Replicated Execution of Workflows

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der Naturwissenschaften
(Dr. rer. nat.) genehmigte Abhandlung

vorgelegt von

David Richard Schäfer

aus Villingen-Schwenningen

Hauptberichter:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Mitberichter:	Prof. Dr. rer. nat. Dr. h. c. Frank Leymann
Tag der mündlichen Prüfung:	4. Oktober 2018

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart
2018

Acknowledgements

I would like to thank Prof. Dr. Kurt Rothermel for giving me the opportunity to work in his research group. The advice he provided, the critical questions he asked, and his guidance and invaluable feedback played an important role for finishing this work successfully. My thanks also go to Prof. Dr. Frank Leymann for being my co-advisor. I would also like to thank my project supervisor Dr. Muhammad Adnan Tariq for many long, fruitful discussions as well as his advice.

Moreover, I would like to thank my former colleagues from the university, who always were ready to discuss research questions, give feedback, and created a wonderful working atmosphere. Special thanks go to Andreas Weiß, Thomas Bach, Florian Berg, Christian Mayer, and Ben Carabelli.

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

My gratitude also goes to my family and friends, who always supported and encouraged me. Special thanks to my wife Annemarie Schäfer, who took care of everything all the times I had neither energy nor time for anything but my research. Special thanks also go to my parents Uwe and Sabine Schäfer for making me the person that I am today.

David Richard Schäfer, October 21, 2018

Contents

Abstract

Workflows are the de facto standard for managing and optimizing business processes. Workflows allow businesses to automate interactions between business locations and partners residing anywhere on the planet. This, however, requires the workflows to be executed in a distributed and dynamic environment, where device and communication failures occur quite frequently. In case that a workflow execution becomes unavailable through such failures, the business operations that rely on the workflow might be hindered or even stopped, implying the loss of money. Consequently, availability is a key concern when using workflows in dynamic environments.

In this thesis, we propose replication schemes for workflow engines to ensure the availability of the workflows that are executed by these engines. Of course, a workflow that is executed by a replicated workflow engine has to yield the same result as a non-replicated execution of that workflow. To this end, we formally define the equivalence of a replicated and a non-replicated execution called *Single-Execution-Equivalence*. Subsequently, we present replication schemes for both imperative and declarative workflow languages. Imperative workflow languages, such as the Web Service Business Process Execution Language (WS-BPEL), specify the execution order of activities through an ordering relation and are the predominant way of specifying workflow models. We implement a proof-of-concept for demonstrating the compatibility of our replication schemes with current (imperative) workflow technology. Declarative workflow languages provide greater flexibility by allowing the reordering of the activities within a workflow at run-time. We exploit this by executing differently ordered replicas on several nodes in the network for improving availability further.

Zusammenfassung

Um den Geschäftsbetrieb einheitlich, reibungslos und kostengünstig zu halten, spezifizieren Unternehmen Geschäftsprozesse, welche die Abläufe innerhalb des Unternehmens definieren. Workflowsprachen, wie die Web Service Business Process Execution Language (WS-BPEL), sind hierfür der faktische Standard in der Unternehmensbrache. Workflows spezifizieren einzelne Aktivitäten und die Reihenfolge, in welcher die Aktivitäten abgearbeitet werden sollen, durch eine Ordnungsrelation. Die Workflowtechnologie erlaubt die einfache Verwaltung und Optimierung von Geschäftsprozessen durch die intuitive graphbasierte Repräsentation der Prozesse. Darüber hinaus ermöglichen Workflows die Automatisierung der Prozesse, wodurch Fehler reduziert und Kosten gespart werden können. Besonders zeitkritische Applikationen, wie die just-in-time Produktion oder das Management von on-demand Cloudresourcen, können durch die Automatisierung wesentlich effizienter erledigt werden, beziehungsweise werden durch die Automatisierung überhaupt erst realisierbar.

Aus diesen Gründen, wird die Workflowtechnologie von Unternehmen genutzt um die Interaktionen zwischen den unternehmenseigenen Standorten und auch die Interaktionen mit Geschäftspartnern zu automatisieren. Hierfür definieren die Standorte und Geschäftspartner Schnittstellen, so genannte Webdienste, welche durch die Aktivitäten eines Workflows aufgerufen werden können und dadurch die Automatisierung der Interaktionen vereinfachen.

Als Folge dieses Trends werden die Workflows in einer verteilten, heterogenen und dynamischen Umgebung ausgeführt, in welcher Fehler regelmäßig auftreten. Diese Fehler können die Ausführung der Workflows unterbrechen oder sogar komplett stoppen. Im schlimmsten Fall kann das dazu führen, dass der Geschäftsbetrieb eines Unternehmens stillsteht. Daher ist es von enormer Wichtigkeit zu garantieren, dass die Workflows auch in fehleranfälligen Umgebungen eine hohe Verfügbarkeit aufweisen.

Diese Dissertation behandelt Mechanismen, welche die Verfügbarkeit von Workflowausführungen in fehleranfälligen Umgebungen sicherstellt. Zunächst werden bestehende Techniken für Fehlertoleranz und Verfügbarkeit auf ihre Tauglichkeit bezüglich unserer Zielsetzung untersucht. Um auch Fehler zu tolerieren, welche einen Knoten ausfallen lassen auf welchem der Workflow ausgeführt wird, muss der Workflow repliziert ausgeführt werden. Wenn also eines der Replikate ausfällt, kann ein anderes die Workflowausführung fortführen.

Nach unserem Wissen existiert bis jetzt jedoch keine Definition über die Korrektheit von replizierten Workflowausführungen. Darüber hinaus bauen bestehende Workflowreplikationstechniken auf der Annahme auf, dass Fehler perfekt erkannt werden können. Wenn also ein Knoten im Netzwerk als ausgefallen erkannt wird, dann ist dieser auch tatsächlich ausgefallen. Allerdings ist es unmöglich einen perfekten Fehlerdetektor zu realisieren, da die Nachricht eines funktionierenden Knotens, welche die Funktionalität an die anderen Knoten signalisieren würde, verloren gehen oder einfach nur zu langsam den Zielknoten erreichen kann.

Diese Dissertation präsentiert die folgenden Beiträge, welche den derzeitigen Stand der Wissenschaft erweitern: (1) Um sicherzustellen, dass eine replizierte Workflowausführung das gleiche Ergebnis liefert, wie es eine nicht-replizierte Ausführung liefern würde, präsentieren wir eine formale Definition über die Äquivalenz von replizierten und nicht-replizierten Ausführungen eines Workflows. (2) Wir entwerfen Replikationsschemata zur replizierten Ausführung von Workflows, welche mit der aktuellen (imperativen) Workflowtechnologie kompatibel sind und keinen perfekten Fehlerdetektor benötigen um korrekt zu funktionieren. (3) Wir präsentieren ein Workflowreplikationsschema für Workflows, welche in einer deklarativen Workflowsprache spezifiziert worden sind. Da deklarative Workflows es erlauben die Ausführungsreihenfolge der Aktivitäten während der Laufzeit zu ändern, nutzen wir diese Eigenschaft um die Verfügbarkeit der replizierten Ausführung weiter zu erhöhen.

1. Introduction

WORKFLOWS are widely used for managing, organizing, and optimizing business operations [SBTR14, KLL09]. Workflows are defined by a process of interrelated activities, where each activity can call a service. Encapsulating functionality in services modularizes complex business operations and, thereby, simplifies maintainability. Today, even small businesses have several locations and partners all over the world. In this global setting, businesses use services to define interfaces for the interactions between business partners and locations. Here, workflows automate and, thus, ease the interactions allowing businesses to stay competitive.

The globalization of businesses, however, requires the workflows to execute in a heterogeneous environment, where device and communication failures occur quite frequently. While it is common knowledge that wireless and mobile connections are often unreliable and not always available [CCE03, TLL08, BMV10, GAP+12, Seg13, PA13], wired networks are typically assumed to be highly reliable. In contrast to this widely spread assumption, wired networks and even the backbone experience frequent failures as well [BK14, BDF+13, TLM+12, TLSS10, MIB+08, FABK03, LAJ99]. Failures are even common in datacenters, where, for example, a Microsoft data-center study found that a data-center on average experience 40.8 failures with end-user impact per day [BDF+13, GJN11]. Thus, contrary to common believe, migrating services into cloud environments does not solve all availability concerns [SSR+10]. Consequently, network partitioning is an important design consideration for any network related application and system [BK14, BDF+13, Dea09, DHJ+07].

In conclusion, a lot of work has verified that networks might fail or partition. But what about failures of the computing nodes? A study has found that a node with typical consumer hardware has a crash failure probability over 0.5 % when running for an accumulated time of 30 days within eight month [NDO11].

Moreover, crash failures are also common on specialized and data-center hardware [Gra85, Bre17], where multi-node failures in datacenters are even more common than network partitioning failures [Bre17].

As a consequence, irrespective of where we deploy a system, we have to build and design the system such that it can tolerate crash and network partitioning failures. Otherwise, our system might become unavailable, where unavailability and even small outages translate to the loss of customers and, thus, money [Sol08, CTX09, Bru09, Loh12, SCGM14].

In the context of workflow technology, these availability requirements mean that a workflow system must be build such that all running workflow executions tolerate crash failures and network partitioning. For example, consider an online platform is using a workflow for managing its cloud resources. The unavailability of this workflow prevents the provisioning of cloud resources, which can render the online platform unresponsive. It is obvious that ensuring availability is essential for staying competitive.

1.1. Contributions

This thesis proposes new replication schemes for ensuring the availability of workflows that are executed in failure prone environments. The concepts and findings presented in this thesis combine and extend our previous publications [SBTR14, SSB+15, SWT+16, STR16b, SRT17, SRT18]. These publications have been a collaborative effort, where we will lay out the contributions of the author of this thesis in the following.

In Part A of this thesis, we describe concepts and replication schemes for workflow engines in order to ensure the availability of workflows specified in imperative workflow languages. The contributions of the thesis towards providing availability for imperative workflows are the following:

1. We formally define the equivalence of a replicated and a non-replicated workflow execution, called *Single-Execution-Equivalence*, and prove that a replicated workflow execution that is Single-Execution-Equivalent is

correct. Defining Single-Execution-Equivalence has been collaborative work [STR16b, SRT18], where the author of this thesis contributed 60 % of the definition.

2. We present a basic majority-based replication scheme that fulfills the defined correctness and that ensures availability in the presence of failures, where the goal of the replication scheme is to minimize the compensation cost that arises through the replication. In specific, we use a primary-backup replication scheme, where multiple workflow engine replicas participate in a workflow execution. The primary workflow engine executes the workflow. The execution state produced by the primary is synchronously replicated on a majority of workflow engine replicas after each activity execution. Upon a primary failure, a new primary is elected from the currently available workflow engine replicas. Because only the current, i.e., latest, primary is able to replicate its state on a majority, old primaries maximally execute one activity before noticing that they are not primary anymore. As a consequence, maximally one activity needs to be compensated per old primary. This work is collaborative effort [STR16b, SRT18], where the author of this thesis contributed 70 % of the concepts.

3. The synchronous replication after each activity basically pauses the execution until a majority of replicas has received the execution state. For reducing the impact of replication on execution time, we present a relaxed majority-based replication scheme. We develop the mechanism of *synchronization groups*, which allows to asynchronously replicate execution state for all activities within the group but requires a synchronous replication after the whole group has been executed. This work is collaborative effort [STR16b, SRT18], where the author of this thesis contributed 90 % of the concepts.

4. The proposed majority-based replication schemes limits the compensation cost by always having only one valid primary, elected via majority consensus, in the system. However, this also limits the availability since a majority needs to be available for electing a new primary. Our flexible failover replication scheme drops this requirement allowing to elect one primary per

partition – even if this partition only contains a single replica. This scheme trades increased availability for increased compensation cost because any partitioned replica will become primary and execute the workflow. In the end, all but one workflow execution have to be compensated. The flexible failover replication scheme is original work of the author [SRT17].

5. We develop a generic architecture for realizing our replication schemes in existing workflow technology. Andreas Weiß, Vasilios Andrikopoulos, and Santiago Gómez Sáez provided the necessary background knowledge on the existing workflow technology and feedback throughout the development of the architecture [SWT+16].

6. The author of this thesis implemented a prototype of the proposed replication schemes and performed extensive evaluations in a geo-distributed setting as well as in a datacenter setup, which show the benefits of the proposed schemes. Lukas Krawczyk ported the replication schemes to the open source workflow engine Apache ODE and, additionally, implemented the proposed architecture in this engine in a study thesis [Kra16] supervised by the author of this thesis and Santiago Gómez Sáez. Andreas Weiß and Santiago Gómez Sáez consulted and provided help in questions on the implementation in the Apache ODE [SWT+16].

In Part B of this thesis, we aim to provide availability for workflows that are specified in a declarative workflow language. In declarative workflow languages, the workflow is defined by a set of activities and a set of constraints, where each constraint restricts the order in which the activities may be executed [PSvdA07]. In other words, the activities may be executed in any order that is not prohibited by the defined constraints. If all constraints are fulfilled, the workflow execution is successfully finished. This way of specifying workflows allows more flexibility in terms of the execution. It, however, also raises the need for checking whether a specific activity execution order is allowed by the constraints.

We propose to use replication for declarative workflows such that each workflow engine replica executes a different activity order, where any executed activity order is allowed by the workflow specification. This decouples the workflow

from transient service failures because the different replicas access the required services at different points in time. Moreover, the results of the activity execution of a replica can be published to the other replicas such that these can reuse the provided result instead of re-executing the activity, speeding up the execution. This work [SBTR14] was inspired by the project proposal of ALLOW Ensembles (Seventh Framework Programme, grant no: 600792) [ALL] and previous work published within the scope of the project [Sch13, Bac13]. In specific, the main contributions are:

7. We provide a metric that allows for evaluating which are the best activity execution orders in terms of availability. This allows to select the activity execution orders that are executed by the workflow engine replicas.

8. As the generation of all allowed activity execution orders is a PSPACE-complete problem [SC85], we provide a heuristic for generating a subset of allowed activity execution orders that provide high availability.

9. We designed a coordination protocol that allows the re-using of results produced by the executions of the other replicas, which speeds up the overall workflow execution. This coordination protocol is inspired by Thomas Bach's coordination protocol [Bac13], which, however, was lacking the support for tolerating crash failures.

10. We implemented a prototype for generating the activity execution orders by extending the SPOT library[1]. The prototype also includes the coordination protocol. The prototype implementation as well as the evaluations has been joined effort with Thomas Bach, where the author of this thesis contributed 50 % [SBTR14].

Unless otherwise mentioned above, the availability concepts for declarative workflow languages are original work of the author [SBTR14].

Note that the concepts of this thesis have partly been developed during the ALLOW Ensembles project and, thus, are also part of the ALLOW Ensembles Deliverables 6.1, 6.2, and 6.3 [TSB+13, SBT15, STR16a].

[1] https://spot.lrde.epita.fr/

1.2. Structure

In the following Chapter 2, we review existing techniques for ensuring fault tolerance, reliability, and availability. The remainder of the thesis is divided into three parts. In Part A, we present replication schemes for workflows specified in imperative workflow languages. In Chapter 3, we define the used workflow and execution model. Subsequently, in Chapter 4, we define the equivalence of a replicated and non-replicated workflow execution, called Single-Execution-Equivalence. In Chapter 5, we present our replication schemes for ensuring the availability of workflow executions in the presence of failures. For realizing the presented schemes with existing workflow technology, we present the requirements for such a replication system as well as a system architecture fulfilling the requirements in Chapter 6. In Chapter 7, we evaluate the presented schemes as well as the presented system with regard to availability, overhead, and scalability. In Chapter 8, we discuss the differences of our workflow replication schemes to related work that has not already been covered by Chapter 2.

In Part B, we present techniques for generating activity sequences from declarative workflow specifications and a coordination protocol for the replicated execution of the generated activity sequences. In Chapter 9, we define the declarative workflow model and present a motivational scenario. In Chapter 10, we present an approach overview. In Chapter 11, we present how to generate and select the activity sequences that conform to a declarative workflow specification such that the availability during the replicated execution is high. In Chapter 12, we present a coordination protocol for the replicated execution of the generated activity sequences. In the following Chapter 13, we evaluate the generation and selection with respect to availability and run-time as well as the execution time of the replicated execution in the presence of failures. In Chapter 14, we discuss the differences of the concepts presented in Part B, to related work that has not already been covered by Chapter 2.

In Part C, we conclude the thesis with a summary of the presented work and a short outlook on future work.

2. Related Work

In this chapter, we will discuss the existing techniques for dealing with failures. In specific, we differentiate between techniques that provide fault tolerance, reliability, and availability. A fault tolerant system can resume to provide its functionality after recovering from a failure in a consistent manner. In contrast, a highly reliable system minimizes the possibility that a failure might occur but might not implement any means of fault tolerance. Finally, a highly available system continues to provide its functionality even when a failure occurs. This requires redundancy such that the redundant component can take over in case the first component fails. Of course, even a highly available system can only continue to provide the functionality up to a specific number of failures.

Because of the high relevance of networked applications in today's always connected world, fault tolerance, reliability, and availability have been extensively researched in the context of distributed systems [AD76, Tho79, Rus80, BT83, Avi85, CL85, SWG92, KM97, EAWJ02, BLKC03, BCH+05, AFB+06, BHK+06].

In the context of workflow technology, handling and tolerating failures has also been studied, where the goal is to ensure that a workflow gracefully finishes when failures occur [AAA+96, LR00, DDGJ01, Gre02, KHC+05, SJP06, SPJ07, TLHL09, LLdSFV08, SPJ11, SS12, SS13]. In the following, we will focus on the techniques that are related to or designed for workflow technology. As our goal is to design a highly available system, we especially discuss the techniques with regard to their advantages and shortcomings regarding availability.

2.1. Crash Recovery

Crash recovery techniques solely provide fault tolerance and do not target providing availability. However, crash recovery techniques [CL85, EAWJ02] are probably the most common method for ensuring fault tolerance and workflow

engines typically implement the techniques in order to guarantee that workflow executions can be continued after a crash failure [BDH+12]. Thus, we will describe them briefly even though it is clear that the techniques do not ensure availability.

Crash recovery techniques solve the following problem: all data kept in volatile memory may be lost through a crash failure. With respect to workflow executions this means that the execution state of all running workflow executions might be lost through crash failures since the execution state is usually kept in volatile memory. Losing the execution state makes it impossible to finish a workflow execution gracefully after recovering from the failure. To avoid losing the data through crash failures, crash recovery techniques store the data in stable storage or in volatile memory of other nodes in the network. Common techniques for crash recovery are checkpointing and logging.

Checkpointing [CL85, EAWJ02, BLKC03, LL06, BHK+06] creates a snapshot, called *checkpoint*, of a – possibly distributed – application such that the application can restart from the last checkpoint after recovering from a failure. However, this means that all progress that the application made after the last checkpoint is lost. With respect to workflow executions, this means that activities might be re-executed after the recovery – without the workflow system being aware that it re-executes a part of the workflow. For example, a payment activity might be executed again, which alters the semantics of the workflow since the payment should have been performed only once. Hence, checkpointing alone cannot guarantee that a workflow execution finishes gracefully in the presence of failures.

Logging [EAWJ02, BLKC03, BCH+05, BHK+06] writes all messages, events, or changes (depending on the use case) of the application to a log. After a crash failure, the application reproduces a consistent state from the log entries. Depending on the length of the log, this recovery procedure might be time consuming. Thus, it is common to combine checkpointing and logging [EAWJ02], which keeps the log small and ensures that no execution progress is lost through crash failures.

2.2. Workflow Failure Handling

As discussed above, crash recovery allows to tolerate the crash failure of a workflow engine. Additionally, workflows usually provide fault tolerance by integrating fault handlers and recovery mechanisms into the workflow model [RvdAtH06, LR00]. This allows to detect and resolve faults that can be detected by the worklfow engine. In this respect, *compensation handlers* [RvdAtH06, LR00] are often specified for each activity of the workflow, where the execution of this compensation handler semantically reverses the effects that the execution of that activity had – even if this activity was only partially executed. If a failure occurs during the activity execution, the compensation handler can be executed, which performs a semantic rollback to the state before the activity was executed. In other words, compensation enables backward recovery for resolving faults.

However, a compensation is not equal to a rollback or an undo. Workflow activities can access stateful web services that, for example, might write to a database – like a payment service reduces your account balance in the bank's database. As the workflow has no access to that database, it cannot rollback the database's state. Thus, once the change committed in the database, it cannot be undone. For example, consider an activity that booked a flight. Compensation would cancel the flight ticket, which returns the money payed for the ticket but might incur a cancellation fee. In contrast, a rollback would require to return to a state before the ticket was booked incurring no fee.

The compensation concept was enhanced by compensation spheres, which allow to encapsulate multiple activities in one sphere such that if one of the activities requires compensation, all activities in the sphere are compensated [LR00]. The idea of compensation originates from Sagas [GMS87, GMGK+91], which we will discuss further below in the context of transactional workflows.

The described compensation concepts allow the semantic rollback of one or several activities of a workflow execution. After the compensation, the workflow can restart from an earlier state, which allows to retry parts of the workflow in case of failures [SK11, SK12, SK13]. Model-as-you-go [SK13] even allows to change the workflow model at run-time, such that the re-execution will use the adapted workflow model. Thereby, model-as-you-go also allows to replace or remove an activity that would prevent the continuation of the execution (e.g.,

because of a failure of the called service). As a consequence, model-as-you-go does not only provide fault tolerance but even improves availability.

However, model-as-you-go – like compensation handlers and other fault handling techniques that are specified in the workflow model– can still only treat failures that can be caught by the workflow engine. However, these techniques cannot ensure availability in case that the workflow engine fails or the device on which the workflow engine resides crashes.

2.3. Transaction Management & Transactional Workflows

There exist techniques to ensure the availability of transaction management system [PCD91, Ady99, PGS03, CRCR09, LFKA11, CL12, WKK12, BDF+13, KKW13, BFF+14]. Workflow technology assumes that there is a workflow client application that sends execution requests to the workflow engine triggering the execution of a workflow [Hol95]. Similar to workflow technology, transaction management techniques also consider clients sending execution requests to a system [WPS+00, PGS03, CRCR09, LFKA11, WKK12, KKW13, BFF+14]. Here, instead of a workflow, the request triggers the execution of a transaction. The changes that a transaction performs on the system's state are only made permanent once the transaction commits. Upon the commit, a reply is send to the requestor. Instead of committing, the transaction might also be aborted, which discards all changes that the transaction issued on the system's state.

In contrast to transactions, the activities of a workflow directly perform changes on variables and other state, e.g., to the account balance when executing a payment activity. Additionally, systems like transaction management systems consider any committed change to be irreversable, i.e., the change cannot be undone [PCD91, Ady99, PGS03, CRCR09, LFKA11, CL12, WKK12, BDF+13, KKW13, BFF+14]. Moreover, these concepts do not support compensation. However, for workflows, where compensation handlers are usually available, this model is too restrictive.

With regard to compensation, Sagas [GMS87, GMGK+91] share more similarities with workflows since the compensation mechanism of workflows originates

from Sagas. In specific, Sagas split long-running transactions into several subtransactions, where subtransactions commit their changes directly, i.e., before the higher level transaction commits. This prevents that the long-running transaction locks accessed variables for a large amount of time, where other transactions cannot proceed as they also require access to these variables. However, this also means that the higher-level transaction of Sagas cannot be aborted because the subtransactions already applied their changes. Thus, when a failure occurs, the subtransactions are semantically reversed by executing a compensation transaction for each executed subtransaction. This is also the basis for transactional workflows, where each workflow activity is equal to a subtransaction of Sagas [RS95, KAGM96, SABS02, Gre02].

There also exist mechanisms for porting transaction semantics to workflows [AAA+96, LR00, DDGJ01]. For example, an atomic sphere [LR00] contains one or multiple activities, where the effects of the activity executions are only made permanent once the sphere is committed. If a failure occurs, the sphere is aborted. However, workflows strive to avoid the overhead that is incurred by requiring transactional properties. Instead a workflow and even the activities of a workflow might be partially executed, where compensation handlers are assumed to semantically reverse the (partially) executed activities. As a consequence, concepts that ensure the availability of transactional workflows or transaction management systems cannot be easily transferred to workflows.

2.4. Fault Tolerant and Highly Available Service Invocation

During the workflow execution, the activities of the workflow may call web services offered by third parties. Actually, workflow languages like the widely used Web Service Business Process Execution Language (WS-BPEL), which is standardized by OASIS[1], specifically target the usage of web services. Such languages allow to compose complex *orchestrations* of web services. Reusing services simplifies the specification of new functionality through workflows,

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

especially since thousands of web services are already available and their number is still growing [PLM+10, ZZL14].

Traditionally, the services used by a workflow are 'hardwired', meaning that an activity has to call one specific service during its execution. In case this service is not available, the workflow execution cannot proceed or fails. This is especially problematic because the reliability and availability of existing services hugely varies [SF07, ZL08a, AMM08, ZZL10, ZZL14]. To this end, several techniques have been proposed to overcome this problem.

A common technique for masking a service failure is to compensate the service call and, afterwards, to call the same service again [Dob06, ZL08a, ZL08c, ZL08b, ZL09, JDF09, EKEF16]. However, if that service is still not available, retrying the same service will not solve the problem. Thus, dynamic service binding, service adapters, and other abstraction mechanisms [GHJV95, KHC+05, SF07, LV07, MSB08] have been proposed to allow calling alternate services in case of a service failure [KHC+05, Dob06, DPEV+06, ES06, SF07, LV07, ES07, MSB08, ZL08a, ZL08c, ZL08b, ZL09, JDF09, EKEF16]. For adhering to execution deadlines and keeping workflow execution times low, some approaches even propose to call multiple alternate services at once or staggered over time, using the result of the service that replies first and compensate all other called services [Dob06, SJP06, SF07, LV07, GHL+07, SPJ07, ZL08a, ZL08c, ZL08b, ZL09, TLHL09, SPJ11, MSM15, BTKR15].

As the dynamic binding of web services allows to select a compatible web services for the required functionality at run-time, some approaches select the service(s) based on measured QoS values [DPEV+06, ES07, ZL08a, ZL08c, ZL08b, ZL09, KIH10]. There even exist techniques for automatically composing a service orchestration with a desired functionality from existing web services using QoS values [LPC+11, LHG+16, EKEF16], where some re-compose the orchestration when a failure occurs [LPC+11, EKEF16].

In conclusion, we can observe that making service invocations highly available has been extensively researched. Even though this decouples the workflow execution's availability from the availability of a specific service, these techniques cannot cope with failures of the workflow engine or crashes of the device on which the workflow engine resides.

2.5. Distributed Workflow Execution

Many solutions for executing a workflow in a distributed manner in a network of workflow engines have been proposed [AMG+95, BD97, MWW+98, MM05, BMM06, CGH+06, TF07, FYG09, HCTS09, LMJ10, EMSU11]. Even though most of these have the goal of providing scalability, some incorporate reliability and availability considerations.

For enabling a workflow to be exectued in a distributed manner, the workflow is split into fragments, which are assigned to different engines. When a fragment finishes its execution, it provides the necessary execution state to the next fragment. Without additional failure handling mechanisms, this increases the failure proneness of a system as the workflow execution requires all involved engines – and, thus, the hardware on which the engines are running – to stay available. One approach for countering this problem is to decide the engine for executing the next fragment at run-time [MM05]. Even though this removes the dependence on one specific engine, the engine which is currently running still might fail causing outages.

Here, fragment mapping techniques strive for improving the reliability of a distributed execution by mapping the fragments to the engines such that the overall failure probability is minimized [SWG92, KM97]. The strategies often have multiple criteria that are considered for deciding a mapping [DO02, AGK04, HB07, BRSR08, BHR08, BHR09a, BHR09b, GEST09, WHP09, PW10, ZRXS10, GWLY11]. When reliability is the optimization goal, these techniques minimize the failure probability of the workflow execution. When reliability is a constraint, the failure probability is at least kept below the given constraint. In both cases, the workflow execution is highly reliable but still unable to mask or tolerate failures. Thus, the workflow execution cannot proceed as soon as a single failure occurs.

