HAWKS: A System for Highly Available Executions of Workflows

David Richard Schäfer[1] Andreas Weiß[2] Muhammad Adnan Tariq[1] Vasilios Andrikopoulos[2]
Santiago Gómez Sáez[2] Lukas Krawczyk[1] Kurt Rothermel[1]
[1] *Institute of Parallel and Distributed Systems (IPVS)*
{*david.schaefer, adnan.tariq, lukas.krawczyk, kurt.rothermel*}*@ipvs.uni-stuttgart.de*
[2] *Institute of Architecture of Application Systems (IAAS)*
{*andreas.weiss, vasilios.andrikopoulos, santiago.gomez-saez*}*@iaas.uni-stuttgart.de*
University of Stuttgart, Stuttgart, Germany

***Abstract*—The workflow technology is the de facto standard for managing business processes. Today, workflows are even used for automating interactions and collaborations between business partners, e.g., for enabling just-in-time production. Every workflow that is part of such a collaboration needs to be highly available. Otherwise, the business operations, e.g., the production, might be hindered or even stopped. Since today's business partners are scattered across the globe, the workflows are executed in a highly distributed and heterogeneous environment. Those environments are, however, failure-prone and, thus, providing availability is not trivial. In this work, we improve availability by replicating workflow executions, while ensuring that the outcome is the same as in a non-replicated execution. For making workflow replication easily usable with current workflow technology, we derive the requirements for modeling a workflow replication system. Then, we propose the HAWKS system, which adheres to the previously specified requirements and is compatible with current technology. We implement a proof-of-concept in the open-source workflow execution engine Apache ODE for demonstrating this compatibility. Finally, we extensively evaluate the impact of using HAWKS in terms of performance and availability in the presence of failures.**

***Keywords*-SOA; workflows; availability; replication; performance;**

I. Introduction

Workflows have gained enormous importance for today's businesses [1], [2]. Healthcare [3], logistic [4], manufacturing, and urban mobility [5] are only few of many areas, where workflows are being used. Especially with the advances of choreographies [6], where complex interactions of many workflows can be managed in an intuitive manner, workflows are used by businesses on a global scale. On the downside, this means that the delay or unavailability of one of the workflow executions hinder the execution of all workflows that participate in the choreography. As a result, highly available workflow executions are fundamental for today's businesses. A single hour of unavailability implies a typical cost of up to $6.48 million [7].

Today's choreographies often comprise multiple business partners scattered across the globe, implying that the participating workflows are executed in a highly distributed and heterogeneous environment. In such environments, failures occur frequently making it challenging to ensure high availability. Even wired networks, usually considered reliable, are failure prone. For example, the IP Backbone experiences failures on median every 3000 s, where failures last for $2-1000$ s [8], [9]. A Microsoft study found that even data centers experience 40.8 network failures with end-user impact per day [8], [10]. Cloud providers like Amazon and Google confirm that network partitioning must be considered when designing network related systems [8], [11].

Moreover, with the advent of mobile devices, businesses are shifting to become more customer-centric. Customers are given the possibility to influence business operations for enabling personalized products and customer-tailored services, strengthening customer loyalty. In these cases, workflows not only interact with each other but also with customers, implying an even stronger need for high availability. Google reported that delaying the reply to the user by only 500 ms led to 25 % less usage of their services with long term impact [12]. Thus, it is desirable to design highly available systems providing reliable workflow executions.

Because of its high relevance, increasing availability and failure tolerance of workflows has already been addressed in the past. For enabling fault-tolerant workflows, failure handling and recovery mechanisms are explicitly specified in the workflow model [13]. For example, if an activity fails or its effects need to be reversed, a compensation handler specified for this activity is executed. Sometimes it is not possible to compensate an activity in isolation. Therefore, compensation scopes can encapsulate multiple activities, where the compensation handler reverses the effects of all the contained activities [14]. However, these approaches cannot mask failures of the computing nodes on which the workflows are executed. In case of a node failure, the workflow execution only recovers and continues once the underlying node has recovered. Thus, the workflow execution does not progress until recovery, which renders the workflow execution unavailable. Consequently, these approaches are infeasible for ensuring high availability.

In distributed systems, redundancy is a commonly used technique for increasing availability in the presence of failures by replicating the same functionality on multiple computing

Published in Proceedings of 2016 IEEE International Conference on Services Computing (SCC), 2016, San Francisco, California, USA

nodes in the system [15]. Unfortunately, the replication concept cannot be directly applied to workflows. Consider that multiple computing nodes execute an instance of the same workflow model. This ensures that even if some of these computing nodes fail, the workflow is still executed. However, if the workflow, for example, contains a payment activity, the replicated execution will pay multiple times – increasing the total cost – instead of paying just once as intended by the workflow designer. Thus, the available replication techniques are not suitable for workflows. For using replication, we need to ensure that the outcome of a replicated workflow execution is identical to a non-replicated workflow execution [16].

