Cloud Native Simulation of Reference Nets

Jan Henrik Röwekamp, Marvin Taube, Patrick Mohr, Daniel Moldt

University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences,
Department of Informatics, http://www.informatik.uni-hamburg.de/TGI/

Abstract. This contribution presents the integration of the cloud-native paradigm into the distributed simulation of reference nets. This is done to reap the benefits of cloud-based environments and by that to increase the scaling capabilities of such simulations on an architectural level. In contrary to most Petri net formalisms, reference net simulations are able to allocate net instances at runtime, like objects are instantiated in object oriented programming. Therefore, reference nets are not static in size and well suitable to natively model dynamic systems with changing demands and net structures. During the architectural discussion the key concepts of operability, observability, agility, and resilience are addressed. Further, a proof-of-concept implementation illustrates the potential of such cloud-based systems. It utilizes the simulator Renew and the Java Spring framework.

Keywords: High-level Petri nets, Cloud infrastructure, Software Architecture, Distributed Simulation, Renew, Containerization

1 Introduction

The interaction of large sets of agents in hierarchical structures is omnipresent in the real-world. While humans are certainly agents in this notion, machines and organizational entities can be as well. Through simulation of these kinds of interactions, information about real-world scenarios can be deduced. In the context of reference nets, a high-level Petri net formalism, that supports concepts similar to object-oriented programming, several publications regarding simulation of hierarchical agents exist.

While several important questions have been addressed by these publications, the simulations usually ran on a single physical machine and by that, we are restricted in its size. While the simulation of traditional Petri nets is usually static in size with the exception of token count in some cases, the size of a reference net simulation cannot be determined at compile time. This is due to their concept of net instances, which may be created dynamically during runtime. Therefore, scalability is an interesting issue to be solved in the context

of simulating reference nets. It addresses the ability to dynamically extend the simulation to more physical notes in an automated and unsupervised fashion.

There has been research to execute a reference net simulation over multiple machines using the DISTRIBUTE plugin for the simulator RENEW [18]. Subsequently, virtualization [13] and container technology [17] have been used to further abstract from specific infrastructure and enable higher portability of RENEW simulators. The control over the simulation extension has been incorporated into the simulation itself to allow for proactive scaling of instances in [16]. While the aforementioned publications only address the core functionality to distribute the simulation and the control of the environment, there are no considerations regarding the software architecture of the simulator in regards to scalability.

Cloud nativity is a paradigm to construct applications, that do not rely on the possibility for operators to be able to access the systems they are running on. With increasing independence on the underlying infrastructure, the application is able to spread to additional physical nodes more easily.

Cloud native applications are also popular in the community of microservice applications. These are usually run in cloud environments, that are constructed and destructed dynamically and applications are deployed automatically.

This paper contributes software architecture related solutions to incorporate the paradigm of cloud nativity into reference net simulators, especially the RENEW simulator. Our foreseen benefits are a more flexible use of the simulator, abstraction from specific protocols like e.g. Java RMI, and better incorporation into scalable environments. Our contribution is focused on the backend-part of the simulator. Therefore any graphical representations or consumers are referred to and used for motivation in the following sections, but are not part of the core contribution. As an exception, we use the ready-made user interface of Spring Boot Admin[1] in the evaluation part of this contribution.

In chapter 2 we explain the basics. The general strategy on how to approach an integration is covered by chapter 3. Following that, chapter 4 describes the conceptual integration of Cloud Native RENEW. In chapter 5 we elucidate our implementation. Lastly, chapter 6 serves as an evaluation of our work, and chapter 7 represents a final conclusion.

1.1 Motivation

When attempting the simulation of large-scale systems scalability of the simulating software components is beneficial. Using a paradigm such as cloud nativity can directly improve this aspect of software. Without dependence on system access to the infrastructure running the simulation, as well as robustness towards other components and agility of the software, many manual steps can be necessary to incorporate additional physical nodes into the simulation.

[1] https://github.com/codecentric/spring-boot-admin - All URLs have been accessed in March 2021

The requirement for the introduction of scalable reference nets originated from the construction of dynamic multi-agent multi-platform systems. The overall approach is far too large to sufficiently summarize here. The gist of the approach is the introduction of a platform management system, that is able to proactively summon heterogeneous reference net based agent platforms into existence and deploy specific functionality to it, as well as to move agents between these kind of platforms.

Looking from the Petri net point of view, there should be no difficulty in firing concurrent transitions on physically distant nodes. However, due to the complexity of the simulation of systems described with reference nets, this can be difficult to achieve. A cloud native simulator also favors more universal protocols for information retrieval, such as HTTP, and by that enables the simulation to be accessible in different ways from different software and systems.

With a universal interface to the software, it is e.g. possible to easily split simulation and representation of the simulation. Additionally, a cloud native simulator could be used without installing it on a local system first, enabling even low spec clients like e.g. netbooks to run complex simulations easily. Also, it is possible to interact with the simulation without a local Java environment installation.

Another useful factor is the ability for the simulator to report other states of the simulator, which would otherwise be difficult to determine from the system's point of view, that is running the simulator. Examples for these states are the number of simulated net instances, loaded plugins of the simulator, liveliness of the simulated reference net, and so on. These could be accessed in the same fashion as external attributes such as CPU load or memory consumption.

1.2 Related Work

Several research papers have been published concerning the integration of Petri nets and service-oriented structures, like [12], that considers the performance of web services but focuses on business processes and service allocation. In a similar fashion, [1] considers performance issues on services but aims mainly at mathematical models. Other publications use different Petri net subformalisms to interact with services, like e.g. [5] or [2].