Some mapping approaches overcome this limitation by incorporating active replication [AGK04, BHR08, BHR09a, BHR09b, GEST09, ZRXS10, CB14]. We will discuss these techniques in the following when exploring replication techniques.

2.6. Replication

Replication is a widely used concept for ensuring availability [CBPS10]. When replicating the functionality that a system shall provide on multiple devices, the advantage is that the remaining devices can continue to provide the functionality even if some of the devices fail or are partitioned.

For example, data availability is achieved by replication, where multiple levels of consistency were defined that each ensure availability in different failure scenarios [LGG+91, LLSG92, FB99, Bre00, SS05, ZP08, LM10, LFKA11, Bre12, BFHD12, TBV15]. For example, with eventual consistency [SS05] reading a variable might return outdated values but – in turn – the variable is always available for reading as long as at least one copy is available. Of course, therefore the replicas have to be placed accordingly, which has also been studied extensively [DW01, HCP03, TX05, KL05, LCZ05, TLX+06]. However, these data replication techniques strive for data availability targeting one level below workflow executions, which operate on that data. Hence, we need extra concepts for ensuring the availability of workflow executions.

The replication techniques that may be applicable to workflows are *passive* and *active* replication techniques. Passive replication [AD76, BMST93, CBPS10] techniques elect a *primary* replica that is responsible for handling and executing client requests while all other replicas – called *backups* – receive state updates from the primary. In case of a primary failure, the backups can continue to provide the functionality by using the previously received state updates and electing one of the backups to be the new primary.

The concept of passive replication has been applied to workflows on different levels. Some approaches only replicate the workflow client requests and replies [LR00, LFCL03, FLLL07, LLdSFV08]. This allows a backup that becomes the new primary to simply repeat replies to workflow client applications that were lost earlier without requiring the new primary to re-execute the workflow. However, these techniques do not save any intermediate state of the workflow executions. Thus, the progress of a started workflow execution will be lost through failures. The new primary has to re-execute the workflow because it only has saved the workflow client request. The re-execution might introduce high failover times – especially for long-running workflows. Moreover, these re-executed workflow

must not interact with services that might change external state. Otherwise, the state might be altered a second time through the re-execution. For example, when the workflow includes a payment activity, the payment is performed a second time, which is obviously undesirable and alters the semantics of the workflow. Additionally, these techniques assume a fail-stop model, which allows to implement a perfect failure detector. However, perfect failure detection is impossible in practice since an unresponsive primary might simply be slow to respond or partitioned [FLP85, CBPS10]. Thus, the possibility that the old primary still might continue to respond to client request or might finish the already started workflow executions is not considered.

Other approaches actually replicate intermediate execution states of the workflow execution [KAGM96, PCD91, FWM03, SWSS04, SS12, SS13]. This has the advantage that already started workflow executions can be continued by the new primary. However, these techniques still require a perfect failure detector and, thus, cannot be used in practice. Moreover, the fail-stop model does not assume a failed node to recover [CBPS10]. Thus, in case that this node partially performed an activity before failing, this partial activity execution cannot be compensated because only the failed node knows about the partially performed activity. Hence, semantic equivalence of the replicated execution to a non-replicated execution of that workflow cannot be guaranteed.

Some approaches rely on transactional workflow properties, where each activity is a subtransaction such that when this activity commits, the execution state is replicated [KAGM96]. Thus, the activity execution is never lost. However, as already discussed, workflows in general do not ensure transactional properties for activity executions avoiding the undesirable coordination overhead. Without transactional properties activities might be partially executed and, then, the question arises when a replicated workflow execution is correct and how to ensure this correctness. To our knowledge, there exists no formal definition on when a replicated workflow execution is correct, so far.

Active replication actively executes each activity on all replicas eliminating the need for transferring execution state updates. A well-known technique of active replication is state machine replication [OL88, Oki88, Lam98, Lam01, CBPS10, LC12, OO14, Ong14, VRA15]. Here, a service is replicated on $2f+1$ replicas, where f crash failures can be tolerated. However, state machine replication

assumes that activities can never be rolled back. Moreover, state machine replication does not support compensation – making each activity execution irreversible. Thus, the activity execution must never be lost through failures and, hence, the replicas have to agree on each activity before executing it. The activities of a workflow typically can be compensated, which allows to save the overhead of an agreement before each activity execution. Additionally, state machine replication assumes an activity only affects the state of the replica's state machine on which the activity is being executed. In contrast, workflows typically interact with other workflows and stateful services [Wes07]. Moreover, the services and, thus, the outcome of an activity execution might be non-deterministic in the following sense: Even though sending multiple identical requests to a service, the service might send different replies. Hence, when an activity calls such a service, the outcome cannot be deterministically deduced from the input. State machine replication approaches, however, require each activity to be deterministic such that after agreeing on the activity that is executed next, all replicas execute the activity resulting in the same state on all replicas.

Some workflow fragement mapping approaches (cf. Section 2.5) also incorporate active replication by executing an activity (or fragment) in multiple workflow engines concurrently [AGK04, SLM05, BHR08, BHR09a, BHR09b, GEST09, ZRXS10, CB14]. However, they assume that the activities are solely computational tasks not altering state except for the execution state of the workflow – similar to state machine replication techniques. Especially in the context of scientific workflows, all activities of a workflow are typically assumed to be solely computational tasks, which only alter the execution state of the workflow. Thus, workflow fragments are repeated in case of a failure or even actively replicated for increasing availability [HK03, DSS+05, YB05, dSdSeSL07, Qua07, KMR08, BWDL08, PPF09, ZMKC09, GdM11, GCB+14]. The result of one of the executions is used for continuing the workflow execution while all other executions are simply discarded. Remember, however, that the assumption of solely computational tasks does not hold for workflows in general. Workflows frequently interact with services and other workflows, which keep their own state [Wes07]. Thus, repeating and actively replicating workflow fragments as used by fragment mapping techniques and scientific workflow approaches cannot be applied to workflows in general.

There are also approaches that combine active and passive replication [PCD91, WPS+00, BDH+12]. Some approaches even support non-deterministic activities [PCD91, WPS+00]. Here, the idea is to use active replication only for deterministic activities, while all non-deterministic activities are replicated using passive replication. However, the techniques also require that the activities do not interact with stateful services.

Another approach targets the usage of non-deterministic services [BDH+12]. Here, all activities are executed by all engines but the service calls are routed through a middleware. The middleware calls multiple services in parallel and selects one of the service replies. The selected service reply is then send to all replicas. For tolerating lost service reply messages, each service call is assigned a unique service call identifier. Each used service has to implement a middleware component that filters duplicate calls based on the unique service call identifier and only repeats the reply message for duplicates (instead of re-executing the service). Even though not explicitly mentioned by the authors, this mechanism actually would also allow to call services that change external state – assuming that for those services only one service is invoked instead of multiple in parallel. However, the described approach requires that each service used by the system needs to be altered, i.e., implement the duplication filtering of the proposed middleware. Thus, existing services cannot be used.

2.7. Summary

Workflow technology provides the means for fault tolerance through fault handlers in the workflow model [RvdAtH06, LR00] and crash recovery mechanisms [BDH+12]. Moreover, the workflow execution's availability has been decoupled from the availability of specific services by using alternate services through adapters and dynamic service binding [KHC+05, Dob06, DPEV+06, ES06, SF07, LV07, ES07, MSB08, ZL08a, ZL08c, ZL08b, ZL09, JDF09, EKEF16] as well as through dynamic scheduling mechanisms [Dob06, SJP06, SF07, LV07, GHL+07, SPJ07, ZL08a, ZL08c, ZL08b, ZL09, TLHL09, SPJ11, MSM15, BTKR15].

However, ensuring the availability of workflow executions in the presence of workflow engine and device failures remains an open problem. Even though there exist solutions that replicate the workflow executions for ensuring availability

[KAGM96, PCD91, FWM03, SWSS04, SS12, SS13], they assume the fail-stop model [CBPS10]. Assuming the fail-stop model, it is possible to implement a perfect failure detector. In practice, however, it is impossible to implement such a failure detector because a workflow engine that is detected as failed might simply be slow to respond [FLP85, CBPS10].

This thesis fills the gap by considering the crash-recovery model [CBPS10], which does not rely on the assumption that each failure can be detected perfectly. We define the correctness of a replicated workflow execution and present replication schemes that enable the highly available execution of workflows in failure prone environments.

Part A.

Replication Schemes for Imperative Workflow Languages

3. System and Imperative Workflow Model

THE underlying system consists of a collection of nodes, which are interconnected by a communication network. The nodes and communication links can experience crash failures at any time. In accordance with the crash-recovery model [CBPS10], we assume that both nodes and links eventually recover from failures.

Nodes are running workflow engines and/or services. Workflow engines execute workflows, which call the available services. We assume that services provide replication transparency to their clients, i.e., to the workflow engines. Because of this transparency, the replication schemes presented in this thesis are totally decoupled from whether or not the called services are replicated.

In the presence of failures, we ensure availability by replicating the workflow engine and, thereby, the workflows executed in the engines. A group of $2f+1$ workflow engines, called *replicas* for short, participate in the replicated execution of the workflow, where f is the number of failures that can be tolerated. In other words, as long as at least $f+1$ replicas are available, the progress of a workflow execution is guaranteed. Each replica $r \in R$ has a unique identifier, where R denotes the set of all replicas.

A workflow is defined by a directed, acyclic graph $G = (mID, A, L, \Sigma)$, where mID is the unique identifier of the workflow model, A is the set of activities, L is the set of links defining the control flow, and Σ is the set of variables needed for the execution, called the *internal state*. The activities $a \in A$ are performed during the workflow execution in the order defined by the links. In specific, a link $l \in L$ is defined as follows: $L : A \times A \times T$, where T is the set of transition conditions. A link $l = (a_1, a_2, t)$ indicates that a_2 is executed after the execution of a_1 completed and the transition condition t is fulfilled. For convenience, we

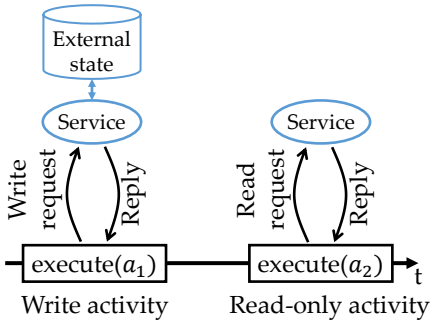


Figure 3.1.: Exemplary execution of a write activity and a read-only activity

define the function $succ(a, G)$ that returns the set of all activities that directly succeed a in the workflow model G.

$E(G, A_o, <)$ (or $E(G)$ for short) denotes the execution of a workflow G, where A_o is the set of executed activities. Because of branches in the workflow model, activities might execute concurrently and, thus, can only be partially ordered. The ordering relation $<$ defines the causal relationship [Lam78] of the activities in A_o. A causally preceding activity of $a_o \in A_o$ is basically any activity that changed the internal state that the activity a_o is using as input. Hence, the causal relationships of the activities are defined through the data flow of a workflow model. We assume that any branches in G that might execute concurrently use disjoint subsets of the internal state. This common assumption [Wes07] prevents race conditions.

Activities may also manipulate or read the *external state* by sending write or read requests to services, where the external state denotes any state that is not in direct control of the workflow engine, e.g., the state of a stateful service. Requests that might modify the external state are called *write requests*. Accordingly, the activities sending write requests are called *write activities*. All other requests are called *read requests* and the corresponding activities *read-only activities*. In the example of Figure 3.1, activity a_1 is a write activity while activity a_2 is a read-only activity. We assume that any activity can send at most one request. This triggers the execution of the service, after which the service returns one reply

message to the activity that sent the request. Later we will relax this interaction model (cf. Section 5.3).

We assume that all activities $a \in A$ are deterministic in the following sense: 1) given the same input internal state, every execution of activity a will always produce the same request (if any) and 2) for a given input internal state and reply message from the service (if any), each execution of activity a will always produce the same output internal state. Any desired non-determinism has to be realized by services, which we do not require to be deterministic.

An activity execution can be compensated by executing its compensation handler, where we assume that every activity has such a compensation handler. In specific, the execution of this compensation handler (semantically) reverses all effects that the execution had on the external state and rolls back the internal state to the state before the execution of the activity started. The compensation handler can be a workflow itself and, thus, we assume that its execution is decoupled from $E(G, A_o, <)$ meaning that we assume A_o does not include any compensation activities. The execution of the compensation handler might induce cost (e.g., monetary cost) to which we refer to as the *compensation cost*. An execution of an activity can, however, only be compensated after all executions of causally succeeding activities have been compensated. This compensation model originates from Sagas [GMS87] and conforms to business processes [LR00].

A workflow execution can only be completed if each of its activities is either completed successfully or compensated. Once a workflow execution has been completed, none of its activities can be compensated anymore. For the purpose of this work, we assume that there is a *Compensation Unit* running in each workflow engine, which is responsible for executing the compensation handlers. A workflow execution can schedule the execution of a compensation handler on the Compensation Unit, where the Compensation Unit guarantees to execute the scheduled compensations eventually even in the presence of failures. Moreover, we assume that there exists a *workflow repository*, which contains all workflow models. Using the unique workflow model identifier mID, a computing node can load the respective model $G = (mID, A, L, \Sigma)$ from this repository. For executing G, the workflow engine instantiates G, which initializes the internal state. Any instance of G has the same initial internal state. The described workflow and

execution model generally complies with any graph-based workflow language, such as the Web Service Business Process Execution Language (WS-BPEL).

4. Correctness of Replicated Workflow Executions

In this chapter, we introduce a property called Single-Execution-Equivalence that defines the correctness of replicated workflow executions. This property is independent of the type of consistency achieved by the underlying workflow model. It ensures that the replicated execution of a workflow has the same effects on the external state as some non-replicated execution of that workflow.

Axiom 1. *The non-replicated execution of a workflow G is correct.*

That is, we assume that any non-replicated execution maintains the consistency of the external state, which it accesses and changes through using services. There are many techniques for ensuring that non-replicated workflow executions will produce a correct result, e.g., [Wes07, vdA97, BGS07]. Some techniques ensure the semantic correctness by checking every possible execution of the workflow model against properties that are specified by the workflow designer [BGS07, vdA97]. Other techniques check the model for generally undesired properties, such as deadlocks [Wes07]. Consequently, there exist a multitude of methods supporting the workflow designer that justify Axiom 1.

The effects of a workflow execution on the external state are caused by the interactions with the services. For each read or write request that an activity sends to a service, the service returns a reply. The received replies typically influence the future behavior of a workflow and, thus, are considered as input. Write requests change the external state and, thus, are considered as output. Informally, a replicated execution of a workflow G is considered Single-Execution-Equivalent if there is a non-replicated execution of G that produces the same output given the same input. In the following, we formally define this property.

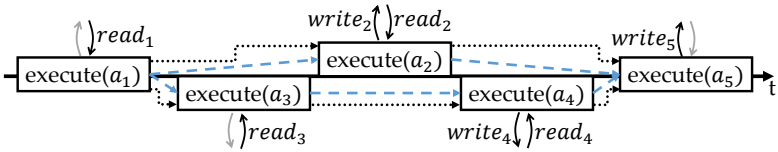


Figure 4.1.: Execution of an exemplary workflow

Definition 1. ***Effective Activity Execution:*** *Let $E(G,A_o,<)$ be the execution of a workflow G. An execution of an activity $a_o \in A_o$ is called effective iff it is not compensated before $E(G)$ completes. Let $A_e \subseteq A_o$ be the set of effective activities in A_o, then $e(E(G)) = (A_e,<)$ denotes the set of effective activities of $E(G)$ and their causal ordering relationship $<$.*

The interactions of an activity execution can only be considered input and output if the activity execution is not compensated. Thus, we only consider effective activity executions.

Definition 2. ***Effective Write Activity:*** *Let $E(G,A_o,<)$ be the execution of a workflow G. Then, $e_w(E(G)) = (A_w,<)$ denotes the effective write activity executions of $E(G)$ and their causal order, i.e., $A_w = \{a_e \in A_e : a_e$ is a write activity$\}$, where $e(E(G)) = (A_e,<)$.*

Compared to a non-replicated execution, the replicated execution has to execute the same effective write activities in the same partial order. The output produced by these activities, i.e., the write requests, should also be same. A write activity $a_w \in A_w$ produces the write request from the internal state and this internal state, in turn, is produced by the causally preceding activities.

Definition 3. ***Effective Causal History:*** *Let $a_x \in A_e$ be an effective activity in $e(E(G)) = (A_e,<)$. Then, the effective causal history of a_x is denoted as $H(a_x) = (A_H,<)$, where $A_H = \{a_e \in A_e : a_e < a_x)$ and $<$ defines the causal order of the activities in A_H.*

That is, $H(a_x)$ defines all causally preceding activities of a_x and their partial order. In other words, $H(a_x)$ contains all activities that might impact the input internal state of a_x. Figure 4.1 shows the execution of a workflow consisting of

activities a_1 to a_5. The dashed arrows indicate the control flow and the dotted arrows indicate the data flow. The set A_H of $H(a_5)$ is $\{a_1, a_2, a_3, a_4\}$ with the partial ordering $a_1 < a_2$ and $a_1 < a_3 < a_4$. For instance, a_2 and a_3 are concurrent and, thus, the execution order does not affect a_5. Likewise, a_4 is not affected by a_2 because the activities are concurrent.

Definition 4. ***Single-Execution-Equivalence:*** *Let $E^R(G)$ be the replicated execution of a workflow G. $E^R(G)$ is Single-Execution-Equivalent iff there exists some non-replicated execution $E^N(G)$ for which the following conditions hold:*

1. $e_w(E^R(G)) = e_w(E^N(G))$
2. $\forall a_w^R \in A_w^R, a_w^N \in A_w^N : a_w^R = a_w^N \Rightarrow H(a_w^R) = H(a_w^N)$,
   where $e_w(E^R(G)) = (A_w^R, <^R)$ *and* $e_w(E^N(G)) = (A_w^N, <^N)$

The first condition states that the replicated execution executes the same write activities in the same order as the non-replicated execution. The second condition states that the replicated and the non-replicated execution have the same Effective Causal History for each of these effective write activities.

Theorem 1. *A replicated execution of a workflow G that is Single-Execution-Equivalent is correct.*

Proof sketch. We show that a replicated execution $E^R(G)$ and a non-replicated execution $E^N(G)$ produce the same output, i.e., the same write requests, given that Def. 4 is fulfilled and given that both the replicated and the non-replicated executions are provided with the same input, i.e., the same reply messages. Any instance of workflow G starts with the same initial internal state. The determinism of each activity ensures that given an internal state and a reply, an activity always produces the same internal state. Because the Effective Causal Histories $H(a_w)$ of a write activity a_w are same in $E^R(G)$ and $E^N(G)$ (cf. Definition 4.2), the internal state that is the input for a_w is same in $E^R(G)$ and $E^N(G)$. Because every activity produces the write request deterministically based on the internal state, the content of the write requests is also same in $E^R(G)$ and $E^N(G)$. Moreover, the effective write activities are executed in the same partial order in $E^R(G)$ and $E^N(G)$ (cf. Definition 4.1). As a consequence, any replicated execution $E^R(G)$ has to produce the same write requests from the same input as a non-replicated

execution $E^N(G)$. From Axiom 1 follows that a replicated execution that is Single-Execution-Equivalent is correct. $\square$

5. Replication Schemes

In order to ensure availability, we propose to replicate the workflow engine and, thereby, the workflows that are executed by the engines. In case of a failure of one workflow engine replica, the remaining replicas can continue the workflow executions of the failed engine. In this chapter, we propose two replication schemes: *majority-based replication* and *flexible failover replication*.

Majority-based replication is based on a primary-backup replication scheme [AD76], where the *primary* replica executes the workflow and replicates the produced state on the other replicas – called *backups*. For electing a primary, we use a majority election [vEVS02]. Consequently, the workflow execution can always continue – even if the primary fails – as long as a majority of replicas is available for electing a new primary. Hence, we call the replication scheme *majority-based replication*.

Flexible failover replication relaxes the requirement for electing a new primary. In specific, the threshold of votes that is is required for being elected as primary can be set flexibly when using the flexible failover replication scheme. This allows the workflow execution to continue even if a majority of replicas has failed, increasing the workflow execution's availability. However, when the vote threshold is set to a low value, there may be many primaries (in different network partitions) that compete for finishing the execution increasing the overhead induced by the replication scheme.

In the following two sections, both schemes will be discussed in detail. This includes a discussion of the advantages and shortcomings of the schemes. For simplifying the presentation and increasing understandably, the schemes will only address workflows consisting of a sequence of activities. In the final section of this chapter, we will extend the schemes for supporting XOR- and AND-branching, loops, non-compensable activities, and complex interactions.

5.1. Majority-based Replication

In the following, we present our majority-based replication scheme. We will first describe a basic version of the scheme, where we strictly rely on synchronous updates when replicating the state, which the primary produces through each activity execution, on the backups. Afterwards, we will relax the basic version by also supporting asynchronous updates, which leads to greater flexibility on controlling the overhead that the state replication implies. Additionally, we extend the scheme to support active replication, such that the scheme can switch between passive replication, i.e., primary-backup replication, and active replication, where all replicas concurrently execute the activities of the workflow. Finally, we describe the *compensation mode*, where the user can initiate the compensation of a part or even the complete replicated workflow execution by coordinating the required compensations on the different replicas.

5.1.1. Basic Majority-based Replication Scheme

An workflow client application [Hol95] can initiate the replicated execution of a workflow by sending a message to the workflow engine replicas of our system (cf. Figure 5.1). The *primary* replica is responsible for executing the workflow while the *backup* replicas store the *execution state* (or *eS* for short), which is sent by the primary, in volatile memory. The primary processes the activities of the workflow, where each activity execution produces an output internal state, which serves as input for the next activity, if any. Whenever an activity execution has been finished, the primary sends an execution state update to the backup replicas (cf. Figure 5.1). Each transferred execution state includes a pointer to the activity that is to be performed next plus this activity's input internal state. Additionally, each execution state is associated with a so-called *state number* (or *s* for short), which is incremented with each activity execution. Backups only accept an execution state update if it has a higher state number than their currently stored execution state. An accepted execution state replaces the currently stored execution state.

When the primary fails or partitions from the network, the available backup replicas elect a new primary. We use a majority election [vEVS02] requiring at least $f+1$ replicas to participate in a successful election. The new primary

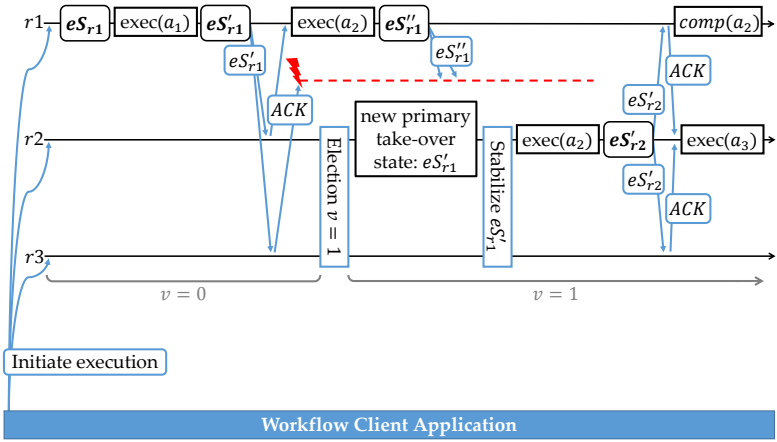


Figure 5.1.: Overview of the functionality of the basic majority-based replication scheme.

collects the execution states of the replicas that participate in the election and selects one of these execution states as the *take-over state*. The selected take-over state tells the primary where to continue the workflow execution, i.e., the activity to be performed next as well the input internal state for this activity. In Figure 5.1, replica $r2$ becomes the new primary and selects eS'_{r1} as the take-over state after the primary $r1$ has partitioned.

Similar to other replication schemes [LC12, OO14], we use the concept of *views* for indicating different phases of the execution. A new view starts whenever a new primary has been elected and, hence, a single primary exists in each view. Every view has a unique *view number* and view numbers are strictly monotonically increasing over time. We call the view with the highest view number the *current view*, and the primary of this view is the *current primary*.

The take-over state mentioned above is determined during the election procedure. Within the context of a workflow execution, an execution state has a unique identifier sID. The identifier consists of a view number $sID.v$ and a state number $sID.s$. The former identifies the view in which this execution state was produced. The latter uniquely identifies the state within this view. Through their identifiers, execution states can be totally ordered: an execution state sID is more recent than a state sID' iff $(sID.v > sID'.v)$ or $((sID.v = sID'.v) \wedge (sID.s > sID'.s))$.

To determine the take-over state, the new primary collects the execution states from a majority of replicas and selects the most recent state out of these. Note that the replicas which do not provide their execution state (e.g., because of slow messages or network partitioning) to the new primary might store a more recent state. In other words, the take-over state might not be the most recent execution state in the system. With respect to Single-Execution-Equivalence, if the old primary processed activities following the take-over state, these activities are invalid because the new primary continues the execution starting from the take-over state. Clearly, all invalid activities need to be compensated, where each compensation may incur compensation cost.

A similar problem arises if there exist multiple primaries at the same time. Due to arbitrary slow messages or partitioning, the backup replicas might start an election although the old primary is still operational, leading to multiple primaries. Now, the problem is that the old primary continues executing the workflow, which increases the compensation cost. In order to stop the old primary, we require a primary to validate whether it is still current primary after each activity execution. Here, we exploit the execution state update that the primary sends after each activity execution. The primary only starts to execute the next activity after a majority of replicas (including itself) have acknowledged the reception of the update (cf. Figure 5.1). Since the election of a new primary also requires a majority, out of a majority always at least one replica knows about the current primary preventing old primaries from receiving enough acknowledgement for continuing the execution. Thus, an old primary at most executes one activity before stopping its execution. Consequently, an old primary might incur the compensation cost of one activity at most.

However, an old primary does not know from which execution state the new primary took over. To ensure Single-Execution-Equivalence, the new primary needs to inform the old primary about the state that it chose as take-over state. Then, the old primary can determine whether or not to compensate the last activity execution. In Figure 5.1, replicas $r2$ and $r3$ change to a new view after the primary $r1$ has partitioned. The new primary $r2$ determines eS'_{r1} as the take-over state. Hence, $r1$ has to compensate the execution of activity a_2.

As pointed out above, the new primary determines the take-over state during the election. After that, the new primary stabilizes this state, i.e., it makes

sure that at least $f+1$ replicas store that state in volatile memory. Only after a successful stabilization, it continues executing the workflow. Since at most f replicas may fail, the selected take-over state will never be lost due to failures. Future elections will never select a take-over state preceding this state because the election protocol collects the execution state from at least $f+1$ replicas and selects the most recent out of these. Consequently, the take-over state is stable in the sense that the activity execution of the old primary that produced this state will never be compensated. The activity executions following the take-over state on the old primary, however, have to be compensated. In other words, for deciding which activities to compensate, an old primary only needs to know from which of its execution states the new primary took over. Even if the replica was primary several times, the replica only needs to learn the last state from which a new primary took over because we require an old primary to schedule the necessary compensations before becoming primary again.

How does an old primary learn about the state from which the new primary took over? Because the take-over state is selected by the new primary during election, the old primary must be informed about this state from the new primary – or other replicas that already learned about the take-over state from the new primary. For this purpose, we introduce the *stable-states vector*, which includes one component per replica. The component of a replica r identifies the last execution state produced by r that was used as a take-over state by a new primary. If r has not been primary yet, r's component is undefined. The stable-states vector is updated during each election: The state number $sID.s$ of the selected take-over state is written into the vector component of the producer of that state. Here, it is not necessary to save the view number $sID.v$ of the take-over state's identifier sID because our replication scheme ensures that a replica compensates all invalid activities before becoming primary in a higher view again. Consequently, a replica's component in the stable-states vector always identifies the last view where the replica acted as primary. For example, assume the execution state eS'_{r1} in Figure 5.1 has the identifier $(v=0, s=1)$. Thus, $r2$ updates the stable-states vector component of $r1$ to $s=1$ during the election.