In this paper, we combine the availability concepts of workflow technology with the replication techniques of distributed systems for ensuring that the outcome is identical to a non-replicated workflow execution. First we provide a formal model for workflows (Section II). We then discuss how replication can be used to increase availability in workflow systems (Section III). Based on this discussion we extract the requirements that a workflow replication system needs to fulfill to enable highly available workflow executions, and subsequently, we present the HAWKS system adhering to the defined requirements (Section IV). The HAWKS system is compatible with existing workflow technology, which we demonstrate by implementing a proof of concept HAWKS prototype in the open-source industry-level workflow engine Apache ODE (Section V). We extensively evaluate the implementation with respect to the impact of using replication on the performance in a cluster of engines (Section VI), before discussing related work (Section VII), and concluding with some future work (Section VIII).

II. Workflow Model

For understanding the challenges arising with the replication of workflows, we first describe the workflow model. A workflow model is defined by a directed acyclic graph $G = (A, L, \Sigma)$, where A is the set of activities, L is the set of links, and Σ the internal state containing the variables needed during execution [14]. The activities $a \in A$ define atomic pieces of work. The control logic is defined by the links L. Each link $l \in L$ is specified by a tuple $L : A \times A \times T$, where T is the set of transition conditions. Thus, the link $l = (a_{start}, a_{end}, t)$ originates from a_{start}, points to a_{end}, and is only triggered if a_{start} was already executed and the transition condition t is fulfilled. For simplicity, we will not consider loops in this work.[1] The internal state Σ contains the variables of the workflow, which can be read and written by the activities during execution.

For executing the workflow, the workflow engine creates an instance $g = (A^i, L^i, \sigma)$ of the workflow model G. The set $A^i = \{(id, a, s) | id \in ID, a \in A, s \in S\}$ contains all activity instances, where $S = \{Ready, Executing, Completed\}$ specifies the execution state of the activity instance. Each activity instance $a^i \in A^i$ is an instance of $a \in A$ that is identified by an id and has an execution state s. The set of links $L^i = \{(l, c) | l \in L, c \in \{true, false\}\}$ defines whether an link is evaluated to $true$ or $false$. The internal state σ is a tuple of the values of all variables. In other words, σ is a snapshot of the current internal state. Each activity $a^i \in A^i$ might have read or write access to the variables of σ depending on the data flow. However, we omit discussing the data flow because it does not impact the concepts of this work. Instead, we assume that each activity can read or write all variables of σ and thereby can transfer the internal state σ_j into a new internal state σ_k, which is denoted by $\sigma_j \rightarrow_{a^i} \sigma_k$. Also, the activity execution might interact with services or other processes by sending or receiving messages.

For reversing the effects of an activity execution, we assume that each activity either has a defined compensation handler or is embedded in a compensation scope that has a defined compensation handler [14]. Since each activity can interact with interaction partners, such as another workflow or a service, the activity execution might have changed state which is not in direct control of the workflow – called the *external state*. Executing the compensation handler of an activity includes (semantically) reversing the effects on the external state. Note that this workflow and execution model is compatible with the WS-BPEL standard[2] and generally compatible to any graph-based workflow description language. Thus, the techniques we present in the following are widely applicable to current workflow technology.

III. Highly Available Workflow Executions

In order to achieve high availability, we can in principle instantiate a workflow model on multiple computing nodes. By executing these replicas of the workflow model in parallel, we can mask failures because if one of the nodes is failed or partitioned from the network, the other replicas continue the execution. However, the replicated execution should affect the external state as a non-replicated execution would have affected the external state.

In specific, a non-replicated workflow execution gets input by receiving messages (denoted as *reads* in Fig. 1) and produces output by sending messages to interaction partners, which write to external state (denoted as *writes*). Not all interaction partners write to external state when they receive a message (denoted as *non-writes*), e.g., stateless services. Consequently, when a replicated execution receives the same input (reads) as a non-replicated execution, it should produce the same output (writes) as the non-replicated execution. Then, we call the replicated execution *consistent* [16]. In the following, we discuss how to achieve this consistency.

[1] Loops can be supported through extending the replication protocol. We, however, leave the discussion of such an extension for future work.

[2] http://docs.oasis-open.org/wsbpel/2.0/

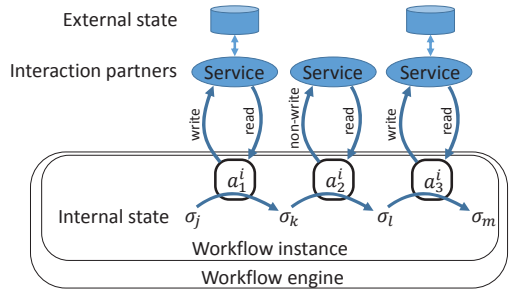

Figure 1. Executions of activities, which synchronously interact with services depicting the effects on internal and external state.