The combination of container technology and Petri nets has been discussed in [3] and [17]. Implementation-wise [15] introduces first ideas regarding the integration of the Spring framework with reference nets.

2 Basics

Introducing the topic, we now explain the core concepts reference nets, Renew, and cloud nativity.

2.1 Reference Nets

Reference nets are colored Petri nets that convey the object-oriented mindset and combine the concept of nets-within-nets [19] with communication via synchronous channels. Reference nets feature the concept of *net instances*, that correspond to objects in object-oriented programming, while nets (or "net definitions") correspond to classes. Like colored nets with arbitrary color sets, reference nets are Turing-complete. Also, setting up net hierarchies is possible by referencing specific instances. Synchronous channels synchronize multiple transitions to fire simultaneously. This way, it allows multidirectional information passing via unification across multiple net instances. The transitions must agree on the channel name and the set of parameters in order to establish the synchronization.

2.2 Renew

Renew [10] is a multi-formalism editor and simulator developed by the Department of Informatics of the University of Hamburg. It offers a flexible modeling approach for reference nets via a user-friendly graphical interface. Renew's key values, openness, and versatility have various reasons. Firstly, it runs on Java, and therefore on almost every single operating system. It can also make use of any Java class by adding them to a transition inscription within a reference net. Lastly, since reference nets are also Java objects, the communication between the code and the reference nets is straightforward. Renew is controlled by a plugin system [4] which offers a high degree of decoupling. Renew is available for download on the projects' website[2].

2.3 Renew Remote

Renew features a "remoting" layer between the simulation core and the user interface. It is based on Java RMI[3] and has been introduced in Renew 1.6 in [11]. As it is currently still the most advanced feature of Renew to provide operability and observability, it is used as a point of reference in this paper. Like all Java RMI-based implementations do, Renew Remoting requires an RMI registry to keep track of remote objects.

Upon starting a remote-enabled simulation, a second instance of Renew may connect to the running simulation over RMI. It is then possible to control the simulation from the second instance as if it was running locally. This includes starting, stopping, resuming, stepping, and terminating a simulation, as well as firing transitions manually. It is also possible to view instantiated net instances, as well as the current marking and transition firings. Renew Remoting, however, is limited to a connection to a single simulator.

[2] http://www.renew.de
[3] *Java RMI* is the Java implementation of the Remote Method Invocation protocol

2.4 Renew Distribute Plugin

The DISTRIBUTE plugin was introduced by Michael Simon [18]. It allows the construction of distributed simulations in RENEW. The plugin is also based on the Remote Method Invocation protocol and by that on Java RMI.

It alters the core of the simulation algorithm in various ways. The details are less important for the considerations in this contribution and are therefore omitted. The publication [18] shows the inner workings of the plugin in more detail. Implementation-wise the DISTRIBUTE algorithm corresponds to a limited extent to a classic distributed commit protocol.

2.5 Modular Renew

A recent contribution to the RENEW simulator is the support of modularity. It is employed using the Java Jigsaw project, which was released with Java 9. Its main benefits are improved encapsulation within the application by using explicitly defined interfaces between modules. Module dependencies are also defined as a directed acyclic graph instead of arbitrary connections.

As an additional step, modular RENEW also uses so-called module layers, which provide a means to apply existing module principles to code, that is loaded at runtime. Module layers also provide the base for the ability to dynamically unload plugins, which is interesting in the context of hot-swapping implementations within a running application. By now, each RENEW plugin resides in its own module, which is loaded inside a separate layer.

Modular RENEW can also be downloaded from the project's website as "preview of RENEW 4.0".

2.6 Cloud Nativity

Cloud nativity is an approach in software development of growing importance, which utilizes cloud computing to create scalable applications in dynamic environments.

Cloud nativity's strength is to even function on an unknown system's infrastructure without the need for operators to manually access it. There are several definitions of cloud nativity. As it fits our intended architecture the best, we will refer to the one presented in [8]. According to Garrison and Nova, cloud nativity has the four key benefits resiliency, agility, operability, and observability.

Firstly, a cloud native system is resilient, when the failure of one component of the system doesn't cause other components to fail also. This is really valuable, as a chain reaction could take down the entire system.

The system is also agile, as quick changes according to new requirements can easily be made. A part of this capability is rooted in the architecture: the system consists of microservices and uses containerization for deployment. Containers are used to package all software needed into one executable package and should be provided using a continuous integration pipeline. This allows the project to

be independent of the environment. Also, agile development approaches are used to enable quick reactions to changing demand.

The application should publish interfaces to administrate the application and by that provide operability. Using them, an administrator can perform configuration tasks or administration of users, without the need to rely on underlying systems.

And lastly, observability is given by the provision of information about ongoing processes, for example, health metrics and log data.

3 Approaching an integration

To realize the integration of cloud nativity with reference net simulations, we first outline the shortcomings in the current architecture of Renew and its relevant plugins in regards to cloud nativity. We do so by covering the four aspects of observability, operability, agility, and resilience separately. To achieve a successful integration, the addressed problems need to be solved. From within each aspect, we will construct specific requirements to be met.

Observability is mainly realized by the *Remote* plugin. The run of a single simulation can be observed through an additional client and the Java RMI protocol. This includes live transitions firings, markings, and an event log. System information and meta-information about the simulator are not included. Consuming the information requires an additional Renew simulator, which again requires an underlying Java virtual machine. Certain situations in a remote simulation, like deadlocks, can be observed manually by an observer knowledgeable in Petri nets, but not automatically.