The updated stable-states vector is stabilized as part of the election and, thus, cannot be lost through failures. The current stable-states vector is included in each execution state update sent from a primary to the backup replicas. Consequently,

each old primary will eventually be informed about its last produced take-over state and, hence, will eventually compensate its invalid activities. In Figure 5.1, the old primary $r1$ receives the execution state eS'_{r2} after reconnecting to the network. The stable-states vector of eS'_{r2} specifies that the new primary selected $r1$'s execution state with the state number $s = 1$ as the take-over state. Thus, the activity executions a_2 is invalid and, hence, compensated.

After the current primary has performed the last activity of the workflow, it starts the termination protocol, which consists of two phases. The first phase ensures agreement on the final outcome of the workflow, and requires at least $f + 1$ replicas to be available. After this phase, the workflow execution is complete and the workflow client application is informed about the outcome of the workflow execution. However, at this point the replicas cannot forget the workflow execution. In particular, they would have to store the stable-states vector forever because they never could be sure that each replica has received the final stable-states vector and compensated its invalid activities. The second phase enables the replicas to forget the workflow execution after all replicas have compensated their invalid activities. While the first phase of termination only needs a majority of replicas to be available, each replica needs to participate in the second phase. However, this is not critical as the second phase of termination is completely decoupled from the workflow client application. In other word, if the termination protocol is blocked in the second phase, this has no effect at all on the workflow client application.

5.1.1.1. Data Structures

Each replica r holds a *compensation log* (or $cLog_r$ for short) on stable storage. The log contains a *compensation record* for each activity executed by r. This record comprises the compensation handler of the executed activity – including all information required for the compensation (e.g., variable values of the internal state) – as well as the state identifier of the produced execution state. The record is synchronously written to the *cLog* before the activity is executed. We assume the compensation handler's logic to be able to identify if the corresponding activity was only partly executed (or not at all) in case the replica fails during (or before starting) the activity execution. To trigger the compensation, the record is

Stable Storage	
$cLog_r$	compensation log
Volatile Memory	
eID_r	workflow execution's identifier
G_r	workflow model that is executed
v_r	current view of replica r
eS_r	execution state currently stored by r
$eS_r.next$	next activity to execute
$eS_r.\sigma$	internal state that is input for the next activity execution
$eS_r.prod$	ID of the replica that produced eS
$eS_r.sSVec$	stable states vector associated with eS_r
$eS_r.sID$	identifier of the execution state
$eS_r.sID.v$	view in which eS_r was produced
$eS_r.sID.s$	state number that the producer assigned to eS_r

Table 5.1.: Data structures kept by replica r for a replicated workflow execution

sent to the Compensation Unit. Through the included unique state identifier, the Compensation Unit can filter duplicated compensation requests.

All other information used by the replication scheme is stored in a *workflow record* in volatile memory. For each ongoing workflow execution, each involved replica r maintains the following information in the workflow record:

- eID_r: execution identifier uniquely identifying the workflow execution.
- G_r: workflow model executed by eID_r.
- v_r: current view number from r's perspective. As in [LC12], we assume a well-known function mapping each view number to the primary replica of this view: function $Primary(v)$ returns the identifier of the replica that is the primary of view v.
- eS_r: eID_r's most recent execution state known to the replica. In the case of a primary, this is the execution state of the last completed activity execution, while for a backup replica this is the most recent execution state received in an update.

For one replicated execution, *eID* and G are identical on all replicas. In contrast, the current view number and the execution state might deviate between replicas. In particular, the stored execution state eS_r comprises the following information:

- $eS_r.next$: pointer to the activity that is to be performed next.
- $eS_r.\sigma$: input internal state for the execution of $eS_r.next$. Note that $eS_r.\sigma$ is the output state of the last executed activity (or the initial internal state).
- $eS_r.prod$: identifier of the replica that produced eS_r.
- $eS_r.sSVec$: stable-states vector associated with eS_r. The stable-states vector includes all information needed by the old primaries to decide which activities to compensate. The vector is implemented as a list of pairs (r_x, s_x), where the former identifies a particular replica r_x, and the latter is the state number s_x of the latest take-over state produced by r_x. An entry only exists for replicas that were primary in the past.
- $eS_r.sID$: unique identifier of the execution state eS_r.

The state identifier *sID* of an execution state is unique across all execution states of a replicated workflow execution. In specific, the state identifier consists of the following variables:

- $eS_r.sID.v$: view number in which eS_r was produced.
- $eS_r.sID.s$: state number assigned by the producer $eS_r.prod$.

We summarized all data structures that are kept in volatile memory and on stable storage for one replicated workflow execution in Table 5.1.

5.1.1.2. Normal Operation

An application initiates the replicated execution of a workflow by sending an INIT message to each involved replica. The INIT(*eID*, *mID*) message contains two fields: the identifier *eID* that uniquely identifies the replicated execution and the identifier *mID* of the workflow model that shall be executed.

A replica, say r, receiving an INIT message generates a workflow record for *eID*, loads the workflow model of *mID* from the workflow repository, and

initializes v_r to 0. In case r is the initial primary (i.e., $r = Primary(0)$), r instantiates the loaded workflow model. The instantiation of the workflow model generates an initial internal state, which is saved in $eS_r.\sigma$. Moreover, the stable-states vector stored in eS_r includes an entry $(r,0)$ for the initial primary $r = Primary(0)$, while no other replica has an entry yet. The execution state identifier $eS_r.sID$ has both $eS_r.sID.v$ and $eS_r.sID.s$ set to 0, which reflects that the initial execution state is produced in view 0 and before any activity was executed (since the state number is incremented with every activity execution).

A primary executes the workflow as follows (cf. Algorithm 1 line 1-11):

1. The primary writes a begin record to the compensation log if it has not been primary before.

2. The primary updates its execution state and writes the compensation record for the activity execution to the compensation log.

3. The primary executes the activity.

4. To replicate the produced state on the backups, the primary sends an UPDATE message to each backup, which contains the produced execution state. The UPDATE messages are sent synchronously, where the primary waits for a majority to acknowledge the reception of the update before it continues the execution by returning to step 2.

Upon receiving an UPDATE message (cf. Algorithm 1 line 12-19), a backup replica, say r, checks whether the UPDATE is valid. An UPDATE is valid if it is (1) sent by the replica that r considers to be the current primary and (2) comprises an execution state, which is more recent than r's current execution state. The first condition is fulfilled if the view number of the UPDATE is higher or equal to r's view number. In specific, the view number can only be higher if a failure caused an election during which the view number is incremented. Thus, we discuss the handling of a higher view number in an UPDATE message (cf. Algorithm 1 line 13-16) when describing the election procedure in the following section. For checking the second condition, r compares the state identifier of its execution state $eS_r.sID$ with the received execution state $eS_x.sID$. All invalid UPDATE messages are discarded. In contrast, a valid UPDATE is applied, where applied

Algorithm 1: Basic majority-based replication scheme on replica r (Part I)

```
   // normal operation
1  if r = Primary(v_r) then
2      if v_r = 0 OR eS_r.sSVec has no entry for r then
3          write (begin, eID) to cLog;
4      while eS_r.next ≠ null AND r = Primary(v_r) do
5          eS_r.sID.s := eS_r.sID.s + 1;
6          eS_r.prod := r;
7          write (eID, eS_r.sID, comp(eS_r.next)) to cLog_r;
8          eS_r.σ := execute(eS_r.next, eS_r.σ);
9          eS_r.next := succ(eS_r.next, G);
10         send UPDATE(eID, eS_r) to all backups;
11         WAIT UNTIL(received ACK(eID, eS_r.sID) from f replicas);
12 upon Receive UPDATE(eID, eS_x) from replica x do
13     if eS_x.sID.v > v_r then // r has an old view number
14         if r = Primary(v_r) then // r is an old primary
15             scheduleCompensations(eS_r.sSVec);
16         v_r := eS_x.sID.v;
17     if eS_x.sID.v = v_r AND eS_x.sID > eS_r.sID then // *
18         eS_r := eS_x;
19         reply ACK(eID, eS_x.sID) to x;
   // * = checking the validity of a message
```

means that r overwrites its execution state with the received one. After applying the execution state, r replies to the primary with an ACK message, which contains the execution state's unique state identifier sID. Since the primary is waiting for a majority of ACK message for its currently produced state, it uses the state identifier sID to filter delayed ACK messages from old UPDATES.

5.1.1.3. Election

The backups monitor the availability of the current primary by means of a heartbeat mechanism. If the current primary crashes or partitions, the backups elect a new primary. In specific, the election consists of three phases. In the first phase, the view number is incremented and, then, stabilized by replicating it on at least $f+1$ replicas. This ensures that a replica will never return to a smaller view number even after a crash failure. In the second phase, the new primary collects

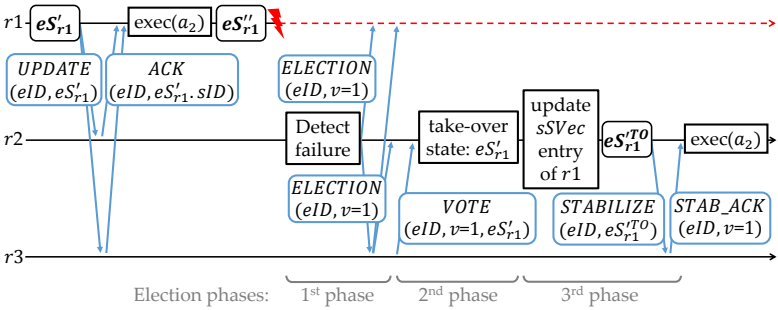


Figure 5.2.: Exemplary election after the crash failure of primary $r1$.

the execution state from a majority of replicas and selects the most recent state as the take-over state. In the third phase, the new primary stabilizes the take-over state before continuing the workflow execution. The stabilization ensures that future elections will never select a take-over state preceding this state. In the following, we will describe these three phases in further detail.

In the first phase, any replica that detects the failure of the current primary increments its view number. Then, the replica sends an ELECTION message to all other replicas, where the message includes the incremented view number (cf. Algorithm 2 line 1-3). In our example depicted in Figure 5.2, replica $r2$ starts an election for view 1 after the primary of view 0 (i.e., replica $r1$) has crashed.

Upon receiving a valid ELECTION(eID, v_x) message (cf. Algorithm 2 line 4-9), a replica r updates its view number v_r and sends the ELECTION message to all replicas as well. Once r has received the ELECTION message from f replicas, the new view number v_x is stabilized (i.e., v_x is saved by f replicas and r itself) and the second phase starts. Now, r sends a VOTE message to $Primary(v_x)$, where the VOTE includes the view number and execution state of the voter.

When $Primary(v_x)$ has received f valid VOTE messages (cf. Algorithm 2 line 10-18), it selects the most recent execution state from the available $f+1$ states (i.e., from the f VOTE messages plus its own state) as the take-over state. Here, a VOTE is valid iff the VOTE's view number is equal to the view number of the ongoing election.

Algorithm 2: Basic majority-based replication scheme on replica r (Part II)

```
   // election
1  upon detect that Primary(v_r) is failed do
2      v_r := v_r + 1;
3      send ELECTION(eID, v_r) to all replicas;
4  upon Receive ELECTION(eID, v_x) from replica x do
5      if v_x ≥ v_r AND v_x > eS_r.sID.v then // *
6          v_r := v_x;
7          send ELECTION(eID, v_r) to all replicas;
8          if received ELECTION of view v_x from f replicas then
9              send VOTE(eID, v_r, eS_r) to Primary(v_r);
10 upon Receive VOTE(eID, v_x, eS_x) from replica x do
11     if v_x = v_r AND received VOTE from f replicas then // *
12         eS_r :=mostRecentStateOf(received VOTE messages);
13         scheduleCompensations(eS_r.sSVec);
14         remove the entry of eS_r.prod from eS_r.sSVec (if any);
15         add (eS_r.prod, eS_r.sID.s) to eS_r.sSVec;
16         eS_r.sID.v := v_r;
17         eS_r.prod := r;
18         send STABILIZE(eID, eS_r) to all backups;
19 upon Receive STABILIZE(eID, eS_x) from replica x do
20     if eS_x.sID.v ≥ v_r AND eS_x.sID > eS_r.sID then // *
21         v_r := eS_x.v;
22         eS_r := eS_x;
23         send STAB_ACK(eID, eS_x.sID.v) to x;
24 upon Receive STAB_ACK(eID, v_x) from replica x do
25     if v_x = v_r AND received STAB_ACK of v_r from f replicas then // *
26         return to normal operation;
   // * = checking the validity of a message
```

Now, the final phase of the election begins. Because the new primary might have been primary before, it compensates all its invalid activities based on the stable-states vector of the selected take-over state. Then, the new primary updates the stable-states vector for informing the execution state producer, i.e., the old primary from which the new primary takes over, about the selected take-over state eS^{TO} as follows: It sets the entry of the take-over state's producer $eS^{TO}.prod$ in the stable-states vector to $eS^{TO}.s$ (cf. Algorithm 2 line 14-15). Afterwards, the new primary changes the view number of the take-over state to the current view number. Thereby, this updated take-over state becomes the most-recent execution state of the system. Any later primary will continue at least from this updated take-over state as soon as the state has been replicated on a majority of nodes, i.e., as soon as the state has been stabilized. Before starting the stabilization, the new primary sets itself as the producer of this updated take-over state. For example, assume that the state eS'_{r1} in Figure 5.2, which $r2$ selects as its take-over state, has the identifier $(sID.v = 0, sID.s = 1)$. Then, the updated take-over state eS'^{TO}_{r1} has the identifier $(sID.v = 1, sID.s = 1)$ and the producer $eS'^{TO}_{r1}.prod = r_2$.

To stabilize the updated take-over state, the new primary sends a STABILIZE message that contains the state to each backup. Upon receiving a valid STABILIZE message (cf. Algorithm 2 line 19-23), a backup saves the new view number (because the replica might not have participated in the election so far) and the included execution state. Afterwards, the backup acknowledges the reception of the STABILIZE message with an STAB_ACK message.

When the primary has received an STAB_ACK message for the current view number from at least f replicas (cf. Algorithm 2 line 24-26), a majority of replicas has saved the take-over state and, thus, the state is stabilized. Now, the election is finished and the new primary returns to normal operation.

When an old primary, say r, reconnects after being partitioned from the network, it will eventually receive an UPDATE from the new primary, which contains the stable-states vector produced during the election. This stable-states vector allows the old primary to identify if any of its activity executions are invalid, i.e., have to be compensated (cf. Algorithm 1 line 13-16). In specific, r's entry in the stable-states vector (r, s) specifies that the execution state from which a new primary took over from r has the state number s. If r has a compensation

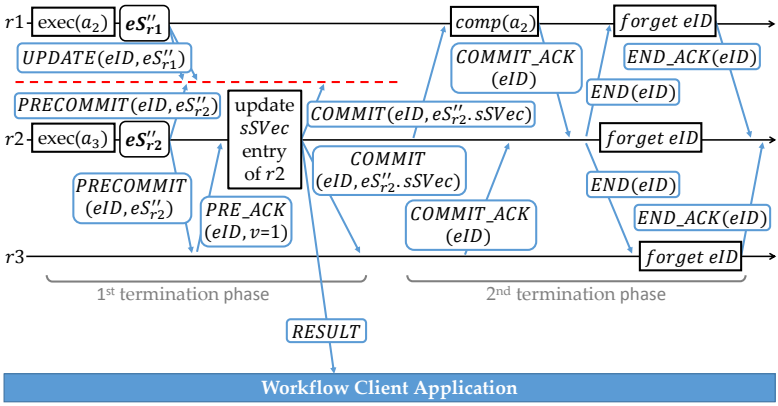


Figure 5.3.: Termination of the execution shown in Figure 5.1.

record in $cLog_r$ with a higher state number $s' > s$, this marks an invalid activity execution. Hence, the compensation record is sent to the Compensation Unit.

If the primary of the ongoing election fails, the backups again time out and restart the election for the next view number.

5.1.1.4. Termination

After the last activity has been executed, the workflow execution terminates in two phases. In the first phase, the replicas agree on the final outcome of the execution, which then is reported to the workflow client application. The second phase is only needed to allow the replicas to forget about the workflow execution, i.e., to drop the workflow execution's data that is stored in volatile memory.

In practice, there might exist multiple primaries, i.e., several old primaries and the current primary. For preventing that these compete for reporting their result to the workflow client application, we only allow the current primary to report its result. In particular, at least $f+1$ replicas have to agree that the current primary's produced state is the final execution state before the primary sends its result.

For this purpose, any replica that assumes to be the current primary and that has executed the last activity, sends a PRECOMMIT message, which includes the current execution state of that primary, to all replicas (cf. Algorithm 3 line 1-2).

Algorithm 3: Basic majority-based replication scheme on replica r (Part III)

```
   // termination
1  upon eS_r.next = null AND r = Primary(v_r) do
2      send PRECOMMIT(eID, eS_r);
3  upon Receive PRECOMMIT(eID, eS_x) from replica x do
4      if eS_x.sID.v ≥ v_r AND eS_x.sID > eS_r.sID then // *
5          v_r := eS_x.sID.v;
6          eS_r := eS_x;
7          scheduleCompensations(eS_r.sSVec);
8          send PRE_ACK(eID, eS_x.sID.v) to x;
9  upon Receive PRE_ACK(eID, v_x) from replica x do
10     if received PRE_ACK of v_x from f replicas then // *
11         remove the entry of eS_r.prod from eS_r.sSVec (if any);
12         add (eS_r.prod, eS_r.sID.s) to eS_r.sSVec;
13         send result to workflow client application;
14         send COMMIT(eID, eS_r.sSVec) to all backups;
15 upon Receive COMMIT(eID, sSVec_x) from replica x do
16     scheduleCompensations(sSVec_x);
17     send COMMIT_ACK(eID) to x;
18 upon Receive COMMIT_ACK(eID) from replica x do
19     if received COMMIT_ACK from all replicas then
20         write (end, eID) to cLog;
21         send END(eID) to all backups;
22         forget eID;
   // * = checking the validity of a message
```

A replica that receives a valid PRECOMMIT message (cf. Algorithm 3 line 3-8), stores the included view number and execution state. Then, the replica schedules the compensation of all invalid activities (if any) based on the execution state's stable-states vector. Afterwards, the replica replies to the PRECOMMIT sender with a PRE_ACK message.

When the primary, say r, receives a PRE_ACK message from f backups for the current view number v_r, the state has been saved by a majority and, thus, is stabilized (cf. Algorithm 3 line 9-14). Hence, the primary updates the stable-states vector accordingly. Now, the primary can report the final outcome to the workflow client application. At the same time, the primary sends a COMMIT message including the final stable-states vector to all backups informing them that the first phase of termination has ended (cf. Figure 5.3).

Crashed or partitioned replicas might not have received the final stable-states vector yet. For example, replica $r1$ in Figure 5.3 did not receive the COMMIT message yet and, thus, has not compensated a_2 so far. Nevertheless, even without the second phase of termination, all replicas will eventually receive the final vector upon recovering from a crash failure (as discussed below) or through starting an election when reconnecting after a network partition. However, no replica knows which replicas have received the final stable-states vector already and which replicas have not. Thus, the replicas have to keep the stabilized stable-states vector in volatile memory.

To be able to delete the workflow execution's data (including the stable-states vector) from volatile memory, the termination has a second phase. Before forgetting the execution, the primary has to know that not only the majority but all replicas have compensated their invalid activities. The desired functionality is similar to the two phase-commit protocol [LS79, ML83], where the nodes only perform an action after all nodes agreed to be ready to perform the action. In our case, this desired action is to forget the execution. Here, the primary repeatedly sends the COMMIT message until all replicas have compensated their invalid activities and have acknowledged this by sending a COMMIT_ACK message (cf. Algorithm 3 line 15-17). Through sending a COMMIT_ACK, the replica agrees to be ready to forget.

After all replicas sent the COMMIT_ACK (cf. Algorithm 3 line 18-22), the system is ready to forget the workflow execution. Hence, the primary writes an *end record* to the compensation log. Before deleting all data of the workflow execution from the volatile memory, the primary sends an END message to all backups informing them that they also can write an end record to the compensation log and forget the execution. Here, the primary also repeatedly sends the END message until all replicas acknowledge the reception through an END_ACK message. For saving space on stable storage, we can now prune the compensation log by deleting all workflow executions which already have written an end record.

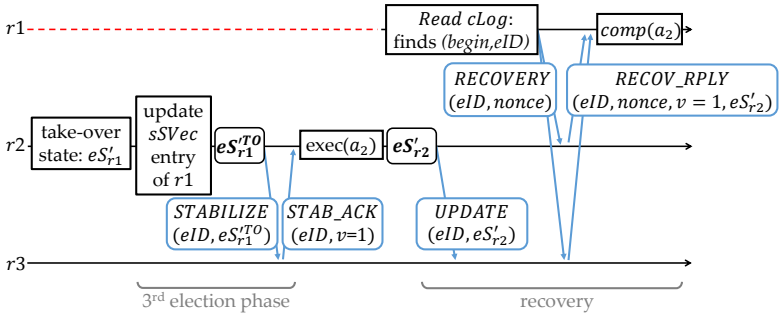


Figure 5.4.: Exemplary recovery of $r1$ after experiencing the crash failure depicted in Figure 5.2.

5.1.1.5. Recovery

Remember that all data of a workflow execution *eID* except the compensation log is stored in volatile memory. Through crash failures the data in the volatile memory is lost. To regain knowledge about the executions in which the replica participated before crashing, the replica reads its compensation log during recovery. Note, however, that a replica that has not been primary during the replicated execution of *eID* has no record on *eID* in its compensation log. Thus, we differentiate the following recovery states:

UNKNOWN: A replica that has either no record of *eID* or both a begin and an end record of *eID* in its compensation log.

ACTIVE: A replica has a begin record but no end record of *eID* in its compensation log.

Obviously, if a replica recovers in the UNKNOWN state, it cannot perform any recovery operation because it does not know any ongoing execution with the execution identifier *eID*. In contrast, a replica that recovers in the ACTIVE state has written a begin record for *eID* and, hence, has been primary before. Thus, it also logged every activity that it executed. Now, the replica has to receive the stable-states vector to determine if any of the logged activity executions are invalid and, thus, need to be compensated. Additionally, the recovering replica needs to receive the current view number and an according execution state for

being able to return to normal operation. In the following, we describe the recovery process by means of an example.

Algorithm 4: Basic majority-based replication scheme on replica r (Part IV)

```
   // recovery
1  upon Recovery from crash failure do
2      broadcast RECOVERY(eID, nonce);
3  upon Receive RECOVERY(eID, nonce) from replica x do
4      send RECOV_REPLY(eID, nonce, G_r.mID, v_r, eS_r) to x;
5  upon Receive RECOV_RPLY(eID, nonce, mID_x, v_x, eS_x) from replica x do
6      if received RECOV_RPLY for nonce from f + 1 then // *
7          load workflow model of mID_x from workflow repository;
8          v_r := highestViewOf(RECOV_RPLYs);
9          eS_r := mostRecentStateOf(RECOV_RPLY messages);
10         scheduleCompensations(eS_r.sSVec);
11         return to normal operation;
   // * = checking the validity of a message
```

Replica $r1$ in Figure 5.2 recovers in the ACTIVE state after its crash failure. The recovering replica requests the required information by sending a RECOVERY message, which includes a *nonce*, to all replicas (cf. Algorithm 4 line 1-2). A replica that receives a RECOVERY message replies with a RECOV_RPLY (cf. Algorithm 4 line 3-4) also including the *nonce*, which enables the recovering replica to filter replies of previous recoveries. Moreover, the reply comprises the workflow model ID *mID* as well as the replying replica's current view number and execution state.

After the recovering replica has received a valid RECOV_RPLY from a majority of replicas (cf. Algorithm 4 line 5-11), it loads the workflow model with the identifier *mID* from the workflow repository. Then, it determines the reply with the highest view number that contains the most recent execution state. It saves both the view number and the execution state of this reply. The recovering replica uses the stable-states vector of the saved execution state to determine its invalid activity executions and schedules the according compensations, if any (cf. Figure 5.2). Afterwards, the replica returns to normal operation.

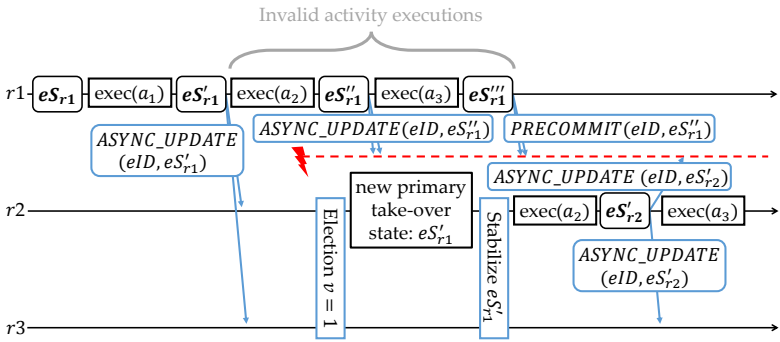


Figure 5.5.: Exemplary execution of a workflow with three activities using the asynchronous update mechanism.

5.1.2. Relaxed Majority-based Replication Scheme

Above we have defined a basic majority-based replication scheme that enables workflow engines to ensure available in the presence of failures. The basic majority-based replication scheme, however, requires a primary workflow engine to wait after each activity execution until a majority of replicas acknowledged the reception (and validity) of the UPDATE message. Thereby, the execution is basically paused after every activity implying a significant execution time overhead. This execution time overhead reduces the performance of executed workflow, which is of course undesirable.

The above problem can be solved by making the updates asynchronous. In specific, a primary sends a ASYNC_UPDATE(*eID*, *eS*) message to all backups after each activity execution. The backups treat ASYNC_UPDATE messages exactly like regular UPDATE messages except that the backups do not acknowledge the reception. Also, the primary does not wait for any acknowledgements. Instead, the primary directly continues the workflow execution by executing the next activity (cf. Figure 5.5).

Unfortunately, using asynchronous updates for the replicated execution of the workflow also imposes problems. When a primary does not check whether its updates are received and accepted by a majority, also old primaries will continue their execution. Instead of producing at most one invalid activity execution, the number of invalid activities are only limited by the remaining activities of

the workflow that still need to be executed. Since all invalid activities need to be compensated, the compensation cost might be tremendous when using asynchronous updates. Moreover, asynchronous updates may even impose a high failover time in the case of a primary failure: The asynchronous updates sent by the failed primary might be lost and, hence, a new primary might not receive the latest execution state (or even miss multiple execution state) leading to re-execution of some or all activities already executed by the old primary.

Since both extremes – synchronous and asynchronous updates – have shortcomings, we introduce *synchronization groups*, which contain a number of consecutive activities of the workflow. All activities of the group are executed using asynchronous updates. However, after the primary has executed the complete group, the primary uses the synchronous update mechanism that was introduced with the basic majority-based replication scheme. Consequently, the synchronization groups allow to flexibly decide between limiting the compensation overhead induced by old primaries – by having synchronization groups contain only few activities – and keeping the execution time overhead of synchronous updates low – by comprising many activities within one synchronization group. For realizing the synchronization groups within our majority-based replication scheme, we discuss only the parts that have to be adapted. In specific, we only need to adapt the normal operation by adding an asynchronous update mechanism. Afterwards, we further relax the synchronization grouping mechanism by allowing some synchronization groups to use active replication in our *relaxed majority-based replication scheme*.

5.1.2.1. Adapting Normal Operation

For enabling the primary to decide between using an asynchronous update or a synchronous update, the primary has to consider the grouping, which – for now – we assume to be given. All activities of the group are executed using asynchronous update messages (cf. Algorithm 5 line 6-13), which also means that there is no need for acknowledging the reception of the asynchronous updates (cf. Algorithm 5 line 16-22). The primary directly continues to execute the next activity of that synchronization group.

