdly, *Req. 3* is changed, allowing more than one downlink e.g. uplink per DSC. Lastly, using one topic for all communication in the communication medium as stated in *Req. 7* is no longer sufficient. Rather we will now use a global topic for registration and deregistration of transitions and create a topic for every synchrnonous channel where all other events that are specific to that channel will be sent on.

Req. 1 (V2) It should be possible to use more than two simulators.

Req. 2 (V2) It should be possible to use more than one distributed channel in a Distributed P/T Net.

Req. 3 (V2) More than one uplink and one downlink should be able to belong to one DSC.

Req. 7 (V2) There should be a well-known global registration topic at the communication medium.

Following these changed requirements two new requirement become crucial. Every up- and downlink needs to be globally distinct for so that any sychnronized firing is clearly linked to its participating transitions. Moreover a topic in the communication medium needs to be dynamically created for every existing DSC whenever any simulation, containing one of its downlinks or uplinks, is started.

Req. 15 Transitions belonging to a DSC must be distinguishable.

Req. 16 There needs to be exactly one topic in the communication medium for every DSC in a running simulation.

A few modifications need to be made to the proof of concept for this prototype as we now allow multiple channels and several uplinks and downlinks for those channels. Subsequently we change *Req. 11* and add a new requirement:

Req. 11 (V2) The proof of concept contains more than one DSC.

Req. 17 The proof of concept should contain more than one downlink and uplink for one channel.

7.2. Design

7.2.1. Simulation

The design of the simulation of the third prototype does not change a lot. As now multiple net instance are allowed, the amount of processes in the simulation may scale up. This is the first step to reach a simulation that is able to use the main advantages of distribution, horizontal scaling that comes with a lot of computation power.

7.2.2. Distributed Synchronous Channel

In this prototype we want to allow an upscaling of the central concept, the DSC. This means we will now allow multiple DSCs to exist in the simulation. Additionally, we allow any DSC to have several up- and downlinks. This allows multiple different pairings of distributed transitions to be possible in a DSC. This upscaling is of great importance, as we want no limit for the amount of different DSCs.

7.2.3. Protocol

Whenever a simulator is starting its simulation, the corresponding channel topic needs to be created if it did not exist yet. Furthermore, the sychnronization service needs to subscribe to the corresponding channel topic whenever receiving a register event. Our protocols are modeled as Agent Interaction Protocol Diagrams, explained in section 2.8, so they still fit to multiple instances of our components, as one role is representative for multiple instances. An exception to this is the synchronizations service, which cannot be scaled yet.

7.2.4. Events

As there now is the possibility for different channels to exist there needs to be a policy to communicate the existing channels to the sychnronization service. This will be done by centralizing all register as well as deregister events to one topic that is well known amongst all components. This leads to the need to include the Name of the DSC of a transition in its registration event.

We have different up- and downlinks running on the same DSC and therefore they listen to and produce events on the same Kafka topic. This means that transitions need to be distinguishable.

Event	Parameters
RegisterUplink / RegisterDownlink	Parameter, Pre-set, Post-set, Parameter/Arc-Mapping, *Identifier*, *Channel*
DeregisterUplink / DeregisterDownlink	*Identifier*
UpdateUplink / UpdateDownlink	Marking of Pre-set, *Identifier*
ConfirmFireable	*Identifier of Downlink*, *Identifier of Uplink*, Parameter values
ConfirmDownlink / CancelDownlink	*Identifier*, *Identifier of Partner*
ConfirmUplink / CancelUplink	*Identifier*, *Identifier of Partner*