We can now formulate the requirements for the integration:

1. The first and most important goal is to introduce a universal interface for the aforementioned functionalities, that can be used without the need to rely on a local Java installation and Renew software. This also means to decouple it from graphical user interface components.
2. Observability should cover the entire distributed simulation, not just singular parts of it.
3. The environment (system resources) of a remote simulator should be observable through the interface.

Operability is also mainly realized by the *Remote* plugin. Upon attaching to a remote simulation the simulation may be halted, stepped, terminated, etc. This, however, is also only possible with a local installation of Renew and over Java RMI. To a degree, the Distribute plugin could also be used to "operate" a remote simulation by firing certain distributed synchronous (send) channels. The operation, however, is only implicit as it is only limited to the inner workings of the simulation itself. Operability was also not the intended use of the Distribute plugin and it would require an additional local Renew simulation.

Requirements regarding operability are therefore:

4. Operability also requires a more flexible interface. This interface should be identical to the one from observability.
5. The simulator should be extensible by new net definitions and not just net instances. Modification of existing net definitions on the other hand potentially leads to non-trivial and large complications and therefore will not be addressed in this context.
6. The simulator should be extensible with additional functionality, that might be required to run specific net definitions.

Agility is available in several aspects of Renew. In regards to agile development, Renew is developed using the Scrum@Scale approach[4]. It is also integrated with extensive continuous integration support and by that using automated pipelines for building. Concerning portability, the publication [13] examined virtualization-based deployment and [17] presented deployments based on containerization. Influence on the execution infrastructure is possible for simulations with proactive scaling, as introduced in [16]. Agility aspects are well covered already, therefore there is not much left to be done in this regard.

As the aspects are well covered through other works in the context of the Renew simulator, we can only express one requirement:

7. The simulator needs to utilize all the features referenced in the "Agility" section simultaneously.

Resilience is mainly relevant in regards to the distribution and inclusion of external services. The Distribute plugin is built on top of Java RMI and therefore requires a single RMI registry for all nodes in the overall deployment. The registry needs to be present during the start of each simulator participating in the simulation. It also must not disconnect during the simulation and net instances of past runs might still be known to the registry leading to inconsistencies. These problems can be alleviated by restarting the registry before an actual simulation, though. Further, relying on external services is not generally addressed on the simulator implementation level. This is due to the fact, that the ecosystem was more or less closed for external services so far. But introducing a flexible and universal interface, as motivated earlier, opens the possibility to incorporate other external (web)services into the simulation runtime. These services should not be addressed directly, to avoid cascading failures or local corruption.

Overall we were able to define the following requirements for the resilience of a cloud native reference net simulation:

8. Failures of global system components must not lead to cascading failures.
9. Failures of local nodes must not endanger the overall simulation.
10. Simulation processes must not directly rely on external services, the structure of their delivered data, or on the quality of their delivered data.

[4] see https://www.scrumatscale.com/

4 Conceptual Integration

In this section we will present the concepts we propose to address the requirements explained in the last section. We believe, that these suffice to achieve a thorough integration of cloud nativity and the reference net simulator Renew. The aspects observability, operability, and resilience will again be addressed one by one. As agility is already well covered, it will not be addressed separately in the conceptual section. An overview of the utilized tools and methods will follow in section 5, which will present our prototype.

We will design the overall system with the global architecture displayed in figure 1 in mind. Several aspects and necessities are only elaborated on further

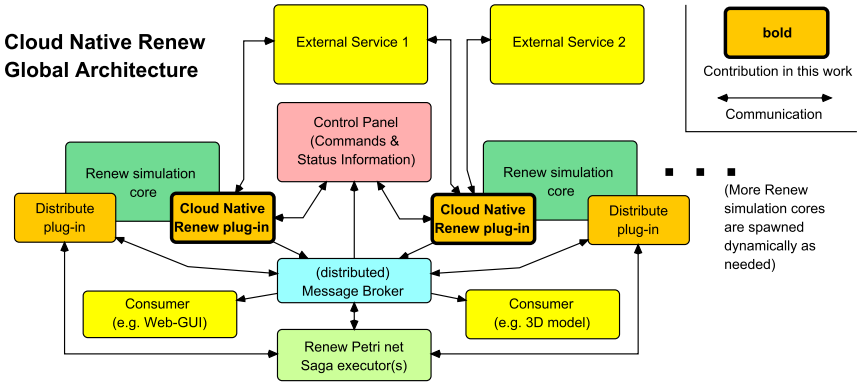

Fig. 1. Global architecture of the cloud native Renew deployment

into this section, however, it will be helpful to deliver the bigger picture early on to explain some aspects more easily. Note, that specific infrastructure elements, like physical nodes, are omitted in this diagram, as the goal of the combination of cloud nativity and the Renew simulator is an infrastructure-agnostic simulation environment in the first place.

The figure shows several separate services, that are required or at least recommended in the deployment of cloud native Renew. Note, that this paper only discusses the cloud native Renew plugin, which is shown in bold font. However, to understand certain implications and design choices it is helpful to be able to see the bigger picture. Elements, that are drawn on top of each other, like for example the cloud native Renew plugin and the Renew simulation core are two parts of the same service, while separate elements are separate services. Not showing the infrastructure also implies, that these services can each run on a different machine or all together on one single machine or something in between.

Some of the prominent parts are the several Renew simulation cores, that are shown in green, alongside their orange colored plugins. There are just two

Renew simulation cores depicted in the figure due to tidiness. As scalability is one of the claims for this contribution, it is intended to be run with much more than just two instances of Renew. The three dots on the right-hand side of the figure also indicate this.