Algorithm 5: Relaxed majority-based replication scheme on replica r

```
   // normal operation
1  if r = Primary(v_r) then
2      if v_r = 0 OR eS_r.sSVec has no entry for r then
3          write (begin, eID) to cLog;
4      while eS_r.next ≠ null AND r = Primary(v_r) do
5          currGroup := SyncGroup(eS_r.next);
6          while eS_r.next ∈ currGroup AND r = Primary(v_r) do
7              eS_r.sID.s := eS_r.sID.s + 1;
8              eS_r.prod := r;
9              write (eID, eS_r.sID, comp(eS_r.next)) to cLog_r;
10             eS_r.σ := execute(eS_r.next, eS_r.σ);
11             eS_r.next := succ(eS_r.next, G);
12             if eS_r.next ∈ currGroup then
13                 send ASYNC_UPDATE(eID, eS_r) to all backups;
14         send UPDATE(eID, eS_r) to all backups;
15         WAIT UNTIL(received ACK(eS_r.sID) from f replicas);
16 upon Receive ASYNC_UPDATE(eID, eS_x) from replica x do
17     if eS_x.sID.v > v_r then // r has an old view number
18         if r = Primary(v_r) then // r is an old primary
19             scheduleCompensations(eS_r.sSVec);
20         v_r := eS_x.sID.v;
21     if eS_x.sID.v = v_r AND eS_x.sID > eS_r.sID then // *
22         eS_r := eS_x;
   // * = checking the validity of a message
```

After the primary has executed the complete group, it uses the synchronous update mechanism that was introduced with the basic majority-based replication scheme (cf. Algorithm 5 line 4-15). In the example depicted in Figure 5.6, the activities a_1 and a_2 are contained in the same synchronization group removing the execution time overhead of waiting for acknowledgements after the execution of a_1. However, the primary has to wait for f ACK messages after executing a_2. This stops the execution of the old primary $r1$, reducing the compensation cost compared to purely using asynchronous updates as depicted in Figure 5.5.

In conclusion, the synchronization groups allow to limit the compensation cost and the failover time imposed by a failure. On the other hand, each synchronization group imposes an execution time overhead for using the synchronous update

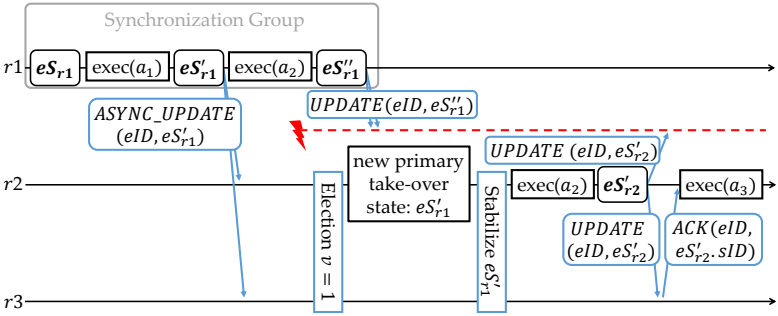


Figure 5.6.: Exemplary execution of a workflow using synchronization groups.

after the last activity of the group. In the following, we will discuss grouping guidelines that strive for an optimal trade-off in terms of compensation cost, failover time, and execution time overhead.

5.1.2.2. Enabling Hybrid Replication

The synchronous update after each synchronization group introduces execution time overhead for replicating the execution state on the backups. In the following, we exploit active replication to mitigate this time overhead.

Using active replication, each workflow is executed on all $2f+1$ replicas independently. Consequently, also each activity of the workflow is executed $2f+1$ times and no state information needs to be transferred between replicas. However, if the workflow contains a write activity, $2f$ executions of that activity need to be compensated to fulfill Single-Execution-Equivalence (cf. Definition 4). In practice, the replicas finish the complete workflow execution and, then, perform a majority consensus to reach an agreement on the execution of one of the replicas. All other replicas compensate their entire workflow execution (cf. Figure 5.7). Of course, compensating all activities of the workflow $2f$ times causes substantial compensation cost.

Now, consider that the entire workflow consists only of read-only activities. Then, compensation is not required because executing read-only activities multiple times does not violate Single-Execution-Equivalence (cf. Definition 4). However, the replicas might produce different results because the called services

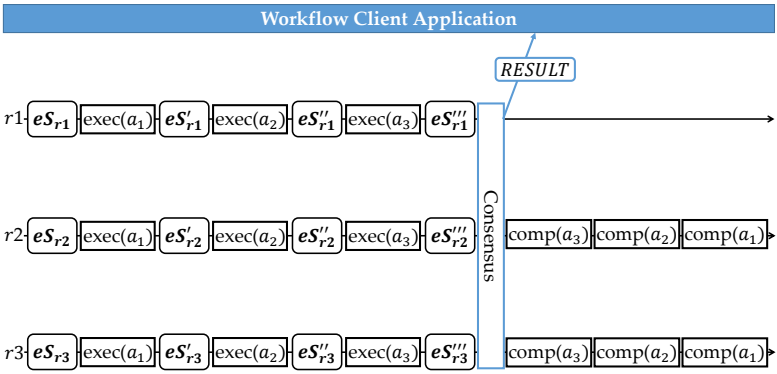


Figure 5.7.: Exemplary execution of a workflow using active replication, where the replicas agree on replica $r1$'s execution meaning that $r2$ and $r3$ have to compensate their executions.

might return different replies to each replica. For example, when calling a random number generator, each replica will receive a different number. Thus, the replicas still would need to perform a consensus to agree on one replica that reports its result to the workflow client application.

Consider now that all activities of the workflow are also *deterministic*, meaning that the activity execution will always produce the same output internal state given the same input internal state. For example, an activity that calls a service offering a function, where the request includes all parameters, is deterministic (such as a calculator service). Consequently, all replicas will produce the same result when all activities of a workflow are deterministic. When the activities are read-only as well, neither consensus for agreeing on one execution nor compensation is required.

Of course, only few workflows are composed solely of deterministic, read-only activities. Hence, we propose a hybrid replication scheme that allows to switch between passive and active replication. The hybrid replication scheme uses active replication for synchronization groups that only comprise deterministic, read-only activities. For any group that comprises a non-deterministic or write activity, the hybrid scheme will use the passive replication scheme that we defined above.

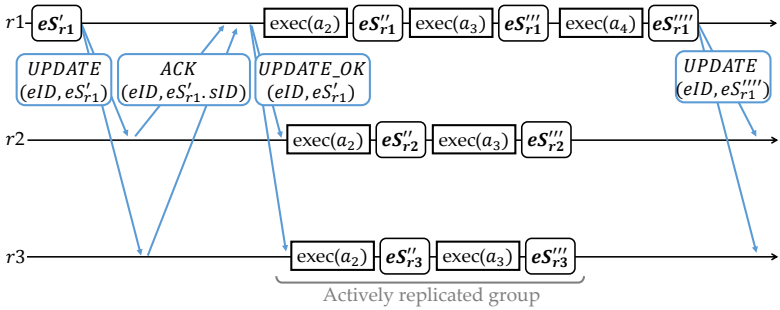


Figure 5.8.: Exemplary execution with our relaxed majority-based replication scheme, which allows combining active and passive replication. The execution states eS'''_{r1}, eS'''_{r2}, and eS'''_{r3} are identical because a_2 and a_3 are deterministic, read-only activities.

In specific, an actively replicated group is executed as follows. After the primary has sent a synchronous update of the previous synchronization group, it sends an UPDATE_OK(eID, eS) message to each backup. Upon receiving a valid UPDATE_OK, the backup saves the execution state and starts to execute the actively replicated synchronization group (cf. Figure 5.8). An UPDATE_OK is valid if it was send by the current primary and includes the same or a more recent state than the one the backup currently stores. After the actively replicated group has been executed, the primary and the backups have produced identical execution states. Thus, the primary directly continues by executing the next group. For example, the primary $r1$ in Figure 5.8 directly executes the passively replicated activity a_4 after finishing the actively replicated group of a_2 and a_3.

Now, we can exploit the benefits of both replication mechanisms when appropriately dividing the workflow into synchronization groups. The workflow designer, as a domain expert, has to specify the grouping at design time. In the following, we present guidelines for helping the designer specify this grouping.

As already stated above, each part of the workflow that solely consists of deterministic, read-only activities should be executed using active replication. Thus, all deterministic, read-only activities are included in actively replicated synchronization groups. But how should a workflow designer specify the passively replicated groups for the remaining activities? The general goal is to

keep the time overhead induced by the synchronous updates after each passively replicated groups low, while restricting the failover time and compensation cost induced by a single failure. For this purpose, the designer specifies a compensation cost threshold $c_t \in \mathbb{N}$ and a failover time threshold $t_t \in \mathbb{N}$ in ms, which define the maximum compensation cost and failover time that a single failure induces. Let $c : A \rightarrow \mathbb{N}$ be the function specifying the compensation cost of an activity and $t : A \rightarrow \mathbb{N}$ specifying the execution time in ms, where these values are either provided by the workflow designer or monitored and learned from past executions.

A passively replicated synchronization group P has to fulfill the following conditions: (1) $t_e + \sum_{a \in P} t(a) \leq t_t$ and (2) $\sum_{a \in P} c(a) \leq c_t$, where t_e is the time needed for an election. The first condition ensures that the failover time, i.e., time needed for an election and re-executing of the whole synchronization group – which might be necessary in the worst case – is below t_t. The second condition ensures that in the case that all activities are executed by both the new and the old primary, the compensation cost stays below c_t.

5.1.3. User Initiated Compensations

A user might need to initiate the compensation of parts of the workflow, e.g., to rerun a specific part [SK11, SK12], or even trigger the compensation of the complete workflow, realizing a functionality that is akin to an abort of a transaction. However, our majority-based replication schemes do not support user initiated compensations so far. Actually, every execution state that has been stabilized, i.e., that has been replicated on a majority of replicas, has the purpose of preventing that the workflow execution returns to an activity that has been executed previous to this execution state. Hence, we need to introduce additional mechanisms to allow a replicated workflow execution to return to any activity of the workflow through compensation.

For enabling support for user initiated compensations, we introduce a *compensation mode* for our majority-based replication schemes. Basically, the compensation mode is working like a reverse execution of the workflow. Previously, we only used compensation to (semantically) reverse the effects that an invalid activity execution had on the external state. Now, the workflow execution shall

return to an activity that already has been executed – or even to the beginning of the workflow where no activity has been executed.

Remember that a compensation (semantically) reverses the effects that an activity execution had on the external state and rolls back the internal state of the workflow to the state that the internal state had before the activity execution was started (cf. Chapter 3). So far, we did not explicitly mention the rollback of the internal state in our replication schemes because any compensated activity was an invalid activity. Hence, there was a valid activity execution of that activity for which the respective primary sent an UPDATE message containing a new internal state. As a consequence, the rolled back internal state produced by the compensation was anyway overwritten by the state contained in the UPDATE message.

Now, we compensate a part of the workflow and strive to continue the execution from this point to which we returned. Hence, it is important that the rolled back internal state is available for continuing the workflow execution. Moreover, we have to provide this rolled back internal state to the other replicas – like an execution state update.

The user initiates the switch to the compensation mode by sending a COMPENSATE message to all replicas, where the message includes the activity to which the workflow execution shall return. When the replicas receive a COMPENSATE message, all the ongoing workflow execution(s) must be stopped before entering the compensation mode.

For being able to return to a specific activity, the compensation records that are written to the compensation log have to include the activity that was executed. Now, a compensation record $(eID, sID, a, comp(a)) \in cLog$ specifies the execution identifier of the replicated execution, the state identifier of the execution state that is produced by the activity execution, the executed activity, and the compensation handler of the executed activity. Basically, we have to compensate activities in the reverse order of execution until we have compensated the activity to which the execution shall return.

In specific, upon receiving a COMPENSATE message (cf. Algorithm 6), a primary is prevented from starting activity executions. Then, the primary switches to the compensation mode by sending an UPDATE message to all backups, where the UPDATE contains an execution state that indicates the

Algorithm 6: Compensation in the basic and relaxed majority-based replication scheme on replica r (Part I)

```
1 upon Receive COMPENSATE(eID, a) do
2     if r = Primary(v_r) then
3         stop execution;
          // Create compensation descriptor
4         eS_r.cD.cTarget := a;
5         eS_r.cD.cNextsID = eS_r.sID;
6         eS_r.sID.s := eS_r.sID.s + 1;
7         send UPDATE(eID, v_r, eS_r) to all backups;
8         WAIT UNTIL(received ACK(eID, eS_r.sID) from f replicas);
9         start Compensation(eID);
```

switch to the compensation mode. In specific, the execution state stores an additional field called the *compensation descriptor* (or *cD* for short). During normal operation, the field is empty indicating the execution mode. When the compensation descriptor contains data, it indicates that the workflow execution has been switched to the compensation mode. In other words, when a primary's compensation descriptor contains data, it will not start any activity executions. In specific, the compensation descriptor comprises the following data:

- $cTarget$: target activity of the compensation, i.e., the activity to which the workflow execution will return.
- $cNextsID$: state descriptor of the compensation record that shall be compensated next.

Now that the compensation records include the executed activity, why do we identify the next activity to compensate based on the state identifier of the produced execution state rather than the activity itself? The answer is that an activity might be executed multiple times by different replicas. Consider our example in Figure 5.9, where the activity a_2 has been executed by replica $r1$ and replica $r2$. Then, the execution switches to the compensation mode, where the target activity of the compensation is activity a_1, i.e., $cTarget = a_1$. In this example, this means that the whole workflow execution shall be compensated.

If the compensation descriptor specifies the next compensation based on the activity, this will possibly start compensations on multiple primaries. However,

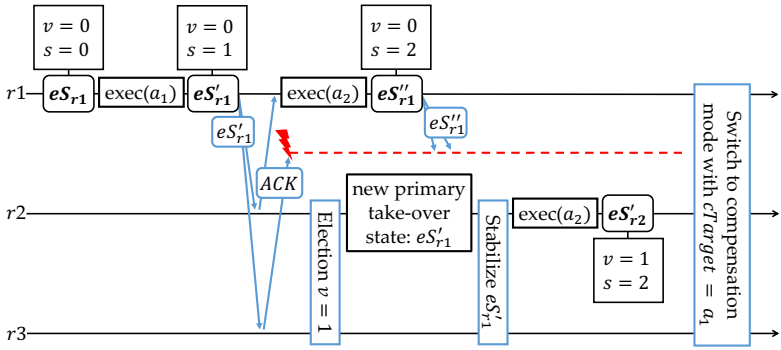


Figure 5.9.: Depicting an election caused by network partitioning, where all execution states are annotated with their state identifiers (i.e., their view and state numbers). Since there are two executions of activity a_2, the compensation descriptor has to use the state identifier to uniquely identify the next activity to compensate.

the rollback must, of course, start from the most recent execution state. To ensure this, only the current primary shall start the compensation. In our example of Figure 5.9, we only want the current primary $r2$ to start its compensation. When the primary $r2$ has finished its compensation, it will notify the previous primary $r1$ that it can start its compensations. Thereby, it is ensured that the activities are compensated in the reverse order of their execution, i.e., that $r1$ will only start to compensate its execution of a_1 after $r2$ has finished its compensations.[1]

Hence, we use the unique state identifiers to identify the next compensation. Directly after switching to the compensation mode in our example of Figure 5.9, the next compensation record that shall be performed is $cNextsID = (v = 1, s = 2)$. The primary uses a synchronous update to replicate the execution state that includes the compensation descriptor, which indicates the switch to the compensation mode. The synchronous update mechanism ensures that only the current primary will receive an acknowledgement from a majority of replicas. Moreover, this ensures that the intention to compensate, saved in that execution state, is stabilized.

[1]Note, however, that based on the stable-states vector, replica $r1$ might still already compensate the invalid activity a_2.

Algorithm 7: Compensation in the basic and relaxed majority-based replication scheme on replica r (Part II)

```
1  upon Compensation(eID) do
2      if eS_r.cD.cNextsID.s = ∞ then
3          eS_r.cD.cNextsID.s := (max(sID.s) | ∀(eID, sID, a, comp(a)) ∈ cLog,
                                   where sID.v = eS_r.cD.cNextsID.v);
4      while eS_r.next ≠ eS_r.cD.cTarget
           AND (eID, sID, a, comp(a)) ∈ cLog, where sID = eS_r.cD.cNextsID do
           // compensation:  execute compensation handler...
5          execute comp(a);
           // ..  and rollback internal state
6          eS_r.σ := rollback(eS_r.σ);
7          write (comp, eID, sID) to cLog;
8          eS_r.sID.s := eS_r.sID.s + 1;
9          eS_r.next := a;
10         eS_r.cD.cNextsID.s := eS_r.cD.cNextsID.s − 1;
11         send ASYNC_UPDATE(eID, v_r, eS) to all replicas;
12     if eS_r.next ≠ eS_r.cD.cTarget then // switch to previous view
13         eS_r.sID.s := eS_r.sID.s + 1;
14         eS_r.cD.cNextsID.s := ∞;
15         eS_r.cD.cNextsID.v := eS_r.cD.sID.v − 1;
16         send COMP_UPDATE(eID, v_r, eS_r) to Primary(eS_r.cD.cNextsID.v);
17         resend COMP_UPDATE until Primary(eS_r.cD.sID.v) acknowledged
             reception;
18     else // Compensation finished, notify current primary
19         send COMP_UPDATE(eID, v_r.eS_r) to Primary(v_r);
20 upon Received COMP_UPDATE(eID, v_x, eS_x) from x do
21     if v_x ≥ v_r then // *
22         v_r := v_x;
23         eS_r := eS_x;
24         send COMP_UPDATE_ACK(eID, eS_x.sID) to x;
25         if eS_r.next ≠ eS_r.cD.cTarget then
26             scheduleCompensations(eS_r.sSVec);
27             Start Compensation(eID);
28         else // Compensation finished, switch to execution mode
29             eS_r.sID.s := eS_r.sID.s + 1;
30             eS_r.cD := null;
31             send UPDATE(eID, v_r, eS_r) to all backups;
32             WAIT UNTIL(received ACK(eID, eS_r.sID) from f replicas);
33             return to normal operation;
   // * = checking the validity of a message
```

Afterwards, the primary starts the compensation (cf. Algorithm 7 line 1-19). The primary has to check whether it has executed the next activity to compensate, i.e., whether its compensation log includes a compensation record for the state identifier $cD.cNextsID$. If the primary's compensation log contains such a record, it compensates the activity as follows (cf. Algorithm 7 line 4-10):

1. Like during the execution, the primary increments the execution state's state number indicating that the execution progressed.

2. The primary executes the compensation handler and rolls back the internal state.

3. The primary updates the execution state. In specific, the activity that has been compensated has to be executed again when continuing the workflow execution. Thus, the next activity to execute in the execution state is set to the activity that has been compensated. Accordingly, the next activity to compensate, which is indicated by the state identifier $cNextsID$ is updated.

4. The primary sends an ASYNC_UPDATE message, containing the new execution state, to all other replicas.

In case that the log does not contain the compensation record of the activity to compensate next (cf. Algorithm 7 line 12-17), the primary of the previous view has to take over. Consequently, we update the compensation descriptor of the execution state accordingly and send a COMP_UPDATE message, which contains the new execution state, to the previous primary. Consider that the replica $r2$ in Figure 5.9 has finished the compensation of activity a_2 which had the state identifier $(v = 1, s = 2)$. Then, the next activity to compensate would be $(v = 1, s = 1)$. However, since r_2 did not execute any activity prior to a_2, it has no such compensation record in the log. At this point, the previous primary, i.e., the primary of view 0 has to take over. Hence, $r2$ sets the next activity to compensate to be $cNextsID = (v = 0, s = \infty)$. Here, we set the state number to infinity because the old primary might have multiple invalid activity execution that have higher state numbers than $s = 1$ that still need to be compensated.

When the previous primary receives the COMP_UPDATE message (cf. Algorithm 7 line 20-32), it starts to compensate its activity executions starting from

the activity that produced the execution state with the highest state number (cf. Algorithm 7 line 1-3). As described above, this is ensured through setting $s = \infty$ when handing over the compensation to the previous primary.

The compensation is finished when the activity to which the execution shall return and the next activity to execute match, i.e., $eS.cD.cTarget = eS.next$. In specific, a replica that reaches this state informs the current primary through a COMP_UPDATE message (cf. Algorithm 7 line 18-19).

When the current primary receives this COMP_UPDATE, it initiates the switch back to the *execution mode* (cf. Algorithm 7 line 28-33). The primary empties the compensation descriptor and then sends the new execution state to all backups using the synchronous update mechanism. As soon as a majority has acknowledged the reception of the execution state, the current primary switches to the execution mode continuing the workflow execution.

Note that the election and recovery work as described previously because the execution state's state number is incremented as during normal execution. Hence, a newly elected primary will get to know of the ongoing compensation through the compensation descriptor in the execution state. Even if after a failure the new primary might not use the newest execution state, each replica is responsible to compensate its own activity execution and will simply skip the compensations that it already performed based on the log entries.

However, since every replica is responsible for compensating the activities that it has executed, the compensation mode requires each primary that has executed activities that need to be compensated to be available. Thus, the compensation mode provides no availability advantage over a non-replicated execution. When the respective primary is failed, the execution (or compensation, rather) is blocked. We, however, assume that the necessity for using user initiated compensations should be rare – especially for highly automated processes where availability is most crucial. Hence, we provide the means for using user initiated compensations with our replication scheme for cases where it is required, even though the compensation mode provides no availability advantage over a non-replicated execution.

5.2. Flexible Failover Replication

In the following, we present a workflow replication scheme that increases availability further but, in turn, causes higher compensation cost than the majority-based replication schemes. For understanding the idea of this new replication scheme, we will lay out the limitations of the majority-based replication schemes presented above.

Assume that a majority of replicas fails. Then, the workflow execution cannot continue when performing a synchronous update. However, when striving purely for availability, this can be cured by encapsulating all activities in one passively replicated synchronization group requiring no synchronous updates at all. However, what severely limits the availability is the fact that the majority-based replication schemes are unable to elect a new primary when a majority of replicas failed. In Figure 5.10, we evaluate a replicated workflow execution with 5 replicas, where we partitioning the network and additionally inject one crash failure such that none of the partitions contain a majority. The failure lasts for 30 s. We measured the downtime, i.e., the time the workflow execution does not make progress, and the compensation cost, where 100 % refers to compensating the complete workflow once. We observe that the majority-based replication, where all activities are contained in a single passively replicated synchronization group, experiences an outage for the complete failure time because it cannot elect a new primary that continues the workflow execution.

The active replication scheme overcomes this problem by executing the workflow on all replicas independently. Note that this active replication is different to our relaxed majority-based replication scheme, where we only use active replication for synchronization groups that imply no compensation cost. Since our goal is to increase availability further by allowing a higher compensation cost, we will here consider a pure active replication scheme meaning that the complete workflow is actively executed by all replicas. Upon finishing the execution, the replicas agree on one of the workflow executions via majority consensus and compensate all other executions (cf. Figure 5.7). Consequently, all but one replica might fail during the execution without impacting availability. Only the consensus during the termination requires a majority to be available.

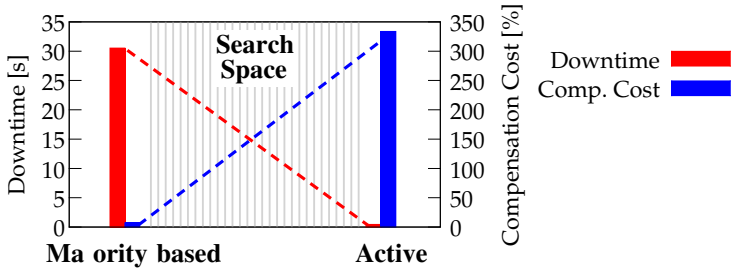


Figure 5.10.: Comparison of our majority-based and an active replication scheme with 5 replicas in the failure scenario of having no majority within a partition. [Lower is better]

The problem of active replication is that it incurs high compensation cost because all but one of the executions have to be compensated. In our scenario of Figure 5.10, the failures do not cause downtime when using active replication and, hence, the execution always stays available. However, the compensation cost is above 300 %. In a failure free execution, active replication with 5 replicas would even cause 400 % of compensation cost. Here, it is lower because one replica experiences a crash failure and, thus, cannot continue the workflow execution.

In conclusion, Figure 5.10 shows that majority-based and active replication constitute two opposing extremes leaving a huge search space in between. In the following, we address this search space by proposing a new replication scheme for obtaining the best of both worlds.

5.2.1. Flexible Failover Replication Scheme

Without any failures, one primary executes the workflow exactly as when using our majority-based replication scheme. However, in the case of a failure, we allow the replicas to elect a new primary *without* requiring a majority. In other words, each partition elects a primary. Assume, for example, that all replicas are partitioned from each other. Then, all replicas will elect themselves as primary and execute the workflow independently – similar to active replication.

Of course, this increases the compensation cost. For controlling the compensation cost, we introduce the *vote threshold* t_v. A replica needs t_v votes for being elected as primary, where $1 \leq t_v \leq \lfloor \frac{|R|}{2} \rfloor + 1$. We call this replication mechanism *flexible failover replication* because the vote threshold determines the failover behavior. A workflow designer, as the domain expert, can decide the vote threshold depending on the desired availability. When setting t_v to 1, each replica can vote for itself and become primary. Here, the availability of the workflow execution is ensured even if all but one replica fail. However, if all replicas are partitioned from each other, setting $t_v = 1$ will incur very high compensation cost. Setting $t_v = \lfloor \frac{|R|}{2} \rfloor + 1$ leads to majority-based replication, where all activities of the workflow are contained in a single synchronization group. Hence, there will be fewer primaries in case of network partitioning but – in turn – the availability is lower since it is impossible to elect a new primary when there is no network partition containing a majority of replicas. Setting the vote threshold between 1 and $f + 1$ realizes a tradeoff.

Even though allowing primaries to be elected without majorities seems straightforward, it poses multiple problems with respect to our existing majority-based replication schemes. First of all, every partition should only elect a single primary even when $t_v = 1$. Otherwise, we again would induce unnecessary compensation cost. On the other hand, when a replica is in a partition on its own, it should quickly become primary.

Secondly, we might have multiple competing primaries when the network partitions. Here, it is not predetermined which of these primaries will send its result back to the workflow client application. In terms of performance, the primary that finishes first should send back its result, while the other primaries should compensate their execution. In comparison, our majority-based replication scheme always has only one valid primary and only this primary can send back its result to the workflow client application. The valid primary is decided by the view number, where the view number is incremented during each majority election and then replicated on a majority of replicas again. Hence, out of a majority, at least one replica knows the highest view number and, thus, the currently valid primary (because there is exists a function mapping the view number to the primary replica of this view). However, with the flexible failover replication scheme, we do not require majorities during elections. Hence, we also

cannot use the view mechanisms because it is based on this majority replication. As a consequence, the stable-states vector cannot be used anymore because it is based on the principle that there is only one valid primary that can make progress on the workflow execution.

Finally, when two partitions reconnect, how do we detect and resolve conflicts without stopping or delaying the workflow execution? Any delay would forfeit availability, where availability is our main goal.

In conclusion, our existing majority-based replication schemes need significant changes for enabling flexible failover replication. In order to ensure understandability with all the required changes, we will describe the adapted scheme as a whole. In specific, we will first give an overview about the scheme and the new mechanisms needed for enabling flexible failover replication before defining the replication scheme in detail.

5.2.1.1. Overview

A replicated workflow execution is started when the workflow client application sends an execution request to all replicas. However, since there is no view number or function to determine the primary from the view number, we need a new mechanism for deciding on a primary: we always elect the replica with the highest identifier as primary. Initially, we can save the overhead of an election. The replica with the highest identifier is automatically the initial primary. The primary executes the activities of the workflow and sends asynchronous updates to the backups after each activity execution.

In case that the primary crashes or the network is partitioned, the backups elect a new primary. For the being elected as primary, a replica only has to receive votes from t_v replicas (including itself). The new primary collects the state from all voters to determine one of the states as the take-over state.

In case of network partitioning, multiple partitions might elect a new primary – depending on t_v. With $t_v = 1$, for example, each partition will elect a primary. The primaries of the different partitions are not aware of each other. Upon re-connecting, the primaries will get to know of the competing executions through the reception of updates from the other primaries. Obviously, all but one execution have eventually to be compensated for ensuring that the overall

workflow execution is Single-Execution-Equivalent (cf. Chapter 4). For example in Figure 5.11, activity a_1 was executed both by $r1$ and $r2$. Thus, either $r1$ or $r2$ has to compensate its execution of a_1 to resolve the conflict.