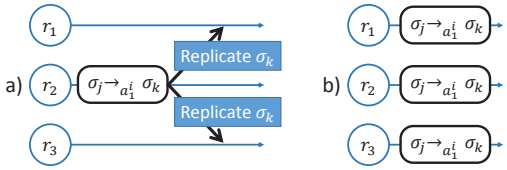

Figure 2. Types of replication: a) Passive replication b) Active replication

In general, we use two replication mechanisms: *passive* and *active* replication (cf. Fig. 2). With passive replication, only one engine, called *master*, executes the activity and replicates the produced internal state to all other replicas. Thereby, all replicas have the same internal state, which means they are synchronized. This synchronization, however, causes a bandwidth overhead as well as a time delay for replicating the state. With active replication, all engines execute the activity. Hence, we save the overhead of synchronizing the replicas because each replica produces the new internal state itself. Unfortunately, active replication might violate the consistency depending on how the executed activity modifies the internal and external state. Therefore, we classify the activities based on how they change the internal and external state.

The external state can by changed by an activity execution if the activity sends a message to an interaction partner. For example, when executing a payment activity, the activity interacts with a payment service, which changes (i.e., writes to) the account balance. All activities which potentially change the external state are called *write* activities. Using active replication for write activities causes inconsistencies because the payment would be performed multiple times, which would not happen for a non-replicated execution. However, as discussed above, not every interaction partner is writing to the external state. For example, reading the account balance through using a banking service does not change the external state. Moreover, an activity might not interact at all or might only receive a message. Such activities are obviously also not writing to the external state. Thus, we summarize all these activities as *non-write* activities. Because non-write activities do not write to the external state, non-write activities can potentially be actively replicated.

However, there is a second classification criteria which has to be taken into account, which specifies how an activity modifies the internal state. Consider an activity for booking a bus seat calling a reservation service. Calling the service with the same message multiple times might result in different replies from the service because a bus has a limited number of seats. Thus, when the replicas use the same internal state for executing the activity, they might produce different internal states. Such activities are considered to be *non-deterministic*. Actively replicating non-deterministic activities can produce diverging internal state on the replicas, e.g., by getting different input from interaction partners. Then, each replica that executes subsequent activities might affect the external state differently. We, however, want to produce the same writes on the external state as one non-replicated execution. This means, we could only use one of the (diverged) replicas further rendering the replication useless. In contrast, assume calculating the sum of all expenses of an investment. When this activity is executed multiple times with the same internal state (the expenses), the sum will always be the same. Such activities are called *deterministic*.

Algorithm 1: The basic protocol functionality, which is running on each replica

```
1 a_c ← NextActivityToExecute();
2 if a_c.isDeterministic() and a_c.isNonWrite() then
3 |   σ ← execute(a_c, σ);
4 else if engine is master then
5 |   σ ← execute(a_c, σ);
6 |   Synchronize(σ);
```

In conclusion, only deterministic non-write activities can be actively replicated without violating consistency. As shown in Alg. 1, all other activities are passively replicated. In general, this strategy allows the replication protocol to avoid inconsistencies. However, in case of passive replication, the master might fail or might be partitioned from the network. Then, the remaining replicas elect a new master for continuing the execution after detecting the failure (cf. Fig. 3). Election protocols are thoroughly researched and any election protocol might be used, e.g., [17]. The old master might have failed before synchronizing (i.e., before executing Alg. 1 line 6). Thus, the execution progress (and produced internal state) is not known during the election. This means the corresponding activity must be re-executed by the new master. To avoid inconsistencies, the execution of the failed master needs to be compensated after recovery. This is realized as follows: 1) the failed master recovers, finishes its execution, and tries to replicate the produced state, 2) the other replicas will reject the old activity execution because they already elected a new master, which is re-executing the activity, 3) upon receiving the rejection, the recovered master compensates its execution. Then, consistency is ensured again.

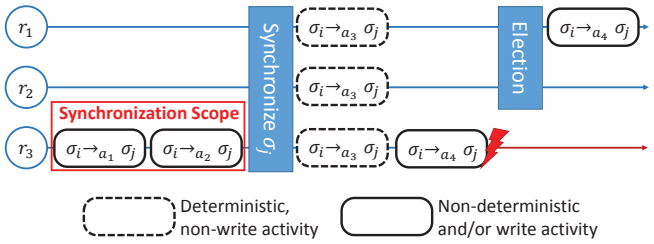

Figure 3. Replication protocol

Since the overhead of synchronizing the internal state after each activity might be very high (cf. Alg. 1 line 5-6), the proposed protocol allows to group consecutive activities into a *synchronization scope* (cf. Fig. 3). Then, the whole synchronization scope is executed before the state is synchronized (by adapting Alg. 1 line 5 accordingly). On the downside, more execution progress is lost when the master fails just before synchronizing. In this case, more activities need to be re-executed increasing the failover time, i.e., the time to regain the execution progress it had before the failure. Moreover, the failed master also has to compensate all activities of the synchronization scope after recovering increasing the compensation cost. By means of this approach, however, we provide the flexibility of letting the workflow designer decide whether more synchronization overhead or more recovery overhead (i.e., failover time and compensation cost) is desirable (cf. [16]).