As the cloud native Renew plugin is designed to offer an interface to incorporate external services, two of them have been drawn into the figure as an example. Note, that this is for future-proofness, as there are no incorporated external services as of now, as also mentioned in the approach section. For convenient access to all operability and observability aspects, the central pink-colored control panel can be used. There is no specific implementation of this for reference nets, but as we will show within the evaluation section, depending on the way the aspects of operability and observability are introduced into the simulation, ready-made tool suites can be used.

Further, one of the additional aspects is the distributed message broker shown in light blue. It acts as the central message passing system in the deployment and is mainly used by an updated Distribute Renew plugin. Additionally, there are some special external services displayed and labeled as consumers. These are able to extract information recorded by the Renew simulation cores from the message broker and consume them in several individual ways. An example for a consumer is a web-based GUI, like the one introduced for reference nets by [9]. The consumer of said publication, however, was not wired to a simulation so far, because of the missing link, that is also filled within the context of cloud native Renew.

Finally, at the bottom of the figure "Renew Petri net Saga executor(s)" are shown in light green. These are intended for eventual consistency in the distributed firing process and are described later on in more detail in section 4.3 about the introduction of resilience into the system.

4.1 Introducing Observability

The first aspect to introduce is observability. Using Renew remote it is possible to use the Renew simulator to connect to a remotely running simulation. This is done using the Java RMI protocol and requires a local Renew simulator. While this solution provides excellent observability of a single simulation it does not provide a flexible interface. This means, that some kind of Java environment is required to consume the interface. It also does not provide means to check on loaded plugins and system states of the remote simulation. Renew remote is also tightly coupled to the graphical user interface of Renew, which again relies on the existence of some kind of output screen, and by that, it might encounter unforeseen difficulties in non-graphical container-based deployments.

To acquire a sufficient degree of observability a universal interface is required to access specific information. We argue that HTTP is a good choice in this regard, as it is in fact universally employed and wide-spread, stateless, and sufficient in terms of expressiveness. All relevant system information, that is required by the control panel should be available. This includes memory consumption, available persistent space on the system, and processor load. As processor load

might experience spiking during the invocation of the request to the information HTTP endpoint it is also desirable to be able to request average load within the last seconds. Also, general information on the healthiness of the process is of interest to be able to decide, whether the specific simulation core might require restarting or even manual attention.

The application should present these kinds of health metrics within a single or multiple HTTP endpoints.

Reference net simulations on Renew can differ from each other in regard to the loaded plugins by the simulator. Incorporating an additional simulation node into a running simulation will only be successful if the node in question is aware of all used classes and is capable of the required functionality. It is not explicitly necessary to match the plugin profile of the remaining simulation cores, but the simulator should provide a minimum baseline of capabilities to fulfill its role in the simulation. Therefore currently loaded plugins should be announced upon start on the message bus and also should be available as information to the control panel. Certainly, the plugins themselves might hold data, that is of interest like the core simulator plugin holds information over liveliness and net instance count of the simulation. A unified interface for plugins to supply this information is required for the internal passing of information from the plugins to the HTTP endpoint.

Finally, to be able to trace certain error states of the simulator, it would be desirable to acquire the stdout logging output of the application. Almost every infrastructure solution nowadays can supply these from the infrastructure itself, but to stay within the requirements of being totally independent on the underlying infrastructure, the simulator should also be able to supply its stdout log upon request.

The last and largest part to support observability of the simulation addresses the access to firing events of a running simulation. The remote plugin supplies this only for one simulator, but a global transparent simulation feed of the overall simulation is more desirable. Construction of a simulation feed for reference nets is not easy though. Due to the nature of the tokens, that, in fact, may be references to arbitrary Java objects, describing them is non-trivial.

As displayed in figure 1, displaying the global architecture, a central event log system will be available for external consumers and inter-simulation communication. This event log will store information about the simulation and will be fed by each individual simulator. Consumers then can access the log and display the simulation e.g. in a user interface or consume the process otherwise, like waiting for a certain transition firing.

4.2 Introducing Operability

As outlined in the approach section, the Remote plugin for Renew supports basic operability with the Java RMI protocol. In the last section, we have shown, that a web-based interface over the HTTP protocol will benefit the application in terms of flexibility and requirements on the clients. Therefore we need to redesign the operability presented from within the Renew Remote plugin to

match the new interface. A basic operability interface needs to at least provide means to start, stop, step, resume and terminate a simulation.

Further, in the context of service-based deployments, it will be desirable to extend a running service without the need to shut it down first. Also, the supply of net definitions to run should be possible easy. RENEW utilizes a construct called *shadow net systems*. A shadow net is a net definition, that is stripped from all of its information, that is non-critical for actual execution. Single or multiple shadow nets can form a shadow net system, which is a set of these. Shadow nets also offer the ability to run net compilers on nets to prepare them for execution. They also use binary serialization, when saved in contrary to regular reference nets, which use a text-based solution and are by that larger in size.

As a shadow net is independent of its graphical representation and holds all information to construct a net instance from it to execute, it perfectly fits for the supplying of running reference net simulator services with new net definitions. It is also possible to verify a shadow net system at least for basic validity by parsing the shadow net system object structure, which we will do. The system should also be able to distinguish between submitting simulation candidates using shadow net systems and commands to actually instantiate a new simulation.

Net definitions alone can offer the possibility to run the arbitrary workload on a remote and distributed reference net simulator. Reference nets are Turing-complete and able to express any programmable functionality. However, often functionality can be reused, which is done in the shape of plugins for the simulator. RENEW plugins can be compiled to jar files and can be loaded by the loader plugin, which is the central component of RENEW.