When the primaries detect a conflict, all but one execution need to be stopped to minimize the compensation cost. For achieving the best performance in terms of execution time, we stop the executions that lag behind while letting the most progressed execution continue. For ensuring that exactly one execution continues we would need to perform an election, where each election, of course, incurs synchronization overhead. Hence, we opted for a more lightweight solution, where the conflict resolution relies on the already integrated asynchronous update mechanism. When a primary receives an update from a further progressed execution (which the primary can decide from the received update message), it directly stops its execution. For example, in Figure 5.11, $r2$ becomes primary after partitioning from $r1$. After reconnecting, $r2$ stops its execution because it receives an update from $r1$, which is already progressed further.

Since we avoid to perform an election, we trade the guarantee given by the election protocol – that there will be at most one primary after the election – against the saved election overhead. In specific, when there are two primaries that execute a workflow at the same speed and started at the same point in time, both might never receive an update from the other primary, which contains a further progressed execution state. Then, both will execute the complete workflow increasing the compensation cost. However, in practice it is unlikely that two (or more) primaries start an execution from the same state at the same point in time and execute the workflow at the same speed over a long period of time.

For completely resolving the conflict of Figure 5.11, replica $r2$ has to compensate its execution of a_1 after stopping its workflow execution. However, $r2$ does not know whether the execution state produced by a_1 was used by another primary as a take-over state – as is the case in Figure 5.11.

In general, a primary can only compensate an activity a_x after all activity executions that depend on a_x (i.e., that used the state produced by a_x as input) have been compensated (if any). This has not been a problem with our previous

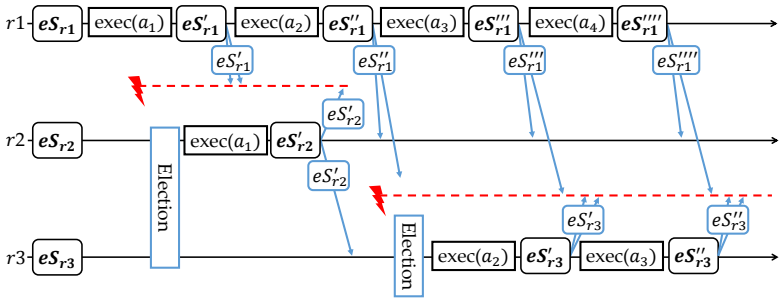


Figure 5.11.: Example of the flexible failover replication scheme with 3 replicas and $t_v = 1$

replication schemes because a majority was informed of a take-over state by a primary. Then, the old primary only produced invalid activity executions afterwards that were never used by another replica. Thus, a replica always could compensate its invalid activity executions in the reverse order of execution without interacting with the other replicas.

However, with flexible failover replication, there might be multiple primaries which execute the same activities concurrently. Which of the activity executions are compensated is decided at a later point in time. Moreover, any of the execution states produced by the activities might be used by other primaries as take-over state. Thus, even if an activity needs to be compensated, other replicas might have to compensate their causally dependent activity executions first. In the example depicted in Figure 5.11, replica $r3$ has to compensate a_2 and a_3 before the replica $r2$ can compensate a_1.

For tracking these causal dependencies, each replica keeps a history of its activity executions, where each history entry includes the information which execution state was used as input and which execution state was produced through the activity execution. This, however, requires each execution state to have a unique identifier, where we cannot use the previous execution state identifier based on views as we already discussed above.

Now, an execution state identifier consists of three parts: a replica identifier, a failover counter, and a state number. The replica identifier specifies which replica was the primary that produced the state. However, a replica might become primary multiple times during the execution. The failover counter indicates how many times a replica tried to become primary through an election, i.e., how many times the replica started a failover. Finally, the state number is incremented with every activity execution – indicating the progress of the workflow execution.

As stated above, a replica can only compensate an activity a_x if all activity executions that used the execution state produced by a_x were compensated. Actually, an execution that used a_x's execution state as a take-over state might be the execution on which the replicas agree in the end. Then, a_x is part of the decided workflow execution and will not be compensated. In our example of Figure 5.11, the replicas might eventually agree on the execution of replica $r3$ meaning that $r2$'s activity execution of a_1 must not be compensated. Instead, $r1$ would need to compensate its execution of a_1 – and, of course, all the following activity executions as well. As a consequence, we only compensate activities after the replicas have agreed on one workflow execution.

This agreement is part of the termination of the workflow execution. In specific, the termination consists of three phases. In the first phase, the replicas perform a majority consensus to agree on one finished workflow execution. The consensus requires a majority of replicas to be available. The replicas send the result of the decided execution to the workflow client application. In the second phase, all replicas learn about the decision, allowing the replicas to compensate the activity executions that are not part of the decided workflow execution. In the final phase, the replicas forget the workflow execution after all replicas have finished all necessary compensations.

During the workflow execution, we can ensure progress as long as t_v replicas are available while the agreement during the first termination phase requires a majority of replicas. The last two termination phases require all replicas to be available. However, this is not critical as the last two phases are completely decoupled from the workflow client application.

5.2.1.2. Data Structures

The flexible failover replication scheme has to maintain data in volatile memory as well as on stable storage for the workflow execution (cf. Table 5.2). In specific, the volatile memory of a replica r contains the following information:

- eID: unique identifier of the replicated execution.
- G: workflow model that is executed by eID.
- t_v: vote threshold required for becoming primary.
- $prim_r$: indicator on whether r is currently primary.
- eS_r: last execution state that r is aware of. In other words, if r is primary, this is the last produced execution state. If r is a backup, eS_r is the last execution state received from a primary.

The execution identifier eID, the workflow G, and the vote threshold t_v are same on all replicas that participate in the execution. All other data, i.e., the primary indicator and the execution state, might be different on each replica. In specific, the execution state consists of the following variables:

- $next$: next activity to execute.
- σ: internal state that is input for the next activity execution.
- sID: execution state's unique identifier.

The identifier of an execution state sID consists of the following variables:

- rID: identifier of the replica that produced the execution state.
- f: value that the failover counter had when the state was produced.
- s: state number that the producer assigned to the execution state.

Volatile Memory	
eID	workflow execution's identifier
G	workflow model that is executed
t_v	vote threshold for being elected as a primary
$prim$	boolean indicating whether the replica is primary
eS	execution state
$eS.next$	next activity to execute
$eS.\sigma$	internal state
$eS.sID$	identifier of the execution state
$eS.sID.rID$	ID of the replica that produced eS
$eS.sID.f$	value the failover counter had when eS was produced
$eS.sID.s$	state number that the producer assigned to eS
Stable Storage	
f	local failover counter
H	execution history

Table 5.2.: Data structures kept for each workflow that is executed using our flexible failover replication scheme

All the above described variables are saved in volatile memory, which is wiped through crash failures. The following variables are saved in stable storage to survive crash failures:

- f_r: current failover counter of r. Each time r starts a failover, the counter is incremented. Initially, f_r is 0.
- H_r: execution history of r, where each activity execution is logged.

The failover counter f_r is saved on stable storage to prevent crash failures from resetting f_r. The execution history H_r is stored on stable storage for ensuring that a replica never forgets an activity execution (which it might need to compensate later). In specific, a replica r writes an *execution record* for every executed activity a_x to H_r. The execution record $(eID, eS_{input}.sID, eS_{prod}.sID, comp(a_x))$ contains the execution identifier eID, the identifier of the execution state eS_{input} that is used as input for the activity execution, the identifier of the produced execution state eS_{prod}, and the compensation handler $comp(a_x)$ of the executed activity a_x. For triggering the execution of a compensation handler, a replica

sends the execution record to the Compensation Unit. The Compensation Unit filters duplicate compensation requests based on the $eS_{prod}.ID$.

5.2.1.3. Normal Operation

The workflow client application sends an execution request message EXEC(eID, mID) to all replicas, where the message contains the unique execution identifier and the identifier of the workflow model that shall be executed. Upon receiving the message, a replica stores the execution identifier and loads the workflow model from the workflow repository. If the replica is the initial primary, i.e., the replica with the highest ID, it instantiates the loaded workflow model and sets its primary indicator $prim_r$ to $true$.

A primary executes the workflow as follows (cf. Algorithm 8 line 1-10):

1. The primary writes an execution record for the activity that is going to be executed into the execution history.

2. The primary executes the activity and updates the variable that indicates the next activity to execute, i.e., $eS.next$.

3. The primary sends an update containing the produced execution state to all replicas. The primary sends the update asynchronously, i.e., the primary continues the workflow execution while sending the update.

Upon receiving an update from the primary, a backup applies the included execution state if the state number of the received state is higher than the one that the backup currently stores (cf. Algorithm 8 line 11-14).

5.2.1.4. Failover

All backups monitor the primary by means of a heartbeat mechanism. If the primary becomes unavailable through crashing or partitioning, the backups elect a new primary. Any partition with at least t_v replicas elects the replica with the highest ID in that partition as primary.

Upon detecting a primary failure (cf. Algorithm 9 line 1-3), a backup increments its failover counter indicating it now starts a failover. Then, it requests VOTE messages from all replicas. All replicas with lower identifiers reply with a

Algorithm 8: Flexible failover replication scheme on replica r (Part I)

```
   // normal operation
1  while eS_r.next ≠ null do
2      if prim_r = true then
           // Save input execution state's ID
3          eSInputID_r := eS.sID;
           // Produce next execution state
4          eS_r.sID.rID := r;
5          eS_r.sID.f := f_r;
6          eS_r.sID.s := eS_r.sID.s + 1;
7          write (eID, eSInputID_r, eS_r.sID, comp(eS_r.next)) to H_r;
8          eS_r.σ := execute(eS_r.next, eS_r.σ);
9          next := succ(G, eS.next);
10         send ASYNC_UPDATE(eID, eS_r);
11 upon receive ASYNC_UPDATE(eID, eS_x) do
12     if eS_x.s > eS_r.s then
13         prim_r := false;
14         eS_r := eS_x;
```

VOTE that includes the execution state of the voter (cf. Algorithm 9 line 4-12). Any replica with a higher identifier sends a REJECT message.

A replica requires t_v votes for becoming primary. If a replica receives a single REJECT message, it will not become primary since this means there is a replica with a higher identifier in the same partition. It, however, might be the case that a replica receives enough VOTE messages before receiving a REJECT. For handling this case, every replica that requested VOTE messages waits for the time threshold $t_t \in \mathbb{N}$, specifying a threshold in ms, even if the replica already received enough VOTE messages.

If a replica has received t_v VOTE messages and no REJECT after waiting for t_t, it becomes primary (cf. Algorithm 9 line 13-17). The replica uses the state with the highest state number from the received VOTE messages as the take-over state. Now, the new primary returns to normal operation.

When there are multiple partitions each hosting a primary, the primaries will receive UPDATE messages from each other after the partitions reconnect. When a primary receives an UPDATE that contains an execution state with a higher state number than its own state, the primary stops its execution (cf. Algorithm 8 line 13). Thereby, competing executions are stopped without introducing addi-

Algorithm 9: Flexible failover replication scheme on replica r (Part II)

```
   // failover
1  upon detected primary failure do
2      f_r := f_r + 1;
3      send REQ_VOTE(eID, r, f_r) to all replicas;
4  upon receive REQ_VOTE(eID, x, f_x) from x do
5      if prim_r = true then // r is primary
6          send REJECT(eID, x, f_x) to x;
7      else if x < r then// r has a higher ID
8          if r is not waiting for votes then
9              trigger event detected primary failure;
10         send REJECT(eID, x, f_x) to x;
11     else
12         send VOTE(eID, x, f_x, eS_r) to x;
13 upon receive VOTE(eID, x, f_x, eS_x) from x do
14     if received VOTE from ≥ t_v replicas for (x, f_x)
15      AND waited ≥ t_t then
16         eS_r := execStateWithHighestStateNumber(VOTEs);
17         prim_r := true;
```

tional coordination overhead. As already described above, there remains a chance that there are two or more competing executions that start from the same state at the same point in time and execute the workflow at the same speed. Hence, the primaries never receive an update, which indicates a further progressed execution, from another primary. In the worst case, the competing executions will execute the complete workflow. However, in practice, this scenario is rather unlikely.

5.2.1.5. Termination

When a primary has executed the last activity of the workflow, it initiates the termination (cf. Algorithm 10 line 1-2). The termination consists of three phases. In the first phase, the replicas agree on one workflow execution via majority consensus. This also decides the result of the workflow execution, which is then sent to the workflow client application. In the second phase, all replicas learn of the decided workflow execution and compensate the activities that do not belong to this workflow execution. In the final phase, the replicas forget the workflow execution after all replicas finished the necessary compensations.

Algorithm 10: Flexible failover replication scheme on replica r (Part III)

```
   // termination
1  upon eS_r.next = null do
2      start Agreement;
3  upon Successful Agreement(eS_x) do
4      forall (eID, eSInputID, eSProdID, comp(a)) ∈ H_r do
5          send COMP_REQ(eSProdID) to all replicas;
6  upon Receive COMP_REQ(sID) from x do
7      if (eID, eSInputID, eSProdID, comp(a)) ∈ H_r, where eSInputID = sID then
8          if (comp, eID, eSProdID) ∈ H_r then
9              send COMP(eID, sID);
10         else if (keep, eID, eSProdID) ∈ H_r then
11             send KEEP(eID, sID);
12         else
13             do nothing; // not yet decidable
14     else
15         send COMP(eID, sID) to x;
16 upon Receive COMP(sID) do
17     if received COMP for sID from all replicas then
18         send (eID, eSInputID_r, sID, comp(a)) to Compensation Unit;
19         write (comp, eID, sID) to H_r;
20 upon Receive KEEP(sID) do
21     write (keep, eID, sID) to H_r;
```

The first phase can be realized by any consensus protocol, such as the Paxos protocol [Lam98]. Using Paxos, a majority of replicas elect a leader which proposes an execution state of a finished workflow execution as the final execution state. If a majority of replicas accept the proposal, it is guaranteed that all replicas will eventually accept the proposal. Thus, a majority can decide on a final execution state and, thereby, on the respective workflow execution which produced this state. Moreover, the final execution state also determines the result that is then sent to the workflow client application.

In the second phase, the proposer repeatedly sends the decided final execution state to all replicas until all replicas have acknowledged the reception. Now, the replicas need to identify which of their activity executions belong to the decided workflow execution and compensate all others. For each executed activity, a replica sends a COMP_REQ message to all replicas (cf. Algorithm 10 line 3-5),

where the message contains the identifier of the execution state produced through the respective activity execution.

Upon receiving a COMP_REQ message (cf. Algorithm 10 line 6-13), a replica, say r, uses the included execution state identifier for checking whether r used the state as input for any of its activity executions. If not, then r can allow the compensation by replying with a COMP message. However, if r used the execution state as input, it can only allow the compensation after compensating the activity execution for which the state was used. Moreover, in case this activity execution belongs to the decided execution, it has to be kept. In this case, r replies with a KEEP message.

A replica can only compensate the activity after all replicas replied with a COMP message (cf. Algorithm 10 line 16-19). For triggering the compensation, a replica sends the execution record of the activity execution to the Compensation Unit. Afterwards, the replicas writes a compensation record $(comp, eS.sID)$ to the execution history. If one replica replies with a KEEP message (cf. Algorithm 10 line 20-21), the activity execution belongs to the decided workflow execution and the activity is not compensated. In this case, the replica writes a keep record $(keep, eS.sID)$.

The replica regularly repeats the COMP_REQ messages until it has written a keep or compensation record for each activity that it has executed. After all records have been written, the second phase is finished. Like in the majority-based replication scheme the workflow execution has now completed. However, the replica has to keep the execution's data in volatile memory because the other replicas still might be in the second phase.

For forgetting the replicated workflow execution, we use the mechanism from the the majority-based replication scheme, i.e., the two phase-commit [LS79, ML83]. The action carried out by two phase-commit is the forgetting of the execution. When all replicas agree to be ready to forget, i.e., when the replicas have finished the second phase, the forgetting is committed. Upon receiving such a commit, the replicas write an $(end, eS.sID)$ record to the execution history and remove all data from volatile memory. For saving space on stable storage any workflow execution with a begin and end record may be pruned from the execution history.

Algorithm 11: Flexible failover replication scheme on replica r (Part IV)

```
   // recovery
1  upon Recover in ACTIVE state do
2      send RECOV_REQ(eID) to all replicas;
3  upon Receive RECOV_REQ(eID) from x do
4      if Terminated and already reached agreement then
5          send DECISION(eID, eS_r) to x;
6      else
7          send RECOV_RPLY(eID, mID, t_v, eS_r) to x;
```

5.2.1.6. Recovery

Through crash failures, a replica loses the data kept in its volatile memory. Upon recovery, a replica reads its execution history. Here, we again differentiate between the UNKNOWN and the ACTIVE state (cf. Section 5.1.1.5).

Obviously, a replica that recovers in the UNKNOWN state cannot initiate any recovery steps for *eID*. When recovering in the ACTIVE state (cf. Algorithm 11 line 1-2), the replica has to retrieve values for the variables in its volatile memory before it can participate in the replicated execution again. Thus, it requests the data from all replicas by sending a RECOV_REQ message.

Upon receiving a RECOV_REQ (cf. Algorithm 11 line 3-7), a replica reacts differently depending on whether or not the workflow execution is already terminating. In specific, if the execution has already finished the first phase of termination, the replica sends a DECISION message to the recovering replica, where the DECISION message contains the decided execution state. Otherwise, the replica sends a RECOV_RPLY, which contains the workflow model identifier, the vote threshold, and execution state that the replica currently stores.

When the recovering replica receives an RECOV_RPLY, it loads the workflow model of the received workflow model identifier from the workflow repository. Additionally, it saves the received vote threshold and executions state. Afterwards, the replica returns to normal operation acting as a backup. In case that the recovering replica receives a DECISION message, the replica also saves the included execution state and returns to normal operation, where it directly starts the second phase of termination.

5.2.2. User Initiated Compensations

As described in Section 5.1.3, a user might initiate the compensation of a part or the complete workflow during the execution. So far, the flexible failover replication scheme does not support compensation during the execution. Compensation is only supported as part of the termination after having agreed on the result of one execution that is sent back to the workflow client application. In the following, we will present mechanisms for supporting user initiated compensations with our flexible failover replication scheme. Similar to the the majority-based replication scheme, we introduce a compensation mode. The compensation mode of flexible failover replication differs from the one of majority-based replication since flexible failover replication does not rely on majorities (except from the consensus during termination).

The logging is extended to include the executed activity. In specific, each execution record $(eID, eSInputID, eSProdID, a, comp(a))$ in the execution history H_r includes the execution identifier of the replicated execution, the identifier of the execution state that was input to the activity execution, the identifier of the execution state that was produced by the activity execution, the executed activity, and the compensation handler of that activity.

Before switching to the compensation mode, every primary needs to stop its execution. With our flexible failover replication, any replica might possibly be a primary when the vote threshold t_v is set to 1. Thus, each replica has to acknowledge that it stopped its execution (if any) before staring the compensation.

Remember that compensation not only strives to semantically reverse the changes of the activity on the external state but also rolls back the internal state. Thus, we also need to roll back to the execution state identifier that was associated with the internal state. Unfortunately, that means that the state number in the identifier will get smaller through the compensations – not reflecting the progress.

Thus, we need an explicit progress indicator for the compensation mode, called the *epoch counter* (or e for short), which the replica stores in volatile memory. Moreover, each execution state identifier stores in which epoch it was executed. Now, the epoch counter serves together with the state number to identify the execution state's progress, where a state sID is more recent than a state sID' if $(sID.e > sID'.e)$ or $((sID.e = sID'.e) \wedge (sID.s > sID'.s))$. Note, however, that

Algorithm 12: Compensation in the flexible failover replication scheme on replica r (Part I)

```
1  upon Receive COMPENSATE(eID, a) do
2      prim_r = false; // stops the execution
       // Create compensation descriptor
3      e_r := e_r + 1;
4      eS_r.cTarget := a;
5      send COMP_MODE(eID, e_r, eS_r.cTarget) to all backups;
6      WAIT UNTIL(received COMP_MODE_ACK(eID, e_r) from all replicas);
7      send START_COMP(eID, e_r) to all replicas;
8      start Compensation(eID);
9  upon Receive COMP_MODE(eID, e_x, cTarget_x) from x do
10     if e_x ≥ e_r then // *
11         prim_r = false; // stops the execution
12         e_r := e_x;
13         eS_r.cTarget := cTarget_x;
14         reply with COMP_MODE_ACK(eID, e_x) to x;
15 upon Receive START_COMP(eID, e_x) from x do
16     if e_x = e_r then // *
17         Start Compensation(eID);
```

the other fields of the state identifier (i.e., the producer *rID* and the failover counter f) are needed as well for uniquely identifying the state even though these fields do not indicate progress.

Similar to the concept of views, the replicas will not accept messages sent from older epochs. Thus, during an election a new primary will use the most recent state as just defined. Moreover, a backup only accepts updates from the current or a newer epoch.

Upon receiving a COMPENSATE message from the user (cf. Algorithm 12 line 1-8), a replica stops any ongoing execution and saves the activity to which the execution shall return (or *cTarget* for short). However, in contrast to the majority-based replication scheme, the *cTarget* is directly stored in the execution state and not part of a compensation descriptor.[2] Afterwards, the replica increases the epoch counter and broadcasts the new epoch and the compensation target with a COMP_MODE message to all other replicas.

[2] We do not need an indicator for the activity to be compensated next because the replicas decide when an activity can be compensated based on coordination and their execution histories.

Algorithm 13: Compensation in the flexible failover replication scheme on replica r (Part II)

```
1  upon Compensation(eID) do
2      while H_r contains uncompensated activities a ≥ eS_r.cTarget do
3          (eID, eSInputID, eSProdID, a, comp(a)) := nextToComp(eS_r.cTarget);
4          send COMP_REQ(eID, eSProdID) to all replicas;
5          WAIT UNTIL received COMP for eSProdID from all replicas;
           // compensation:  reverse changes on external state and...
6          execute comp(a);
           // ...roll back internal state if this state is available
7          if eS_r.sID = eSProdID then
8              eS_r.σ := rollback(eS_r.σ);
               // Revert sID to match rolled back state
9              eS_r.next := a;
10             eS_r.sID := eSInputID;
11         write (comp, eID, sID) to H_r;
12         if eSInputID.prod ≠ r then // input state produced by other
            replica
13             WAIT UNTIL COMP_REQ(eID, eSInputID) is received;
14     Start LeaveCompMode(eID);
15 Function nextToComp(a)
16     return (eID, eSInputID, eSProdID, a, comp(a)) ∈ H_r,
                where max(esProdID.e)|max(esProdID.f)|max(esProdID.s)
                    AND (comp, eID, eSProdID) ∉ H_r;
   // * = checking the validity of a message
```

Upon receiving a COMP_MODE message (cf. Algorithm 12 line 9-14), the replica checks the validity of the message by checking whether the received epoch is greater or equal to the locally stored epoch. If the message is valid, the replica stores the received epoch and the compensation target. Afterwards, it acknowledges the reception of the message.

After all replicas have stored the compensation target and switched to the new epoch, it is ensured that all primaries have stopped their execution. Thus, the compensation may now start (cf. Algorithm 12 line 6-8 & line 15-17).

Now, the replica may start its compensations (if any). Here, all the activities have to be compensated in the reverse order of execution (cf. Algorithm 13 line 15-16). However, any execution state produced by a primary might be used by another primary as take-over state. Hence, the replica has to request all other

replicas for whether it can compensate an activity execution similar to the second phase of termination (cf. Algorithm 13 line 3-5). Thus, the replica sends a COMP_REQ message, which includes the state identifier of the execution state produced by the activity execution that shall be compensated, to all replicas.

Upon receiving such an compensation request (cf. Algorithm 14 line 1-11), a replica validates whether it needs to compensate any of its own executions before allowing the compensation. If the replica used the state that shall be compensated as input for an execution, the replica waits until it has compensated the respective activity execution.

In contrast to the termination phase, we have to ensure that after the compensation a rolled back internal state is available from which we can continue the execution. Thus, the output internal state of an activity execution, which will be rolled back during the compensation, has to be available before starting the compensation. As a consequence, a replica provides the execution state to the compensation requestor if the state identifier for which the compensation permission is requested matches the replica's current execution state.[3]

After receiving the compensation allowance from all replicas, the compensation and rollback can be started (cf. Algorithm 13 line 5-13). Then, the compensation is written to the log.

Before starting the next compensation, the replica has to check whether another replica might need this state as input to start its compensation (cf. Algorithm 13 line 12-13). In specific, if another replica produced the execution state – which is determined from the logged execution state identifier – then this execution state has to be provided to the producer before the execution state can be overwritten through another compensation.

After a replica has compensated all activities that it executed following the activity to which the workflow execution shall return, the compensation has finished (cf. Algorithm 13 line 14). However, before switching back to the execution mode, the replicas have to receive an execution state update that includes the rolled back execution state. In specific, the replica that compensated the compensation target has produced the execution state that has to be replicated

[3] Of course, some of the execution state might not be available anymore since they were dismissed because there was a more progressed execution. However, this is not severe since it is ensured that there are some execution states from which the execution can be rolled back.

Algorithm 14: Compensation in the flexible failover replication scheme on replica r (Part III)

```
1  upon Receive COMP_REQ(eID, sID_x) from x do
2      if (eID, eSInputID, eSProdID, a, comp(a)) ∈ H_r, where eSInputID = sID_x then
3          if (comp, eID, eSProdID) ∈ H_r then
4              if sID_x = eS_r.sID then
5                  send COMP(eID, sID_x, eS_r);
6              else
7                  send COMP(eID, sID_x, null);
8          else
9              do nothing; // not compensated own execution yet
10     else
11         send COMP(eID, sID_x, null) to x;
12 upon Receive COMP(eID, sID_x, eS_x) from x do
13     if eS_x ≠ null AND (comp, eID, sID_x) ∉ H_r then
           // Received output internal state of activity that is
              compensated next
14         eS_r := eS_x;
```

on all replicas for continuing the workflow execution. There actually might be multiple replicas that executed the activity that is the compensation target. Any of the execution states might be used for continuing the execution. However, it needs to be ensured that after a failure a recovering replica will always choose one of these states to continue the execution from.

Thus, the execution state is made the most recent execution state by incrementing the epoch counter of the replica and setting the epoch counter in the state identifier to the new epoch counter value (cf. Algorithm 15 line 1-9). As this creates a new execution state, this has to be logged in the execution history to keep the execution history intact. Afterwards, the replica sends this new execution state to all replicas, which save the received state, and returns to normal operation for continuing the execution (cf. Algorithm 15 line 10-17).

In comparison to the compensation mode of majority-based replication, flexible failover replication requires all replicas to be available for every activity that needs to be compensated. Thus, the compensation mode of flexible failover replication actually has a lower availability than a non-replicated execution. Again, we assume the necessity of user initiated compensations to be rare – especially

Algorithm 15: Compensation in the flexible failover replication scheme on replica r (Part IV)

```
1  upon LeaveCompMode(eID) do
2      if eS_r.next = eS_r.cD.cTarget then
           // reverted to the execution state that has been input for
              the activity execution to which we reverted
3          e_r := e_r + 1;
           // log phantom execution
4          eSInputID := eS_r.sID;
5          eS_r.sID.e := e;
6          eS_r.sID.prod := r;
7          eS_r.sID.f := f;
8          eS_r.cTarget := null;
9          write (eID, eSInputID, eS_r.sID, null, null) to H_r;
10         send COMP_FINISHED(eID, eS_r) to all replicas;
11         return to normal operation as primary;
12 upon Receive COMP_FINISHED(eID, eS_x) from x do
13     if finished all compensations then
14         if eS_x.e ≥ e_r then // *
15             e_r := e_x;
16             eS_r := eS_x;
17         return to normal operation as backup;
   // * = checking the validity of a message
```

for highly automated processes, where even fractions of seconds count. However, there are cases in which it might be necessary to compensate a part of the workflow or even the complete workflow [SK11, SK12]. For workflows, which require a user to supervise the execution and frequently initiate compensations, majority-based replication is the better choice.