In the following, we present an architecture for realizing the protocol efficiently with existing workflow technology.

IV. The HAWKS System

In this section, we provide the architecture for realizing highly available workflow executions with existing technology. The architecture is kept generic and, thus, can be used for adding the replication protocol to any existing workflow engine that is based on a graph-based workflow language. However, before we describe the architecture, we extract the requirements that the architecture needs to fulfill.

A. System Requirements

The system should fulfill several requirements regarding the generality of its applicability. The system especially should be easily usable and, thus, work with as few manual interventions by users as possible. In specific, we identified the following requirements:

Automated deployment: The system must be able to automatically distribute a workflow model to several workflow engines. The number of workflow engines receiving the model has to be identical to the specified replication degree.[3]

[3]We assume that the desired replication degree is specified in the execution request. The replication degree might be specified by the workflow designer or calculated by a probabilistic model taking QoS properties into account. We can support any arbitrary method.

Automatic execution: The replicated execution of a workflow must be triggerable by sending a message to one endpoint of the system. Linking and synchronizing the workflow engines for the exchange of synchronization messages must be done automatically.

Scalability: The system has to be scalable. Thus, it must be possible to easily add and remove workflow engines, which can participate in the replicated execution of workflows.

Transparency: The replication must be transparent to any interaction partner of the workflow instance. Thus, other workflows participating in the choreography can use different replication degrees, or might not be replicated at all.

We realize these requirements in our **H**ighly **A**vailable **W**or**K**flow execution**S** (HAWKS) system, which we present in the following.

B. Architecture of the HAWKS System

As depicted in Fig. 4, the HAWKS system is a middleware solution consisting of the *Synchronization Units*, which are attached to each *Execution Engine*, and the *HAWKS Controller*. The Synchronization Unit is responsible for controlling the workflow executions in the engines according to the replication protocol. The HAWKS Controller is responsible for starting replicated executions and routing messages between the engines. In the following, we describe each component in detail.

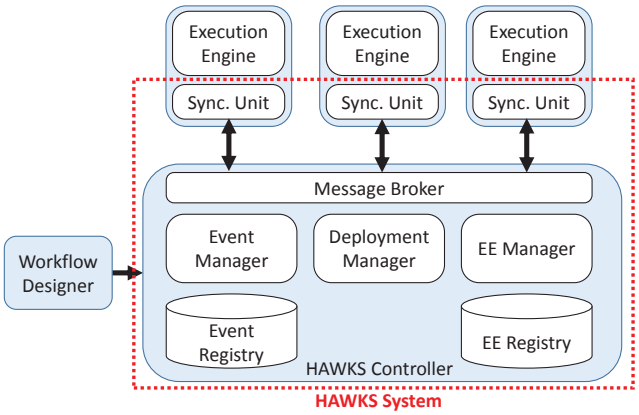

Figure 4. Architecture of the HAWKS system

1) Synchronization Unit: The Synchronization Unit runs the replication protocol and controls the execution in the Execution Engine accordingly. More specifically, the Synchronization Unit i) suspends and resumes the execution, ii) tracks changes of the internal state during execution, iii) sends the changes to the other replicas (by sending them to the Message Broker), iv) applies state changes it receives and skips the corresponding activities, and v) manages elections after failures. Thus, any existing workflow engine can be extended by a Synchronization Unit, which enables the usage

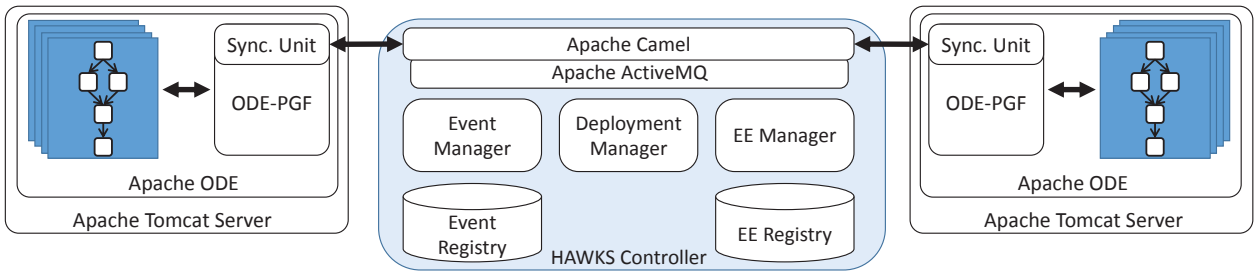

Figure 5. Overview of the prototypical implementation and used technology

of our protocol. In conclusion, the Synchronization Unit fulfills the automatic execution requirement.