An interface providing operability should be able to accept additional plugins, as well as loading them into the simulation code. We are aware of the security implications of this behavior, as a potential attacker could just upload malware and execute it most easily. This also applies to the upload of net definitions, as they are Turing-complete, as stated before. As for now, however, we do not address this aspect in this contribution and assume all involved parties are trustworthy. For future versions, there is currently undergoing research and implementation work is done in our workgroup regarding net definition signing and plugin signing prior to loading to tackle these problems.

While loading a plugin (or generally speaking a jar library) is straightforward in Java, unloading them is not. Therefore we need to assume, that supplied classes are not yet existent under the specified name (classpath). Unloading might become possible using the Java module system, which was outlined earlier.

Another aspect of operability is the influence on the simulation environment. As this has already been addressed in the publication [16] in the shape of RENEWKUBE, we will omit this aspect here and assume infrastructure interaction is possible and available.

4.3 Introducing Resilience

One of the major concerns regarding resilience is the avoidance of single points of failure and the avoidance of cascading failures. As outlined earlier, the DIS-

TRIBUTE plugin is one problematic part, especially the used RMI registry and RMI calls. The left part of Table 1 shows what parts of the algorithms in the DISTRIBUTE plugin use which approaches.

Approach so far	**Algorithm part**	**New approach**
Registry	*Net instance referencing*	Message broker
Direct via RMI call	*Firing*	
	Binding	Direct via HTTP
local	*Unification*	local

Table 1. Overview over the approaches used to introduce resilience.

We argue, that both, RMI registry and RMI calls can be replaced by using a stateless protocol like HTTP and a persistent message broker system, which is replicated (sharded). Stateless protocols demand much less constrained environments than Java RMI does. It also can be inspected more easily for debugging and monitoring considerations.

With the introduction of a message broker system, the referencing to remote instances can be modeled just as with an RMI registry. The advantages are, however, that using general message brokers can be accessed more flexibly and do not require a specific running Java application. It also persists the net instance information and does not require ongoing communication to the service hosting the remote net instance. To differentiate between separate simulations, we propose to apply a UUID[5] identifier to each simulation upon start. The identifier is then passed to new simulation participants using the infrastructure (see RENEWKUBE [16]), which is not addressed here. To alleviate possible failures of the message broker, a replicated and sharded system like e.g. Apache Kafka[6] should be used.

Upon retrieval of remote net instances, the distributed binding search should be triggered over HTTP by directly contacting the RENEW instance, that the net instance is hosted on. This can be done in an easy fashion, as binding search is a read-only operation [7]. When a binding is found it can be fired. The DISTRIBUTE plugin uses RMI calls for this and by freezing the related parts, as described earlier, it effectively handles distributed transition firing in a similar manner, as a distributed commit protocol. This behavior, however, introduces the possibility for cascading failures, or more likely non-responsiveness of several

[5] Universally unique identifier: These can be generated locally and global collisions are as low as 1 in 2^{122} and by that negligible.

[6] https://kafka.apache.org/

[7] Note, that the internal implementation of the binding search in RENEW is complex and requires evaluation of possible bindings. It uses so-called state recorders to effectively create an overall read-only operation.

system components, once a badly timed outage happens during distributed firing. Therefore we argue to introduce an approach featuring eventual consistency and availability orientation.

One of these approaches is the Saga pattern [7], which rather recently found its revival in the context of microservices [14]. We can utilize Petri net Sagas to implement distributed transition firing, which is discussed in the separate publication "Petri Net Sagas" within these proceedings of PNSE 2021.

This overall concept covers the challenges introduced by the DISTRIBUTE plugin but external services are yet to cover. Upon relying on external services, the central processes of the simulator should never directly rely on the availability of the service or the integrity of the delivered data. Therefore we argue to introduce a convention to apply so-called Anti-Corruption-Layer (ACLs) in front of external services. These encapsulate the interface towards the service and behave in a deterministic and foreseeable way towards the core implementations. The beneficial aspects of this are stronger encapsulation of the external behavior and resilience against potentially malicious behavior of the service.

5 Implementation

As outlined in the conceptual section, the implementation of the cloud nativity aspects requires the introduction of new functionality, as well as adjustments to existing functionality. Adjustments mainly address the DISTRIBUTE plugin and resilience considerations along with integration problems with message broker systems. The implementation of both would exceed the scope of a single paper. Therefore, we only present an implementation for the introduction of new functionality, mainly regarding observability and operability. These are also the aspects, that serve as a foundation for the introduction of resilience later on.

Since the implementation of a web service is quite complex, we decided to use the Spring framework for our prototype. The procedure and problems that occurred, as well as their solutions, are described in the following sections.

5.1 Spring Framework and Dependency Injection

Java Spring is a framework that is often used for web applications. In addition, Spring Boot is a variant of the framework that allows a "ready to run" version, with minimal features for web functionality, to be set up quickly. This further reduces the already shortened implementation time. Besides the faster implementation time, it is also much easier to integrate a Spring context into RENEW than to implement all elements needed for a minimal web application. More on that in the "Integration with RENEW " section. Because of the minimal version of Spring we are using, we can easily integrate all the features that Spring provides and add only the functionality we need. A currently used extension is Spring Boot Admin, which provides a control panel and an interface for health metrics. Other interesting extensions offered by the framework include database integration and Object-Relation-mapping through Hibernate, as well as an integrated cloud architecture.

Java Spring also allows us to use the concept of dependency injection. This concept gives the possibility of providing a kind of blueprint for objects of a class that we can then pass to the required location. If an object of this specific class is needed, one can be inserted by injection of the required object [6]. The framework provides automated initialization of objects by accessing the initialization definition provided by the developer of the corresponding class. Thus, users of an object of the class do not need to know how to initialize that object. We have chosen the form of constructor injection for the implementation because it allows us to see how and where objects are used and initialized easier. In addition, the use of dependency injection ensures that we achieve an increase in the decoupling of the components, which is in line with the concept of cloud nativity.