5.3. Extensions

The above described basic and relaxed majority-based replication schemes as well as the flexible failover replication scheme can only handle a sequence of compensable activities. In the following, we extend the schemes to support branching, activities which cannot be compensated, and more complex interaction patterns than request-reply such as choreographies.

5.3.1. Branching

In our workflow model, AND- and XOR-branches are created through having multiple outgoing edges from one activity with according transition conditions. In the following, we describe the mechanisms for supporting AND- and XOR-branching. This implies that we also support any branching that can be expressed as a combination of AND- and XOR-gateways.

AND-branching: Through AND-branching, the workflow can concurrently execute branches of the workflow. So far, our presented replication schemes maintain a varialbe that indicates the activity that shall be executed next. Obviously, when only maintaining a single activity which is up for execution, this works only for sequences. We overcome this limitation by executing each branch as it would be a separate workflow because then each branch again is simply an activity sequence. This is possible because all concurrent AND-branches use disjoint subsets of the internal state (cf. Chapter 3). Since this means that the branches do not share data, the independent execution of the branches is straightforward [KL12].

We realize the AND-branching support as follows: the primary creates a *branch execution state* for each branch, which only contains the respective subset of the internal state. Basically, the primary executes each branch as it would be an separate workflow using the respective branch execution state. For identifying the branch execution, we associate the *eID* with a branch identifier *bID*, i.e., (eID, bID) uniquely identifies one branch. Each branch sends independent UPDATES using the new identifier, where the UPDATE includes only the respective subset of the internal state.

For joining the branches back together, each branch execution has to be finished. Then, the primary joins the subsets of the internal state by copying the new variable values of the subsets into its regular execution state. Afterwards, the primary continues the workflow execution.

XOR-branching: Actually, XOR-branching is naturally supported by our schemes because the execution state includes a pointer to the activity that shall be executed next. So, after the activity execution is finished, the transition conditions of the outgoing edges are evaluated deciding the branch that is going

to be executed. Thus, the UPDATE sent after the activity execution indicates which branch was chosen.

5.3.2. Loops

For realizing loops within a workflow, it is common to define loop activities that contain as sequence of activities (and possibly also AND- and XOR-branches), which are repeated until the loop's condition is fulfilled.[4] Thus, the activities that are contained in the loop activity might be executed multiple times. With respect to our replication schemes, this means that some activities will be executed again but the state number of the produced execution states will continuously increase. Since the states are still uniquely identifiable, our schemes support loop activities without any extension.

Note, however, that usually the iterations of a loop activity are assigned a unique identifier [SK12] such that the user can refer to an activity execution of a specific iteration of the loop, e.g., for monitoring purposes. In principle, we can add such a unique identifier when logging the activity execution for allowing a user to invoke the compensation to a specific point in one iteration.

5.3.3. Non-compensable Activities

So far, we assumed that each write activity has a compensation handler that semantically reverses the changes that the activity performed on the external state. However, there also exist activities which cannot be compensated [LR00]. For example, some discounted airplane tickets do not allow to cancel the ticket after booking. A replica might only start this activity execution if it is ensured that no other replica restarts (or already started) the execution of that activity. Moreover, when the activity is contained in an XOR-branch, it must be ensured that the activity execution is only started if the replicas agree on that branch.

As a consequence, calling a service of a non-compensable activity is similar to informing the workflow client application about the result of the workflow execution. Once the message (i.e., the request message to the service or the result message to the workflow client application) is sent, the effects on the

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

external state are irreversible. Because of the similarities, we reuse the first phase of termination of the respective replication scheme for reaching consensus on one workflow execution. When having decided on the workflow execution, the primary of the decided workflow execution becomes the *exclusive primary* until the UPDATE of the non-compensable activity was distributed. While there exists an exclusive primary, no elections may start. Thus, if the primary or the called service fails, the workflow execution cannot progress, i.e., the execution has to wait for the recovery of the primary or service. For this reason, this approach of enabling the support of non-compensable activities is not optimal.

Similar to Behl et al. [BDH+12], we propose to increase the fault tolerance by introducing a unique interaction identifier (UIID) that identifies the interaction with that service. The service can filter duplicate calls based on the UIID, which allows a new primary to repeat the request to the service. As described above, the replicated execution still performs the consensus on one workflow execution.

The consensus – reused from the first phase of termination – requires a majority of replicas to be available both for the majority-based replication schemes as well as for the flexible failover replication scheme. As part of this consensus, the coordinator proposes one workflow execution to decide upon. Furthermore, this proposal includes the service to be called by the non-compensable activity as well as a UIID. Thus, once the replicas have decided on one workflow execution, they also have decided on the service to call as well as the UIID.

When calling that service, the primary includes the UIID in the request message. Now, assume that the primary fails and a new primary is elected. The majority-based replication schemes will automatically get to know of the UIID and called service since the election is based on a majority. The flexible failover replication scheme might elect a primary that does not know of the agreed workflow execution, called service, and UIID. However, this simply means it will start the consensus again through which the primary will learn of the decision.

Consequently, any newly elected primary will call the same service again using the same UIID. The service will filter the duplicate message based on the UIID and simply repeat the reply. Hence, the service, for which no compensation handler is available, will not be executed again. Of course, this requires each service of a non-compensable activity to implement such a filtering mechanism. Alternatively, a middleware for filtering might be inserted. Then, existing services

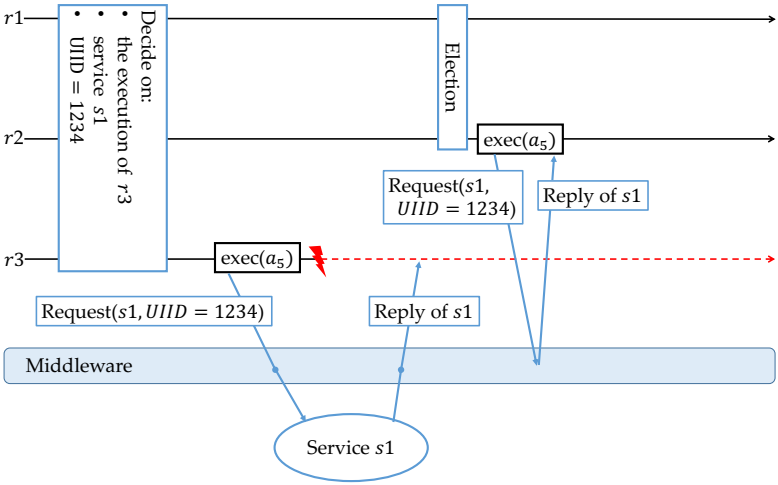


Figure 5.12.: Using a middleware and a UIID for filtering duplicated service calls of a non-compensable activity a_5.

without such a filtering mechanism may be used. We have depicted how a middleware can be used for filtering in Figure 5.12.

5.3.4. Complex Interactions and Choreographies

Until now, we assumed that each service call is realized by a request-response pattern, where the request and the response are sent and received by the same activity. If, however, the request-reply interaction spans multiple activities, the service interaction is called *asynchronous* [BCPR04]. In other words, with an asynchronous service call one activity, called *request activity*, sends the request to the service while another activity, called *reply activity*, is responsible for the reception of the reply. When a replica sends a request to a service, the service will send the reply to the requesting replica. In case of a primary failure, the new primary will not re-execute the request activity if the old primary replicated an execution state following the request activity. Figure 5.13 depicts such a scenario where the primary $r3$ fails after sending the request to the service. The new primary $r2$ continues the execution from an execution state following the request

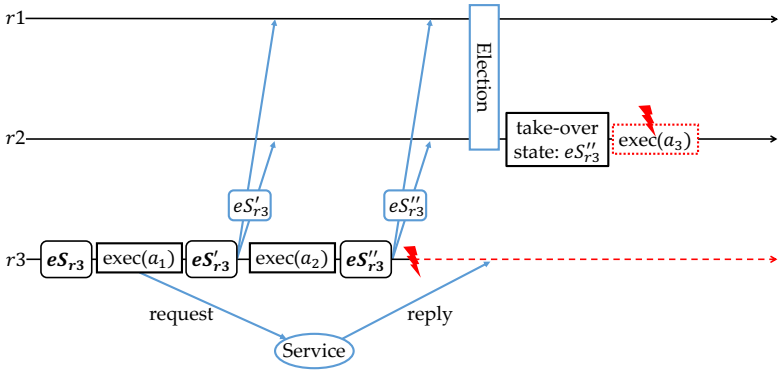


Figure 5.13.: Example of an asynchronous service call, were activity a_1 is the request activity and activity a_3 is the reply activity. The new primary $r2$ does not receive the service reply because the service is sending the reply to the request sender, i.e., the failed old primary $r3$.

activity. Hence, the new primary does not re-execute the request activity a_1 and, for this reason, also does not receive a reply from the service because the service sends the reply to the old primary $r3$ that has sent the request.

A similar problem arises for choreographies, where multiple workflows interact – in a possibly complex manner – by sending or receiving messages according to a specified interaction pattern [Wes07]. In Figure 5.14, for example, two workflows are interacting according to a (rather simple) choreography, where workflow 1 is sending a message to workflow 2 and, later, receives two messages from workflow 2 in return. Workflow 1 is executed in a replicated manner, where the replica $r3$ is primary and, thus, the interaction partner for workflow 2. If the replica $r3$ would fail, workflow 2 will still send its messages to the failed replica.

Workflow engines typically correlate messages to workflow instances by means of the sender's endpoint address, e.g., the IP address and port number of the message sender. With respect to our example in Figure 5.14 this means only messages sent from replica $r3$ are correlated with workflow 2. Even if replica $r2$ would try to continue the interaction, the workflow engine of workflow 2 would not deliver the messages to workflow 2 since the correlation would fail.

For overcoming the limitation, we extend the middleware that we already proposed for supporting non-compensable activities (cf. Figure 5.12). In specific,

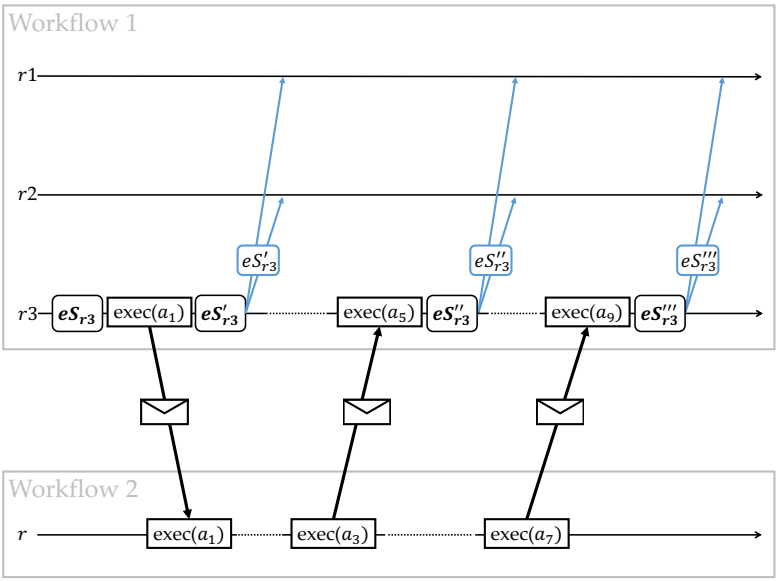


Figure 5.14.: Example of choreography between two workflow executions, where workflow 1 is executed in a replicated manner.

we propose to route all asynchronous service calls and choreography messages through a middleware making the workflow replication transparent.

Like with non-compensable service calls, the replicas have to decide on one workflow execution and on a unique interaction identifier (UIID). Then, the middleware uses the UIID for correlating the messages of the conversation with a workflow execution instead of using endpoint references of the replicas for the correlation. Any replica might send messages for the conversation to the middleware. The middleware forwards the message to the service or workflow. The service or workflow observes the middleware as one endpoint making the replication transparent. Consequently, any replica might continue a conversation. Moreover, the reply of the service or workflow will be routed to any available replica when letting the middleware monitor the availability of the replicas. In the Chapter 6, we will present how such a middleware is realized in detail.

6. HAWKS: A Middleware for Workflow Replication

FOR realizing the replication schemes with existing workflow engines, we will provide a software architecture in this chapter. The architecture is kept generic and, thus, can be used for adding the replication schemes to any existing workflow engine that manages imperative workflows. However, before we describe the architecture, we extract the requirements that the architecture needs to fulfill.

6.1. System Requirements

The system should fulfill several requirements regarding the generality of its applicability. It 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.[1] In other words, not all workflow engines have to serve as a replica during a replicated workflow execution enabling the possibility for balancing the workload between the registered workflow engines.

Automatic execution: It must be possible to trigger the replicated execution of a workflow by sending a message to one endpoint of the system. Linking and

[1] 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.

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. 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.

6.2. Architecture of the HAWKS System

As depicted in Figure 6.1, 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 used replication scheme. The HAWKS Controller is responsible for starting replicated executions and routing messages between the engines. In the following, we describe each component in detail.

6.2.1. Synchronization Unit

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

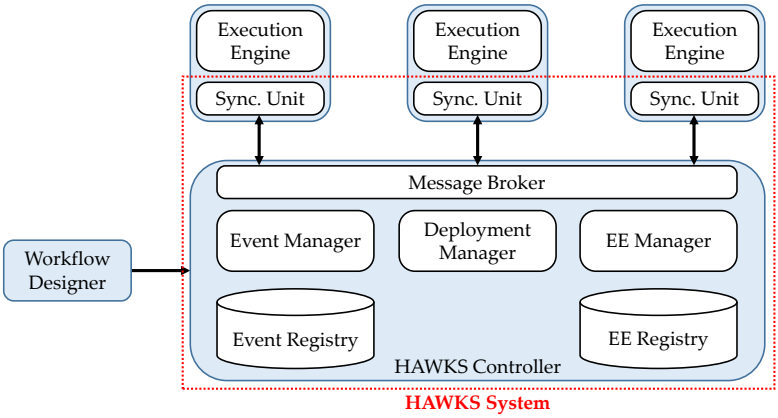


Figure 6.1.: Architecture of the HAWKS system

6.2.2. Message Broker

The Message Broker routes 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 makes it transparent to an interaction partner of a workflow execution with which replica they are interacting. This transparency is required for enabling the tolerance of replica crashes – especially for asynchronous service calls, non-compensable services, and choreographies as we already discussed in Section 5.3. Thus, the replicas send the according messages to the Message Broker, which then sends the messages to the interaction partner. Through this routing mechanism, the interaction partner perceives the replicated execution as a single endpoint satisfying the transparency requirement. Moreover, the Message Broker filters duplicate messages based on the unique interaction identifier as described in Section 5.3.

6.2.3. EE Manager and EE Registry

The Execution Engine Manager (EE Manager) and the Execution Engine Registry (EE Registry) 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.

6.2.4. 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 [WKK+10]). 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.

6.2.5. 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.

7. Evaluations

FOR showing the applicability of the HAWKS system to current workflow technology, we implemented a proof of concept in an existing open-source workflow engine. In the following, we present the proof of concept implementation. Subsequently, we evaluate our replication schemes – first our majority-based replication schemes and then our flexible failover replication scheme – with regard to availability, replication overhead, and performance. Finally, we will evaluate the scalability of our HAWKS proof of concept implementation.

7.1. Prototype

We show the applicability of the HAWKS system to existing workflow technology by extending the open-source workflow engine Apache ODE[1] and the pluggable framework for extended BPEL (ODE-PGF)[2], which is able to orchestrate workflows based on the Web Service Business Process Execution Language (WS-BPEL). WS-BPEL is standardized by OASIS[3] and widely used in industry. The Apache ODE is itself running in an Apache Tomcat[4] servlet container.

For realizing the communication between the engines, we use Apache ActiveMQ[5]. 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[6]. Furthermore, we also use it for routing messages sent from

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

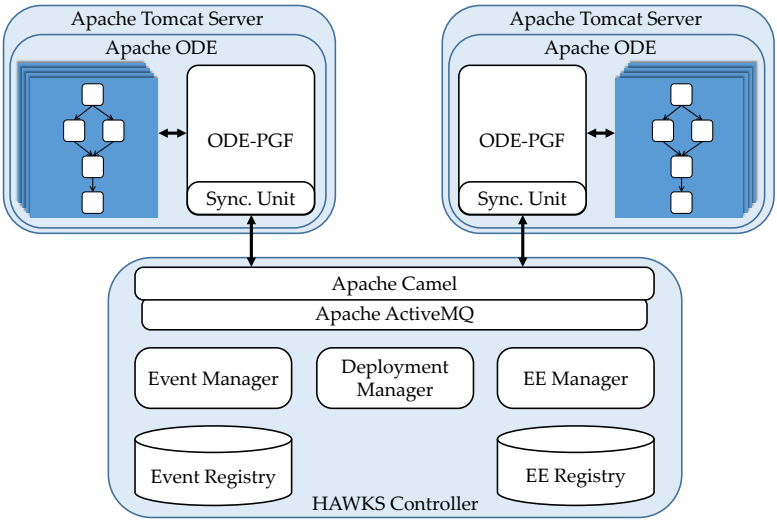


Figure 7.1.: Overview of the prototypical implementation and used technology

interaction partners to the replicas. The realization of the HAWKS system with the discussed technologies is depicted in Figure 7.1.

For reducing the synchronization overhead, we do not include the complete internal state of the workflows in the execution state updates messages. Instead, we use incremental updates, i.e., we track which variables are modified during activity executions and only send these modifications to the backups. In turn, the backups check that they received and applied all preceding execution state update messages. They can easily check this because the state number of the execution states is incremented with every update. We do not apply the update if a preceding update is missing.

7.2. Evaluation Setup

We evaluate our workflow replication system with regard to availability, replication overhead, performance, and scalability. Therefore, we evaluate the system on two different platforms: OpenStack and PlanetLab Europe. On Open Stack,

we use VMs with 1 vCPUs and 2 GB RAM. In PlanetLab, the nodes are heterogeneous because they are provided and shared among users. Consequently, we cannot specify the used hardware.

We generate random workflow models consisting of 100 activities, which is in the range of typical workflow lengths [DDGB09]. We assume the compensation cost to be monetary, where the compensation cost of the activities is set using a uniform distribution between \$0 and \$100. Deterministic, read-only activities occur with a probability of $\frac{1}{4}$. Each activity might either perform a local computation or call a prototypical service, where all service calls are synchronous. In specific, all read-only activities perform local computations, while all other activities call services.

When stabilizing the state or when performing an election, a primary has to receive a response from a majority of replicas to be able to continue the execution. The primary resends the message when not receiving the required responses within 150 ms. During the workflow execution, the primary sends a heartbeat message to each replica every 100 ms. The timeout for detecting a primary failure is 400 ms. The values have been determined experimentally and ensure low execution times while only triggering elections when the primary is actually failed.

We compare our replication schemes to a non-replicated execution as well as to pure active replication. When using active replication, the executions on the different replicas are completely independent of each other. In the end, the replicas perform a majority consensus to agree on one of the executions and compensate all others. Overall, we performed more than 100,000 workflow executions and averaged over these results.

7.3. Majority-based Replication Schemes

In the following, we will evaluate our basic and relaxed majority-based replication scheme with regard to availability, execution time overhead, and compensation overhead. Since our relaxed majority-based replication scheme requires the workflow to be divided into synchronization groups for supporting the switching between passive and active replication, we use our grouping guidelines for deciding the grouping. In specific, we set the failover time threshold t_t (cf. Section 5.1.2.2)

to 500 ms (*Relaxed MBR* 500) and 10000 ms (*Relaxed MBR* 10000). We only limit the failover time (but not the compensation cost) because the failover time has a direct influence on the availability in case of failures. We will compare the relaxed majority-based replication *Relaxed MBR* 500 and *Relaxed MBR* 10000 to the basic majority-based replication scheme (*Basic MBR*), where the state is replicated synchronously after each activity execution. As already mentioned, we additionally compare our replication schemes to a non-replicated execution and pure active replication.

7.3.1. Availability

The main purpose of replication is to improve availability in the presence of failures. In general, the availability is the probability (or percentage of time) that a system is available (i.e., providing the intended functionality) [Woo95, CDK05]. In other words, the availability α is the uptime of the system t_{up} (i.e., the time the system is available) divided by the uptime plus the downtime t_e [Woo95]:

$$\alpha = \frac{t_{up}}{t_{up} + t_{down}} \tag{7.1}$$

For each workflow execution, we measure the downtime t_{down} and the overall execution time t_{exec}. The downtime is the time that the workflow execution does not make progress due to failures. The execution time is the time from initiating the workflow execution until receiving the result of the workflow execution. Because $t_{exec} = t_{up} + t_{down}$, we can reformulate Equation 7.1 as follows:

$$\alpha = \frac{(t_{exec} - t_{down})}{t_{exec}} \tag{7.2}$$

For evaluating the availability in the presence of failures, we randomly inject failures during the workflow execution. With a mean time to recovery of 10 s, failures are short lived because typical failures even last up to multiple days [BFF+14]. However, replication is obviously beneficial for long lived failures. Instead, we show our replication scheme to be worth its overhead even when failures are short lived. Moreover, we assume that a failed engine can resume any started workflow execution after recovering from the failure.

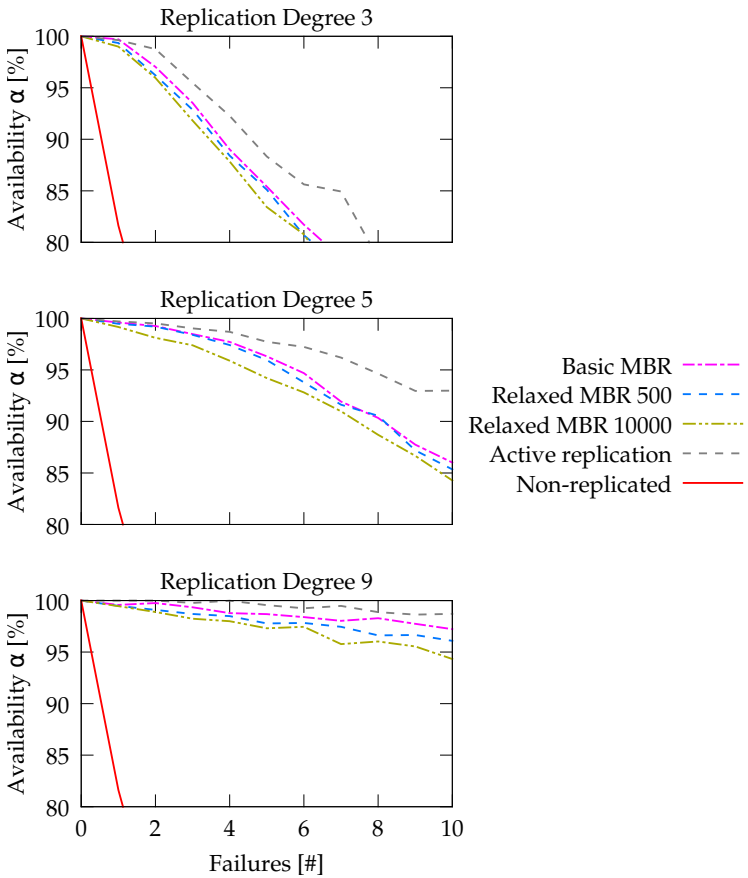


Figure 7.2.: Plotting the availability α (cf. Equation 7.2) against the number of injected failures for our majority-based replication schemes running on OpenStack. [Higher is better]

The Open Stack measurements of Figure 7.2 show the availability plotted against the number of injected failures. We can observe that the availability of the availability of the non-replicated execution steeply decreases with an increasing number of failures. Since the execution is not replicated, each failure causes an outage lasting for the mean time to recovery leading to the steep decrease of availability. Here, the non-replicated execution performs significantly worse than any replicated execution. This shows that workflow replication is a valid approach for increasing the availability. In the following, we will compare the different replication schemes.

In terms of availability, active replication outperforms all other replication strategies. The workflow execution provides over 99.5 % availability with up to f failures. With more than f failures, the majority consensus during termination might be delayed causing a linear decrease in availability. However, active replication is anyway not desirable because of its compensation overhead as we will see in Section 7.3.3.

Now, we compare the basic majority-based replication scheme to our relaxed majority-based replication scheme, which uses both active and passive replication. The basic majority-based replication scheme provides the best availability, followed by *Relaxed MBR* 500 and, finally, *Relaxed MBR* 10000. This is due to the grouping of the activities. In case of a primary failure, the basic majority-based replication scheme has to re-execute at most one activity, while *Relaxed MBR* 500 and *Relaxed MBR* 10000 might need to re-execute the complete synchronization group in the worst case (cf. Section 5.1.2). When increasing the threshold from $t_t = 500\,\text{ms}$ to $t_t = 10000\,\text{ms}$, the passively replicated synchronization groups contain more activities and the re-execution might take even longer. Thus, the average failover time is increased causing longer times of downtime. This can especially be observed for replication degree 9 in Figure 7.2.

In general, we observe that replication significantly improves availability. Higher replication degrees tolerate more failures and, thus, improve availability further. Our relaxed majority-based replication schemes *Relaxed MBR* 500 and especially *Relaxed MBR* 10000 reduce availability compared to active replication and our basic majority-based replication scheme. We, however, have to put these results into context because replication always implies overhead. Thus, we will

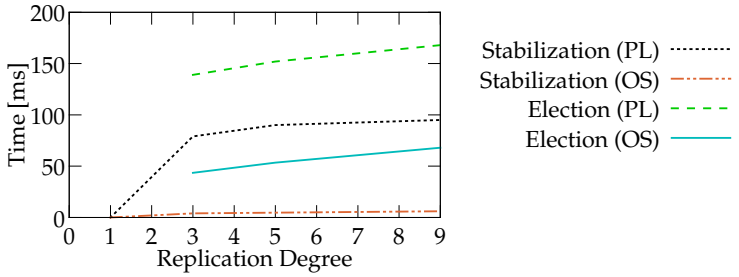


Figure 7.3.: Plotting the time required for elections and stabilizations against the replication degree. The measurements were performed on PlanetLab (PL) and on Open Stack (OS). [Lower is better]

evaluate the execution time overhead of the replication strategies in the next section.

7.3.2. Execution Time Overhead

The replication implies execution time overhead for stabilizing an execution state at the end of a synchronization group as well as for electing a new primary. In Figure 7.3, we depict the overhead of stabilizing the state and of performing an election in Open Stack as well as in PlanetLab.

Figure 7.3 shows that the time overhead for a stabilization increases linearly on Open Stack and PlanetLab from the replication degrees 3 to 9. On Open Stack, the overhead stays below 10 ms even for up to 9 replicas, while in PlanetLab the overhead is around 100 ms for 9 replicas. Electing a new primary takes around 45 ms with 3 replicas and linearly increases to around 60 ms for 9 replicas on Open Stack. On PlanetLab, the election time increases linearly, taking 140 ms for replication degree 3 and 170 ms for replication degree 9. Even though the overhead in PlanetLab is higher, it is still relatively low when considering the distances between the geo-distributed replicas in PlanetLab compared to the Open Stack setup, where all replicas are placed in one server rack.

For evaluating the impact of the different replication degrees on the performance on a complete workflow execution, we execute the workflows in PlanetLab. This setting imposes the highest execution time overhead for stabilizations.

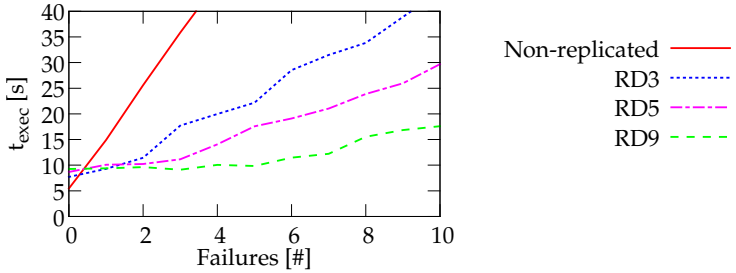


Figure 7.4.: Plotting the overall execution time t_{exec} of our basic majority-based replication scheme on PlanetLab against the number of injected failures. [Lower is better]

Additionally, we use our basic majority-based replication scheme forcing a stabilization after each activity. In conclusion, we use the worst case configuration with respect to execution time for making the differences between the replication degrees obvious. In this measurement, we depict the absolute execution time of the workflow executions. Hence, we generate a single workflow used for all executions to make the execution times comparable. Again, we randomly inject failures with a mean time to recovery of 10 s.