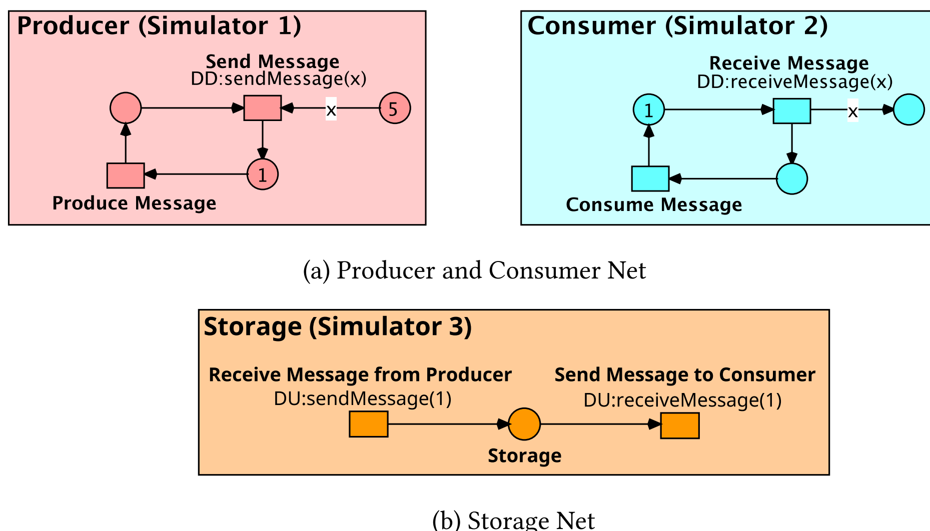
Table 3
Adjusted Design of Update Events

To do this we decided to give every transition of a DSC a unique identifier. This unique *identifier* differs from the formal identifier that was specified in 2.2 as it is a technical necessity. It needs to be passed on in every interaction to avoid possible mixups between any two up- or downlinks of a DSC during one singular firing interaction. Therefore the parameters of all events sent by a simulator need to be adjusted to always include the identifier of the transition that sent the event.

Furthermore the event ConfirmFirable needs to include the unique *Identifier of Downlink* as well as the unique *Identifier of Uplink* that the successful unification was done with. Lastly the events ConfirmDownlink, ConfirmUplink, CancelDownlink as well as CancelUplink need to also specify which partner they were mapped to for that specific firing to avoid other transitions from reacting to them. This is done by including the unique *Identifier of its Partner* for that specific firing. Following the Table 2 and section 6.2.3 the events are mainly changed by adding specific identidiers to the parameters. Besides, the register events now contain the repective channel. The adjusted design of the event can be found in table 3.

7.2.5. Proof of Concept

According to *Req. 11 (V2)* the proof of concept should now contain multiple distributed channels. Therefore we extend 6.2.4 by a storage component. This means the producer and consumer do not interact directly anymore but instead exchange messages through the storage. These changes are visualized in Figure 12.



(a) Producer and Consumer Net

(b) Storage Net

Figure 12: Components of the producer storage consumer example

Additionally *Req. 17* demands multiple up- and downlinks for each channel of the proof of concept. We accomplish this by allowing every component to exist multiple times on different Simulators.

7.3. Implementation

7.3.1. Simulator

Whenever a simulator is initializing its simulation for a Distributed P/T Net, it will check for any DSCs. For each found DSC a new topic will be created in the communication medium as long as it did not exist beforehand. Finally, the simulator will subscribe to all topics that represent its DSCs to establish the communication to other components of the simulation via the communication medium Kafka.

Register and deregister events will then be sent on a seperated topic. All events that concern a specific channel will be sent on the topic of that channel.

All simulators now assign a unique identifier to every transition of a DSC in the local net it simulates so that different up- and downlinks can be differentiated by other components. This is implemented by assigning a Universally Unique Identifier (UUID)[34] to every transition. This also allows the possibility to have multiple distributed transitions in one simulator because it is always clear which transition was called by an event.

7.3.2. Sychronization Service

Since a DSC now consists of multiple up- and downlinks, the synchronization service now has to save the unique identifier of any transition registering to it. So that it knows exactly which transition is part of a specific unification and who wants to unify with who. Subsequently, it needs to specify which transitions were part of its successful unification whenever sending a "ConfirmFirable" event.

The sychnronization service now subscribes to any channel topic that is specified in a registration event. Subsequently "ConfirmFirable" events are now sent on the topic that represents its corresponding channel.