2) Message Broker: The Message Broker is responsible for routing the messages between the Execution Engines such that the replicas of a workflow can interact. This is necessary because the Execution Engines do not know the other Execution Engines that participate in the replicated execution. The Message Broker also correlates the messages to the corresponding workflow instance (i.e., replica) of the Execution Engine (since the Execution Engine can execute multiple workflow instances in parallel). Additionally, the Message Broker is handling the communication of the replicas with the interaction partners. Thus, the replicas send the messages to the Message Broker, which then sends it to the interaction partner. Through this, the interaction partner perceives the replicas as a single endpoint satisfying the transparency requirement.

3) EE Manager and EE Registry: These components are responsible for managing the Execution Engines that can currently participate in a replicated workflow execution. On start-up, each Execution Engine registers itself at the HAWKS Controller. Therefore, the Synchronization Unit sends a registration message to the Message Broker. The Message Broker routes the message to the EE Manager. Then, the EE Manager stores the Execution Engine together with its endpoint reference in the EE Registry. The EE Manager is also responsible for checking if Execution Engines become unavailable. If an Execution Engine is unreachable, the EE Manager will (eventually) remove the Execution Engine from the EE Registry.

4) Deployment Manager: When a workflow, which was modeled in a Workflow Designer, is to be executed, an execution request containing the workflow model is sent to the Message Broker. Then, the Deployment Manager determines the Execution Engines that participate in the replicated execution based on the workload of the engines. More specifically, it selects the engines which have the lowest workload ensuring workload balancing (e.g., by monitoring the engines [18]). The Deployment Manager sends the workflow model to these engines and sets up the routing paths between the replicas in the Message Broker. The Deployment Manager satisfies the automated deployment requirement. Because the Deployment Manager selects the Execution Engines using workload balancing and the EE Manager and EE Registry allow to easily add more Execution Engines, the scalability requirement is fulfilled as well.

5) Event Manager and Registry: The Event Manager tracks events and saves these in the Event Registry. The tracking includes the start and end of a replicated execution. Thereby, the HAWKS Controller can identify started and not finished executions. Moreover, the tracking can be easily extended for further monitoring of the HAWKS system.

V. Proof of Concept

We show the applicability of the HAWKS system on existing workflow technology by extending the open-source workflow engine Apache ODE and the pluggable framework for extended BPEL (ODE-PGF), which is able to orchestrate workflows based on Web Service Business Process Execution Language (WS-BPEL).[4] WS-BPEL is standardized by OASIS and widely used in industry. The Apache ODE is itself running in an Apache Tomcat servlet container.

For realizing the communication between the engines, we use Apache ActiveMQ. Thus, the Synchronization Unit that is attached to Apache ODE is reading from and writing to a message queue for controlling the ODE such that the replicas are synchronized. For routing the synchronization messages, we use Apache Camel. We also use it for routing messages sent from interaction partners to the replicas. The HAWKS system's realization is depicted in Fig. 5.

For reducing the synchronization overhead, we do not include the complete internal state of the workflows in the synchronization messages. Instead, we track which variables are modified during activity executions and only send these modifications to the other replicas. In turn, the replicas check that they received and applied all preceding state updating synchronization messages. They can easily check this using the workflow model because each state update message contains the activity that produced the state. We validate the activity against the execution progress and do not apply the update if a preceding update is missing. Our implementation of HAWKS is open-source and available on GitHub.[5]

[4]http://ode.apache.org, http://camel.apache.org, http://activemq.apache.org, http://tomcat.apache.org, http://docs.oasis-open.org/wsbpel/2.0/, http://www.iaas.uni-stuttgart.de/forschung/projects/ODE-PGF/

[5]https://github.com/TheRingbearer/HAWKS/

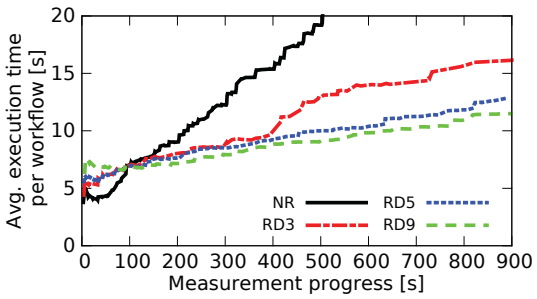

Figure 6. Average workflow execution time in the presence of failures

VI. Evaluations

We evaluate the HAWKS system by using up to 12 virtual machines (4 vCPUs, 8 GB RAM) running in OpenStack[6] for hosting the Apache ODE. The HAWKS Controller is distributed to 10 additional VMs. In the following, we evaluate the HAWKS system in terms of availability, replication overhead, and scalability. For all measurements, we used an exemplary workflow, which also can be found in the HAWKS GitHub repository. Overall we measured over 50,000 workflow executions.