Figure 7.4 depicts the number of failures plotted against the execution time for different replication degrees. For a non-replicated execution, the execution time increases linearly with the number of failures. Intuitively, each failure stops the execution until recovery. Because the non-replicated execution does not replicate state and, hence, has no stabilization overhead, it performs better in the failure-free case. However, even with a single failure, the replicated executions perform better. A replication degree of 3 (*RD*3) can already tolerate one failure. For more failures, the execution time linearly increases. However, less steep than the non-replicated. The behavior of an increasing failure tolerance for increasing numbers of replicas can be observed across all replication degrees. However, higher replication degrees also imply a higher execution time overhead, where the replication degree 9 (*RD*9) takes almost 2 s longer than *RD*3. On the other hand, replication degree 9 remains unaffected with up to 5 failures.

In summary, we lose performance in the failure free case in terms of execution time when increasing the availability by using higher replication degrees. However, the execution time overhead of stabilizing the state is negligible when a single failure occurs – even in a geo-distributed setting like PlanetLab when forcing a stabilization after each activity.

Now, we will evaluate the execution time overhead in Open Stack, which mirrors a datacenter setup, where the different replicas are executed closely together, e.g., in the same server rack. Here, we will also inspect how our synchronization groups can be used to reduce the stabilization overhead.

Figure 7.5 depicts the different replication approaches with respect to their execution time overhead. Note that the generated workflows have different execution times, where each activity is assigned a random execution time (using the absolute values returned from a random generator based on a Gaussian distribution with a mean of 0 ms and a standard deviation of 500 ms). To make different workflow models comparable, we normalize the measured execution time by the minimum time for executing the whole workflow. Thus, the best possible execution time is 100 %. An execution time above 100 % in the failure free case is caused by the overhead for stabilizing the state. The increase of the execution time with an increasing number of failures shows the time that the execution is delayed through the injected failures. This allows to compare the overall performance of the replication schemes and puts the availability of the different replication schemes (cf. Figure 7.2) into context.

As depicted in Figure 7.5, active replication does imply almost no time overhead compared to a non-replicated execution in the failure free case. The basic majority-based replication scheme implies significant execution time overhead through the stabilization that is required after each activity execution. The workflow is delayed by around 6 % compared to a non-replicated execution, no matter how many replicas are being used. In other words, the replication degree has basically no impact on the execution time overhead when all replicas are located in one server rack. The overhead of the basic majority-based replication scheme is, however, still significant. Our relaxed majority-based replication scheme reduces the overhead considerably, i.e., $t_t = 500$ ms saves around $\frac{1}{2}$ and $t_t = 10000$ ms saves more than $\frac{2}{3}$ for failure free executions.

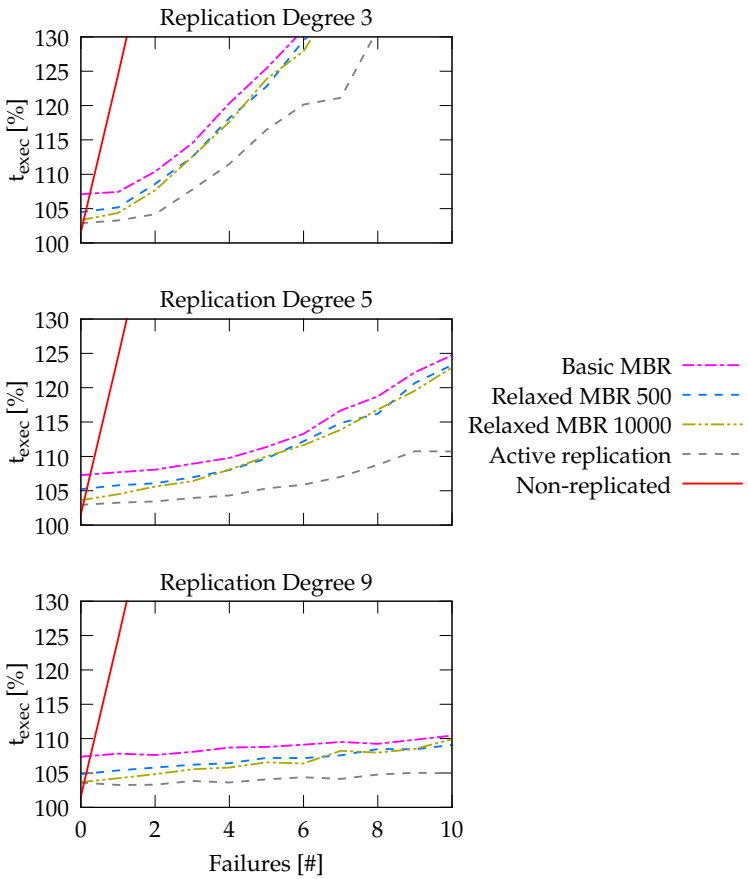


Figure 7.5.: Plotting the execution time t_{exec} of our majority-based replication schemes against the number of injected failures. [Lower is better]

This, however, means that in the worst case we need to re-execute a complete synchronization group when a primary fails. As a consequence, *Relaxed MBR* 10000 quickly loses the execution time advantage that it has over *Relaxed MBR* 500 because the groups contain more activities. In specific, when using 3 replicas and there are at least 3 failures, both *Relaxed MBR* 500 and *Relaxed MBR* 10000 perform basically identical in terms of the execution time. With 5 replicas, *Relaxed MBR* 500 and *Relaxed MBR* 10000 perform identical when there occur more at least 4 failures. With 9 replicas, this point is even delayed to at least 7 failures. Additionally, we can observe that the execution time advantage that the relaxed majority-based replication scheme, i.e., *Relaxed MBR* 500 and *Relaxed MBR* 10000, has over the basic majority-based replication scheme reduces with an increasing number of failures. This trend is especially noticeable for replication degree 9.

Surprisingly, the execution times of the relaxed majority-based replication do not increase beyond the basic majority-based replication – even for $t_t = 10000\,\text{ms}$. In other words, it is mostly more efficient to tolerate longer downtime by increasing t_t for reducing the overall execution time. This, however, is also due to our grouping guidelines, where all deterministic, read-only activities are grouped for using active replication (independent of t_t). With these synchronization groups already given, it is then more efficient (in terms of execution time) to choose a high value for t_t, i.e., to tolerate higher failover times than to stabilize more often.

In conclusion, our relaxed majority-based replication scheme increases the performance of workflow replication significantly compared to the basic majority-based replication scheme. We save more than $\frac{2}{3}$ of the execution time overhead that the basic majority-based replication scheme implies. However, this means a complete synchronization group might be re-executed in case of a primary failures, which also increases the compensation cost. Thus, we will now evaluate the compensation overhead implied by the different replication schemes.

7.3.3. Compensation Cost

We measured the number of compensated activities as well as the compensation cost in the presence of failures, which is depicted in Figure 7.6 and Figure 7.7. Remember that each activity of our generated workflow models has a random

compensation cost between \$0 and \$100. To make the different workflow models comparable, we normalize the compensation cost. A compensation cost of 100 % is equal to compensating all activities of the workflow model once. The optimal value is 0 %.

The basic majority-based replication scheme implies the smallest compensation cost because at most one activity has to be re-executed and, hence, compensated in case of a primary failure. While the number of compensated activities increases mostly linearly with the number of failures for all replication schemes (cf. Figure 7.6), the relaxed majority-based approach *Relaxed MBR* 500 compensates around twice as many activities compared to the basic majority-based replication scheme. *Relaxed MBR* 10000 even compensates around four times as many activities as the basic majority-based replication scheme.

In Figure 7.7, we can observe that active replication is no feasible solution. In the failure free case, it implies 200 % compensation cost with replication degree 3, 400 % with 5 replicas, and 800 % with 9 replicas. The compensation cost gets smaller with an increasing amount of failures because crashed replicas do not execute activities and, thus, reduce the compensation cost.

The observations of Figure 7.6 are also reflected by the compensation cost in Figure 7.7. For example, with 10 failures and replication degree 3, the basic majority-based replication scheme implies around 3 % compensation cost, while *Relaxed MBR* 500 implies 6 %, and *Relaxed MBR* 10000 incurs 12 %. With the higher replication degrees of 5 and 9 replicas, the number of compensated activities and the compensation cost grows slower. The reason is that a higher replication degree decreases the probability that a failure affects the current primary. The workflow designer might directly set a maximum compensation cost that is allowed per failure by setting c_t. This provides a finer control over the absolute compensation cost. The principle of setting c_t is similar to setting t_t: Lower values for c_t decrease the size of the groups and, thereby, increase the number of required stabilizations.

7.3.4. Communication Cost

The primary coordinates the execution by sending and receiving messages. The main communication cost is imposed by the messages that contain the inter-

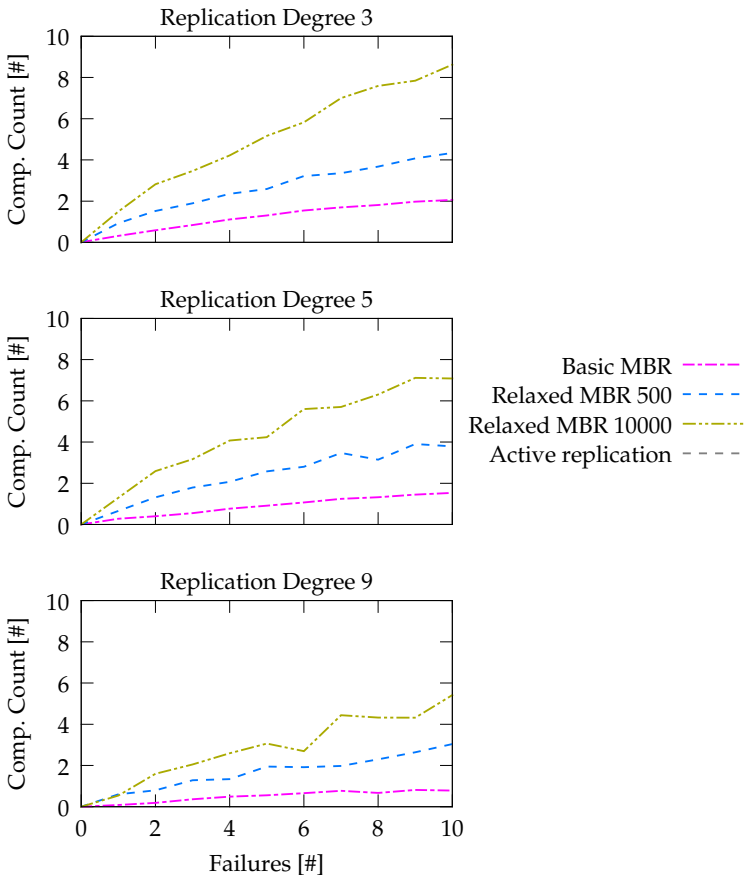


Figure 7.6.: Plotting the number of compensated activities of our majority-based replication schemes against the number of injected failures. [Lower is better]

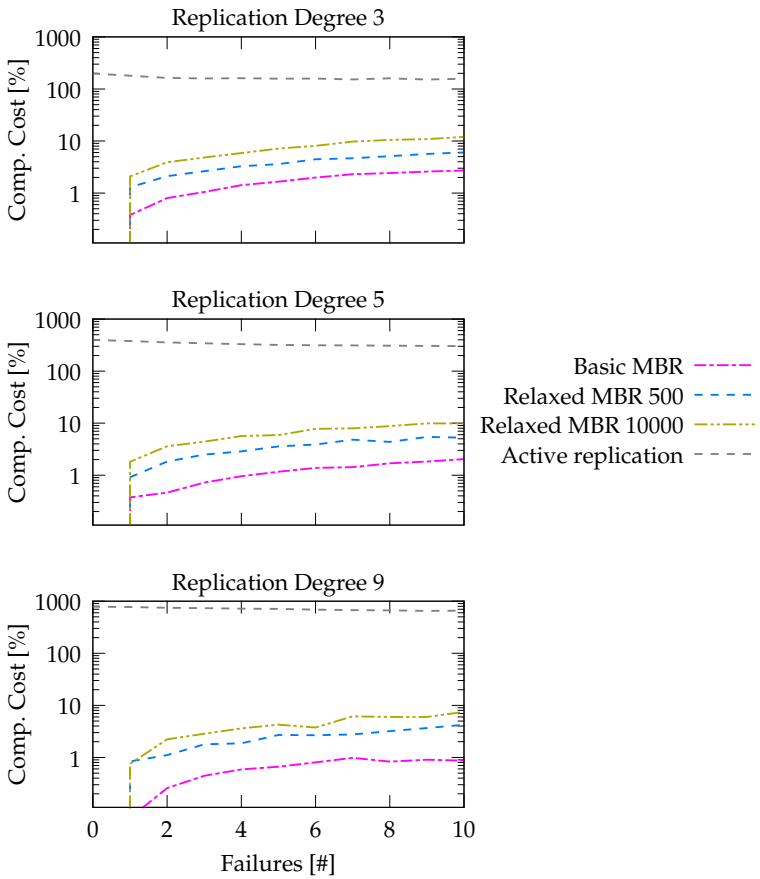


Figure 7.7.: Plotting the compensation cost of our majority-based replication schemes against the number of injected failures. [Lower is better]

nal state of the workflow, i.e., the UPDATE and STABILIZE messages. The actual cost depends on the size of the internal state and how state information is transferred (e.g., incremental versus complete). Within a synchronization group, the primary sends exactly one UPDATE message per activity and backup. STABILIZE messages might be sent more than once in case the primary times out when waiting for a majority of replicas to respond. In our experiments, the primary sends approximately 1.1 STABILIZE messages per passively replicated synchronization group and backup – independent of the used replication degree (i.e., 3, 5, or 9).

7.4. Flexible Failover Replication Scheme

In the previous section, we have evaluated our majority-based replication schemes. The measurements have revealed that there is an obvious gap between active replication and our majority-based replication in terms of availability and execution time overhead (cf. Figure 7.2 and Figure 7.5). Our flexible failover replication scheme strives to close this gap. Additionally, flexible failover replication tolerates partitioning failures that partition the replicas such that none of the partitions contain a majority of replicas. Thus, each injected failure now either partitions the network or crashes one replica. The ratio of partition to crash failures is 1 to 4 reflecting that partitioning failures occur less frequently than multi-crash failures [Bre17]. Both partition and crash failures have again a mean-time to recovery of 10 s.

For comparison, we again evaluate an active replication strategy and a non-replicated execution. Additionally, when setting the vote threshold t_v to $\lfloor \frac{|R|}{2} \rfloor + 1$, our flexible failover replication scheme is identical to the majority-based replication scheme with the complete workflow encapsulated in one synchronization group. Since this is the configuration, where the majority-based replication has the fewest execution time overhead, this is the best benchmark for comparing it to our flexible failover replication scheme.

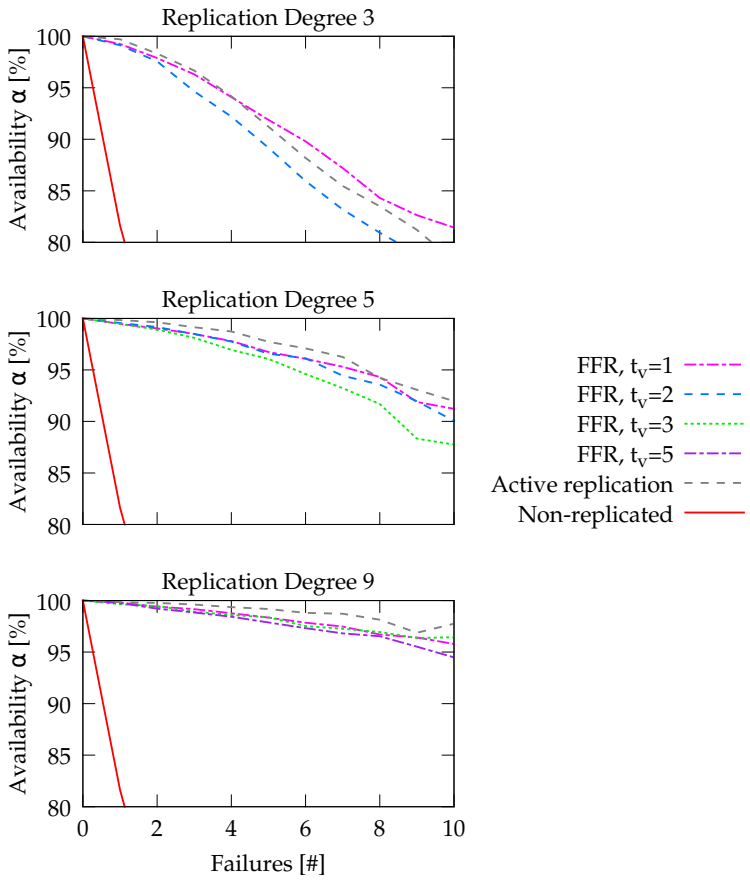


Figure 7.8.: Plotting the the availability α (cf. Equation 7.2) of our flexible failover replication scheme against the number of injected failures. [Higher is better]

7.4.1. Availability

In Figure 7.8, we evaluate the replicated workflow executions with 3, 5 and 9 replicas as well as a non-replicated execution. The availability α (cf. Equation 7.2) of the non-replicated execution is steeply decreasing with failures. Actually, each failure delays the non-replicated execution by the mean time to recovery. Any replicated execution clearly outperforms the non-replicated execution.

With only one failure, the availability of active replication and the flexible failover replication scheme (with the different settings of the vote threshold) remains almost unaffected. This is not surprising because all approaches can tolerate one partitioning or crash failure. Only short failover times for (possibly) electing new primary after a failure impact the availability. With more than one failure, the approaches start to diverge. In specific, one partitioning plus one crash failure can create two partitions, where no partition contains a majority of replicas. This means that majority-based replication (i.e., $t_v = 2$ for 3 replicas, $t_v = 3$ for 5 replicas, and $t_v = 5$ for 9 replicas) cannot elect a primary. Here, our flexible failover replication with $t_v = 1$ performs substantially better because it continues the execution even if all but one replica fail – similar to active replication. With replication degree 3, our flexible failover replication scheme with $t_v = 1$ even outperforms active replication for more than 4 failures. The reason for this behavior is that active replication executes the workflows on all replicas independently. Thus, when all replicas fail during the execution, each execution is delayed by the mean time to recovery. In contrast, our flexible failover replication scheme sends asynchronous updates to all other replicas. Thus, when a replica recovers from a failure it might receive an execution state from another replica which it will use for continuing its execution – in case it becomes primary later. With higher replication degrees, our flexible failover replication scheme cannot outperform active replication anymore. It becomes very unlikely that the randomly injected failures delay all the replicas of the active replication. Thus, on average, the effects of this failure scenario cannot be observed anymore.

The advantage of the flexible failover replication scheme – no matter which value was chosen for the vote threshold – over the majority-based replication

scheme gets smaller when comparing replication degree 5 to replication degree 3. This is due to the increased failure tolerance of higher replication degrees. This trend continues for replication degree 9, where all replication schemes almost behave identical.

In conclusion, when striving for availability, setting low values for t_v allows to reach near active replication performance. Especially, with a low replication degree, such as replication degree 3, setting $t_v = 1$ increases availability significantly compared to our majority-based replication scheme.

7.4.2. Execution Time Overhead

In Figure 7.9, we depict the execution time of the flexible failover replication scheme. Since we are still using the randomly generated workflows as described in Section 7.3.2, we again normalize the measured execution time by the minimum time for executing the whole workflow. Thus, the best possible execution time is 100 %. An execution time above 100 % in the failure free case is the overhead of the replication scheme. The increase of the execution time with an increasing number of failures shows the time that the execution is delayed through the injected failures. This allows to compare the overall performance of the different replication schemes and put the availability (cf. Figure 7.8) into context.

We can observe that basically all of the replication strategies have the same overhead for failure free executions. Compared to a non-replicated execution, the replication strategies need 1 % longer. This is the time required for the termination, where the replicas perform a majority consensus to agree on one workflow execution. Thus, we match the execution time overhead of the active replication scheme.

The increased amount of execution time with an increasing number of failures depicts the downtime. Since all replication schemes have the same execution time overhead, a discussion about the execution time in the presence of failures is congruent to discussing the availability (cf. Section 7.4.1).

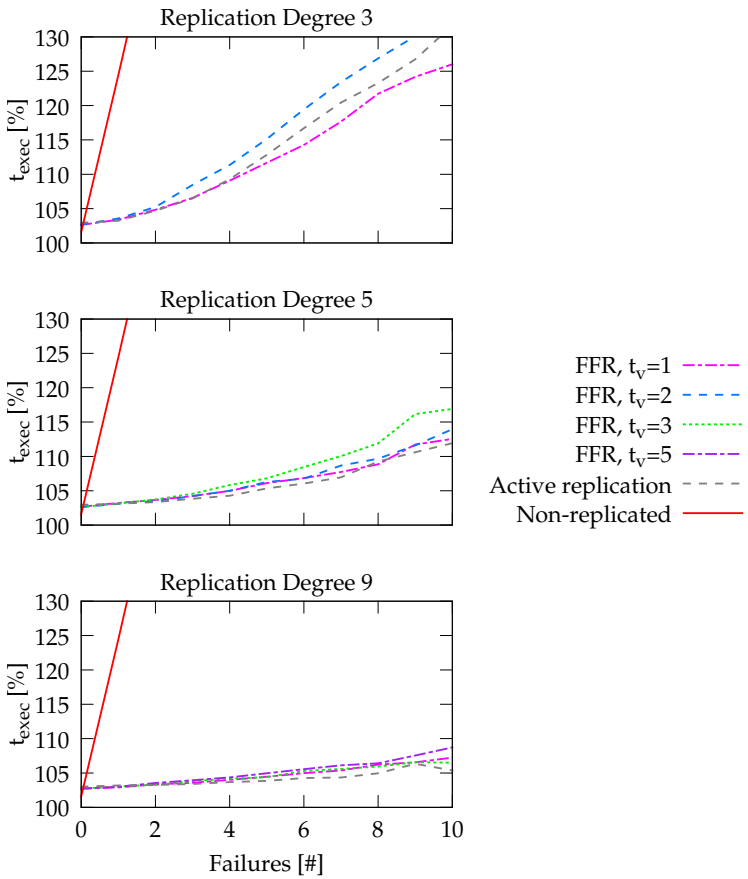


Figure 7.9.: Plotting the execution time t_{exec} of our flexible failover replication scheme against the number of injected failures. [Lower is better]

7.4.3. Compensation Cost

The increased availability through our flexible failover replication scheme, of course, comes at a cost. In the worst case, when setting $t_v = 1$, all replicas are partitioned from each other and execute the workflow independently like active replication. Even though this scenario is unlikely, it shows that the flexible failover replication scheme increases availability at the cost of compensation.

Figure 7.10 shows the number of compensated activities plotted against the number of failures, while Figure 7.11 shows the cost. Like in Section 7.3.3, we again normalize the compensation cost to make the different workflow models comparable. A compensation cost of 100 % is equal to compensating all activities of the workflow model once. Here, the optimal value is 0 %.

The compensation cost again shows that active replication is no feasible solution because it implies 200 % compensation cost with replication degree 3, 400 % with 5 replicas, and 800 % with 9 replicas. All other replication schemes, i.e., our majority-based and our flexible failover replication scheme, do not cause any compensation cost in the failure free case. With more than one failure, flexible failover replication incurs higher compensation cost when the vote threshold t_v is decreased. As decreasing t_v allows smaller partitions to elect a primary, partitioning failures lead to more competing workflow executions and, eventually, to more compensations. However, compared to our majority-based replication scheme, the compensation cost in the failure case is tremendous – especially when considering that with smaller synchronization groups the compensation cost can be reduced further. With replication degree 3, reducing the vote threshold from $t_v = 2$ (i.e., our majority-based replication scheme, where all activities are encapsulated in a single synchronization group) to $t_v = 1$ doubles the number of compensated activities when failures occur. With 5 and 9 replicas, the amount of compensated activities is also doubled when setting the vote threshold to $t_v = 1$ compared to the majority-based replication scheme. Here, setting intermediate values for the vote threshold allows to exploit the space in between.

In conclusion, flexible failover replication allows to reach near active replication performance while inducing no compensation cost in the failure free case like passive replication. This is especially useful with low replication degrees, such as

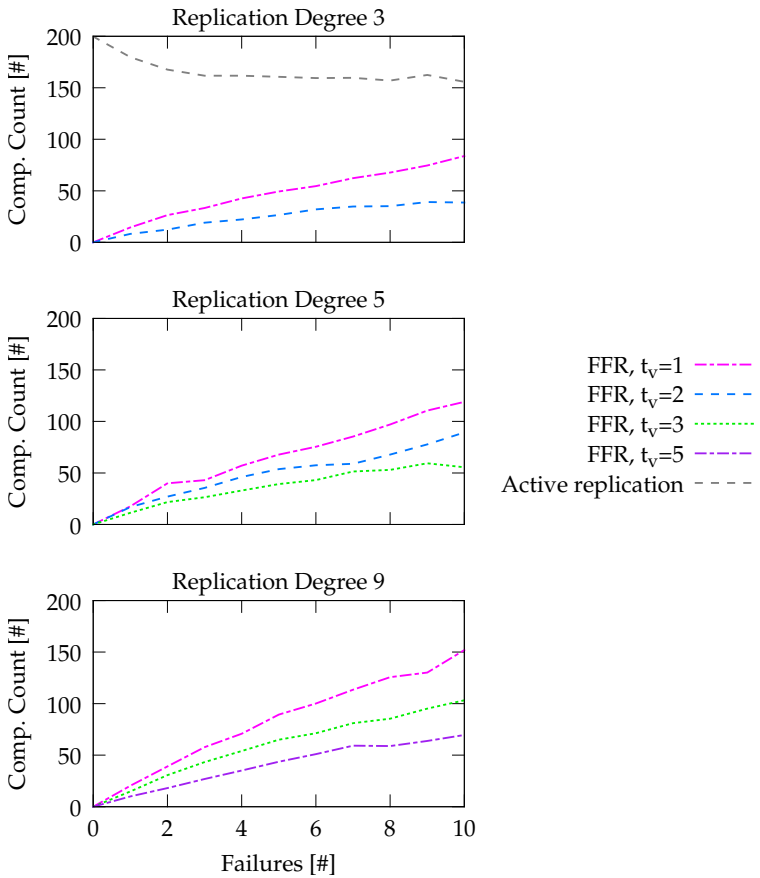


Figure 7.10.: Plotting the number of compensated activities of our flexible failover replication scheme against the number of injected failures. [Lower is better]

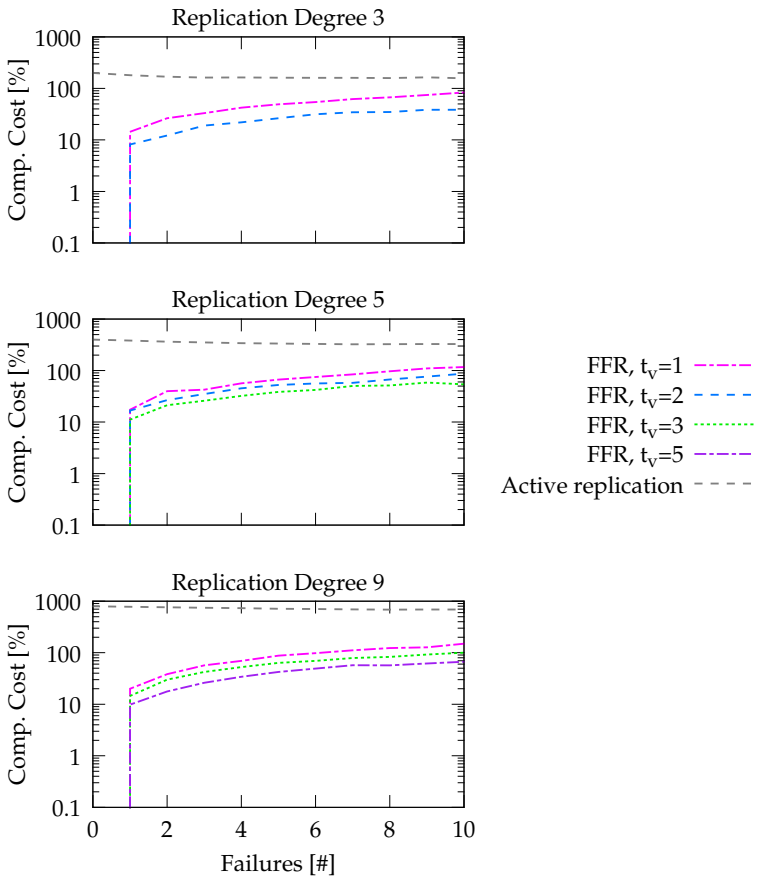


Figure 7.11.: Plotting the compensation cost of our flexible failover replication scheme against the number of injected failures. [Lower is better]

replication degree 3, where setting $t_v = 1$ and tolerating the higher compensation cost in failure cases increases availability significantly.