7.4. Evaluation

Req. 1 (V2) was made possible by allowing multiple up- and downlinks for a DSC as well as allowing multiple channels. Both of these changes allow for more than two different distributed transitions to exist in a simulation. Since the implementation was already able to handle multiple simulator instances by default, this requirement is met.

To satisfy *Req. 2 (V2)* we added a way to differentiate between channels by either including the channel in an event or sending the events to a new topic that represents the channel. This simultaniously accomplishes *Req. 16*.

Req. 7 (V2) was fullfilled by introducing a globally known topic that is specifically used to send all registration and deregistration events from a simulator.

To realize *Req. 3 (V2)* we assigned a UUID to every uplink and downlink. This allows us to be able to differntiate between every transition of a synchrnonous channel, which subsequently completes *Req. 15*. *Req. 11 (V2)* and *Req. 17* were fulfilled as described in 7.2.5.

The scalability of the components was partially resolved as there is now the possible to increase the scale of the simulators to more than two instances. Additionaly there is now the ability to have more than one uplink and one downlink for a DSC. Furthermore this prototype allows multiple channels to exist in the simulation, which enables the simulation of far more complex nets. This has removed a further restriction in regard to size, although this can be further improved.

8. Discussion

One of the main advantages is the potential performance improvements through distribution. However, this requires more computers overall, which means that the performance requirements are higher. However, they can now also do more if the distribution works well. The fact that the decision for synchronization is made centrally in the synchronization service means that this service is crucial

for the performance of the overall system. Improvements will be made to this in future versions, for example by improving scalability.

Furthermore, the distribution allows to model and simulate one net that is created on different computers if the channels that will be used are agreed on. Therefore the Kafka broker must be reachable for every participating part of the net. However, a simulation must be coordinated to avoid inconsistencies in the simulation.

Another advantage is the centralization of events at Kafka. By using this characteristic it might be possible to restore a point in a simulation more easily. Following the usability can be increased. Moreover, crashed instance of a simulation may be restarted without canceling the whole simulation.

Obviously, there are at least two different types of synchronous channels. The local synchronous channels, which are executed in a simulator, and the DSCs, which are executed in a distributed manner. One disadvantage of DSC compared to local synchronous channels is performance, as the synchronization process is more complex.

One limitation of the presented prototype is that the concurrent firing of a single DSC is not implemented by now. However, this will be addressed in future versions.

Another current limitation is the lack of scalability regarding the synchronization service. This requires some technical adjustments to the synchronization service and execution in a container environment including container orchestration, such as Kubernetes [35].

In addition, this prototype only implements the distributed simulation for P/T nets with synchronous channels and not, for example, for reference nets. However, extensions in this direction are planned.

9. Related Work

In [36, 37, 38] first experiments with Renew in the Kubernetes context were undertaken to perform reference net simulations. The decision to synchronize synchronous channels is made in a distributed manner. Java Remote Method Invocation is used as the communication medium for this. This requires a central registry, which represents a single point of failure.

One of the main differences to this work are that this prototype makes the decision about synchronization centrally. Another main difference is that a event-based approach is chosen for the communication medium. This is intended to create a additional synchronization option for synchronous channels for distributed simulation. In addition, the prototypes presented here only implement P/T nets instead of coloured or reference nets.

Whereas the works [39, 40, 41, 42] deal with the distributed simulation of timed Petri Nets. The approaches used in these works use different strategies to ensure the simulation of timed Petri Nets. This work distinguishes itself from these approaches in that the P/T net class used in this paper does not recognize a notion of time.

10. Conclusion

Section 2 covered all necessary foundations, crucial to understand this article. Section 3 covered the objectives of this work, namely a prototype implementation, the establishment of a distributed execution environment and the usage of a classical computer science example as a proof of concept. The following section, section 4 treated the distributed system consisting of a communication medium, a synchronization service and multiple simulators.