For evaluating the availability, we assume that a failed engine can resume any started workflow execution after recovering from the failure. If the engine would not provide this recovery mechanism, any started workflow execution would be lost when an engine fails. Then, replication would obviously outperform a non-replicated execution. We instead show that replication is beneficial even when all engines provide this recovery mechanism.

For simulating failures, we fail one workflow engine of the system with the probability f every second. Initially, f is set to 0. Then the failure probability is increased by $\frac{1}{120}$ every 30 s. For controlling the mean time to recovery of a failed engine, we simulate failures by halting the engines. Our mean time to recovery is set to 10 s, which is short since typical failures last up to multiple days [19]. However, replication is obviously beneficial for long lived failures. In contrast, we want to show that replication is also beneficial in the presence of short lived failures. In this measurement, we use a stable workload of one request every 10 s, i.e., one workflow execution starts every 10 s.

In Fig. 6, the non-replicated execution (NR) has the lowest average workflow execution times when the failure probability f is low or zero. With this low f, increasing replication degrees imply higher synchronization overhead because the state has to be replicated on more replicas before continuing the workflow execution. Thus, higher replication degrees have higher average workflow execution times.

However, when the failure probability increases, NR shows the trend of a linear increase because every failure delays the execution by the mean time to recovery. The average workflow execution times are quickly higher than any of the replicated workflow executions. This means that even though NR has no synchronization overhead and, therefore, can execute more workflows during the failure-free phases, it still cannot compensate for the time it is failed. This shows that replication keeps the average execution times low in failure prone environments. Moreover, Fig. 6 also shows that higher replication degrees can tolerate more failures. Until 400 s of measurement progress (i.e., $f < \frac{13}{120}$) the execution time of replication degree 3 (RD3) increases slowly. This increase is due to the time needed for electing a new master when a master fails. However, there is a sudden increase at 400 s because f becomes so high ($f = \frac{13}{120}$) that a majority of replicas fails concurrently, which prevents the election of a new master. Thus, the workflow execution is halted until a majority recovered. However, both the replication degree 5 (RD5) and 9 (RD9) do not show this sudden increase even up to 900 s, i.e., $f = \frac{1}{4}$. In conclusion, higher replication degrees can tolerate more concurrent failures providing high availability in failure prone environments.

Above we have already seen that without failures the increased synchronization overhead of higher replication degrees increases the workflow execution time. The measurement, however, was specific to the used workload. Now, we will investigate the overhead further under different workloads without injecting failures. In specific, we double the workload every 300 s. Initially, we start one workflow execution every 60 s, then every 30 s, and so on. In Fig. 7a, the non-replicated workflow execution (NR) again has the lowest execution times because it has no synchronization overhead. Accordingly, the average workflow execution time increases with higher replication degrees. Additionally, we can observe that RD3 becomes overloaded around 700 s of measurement progress (starting a workflow every 15 s), RD5 at 1000 s (starting a workflow every 7.5 s), and RD9 at 1250 s (starting a workflow every 3.75 s). At some point any system – also NR – becomes overloaded when the workflows executions are started with a higher rate than they can be executed by the engines. However, the replicated executions become overloaded before a non-replicated execution because of the synchronization overhead. When comparing the replicated executions to each other, higher replication degrees become overloaded later. The reason is that the bottleneck of the replication protocol is the master which executes the activities and synchronizes the produced state when using passive replication. When using more replicas the load of being master can be distributed to more computing nodes.

Because HAWKS is designed to be scalable, we can add more engines for handling higher workloads without increasing the replication degree. For evaluating the scalability, we first evaluate the replication degree 3 with 3 engines (RD3E3), i.e., all engines execute a replica of every workflow. We also evaluate the setup of replication degree 3 with 6 engines (RD3E6) and with 12 engines (RD3E12). In

[6]https://www.openstack.org

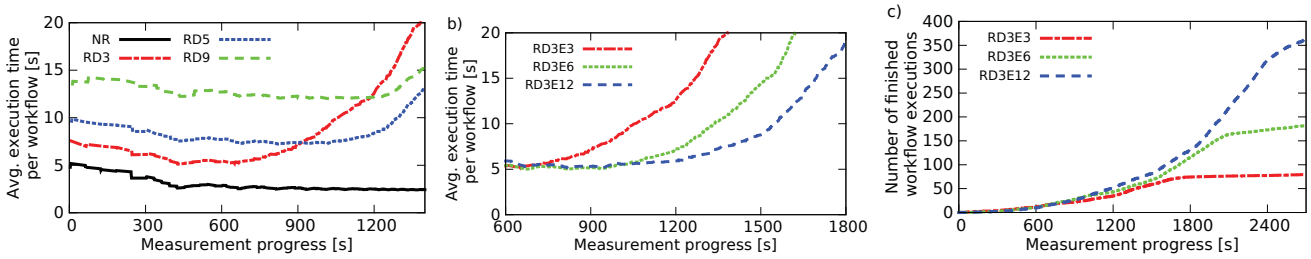

Figure 7. The figures show the replication overhead (a) and scalability with respect to execution time (b) and the number of finished workflows (c).