5.2 Integration with Renew

As mentioned before we used the Java Spring framework which we integrated into Renew. Figure 2 shows that we put Spring in a separate module layer to create a system where Spring is integrated in Renew. This way we can guarantee

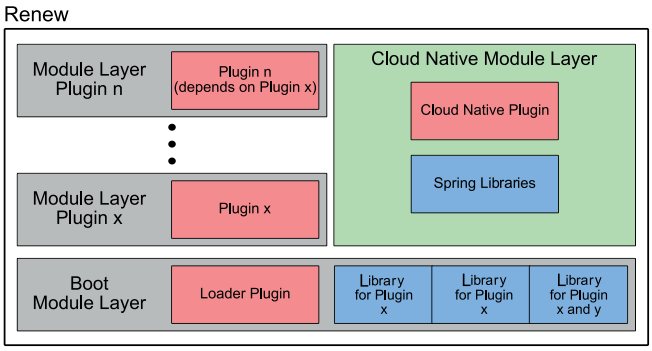

Fig. 2. Renew Module Layer

that all Spring functionalities that do not support a module system can work without any problems.

In our attempt to integrate Spring with Renew, we created a new plugin and added a new Spring Boot project to it. After some adjustments to the module properties according to the Renew specifications, we were able to identify two core problems.

Primarily all Spring modules and libraries were not present in the project. The problem comes from the fact that the Java module system is not thoroughly integrated into Spring and this conflicts with the modular order given by Renew. Furthermore, Spring Boot creates a fat jar that contains all dependencies and necessary libraries if Spring is a standalone application. However, the Renew

loader can do little with it. To integrate the contents in the build process of Renew we made adjustments to the location settings, where we were able to place the contents of the fat jar separately where Renew organizes the modules and libraries.

The other problem occurs after deploying the required modules and libraries, Spring does not have access to them. Due to the module layer structure, the dependencies for Spring and our module are on different layers, and thus cannot have access to each other. Our solution was to implement a new module layer only for Spring and its required modules, as shown in figure 2.

Since for many internal Renew operations, certain outputs or errors are issued for certain behavior, we had to introduce a system in which we can react to various internal events. Generally, we use the HTTP status codes, as well as a JSON response with more detailed information as return value for each request. We obtained this by implementing our own response class that provides reliable responses independent of the internal Renew process.

To fulfill observability aspects, we implemented a log output and a status overview as a health metric. For the implementation of the log output, we redirected the two Java output streams System.out and System.err, and formatted the contents of the streams for the respective endpoints. The health metrics are provided by Spring Actuator and include simple information about the system where Spring, i.e. Renew, is running on. After some additions to this given implementation, it is now possible to receive information like memory, CPU usage, RAM usage, and their corresponding averages about the host system. Furthermore, we also provide an overview of all loaded Renew plugins and created another plugin that acts as a communicator between all other Renew plugins and Spring. This new plugin gives all components the ability to display a message in the health metric. We decided to create a new plugin for this because if we had provided this functionality in our cloud native layer, there could be a circular dependency on plugins in the module system. Another solution would have been to integrate it directly into the Spring plugin, but then Spring would need to know the implementation details of the plugins it uses. However, both other approaches are in contradiction to the architecture of Renew.

In consideration of the operability aspect, we have created a possibility to upload reference nets, as well as a way to control these nets. When uploading the nets, we accept a file and check whether it is the correct file format by attempting to parse its shadow net structure. If this is the case, the file is deposited in the appropriate path. We also implemented an endpoint to upload additional Renew plugins, which then can be loaded into the running simulator at runtime. As outlined earlier, we do not check for malware yet in both cases. For the control of the simulation, we currently provide four commands: "run" for starting a simulation, "step" for stepping a simulation, "stop" for stopping a simulation, and "term" for terminating a simulation. Here we used the given functionality of the simulator plugin in Renew and mapped them to the corresponding HTTP endpoint. A key problem is that at the time of receiving the request and sending a corresponding response, only the submission of the command into the simulator

can be guaranteed, due to the multi-threaded and concurrent behavior of the simulation core of Renew.

6 Evaluation

For evaluation purposes, the implementation was run on a physical machine running Windows 10 20H2 with 32 GB memory and an AMD Ryzen 9 3900X 12-core CPU. The specific evaluation criteria introduced in section 3 were used. As motivated earlier, we have split the implementation between the cloud native Renew plugin and the upgrades to the Distribute plugin. As the latter is addressed in another upcoming contribution, the evaluation also only covers the aspects realized within the cloud native Renew plugin itself. The implementation of the cloud native Renew plugin can also be downloaded from the projects website[8]. An overview of the fulfilled requirements can be found in table 2.

Requirement	1	2	3	4	5	6	7	8	9	10
Fulfilled?	✓	✓	✓	✓	✓	✓	✓	(✓)	(✓)	(✓)

Table 2. Overview over the fulfilled requirements.

With the implementation, the simulation can be run on a server. It is possible to use the endpoint "/simulation/start" to start a simulation over HTTP. It is not required to have a local installation of Renew or Java to do so, but it is still necessary to have any HTTP client installed. We thereby meet requirement 1. The simulator also supports control of the simulation via the respective endpoints. Commands to control the simulation can be submitted via HTTP, which fulfills requirement 4. Further, it is possible to upload new net definitions as shadow net system, which subsequently can be instantiated with the start simulation endpoint, fulfilling requirement 5. Additionally, Renew plugins can be uploaded and loaded during runtime through the web interface, fulfilling requirement 6.