The prototype "Firing of one Distributed Synchronous Channel" in section 5 represents the first version of distributed simulation of Distributed P/T Net. This prototype includes exactly two simulators with one DSC consisting of one up- and one downlink, as well as one communication medium and one synchronization service. The main protocols of this prototype were the registration and the firing.

The prototype "Update global marking of a Distributed P/T Net" builds upon the prototype and is described in section 6. As the prototype was limited to one firing of a synchronous channel, this

prototype was meant to enable multiple firing. That is why a new concept was introduced, namely the update of markings so that every component that needs to react to these changes can do it.

The third prototype "Handling several Simulators, Distributed Synchronous Channels and Uplinks and Downlinks" focuses on the limitation that various components are limited the instances of the components and is explained in section 7. This limitation is resolved for several simualtors, several DSCs and up- and downlinks in this prototype. The upscaling especially demands the adaption of the events so that multiple instances are distinguishable. The discussion in section 8 reviews the results deriving from this paper and points out the advantages, disadvantages and limitations of the proposed prototype.

There are still some limitations left regarding this implementation of Distributed P/T Nets. Regarding Kafka, it is predictable that high availablity becomes significant.

Another interesting point starting from this article may be the integration of the distributed simulation into plattforms like Docker [43] and Kubernetes [35] that allow to scale the computation power of a simulation or the amount of simulation.

The extent to which the prototype can be used in the context of verification for distributed model checking will also be investigated.

Additionally, this proposal of distributed simulation of nets is limited to P/T nets with channels. Therefore the expansion to simulate colored nets or reference nets might as well be interesting.

References

[1] I. Mitrani, Simulation techniques for discrete event systems, volume 14, Cambridge University Press, 1982.

[2] A. Ferscha, S. K. Tripathi, Parallel and distributed simulation of discrete event systems, Citeseer, 1998.

[3] E. Jessen, R. Valk, Rechensysteme: Grundlagen der Modellbildung, Studienreihe Informatik, Springer-Verlag, Berlin Heidelberg New York, 1987.

[4] S. Christensen, N. D. Hansen, Coloured Petri nets extended with channels for synchronous communication, in: R. Valette (Ed.), Application and Theory of Petri Nets 1994, 15th International Conference, Zaragoza, Spain, June 20-24, 1994, Proceedings, volume 815 of *Lecture Notes in Computer Science*, Springer, 1994, pp. 159–178.

[5] C. Lakos, From coloured Petri nets to object Petri nets, in: International Conference on Application and Theory of Petri Nets, Springer, 1995, pp. 278–297.

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

[7] 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.

[8] K. Jensen, Coloured Petri nets, in: Petri Nets: Central Models and Their Properties, Springer, 1987, pp. 248–299.

[9] A. V. Ratzer, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, K. Jensen, Cpn tools for editing, simulating, and analysing coloured Petri nets, in: International Conference on Application and Theory of Petri Nets, Springer, 2003, pp. 450–462.

[10] C. A. Petri, Nets, time and space, Theoretical Computer Science 153 (1996) 3–48.

[11] J. Desel, W. Reisig, Place/transition petri nets, in: Advanced Course on Petri Nets, Springer, 1996, pp. 122–173.

[12] K. Lautenbach, Linear algebraic techniques for place/transition nets, in: W. Brauer, W. Reisig,