these cases, the workload can be distributed to the different engines. Fig. 7b shows that adding more engines improves the performance by delaying the point where the system becomes overloaded. While RD3E3 becomes overloaded at 700 s (starting a workflow every 15 s), RD3E6 becomes overloaded at 1000 s (starting a workflow every 7.5 s) and RD3E12 delays the point even to 1200 s (starting a workflow every 3.75 s). To make the effects of delaying this point clearer, we show the accumulated number of finished workflow executions in Fig. 7c. RD3E3 is overloaded after executing around 70 workflows, whereas RD3E6 and RD3E12 can handle 150 and 340 workflow executions before being overloaded. This clearly shows the scalability of HAWKS.

In conclusion, HAWKS ensures highly available workflow executions, while implying only small overhead for low replication degrees. Moreover, in a failure prone environment, the replication overhead is negligible.

VII. Related Work

Availability is an important property for any network related system. Therefore, there has been a lot of research targeting availability. One approach for providing availability for workflow executions is the dynamic composition of web services [20], where it is possible to use the availability of the individual services as a QoS property for selecting the services [21]. However, service composition approaches require a description of the desired functionality in a declarative language (e.g., [22]) and, thus, suffer from state explosion [23]. Additionally, declarative languages are not compatible with the imperative workflow languages, i.e., graph-based workflow languages, which are the predominant way of modeling workflows.

Other approaches improve availability by providing the functionality to switch from a used service that is failed to an alternative service [24], [25]. Some approaches even call multiple alternative services in parallel and use the first reply they receive and cancel or compensate the other executions [26], [27]. These approaches, however, only target masking service failures. The approaches cannot handle failures, where the computing node on which the engine is running fails or is partitioned from the network. On the other hand, HAWKS does not tackle service failures and, thus, incorporating the mentioned strategies in HAWKS can further improve the availability. Another solution would be to use HAWKS for executing the services (if these are modeled as workflows) making the services themselves highly available.

Data replication techniques usually consider the trade-off between consistency and availability [28], while workflow replication techniques do not consider this trade-off. Instead, some workflow replication techniques consider all activities to be not writing to external state, e.g., [29]. These approaches actively replicate activities and take the result of the first finished activity execution for continuing the workflow execution. Thereby, they completely avoid the need for considering consistency problems. However, workflows in general interact with services and other workflows implying that they can change the external state. Thus, such techniques can only be applied to few workflows. Other workflow replication techniques solely rely on passive replication [30], where the state is transferred to a backup system after each activity execution. However, these techniques imply overhead in terms of bandwidth for transferring the state after each activity execution, as well as a delay until the transfer is complete. Also, in case of a failure the triggered re-execution does not consider consistency and transparency for interacting with interaction partners. In this work, we explicitly specify how to achieve a consistent replicated execution. During the execution we allow temporal inconsistencies (when activities are re-executed in case of failures), which are resolved eventually. Thus, the achieved consistency is congruent to eventual consistency [31].

VIII. Conclusion

We presented the HAWKS system which enables highly available workflow executions by replicating the executions on multiple computing nodes. Therefore, we identified the requirements that the system needs to fulfill to be usable with existing workflow technology without manual interventions. We showed that the HAWKS system fulfills these requirements. Moreover, we implemented a prototype in the open-source workflow engine Apache ODE proving the applicability to state-of-the-art workflow technology. Finally, we evaluated the HAWKS system and showed that it induces only a small overhead. We also showed that the performance loss is outweighed by the gains in terms of availability in the presence of failures. In future work, we will extend the HAWKS system i) to support loops in the workflow model and ii) consider activities that are bound to specific workflow

engines (e.g., because of privacy concerns). We also plan an empirical study, where we will show the impact of replication in different scenarios and use-cases.

ACKNOWLEDGMENT

The authors would like to thank the European Union's 7th Framework Programme for partially funding this research through the ALLOW Ensembles project (600792).

REFERENCES

[1] D. Schäfer, T. Bach, M. Tariq, and K. Rothermel, "Increasing availability of workflows executing in a pervasive environment," in *SCC'14*, pp. 717–724.

[2] R. K. Ko, S. S. Lee, and E. W. Lee, "Business process management (bpm) standards: a survey," *Business Process Management Journal*, vol. 15, no. 5, pp. 744–791, 2009.