Health metrics can indicate the correct functioning of the simulator. We enriched these with environment information (meeting requirement 3) and a means for different parts of the application to present custom status data. As each simulator implements these endpoints, we can use a centralized monitoring system to access all states simultaneously, fulfilling requirement 2. For our evaluation, we ran the tool-suite Spring Boot Admin to consume the endpoints emitted by the simulators. The tool-suite corresponds to the "Control Panel" in figure 1. A view of the user interface of Spring Boot Admin for our simulator is visible in figure 3.

[8] https://paose.informatik.uni-hamburg.de/paose/wiki/CloudNativeRenew

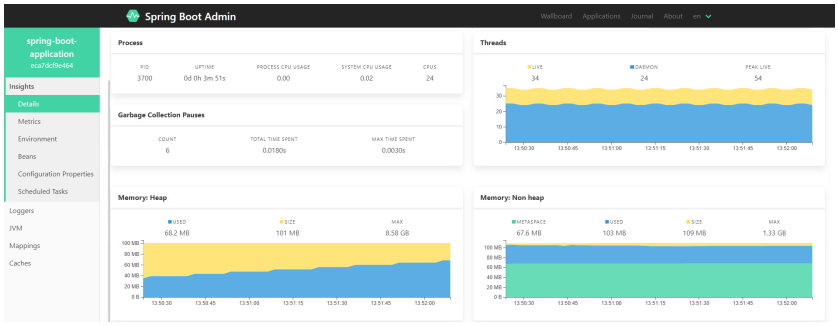

Fig. 3. Spring Boot Admin overview of a cloud native Renew instance

The evaluation of the agility aspects of requirement 7 is not straightforward, as it requires other dependent projects. Mainly the contribution presented in [16] (RenewKube) supplied means to run Renew simulations containerized and in an organized fashion with control from within the reference net simulation. Throughout the development CI/CD by Gitlab[9] has been used, as well as a Scrum-based development approach.

The remaining requirements can be evaluated in regards to the cloud native Renew plugin on its own, but are more interesting when applied to the upcoming redesign of the Distribute plugin, as it is the central part, that is in need of resilience optimizations. However, for completeness, we consider them in the context of the cloud native Renew plugin as well. As the endpoints are emitted regardless of other components, failures (including the control panel) do not affect other components (requirement 8) and the simulation (requirement 9). As there are no incorporated external services yet, requirement 10 is fulfilled trivially.

Overall we could achieve a very solid detachment from local system access, resulting in a reference net simulator, that is well suited to be run in cloud environments. Upon completion of the remaining projects addressed throughout the evaluation and paper, a thorough evaluation in a real cloud environment can be conducted. The outlook section 7 will point to follow-up research topics in this regard.

6.1 Limitations

All shown limitations refer to our implementation. When using the simulation controls, there is currently no feedback about the success or failure of the operation. This is due to the concurrent nature of the simulator and the fact, that HTTP server calls should not be implemented in a blocking fashion. Later this can be solved using polling, webhooks, or websockets.

[9] https://about.gitlab.com/

Detailed feedback about the simulation run is also not yet available in the shape of a simulation feed. This also has implications on the DISTRIBUTE plugin, as distributed firing and recording of firings share similarities and should be implemented together.

There is also no cleanup mechanic implemented yet. Therefore very long runs of plugins might require restarting of the containerized application to free up space. This can be achieved using RENEWKUBE [16], but is not yet built into the application itself.

6.2 Discussion

Overall the contribution resembles groundwork for scalable applications, by having a simulator, that can be run without machine-local interaction and without need to access the underlying system. These aspects enable all kinds of automation, by accessing respective endpoints. While introducing an increased complexity, the interfaces are now much more streamlined and consumable by a variety of tools.

Within the prototype implementation new dependencies on third party libraries have been introduced, to severely speed up the development process and to avoid repeated work on solved problems at the risk of the possibility of discontinuation of said dependencies. The method also introduces an overhead in generating messages on the universal interface. However, as it also paves the way for scabale applications, this effect is expected to be compensated by the now possible sizes of reference net applications.

In the larger context of the multi-agent applications mentioned in the motivation section, the cloud native plugin for RENEW will serve as essential platform extension to enable interoperability with an abstract platform management (compound) system.

7 Conclusion

The conceptual integration of the cloud nativity paradigm into the RENEW simulator for reference nets was presented. While cloud nativity is a heavily used term, especially in marketing departments, the here used interpretation is based on the concepts of operability, observability, agility, and resilience. Necessary changes to the architecture have been motivated and include the introduction of an HTTP-based interface for monitoring and control features, the usage of agile development, deployment via CI/CD, and usage of containerization. An implementation of the aspects relevant to the here presented "cloud native plugin" for RENEW has been presented. It includes most of the observability and operability aspects. Within the evaluation, it was shown, that the implementation met all the relevant requirements. Overall we could provide a future proof improvement over the dated RENEW Remoting plugin and related solutions.

Outlook

Further work will include the implementation of the aspects concerning other projects with relations to cloud native Renew, especially those regarding the Distribute plugin and resilience aspects. Additionally, overall integration of all related projects (Resilient Distribute, Petri net Sagas, RenewKube [16]) with Cloud Native Renew is of interest. In addition, the next step is to deploy the prototype in a common environment such as AWS or Azure to ensure that the experimental approach works in a real cloud environment. Other open questions address the safety of remote code execution operations regarding net and plugin signatures.