G. Rozenberg (Eds.), Petri Nets: Central Models and Their Properties, Springer Berlin Heidelberg, Berlin, Heidelberg, 1987, pp. 142–167.
[13] M. Jantzen, R. Valk, Formal properties of place/transition nets, in: W. Brauer (Ed.), Net Theory and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 1980, pp. 165–212.
[14] H. Heilmann, Organisatorische flexibilität im intelligenten unternehmen—potentiale von workflow-management, Wissensmanagement: Schritte zum intelligenten Unternehmen (1998) 109–128.
[15] W. M. van der Aalst, K. M. van Hee, A. H. ter Hofstede, N. Sidorova, H. Verbeek, M. Voorhoeve, M. T. Wynn, Soundness of workflow nets with reset arcs, Transactions on petri nets and other models of concurrency III (2009) 50–70.
[16] H. M. Verbeek, W. M. van der Aalst, A. Kumar, Xrl/woflan: Verification and extensibility of an xml/petri-net-based language for inter-organizational workflows, Information Technology and Management 5 (2004) 65–110.
[17] T. Jacob, Implementierung einer sicheren und rollenbasierten Workflow"=Managementkomponente für ein Petrinetzwerkzeug, Diploma thesis, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, 2002.
[18] A. Varga, Discrete event simulation system, in: Proc. of the European Simulation Multiconference (ESM'2001), volume 17, 2001.
[19] R. M. Fujimoto, Research challenges in parallel and distributed simulation, ACM Transactions on Modeling and Computer Simulation (TOMACS) 26 (2016) 1–29.
[20] 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.
[21] 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. doi:10.1007/978-3-031-33620-1_12.
[22] 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=.
[23] R. Valk, Petri nets as token objects - an introduction to elementary object nets, in: J. Desel, M. Silva (Eds.), 19th International Conference on Application and Theory of Petri nets, Lisbon, Portugal, number 1420 in Lecture Notes in Computer Science, Springer-Verlag, Berlin Heidelberg New York, 1998, pp. 1–25. URL: https://doi.org/10.1007/3-540-69108-1_1.
[24] Apache, Introduction, https://kafka.apache.org/intro, 2024. Accessed: 2024/04/29.
[25] N. Naik, Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http, in: 2017 IEEE International Systems Engineering Symposium (ISSE), 2017, pp. 1–7. doi:10.1109/SysEng.2017.8088251.
[26] T. Bray, The JavaScript Object Notation (JSON) Data Interchange Format, RFC 7159, 2014. URL: https://www.rfc-editor.org/info/rfc7159. doi:10.17487/RFC7159.
[27] Foundation for Intelligent Physical Agents, FIPA interaction protocol library specification, 2000. URL: http://fipa.org/specs/fipa00025/PC00025C.html.
[28] J. J. Odell, H. Van Dyke Parunak, B. Bauer, Representing agent interaction protocols in uml, in: Agent-Oriented Software Engineering: First International Workshop, AOSE 2000 Limerick, Ireland, June 10, 2000 Revised Papers 1, Springer, 2001, pp. 121–140.
[29] S. FIPA00003, Fipa interaction protocol library specification (2000).
[30] M. F. Wood, S. A. DeLoach, An overview of the multiagent systems engineering methodology, in: International Workshop on Agent-Oriented Software Engineering, Springer, 2000, pp. 207–221.
[31] O. M. G. Inc., OMG Unified Modeling Language – version 2.5.1, https://www.omg.org/spec/UML/2.5.1, 2017. URL: https://www.omg.org/spec/UML/2.5.1, last accessed: 2024-05-01.
[32] L. Cabac, Modeling Petri Net-Based Multi-Agent Applications, Dissertation, University of Hamburg,

Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2010. URL: https://ediss.sub.uni-hamburg.de/handle/ediss/3691.

[33] L. Cabac, D. Moldt, Formal semantics for AUML agent interaction protocol diagrams, in: J. Odell, P. Giorgini, J. P. Müller (Eds.), The Fifth International Workshop on Agent-Oriented Software Systems (AOSE-2004). Proceedings, Columbia University, New York, USA, 2004, pp. 97–111. URL: http://dx.doi.org/10.1007/978-3-540-30578-1_4.

[34] P. J. Leach, R. Salz, M. H. Mealling, A Universally Unique IDentifier (UUID) URN Namespace, RFC 4122, 2005. URL: https://www.rfc-editor.org/info/rfc4122. doi:10.17487/RFC4122.

[35] Google, Kubernetes Documentation — kubernetes.io, https://kubernetes.io/docs/home/, 2024. [Accessed 29-04-2024].

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

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

[43] Docker, Home — docs.docker.com, docs.docker.com, 2013-2024. [Accessed 29-04-2024].