[3] H. Wolf, K. Herrmann, and K. Rothermel, "Flexcon robust context handling in human-oriented pervasive flows," in *OTM'11*, ser. LNCS. Springer, vol. 7044, pp. 236–255.

[4] ——, "Modeling dynamic context awareness for situated workflows," in *OTM'09 Workshops*, ser. LNCS. Springer, vol. 5872, pp. 98–107.

[5] K. Herrmann, K. Rothermel, G. Kortuem, and N. Dulay, "Adaptable pervasive flows - an emerging technology for pervasive adaptation," in *SASO'08 Workshops*, pp. 108–113.

[6] G. Decker, O. Kopp, and A. Barros, "An Introduction to Service Choreographies," *Information Technology*, vol. 50, no. 2, pp. 122–127, 2008.

[7] V. Solutions, "Assessing the financial impact of downtime - understand the factors that contribute to the cost of downtime and accurately calculate its total cost in your organization." White Paper, 2008.

[8] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *VLDB'13*, vol. 7, no. 3, pp. 181–192.

[9] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, Y. Ganjali, and C. Diot, "Characterization of failures in an operational ip backbone network," *IEEE/ACM Trans. Networking*, vol. 16, no. 4, pp. 749–762, 2008.

[10] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," *SIGCOMM'11*, vol. 41, no. 4, pp. 350–361.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS'07*, vol. 41, pp. 205–220.

[12] M. Mayer, "In search of... a better, faster, stronger web," Keynote at O'Reilly Velocity: Web Performance and Operations Conference 2009.

[13] N. Russell, W. van der Aalst, and A. ter Hofstede, "Workflow exception patterns," in *CAiSE'06*, ser. LNCS. Springer, vol. 4001, pp. 288–302.

[14] F. Leymann and D. Roller, *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 2000.

[15] B. Charron-Bost, F. Pedone, and A. Schiper, Eds., *Replication: Theory and Practice*, ser. LNCS. Springer, 2010, vol. 5959.

[16] D. R. Schäfer, M. A. Tariq, and K. Rothermel, "Highly availabile process executions," Institute of Parallel and Distributed Systems, University of Stuttgart, Tech. Rep. 2016/02, 2016.

[17] B. Awerbuch, "Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems," in *STOC '87*, 1987, pp. 230–240.

[18] B. Wetzstein, D. Karastoyanova, O. Kopp, F. Leymann, and D. Zwink, "Cross-organizational process monitoring based on service choreographies," in *SAC'10*, pp. 2485–2490.

[19] P. Bailis and K. Kingsbury, "The network is reliable," *ACM Queue*, vol. 12, no. 7, pp. 20:20–20:32, 2014.

[20] I. Abbassi, M. Graiet, S. Boubaker, M. Kmimech, and N. Ben Hadj-Alouane, "A formal approach for verifying qos variability in web services composition using event-b," in *ICWS'15*, pp. 519–526.

[21] L. Sun, S. Wang, J. Li, Q. Sun, and F. Yang, "Qos uncertainty filtering for fast and reliable web service selection," in *ICWS'14*, pp. 550–557.

[22] M. Pesic, H. Schonenberg, and W. van der Aalst, "Declare: Full support for loosely-structured processes," in *EDOC'07*, pp. 287–287.

[23] A. P. Sistla and E. M. Clarke, "The complexity of propositional linear temporal logics," *J. ACM*, 1985.

[24] D. Karastoyanova, A. Houspanossian, M. Cilia, F. Leymann, and A. Buchmann, "Extending bpel for run time adaptability," in *EDOC'05*, pp. 15–26.

[25] P. Melliar-Smith and L. Moser, "Conversion infrastructure for maintaining high availability of web services using multiple service providers," in *ICWS'15*, pp. 759–764.

[26] T. Bach, M. Tariq, B. Koldehofe, and K. Rothermel, "A cost efficient scheduling strategy to guarantee probabilistic workflow deadlines," in *NetSys'15*. IEEE, pp. 1–8.

[27] S. Stein, T. Payne, and N. Jennings, "Robust execution of service workflows using redundancy and advance reservations," *IEEE Trans. on Services Computing*, vol. 4, no. 2, pp. 125–139, 2011.

[28] E. Brewer, "Cap twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, Feb 2012.

[29] R. Calheiros and R. Buyya, "Meeting deadlines of scientific workflows in public clouds with tasks replication," *IEEE Trans. on Parallel and Distributed Systems*, vol. 25, no. 7, pp. 1787–1796, 2014.

[30] J. Lau, L. C. Lung, J. da Fraga, and G. Santos Veronese, "Designing fault tolerant web services using bpel," in *ICIS '08*, pp. 618–623.

[31] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Comput. Surv.*, vol. 37, no. 1, pp. 42–81, 2005.