References

1. Bernine, N., Nacer, H., Aïssani, D., Alla, H.: Towards a performance analysis of composite web services using petri nets. Int. J. Math. Oper. Res. **17**(4), 467–491 (2020). https://doi.org/10.1504/IJMOR.2020.110847
2. Bhandari, G.P., Gupta, R.: Fault diagnosis in service-oriented computing through partially observed stochastic petri nets. Service Oriented Computing Applications **14**(1), 35–47 (2020). https://doi.org/10.1007/s11761-019-00279-5
3. Buchs, D., Klikovits, S., Linard, A., Mencattini, R., Racordon, D.: A model checker collection for the model checking contest using docker and machine learning. In: Khomenko, V., Roux, O.H. (eds.) Application and Theory of Petri Nets and Concurrency - 39th International Conference, PETRI NETS 2018, Bratislava, Slovakia, June 24-29, 2018, Proceedings. LNCS, vol. 10877, pp. 385–395. Springer (2018). https://doi.org/10.1007/978-3-319-91268-4_21
4. Duvigneau, M.: Konzeptionelle Modellierung von Plugin-Systemen mit Petrinetzen. Dissertation, Universität Hamburg, Department Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg (Oct 2009), `https://ediss.sub.uni-hamburg.de/handle/ediss/3023`
5. Entezari-Maleki, R., Etesami, S.E., Ghorbani, N., Niaki, A.A., Sousa, L., Movaghar, A.: Modeling and evaluation of service composition in commercial multiclouds using timed colored petri nets. IEEE Trans. Syst. Man Cybern. Syst. **50**(3), 947–961 (2020). https://doi.org/10.1109/TSMC.2017.2768586
6. Fowler, M.: Inversion of control containers and the dependency injection pattern (january 2004), `https://martinfowler.com/articles/injection.html`
7. Garcia-Molina, H., Salem, K.: Sagas. In: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data. pp. 249–259. SIGMOD '87, ACM, New York, NY, USA (1987). https://doi.org/10.1145/38713.38742
8. Garrison, J., Nova, K.: Cloud Native Infrastructure: Patterns for Scalable Infrastructure and Applications in a Dynamic Environment. O'Reilly Media, Inc., 1st edn. (2017)
9. Kilian, T.: Entwicklung einer modularen JavaScript-Bibliothek zur Modellerstellung. Bachelorarbeit, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg (2019)
10. Kummer, O.: Referenznetze. Logos Verlag, Berlin (2002), `http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=0035&lng=eng&id=`

11. Kummer, O., Wienberg, F., Duvigneau, M., Köhler, M., Moldt, D., Rölke, H.: Renew – the Reference Net Workshop. In: Veerbeek, E. (ed.) Tool Demonstrations. 24th International Conference on Application and Theory of Petri Nets (ATPN 2003). International Conference on Business Process Management (BPM 2003). pp. 99–102. Department of Technology Management, Technische Universiteit Eindhoven, Beta Research School for Operations Management and Logistics (Jun 2003)
12. Lacheheub, M.N., Hameurlain, N., Maamri, R.: Resources consumption analysis of business process services in cloud computing using petri net. J. King Saud Univ. Comput. Inf. Sci. **32**(4), 408–418 (2020). https://doi.org/10.1016/j.jksuci.2019.08.005
13. Moldt, D., Röwekamp, J.H., Simon, M.: A simple prototype of distributed execution of reference nets based on virtual machines. In: Bergenthum, R., Kindler, E. (eds.) Algorithms and Tools for Petri Nets Proceedings of the Workshop AWPN 2017, Kgs. Lyngby, Denmark October 19-20, 2017. pp. 51–57. DTU Compute Technical Report 2017-06 (2017)
14. Richardson, C.: Microservices Patterns: With Examples in Java. Manning Publications (2019)
15. Röwekamp, J.H.: Investigating the java spring framework to simulate reference nets with Renew. pp. 41–46. No. 2018-02 in Reports / Technische Berichte der Fakultät für Angewandte Informatik der Universität Augsburg (2018), `https://opus.bibliothek.uni-augsburg.de/opus4/41861`
16. Röwekamp, J.H., Moldt, D.: RenewKube: Reference net simulation scaling with Renew and Kubernetes. In: Donatelli, S., Haar, S. (eds.) Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11522, pp. 69–79. Springer (2019). https://doi.org/10.1007/978-3-030-21571-2_4, `https://doi.org/10.1007/978-3-030-21571-2`
17. Röwekamp, J.H., Moldt, D., Feldmann, M.: Investigation of containerizing distributed Petri net simulations. In: Moldt, D., Kindler, E., Rölke, H. (eds.) Petri Nets and Software Engineering. International Workshop, PNSE'18, Bratislava, Slovakia, June 25-26, 2018. Proceedings. CEUR Workshop Proceedings, vol. 2138, pp. 133–142. CEUR-WS.org (2018), `http://ceur-ws.org/Vol-2138/`
18. Simon, M., Moldt, D.: Extending Renew's algorithms for distributed simulation. In: Cabac, L., Kristensen, L.M., Rölke, H. (eds.) Petri Nets and Software Engineering. International Workshop, PNSE'16, Toruń, Poland, June 20-21, 2016. Proceedings. CEUR Workshop Proceedings, vol. 1591, pp. 173–192. CEUR-WS.org (2016), `http://CEUR-WS.org/Vol-1591`
19. Valk, R.: Petri nets as token objects - an introduction to elementary object nets. In: Desel, J., Silva, M. (eds.) 19th International Conference on Application and Theory of Petri nets, Lisbon, Portugal. pp. 1–25. No. 1420 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, New York (1998). https://doi.org/10.1007/3-540-69108-1_1