7.4.4. Communication Cost

The flexible failover replication scheme shares the properties of the majority-based replication scheme with regard to communication cost. The main communication cost is imposed by the messages that contain the internal state of the workflow, i.e., the UPDATE. The actual cost depends on the size of the internal state and how state information is transferred (e.g., incremental versus complete). The primary sends exactly one UPDATE message per activity and backup.

7.5. Scalability of the HAWKS System

So far, we reserved the engines exclusively for the execution of a single (replicated) workflow execution. This ensured that the execution times were not influenced through overloaded engines. However, the engines are capable of executing multiple workflows concurrently. Now, we investigate our HAWKS system regarding high workloads without injecting failures in OpenStack, using VMs with 4 vCPUs and 8 GB RAM. Because the basic majority-based replication implies the highest message overhead because every state update requires replies from the backups, we use this replication scheme for evaluating the scalability.

We double the workload every 300 s until the point when the workflow engines become overloaded. Initially, we start one workflow execution every 60 s, then every 30 s, and so on. For evaluating the scalability, we add more computing nodes running workflow engines. Then, for each replicated execution, we select some of the computing nodes to participate in the specific replicated execution. We first evaluate the replication degree 3 with 3 engines (RD3E3), i.e., all engines host one replica per workflow execution. We also evaluate the setup of replication degree 3 with 6 engines (RD3E6) and with 12 engines (RD3E12). In these cases, the workload can be distributed to the different engines. Figure 7.12 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

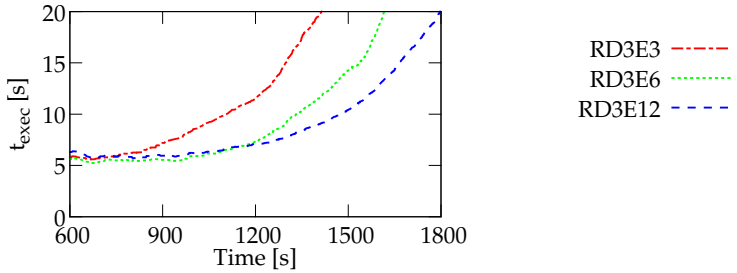


Figure 7.12.: Plotting the execution time t_{exec} against the time that the experiment is running, where the workload is increasing over time. [Lower is better]

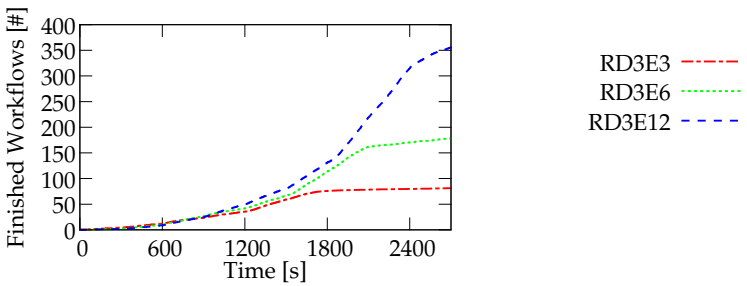


Figure 7.13.: Plotting the number of finished workflow executions against the time that the experiment is running, where the workload is increasing over time. [Higher is better]

a workflow every 7.5 s) and RD3E12 even delays the point to 1200 s (starting a workflow every 3.75 s).

For making the effects of this clear, we show the accumulated number of finished workflow executions in Figure 7.13. RD3E3 is overloaded after executing around 70 workflow executions, RD3E6 after around 150 workflows, while RD3E12 finishes around 340 workflows before being overloaded. This clearly shows the scalability of the HAWKS system.

8. Related Work

THERE is a huge body of work on replication in the field of transaction-oriented systems [BDF+13]. Those replication schemes are based on the assumption that groups of activities are performed as atomic units of work. There is also an extensive literature on active and passive replication of services [CBPS10, OO14, LC12, Lam98] performing (non-atomic) activities as we already described in Chapter 2. We borrow ideas from those approaches like the structure of the election, consensus, and recovery procedure. However, those approaches are based on the assumption that activities cannot be undone or compensated once they are performed. In contrast, we exploit the possibility of compensation to relax normal processing: Rather than performing consensus for every activity, we only require consensus during termination, primary election, and state stabilization.

There are a few other approaches aiming at fault-tolerant workflow engines [LLdSFV08, SS13]. These approaches assume a fail-stop model [CBPS10], where each engine that is detected as failed is actually failed. However, such a perfect failure detector cannot be achieved in practice because an engine might simply be slow to respond [FLP85, CBPS10]. Our scheme assumes the crash-recovery model [CBPS10], meaning that we detect every failure eventually but we might falsely assume an operational engine to be failed. We precisely defined a correctness criteria for workflow replication. To fulfill the criteria, old primaries have to compensate activities that are re-executed by the new primary. In contrast, the fail-stop model implies that an engine that fails stays failed indefinitely [CBPS10]. Hence, old primaries cannot compensate any of its activities. Instead, the approaches focus on the new primary correctly taking over the execution from the last received execution state. Other approaches propose active replication for workflows consisting solely of read-only activities [AGK04, BHR08, BHR09a, BHR09b, GEST09, ZRXS10, CB14]. This limits the

workflow to only use activities that do not change a service's state. Hence, our replication schemes are more general.

There exist many approaches to increase the availability of services (cf. Chapter 2), such as replicating the servers of a service [OO14, LC12, Lam98], or using alternative services in case of failures [BTKR15, SPJ11, KHC+05]. However, as mentioned above, those schemes are complementary to our proposed replication schemes for workflow engines.

Part B.

Replication Schemes for Declarative Workflow Languages

9. System and Declarative Workflow Model

THE concepts presented in Part A increase the availability of workflows that are defined in an imperative workflow language (cf. Chapter 3). Imperative workflow languages define the order in which the activities of the workflow must be executed through an ordering relation. Thus, there is a predefined execution order for the activities. There, however, exist workflow languages that allow greater flexibility with regard to the execution order of the activities: *declarative workflow languages*. These languages define constraints on the execution order, where any execution order that does not violate a constraint is allowed.

Of course, this greater flexibility raises the question of how much of the previously defined concepts for imperative workflow languages can be applied to declarative workflows. In order to answer this question, we first define the declarative workflow model and introduce a scenario. Based on this scenario, we can discuss the differences to our previous concepts in the following Chapter 10.

As in Part A, we consider a system that consists of a collection of nodes, connected by a communication network. The nodes and the communication links might experience crash failures at any point in time. According to the crash recovery model, we assume that every failed node or communication link eventually recovers from the failure [CBPS10].

Each node of the network might run a workflow engine, a service, or both. The engines execute workflows, which call the available services. The workflows are modeled in a declarative workflow language based on *linear temporal logic* (LTL), such as Declare [PSvdA07, Pes08]. The declarative workflow language allows us to constrain the order in which the activities of a workflow are allowed to execute by constructing an LTL formula on the activities from a defined set of

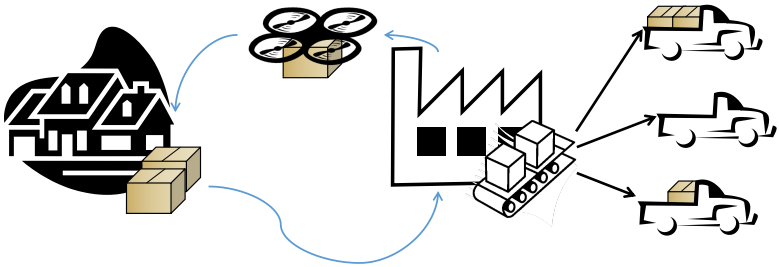


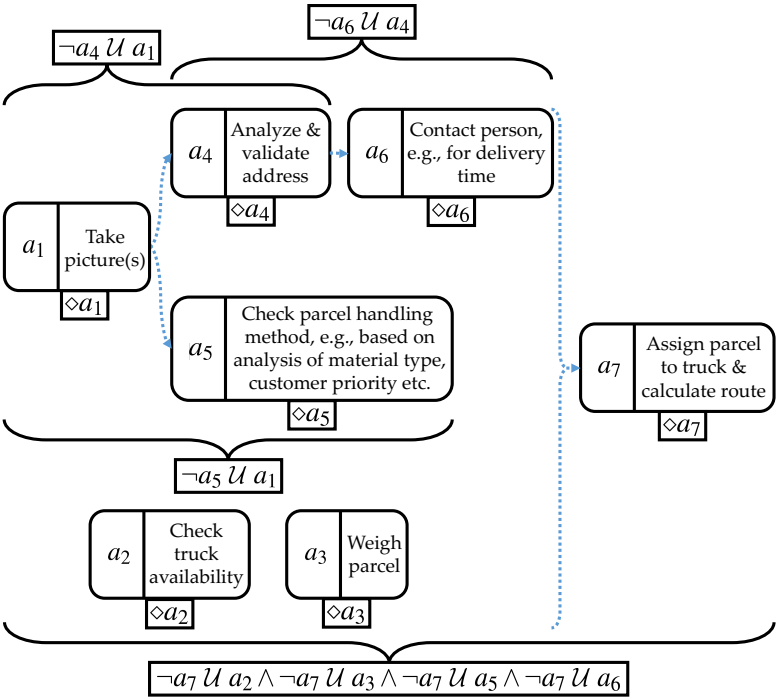
Figure 9.1.: A drone picks up parcels and delivers them to the next post office and vice versa

linear time operators [EH86, Pes08]. All other activity sequences, which are not explicitly forbidden by the LTL formula, are allowed. In the following, the term *workflow specification* (or W for short) is used for the LTL formula that specifies a workflow.

To clarify our contributions, we constructed a scenario in the context of drone parcel delivery services (cf. Figure 9.1). Drone delivery has gained an enormous attention lately. DHL already successfully tested a drone delivery and companies like Amazon have shown great interest in the technology.[1] The parcel delivery companies usually compete for delivery times. Drones are especially interesting to these companies because drones are not hindered by traffic jams and can deliver parcels around the clock without much human involvement. In particular, drones can be used to pick up (one or more) parcels from customers and deliver them to the nearest post office (or warehouse) and vice versa. However, between post offices (or warehouses), it still will be more efficient to use large trucks because of the amount of parcels to be transported. The whole parcel delivery process chain can be automated and modeled as a workflow.

Figure 9.2 shows an exemplary workflow for picking up one parcel. To keep delivery times low, the execution of the workflow should be finished latest when the drone reaches the post office. The LTL formulas in Figure 9.2 depict that all activities in the workflow specification need to be executed. This is specified by the $\diamond$ (*finally*) operator. For example, $\diamond a_1$ specifies that activity a_1 has to happen

[1] http://www.cbsnews.com/news/dhl-testing-delivery-drones/



<u>Workflow Specification</u>

$$\diamond a_7 \wedge \neg a_4 \mathcal{U} a_1 \wedge \neg a_6 \mathcal{U} a_4 \wedge \neg a_5 \mathcal{U} a_1 \wedge \neg a_7 \mathcal{U} a_2 \wedge \neg a_7 \mathcal{U} a_3 \wedge \neg a_7 \mathcal{U} a_5 \wedge \neg a_7 \mathcal{U} a_6$$

Figure 9.2.: Workflow specification of a drone picking up one parcel. An arrow means that the activity at the end of the arrow is only allowed to execute after the activity at the root of the arrow has terminated (inspired by Declare [Pes08]). The boxes show the LTL formulas that model the specific behavior.

finally, i.e., it needs to be executed at least once. Also, a_4 can only be executed after a_1 has finished because a picture can only be analyzed after it has been taken. The respective LTL formula $\neg a_4 \, \mathcal{U} \, a_1$ specifies that a_4 must not execute *until* a_1 was executed and that a_1 finally has to be executed. Combining all the LTL constraints depicted in Figure 9.2 leads to the workflow specification, which is shown at the bottom of the figure.

Activity a_7 should be executed only once because otherwise the system assigns one parcel multiple times. This leads to trucks that reject other parcels because the system falsely assumes the capacity limit of the truck is already reached. Hence, activity a_7 is *non-idempotent*. In contrast, activity a_2, checking the truck availability, can be executed arbitrarily often without any harm. Thus, this is an *idempotent* activity. We assume that the workflow designer defines a *cardinality* $\theta(a)$ for every activity a that specifies how many times the activity can be executed maximally. We consider the cardinality to be part of the workflow specification because it can be specified within LTL. Returning to our previous example, we specify the cardinality of the non-idempotent activities a_6 and a_7 to be $\theta(a_6) = 1$ and $\theta(a_7) = 1$, whereas all other activities are idempotent.

The workflow engines require a sequence of activities as input for executing the workflow. This means that a sequence of activities that does not violate the workflow specification needs to be generated and send to the workflow engine. For example, $[a_1, a_2, a_3, a_4, a_5, a_6, a_7]$ is a valid sequence of the workflow specification of Figure 9.2.

To execute the activity sequence, the engine instantiates the sequence, which initializes the internal state of the workflow. Like in Part A, the internal state is a set of variables needed for the execution. To execute an activity, the workflow engine invokes a service that implements the functionality required by that activity. The service can be either available locally on the device that is running the workflow engine or offered by a service provider. For example, when executing activity a_4, the workflow engine calls an image analysis and address validation service. This service returns if the address is valid or not.

We assume that the constraints that enforce an activity execution order, specify the data flow. Hence, the dashed arrows also define the data flow in our example of Figure 9.2. For example, the output internal state produced by activity a_2 (i.e., the variables of the internal state that the execution of activity a_2 changes) is

used as input for activity a_7. However, no other activity uses the output state of activity a_2. Hence, the execution of activity a_2 is completely decoupled from activities a_1, a_3, a_4, a_5, and a_6.

We assume that each workflow engine is using logging, which is typical for workflow engines [BDH+12]. This allows the engine to continue the workflow execution after a crash failure. However, without using replication, this does not solve availability concerns because the workflow executions of that engine still will not make progress while the engine is failed.

The goal is to ensure the availability of the workflow execution in the presence of failures. For instance, in the drone delivery scenario, the workflow that needs to be executed for every picked up parcel should be finished before reaching the post office such that the parcels can be dropped directly to their allocated trucks and the shipment of the parcels proceeds without delays.

10. Approach Overview

THE basic idea for ensuring availability is same as in Part A. We replicate the workflow execution, where each workflow engine participates in the replicated execution. Hence, we call the workflow engines *replicas*.

When assuming that each activity has a compensation handler (as we did in Part A), we can generate one valid activity sequence from the declarative workflow model. This activity sequence has a defined activity execution order and can be specified in an imperative workflow language. Then, we can use our replication schemes from Part A to ensure the availability during the workflow execution. Moreover, we could even decide the next activity to execute at run-time (after each activity execution) and announce the decided activity as part of the update messages.

However, declarative workflows typically do not consider activities to have compensation handlers, e.g., Declare [Pes08]. This might be due to the ability to specify these as part of the workflow, e.g., by specifying that an activity might only be executed a second time if another activity (that realizes the compensation) is executed before. As a consequence, a non-idempotent activity is similar to a non-compensable activity of Part A because we might execute it only once and compensation is not available (cf. Section 5.3.3). When assuming non-compensable activities, our replication schemes only improve availability when assuming that a middleware makes the replication transparent such that every replica might receive the reply from the already called service (cf. Section 5.3.3). Even though this allows to tolerate replica crash failures, it requires a great overhead because then we need to perform a consensus before the execution of each activity to agree on the called service, an UIID, and the workflow execution that is allowed to call the service.

So far, we, however, disregarded that declarative workflows are more flexible than imperative workflows in the sense that the execution order of the activities

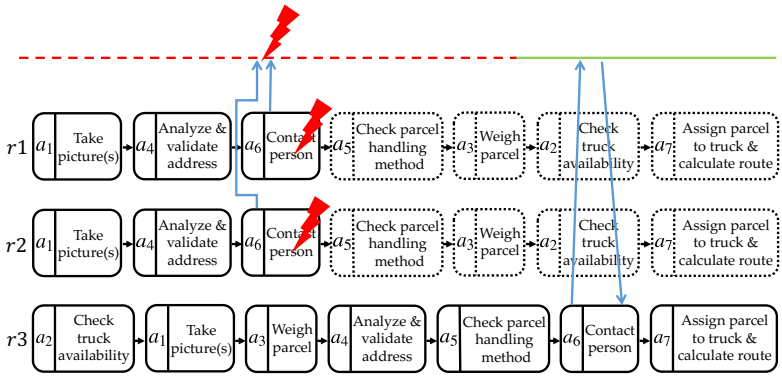


Figure 10.1.: Showing the impact of the temporal failure of the service that activity a_6 calls on the execution of three replicas.

can be changed without violating the workflow specification. When executing a differently ordered activity sequence on each replica, we can increase availability without requiring the synchronization overhead that we described above. In order to exploit this flexibility, we, however, need to make an additional assumption on the workflow. In the following, we will first describe how reordering improves the availability before describing why we need to make this additional assumption.

We will show the increased availability based on the drone delivery scenario in the following. Figure 10.1 depicts the replicated execution of our drone example workflow, where the replicas $r1$ and $r2$ execute the same activity sequence – $[a_1, a_4, a_6, a_5, a_3, a_2, a_7]$ – while $r3$ executes a reordered activity sequence – $[a_2, a_1, a_3, a_4, a_5, a_6, a_7]$. The replicas $r1$ and $r2$ both cannot continue the execution upon executing activity a_6 because the service that activity a_6 has to access is unavailable, e.g., due to the smartphone of the parcel receiver currently having no connectivity. To overcome this limitation, we use reordered activity sequences on the participating replicas, where every used activity sequence has to conform to the workflow specification. In our example, the activity sequence of $r3$ is not delayed because it executes a_6 at a different point in time, where the service is available again. In other words, the different activity sequences reduce the impact of transient service failures.

If the service is, however, permanently unavailable, all replicas will be unable to finish the workflow execution. Permanent failures of services can be masked by exploiting the possibility of defining alternative activities in the workflow specification. Consider that activity a_4 in Figure 10.1 has the alternative to execute activities a_8 and a_9 instead, such that $\neg a_9 \, \mathcal{U} \, a_8$. Instead of using a service that directly analyzes and validates the address (as required by activity a_4), a_8 analyzes the picture and a_9 validates the address. Thus, these activities rely on different services. If the service required for the execution of activity a_4 permanently fails, another replica executing the activity sequence $[a_2, a_1, a_3, a_8, a_9, a_5, a_6, a_7]$ can still execute. In conclusion, the structure of the activity sequences on the different replicas has a significant influence on the availability of a workflow execution. In the subsequent chapter, we address the problem of efficiently creating k structurally different activity sequences, which ensure high availability during execution.

In the presence of cardinality constrained activities, such as non-idempotent activities, it is necessary to synchronize the replicas to ensure that the respective activity is only executed as often as allowed by its cardinality constraint. This, however, is a problem because that means if a replica, for example, executes activity a_6, no other replica can re-execute the activity. Other replicas can, then, only continue their workflow execution if they reuse the result produced by the single execution of a_6. However, the outcome of a_6 is dependent on the input meaning that if another replica reuses the result of a_6, it has to be ensured that it would have provided the same input to a_6. However, since the different replicas might execute different activities before the execution of a_6, the internal state might differ.

We can overcome this limitation with the following assumption: every activity a for which $\theta(a) > 1$ is deterministic. In order to make that assumption applicable to any declarative workflow in general, each activity that is non-deterministic is considered and handled as a non-idempotent activity. With this assumption, any execution of an activity, which is allowed to be executed multiple times, will produce the same output for the same input.[1] Additionally, remember that all activities which have a causal relationship also have a constraint defining their

[1] Note that this requires that also the called service returns the same reply when sending identical requests.

execution order and, thereby, the data flow (cf. Chapter 9). Now, consider that a workflow consists solely of idempotent (and, thus, deterministic) activities. Then, any activity sequence that does not violate the workflow specification (i.e., the ordering constraints) will generate identical results.

However, a workflow usually includes non-idempotent/non-deterministic activities. These have to be executed by only one replica for ensuring the correctness. The result of this activity execution is then shared with all other replicas such that these can reuse the result of that execution. As discussed above, we can, however, only reuse the result if the respective replica would have used the same input internal state for executing the non-idempotent activity.

The first non-idempotent (and possibly non-deterministic) activity has the same input internal state regardless on which replica and in which order the activities are executed[2] because all preceding activities are deterministic. Hence, the other replicas can reuse the shared result. Basically, this reusing of the result makes any non-deterministic activity a deterministic activity. By induction, all replicas will produce the same result for the complete workflow execution no matter which activity execution order is used.

For imperative workflow languages, we defined Single-Execution-Equivalence, which specifies when a replicated workflow execution is equivalent to a non-replicated execution, where any replicated execution that fulfills Single-Execution-Equivalence is correct (cf. Chapter 4). We can still apply the definition, which however, is fulfilled rather obviously under our current assumptions.

Each activity sequence executed by a replica is allowed by the workflow specification. Because the data flow is expressed as part of the constraints, each write activity that causally succeeds another write must only be executed after that activity. Every activity sequence of that workflow will enforce this order. The causally succeeding write activity might only be executed if the replica received a result of the preceding write or the replica executed the first write itself. Hence, the writes are carried out in the same partial order as some non-replicated execution. By induction, this holds for all causally related write activities. Any write activity has the same input internal state as some non-

[2] Of course, this is only true if the workflow specification is not violated by the execution order of the activities.

replicated execution because all causally preceding activities are deterministic (or behave deterministically through the reusing of results).

As a consequence, our replication scheme for declarative workflow executions only has to fulfill the following conditions to ensure Single-Execution-Equivalence: (1) any replica might only execute activity sequences that are allowed by the workflow specification and (2) every activity might only be executed as often as allowed by its cardinality constraint.

Since we already need a mechanism for sharing results of non-idempotent activities, we can reuse the mechanism for also sharing the results of all other activities. These activities are required to be deterministic and, thus, the re-execution will produce the same result. However, the replicas will execute the activities at different points in time because each replica has a reordered activity sequence. If a replica already received a result of an idempotent activity, it can reuse the result instead of re-executing the activity speeding up the workflow execution. If the replica did not receive the result yet, the replica re-executes the activity and produces the result again.

In the following chapters, we will present a replication scheme for declarative workflows, which consists of two parts. In Chapter 11, we present algorithms for generating activity sequences that comply to the workflow specification. In Chapter 12, we present a protocol that coordinates the replicated execution of these activity sequences such that every cardinality constrained activity is only executed as often as allowed by the constraint.

11. Generating Activity Sequences

So far, we motivated that using structurally different activity sequences increases the availability of a replicated workflow execution and showed the challenges that arise with it. In this chapter, we present a metric that rates a set of activity sequences depending on their structure to predict the availability during a replicated execution. Then, we show how to generate the activity sequences from a workflow specification. Afterwards, we analyze the complexity of the generation process and present more efficient techniques.

11.1. Availability Metric

Our goal is to select a set of activity sequences S that provides high availability when executing the activity sequences concurrently on different replicas. We first define a metric that rates a sequence set according to the expected availability during the concurrent execution of the sequences $s \in S$. As explained in Chapter 10, a sequence set S provides higher availability during execution, the more the sequences $s \in S$ structurally differ. Therefore, the proposed availability metric rates sequence sets according to the requirements that

1. the time offset between two executions of one activity in different sequences should be as big as possible,
2. alternative activities should be used as much as possible, and
3. sequences with few activities should be preferred.

The first requirement decreases the impact of transient failures of a service that has to be accessed by one activity. The second requirement ensures that alternate

activities are used whenever possible to reduce the dependency on a specific service accessed by one activity. Finally, the third requirement prefers sequences with few activities because every activity might possibly incur a failure.

In the following, we first define the availability rating of a set of two activity sequences s_1 and s_2 denoted as $D(s_1, s_2)$. Let c_{max} be the number of activities of the longest possible activity sequence that complies with the workflow specification. Likewise, let c_{min} be the number of activities of the sequence with the fewest activities.

To account for the above mentioned first requirement, we calculate the offset between the two executions of one activity a in the two sequences s_1 and s_2. The offset of an activity a is denoted as $O(a, s_1, s_2)$ and its calculation is based on the position of the activity a in sequence s_1 and the position in sequence s_2.

To incorporate the second requirement, we need to consider the cases where an activity a might not occur in one of the activity sequences because of alternatives. For these activities, $O(a, s_1, s_2)$ returns the maximum possible offset c_{max}.

To calculate the rating, we add the offsets of all activities occurring in the two sequences. This gives a metric on how much these two sequences differ. In general, the offset based rating increases with the number of activities in a sequence. However, if a sequence has more activities, more failures can occur. To fulfill the third requirement, the rating of sequences with many activities should be decreased. Let function $L(s_1)$ calculate how many activities the sequence s_1 has more than c_{min}, i.e., $L(s_1) = |s_1| - c_{min}$. The rating is reduced by c_{max} for every additional activity, i.e., it is reduced by $L(s_1) \cdot c_{max}$. Thus, the availability rating $D(s_1, s_2)$ is calculated by adding the offset of all activities in the two sequences and, then, reducing the rating for the additional activities (cf. Equation 11.1).

$$D(s_1, s_2) = \left(\sum_{a \in s_1 \cup s_2} O(a, s_1, s_2) \right) - (L(s_1) + L(s_2)) \cdot c_{max} \tag{11.1}$$

Now, we generalize Equation 11.1 to rate a sequence set S of k sequences. We simply sum up the availability rating of all pairs of sequences (Equation 11.2).

$$F_{AR}(S) = \sum_{(s_i \in S)} \sum_{(s_j \in S \setminus s_i)} D(s_i, s_j) \tag{11.2}$$

Note that we defined the offsets based on the positions of the activities in the activity sequences. This means that the availability metric is only accurately representing the time offset between activities if all activities have the same execution time. In reality, the activities may have varying execution times. However, in our evaluations, we show that even if we vary activity execution times, the sequence availability is not effected (cf. Chapter 13).

11.2. Generation and Selection of Activity Sequences

In this section, we first present a method to generate differently structured activity sequences from a workflow specification. Afterwards, we select the highest rated set of sequences with respect to the availability metric.

To generate the sequences, we use model checking methods to translate the LTL formula that represents the workflow specification into an automaton by expanding the LTL formula step-by-step. The automaton provides the information to deduce all activity sequences that conform to the workflow specification. First, the LTL formula is associated with the entry node of the automaton (cf. Figure 11.1 node X). Then, the automaton node is expanded according to a set of rules [Cou99, DLP04] to simplify the formula into three parts (cf. Figure 11.1):

I) Activities that need to be executed directly,

II) promises ($\mathcal{P}$) that need to be fulfilled eventually, and

III) a formula that defines what needs to hold next ($\circ$), i.e., what needs to hold after the activities of part I) have been executed [DLP04].

Part I is associated with the label of the edge to the next node, i.e., the activities need to be executed to reach the next node. In our example of Figure 11.1 the activity a_1 needs to be executed to reach automaton node Y from node X. Part II – the promises – is used to check if the current activity sequence fulfills the workflow specification. If there are promises, then the workflow specification is not yet fulfilled, i.e., the automaton node after this transition needs to be expanded further. This information is stored in the edge between the respective

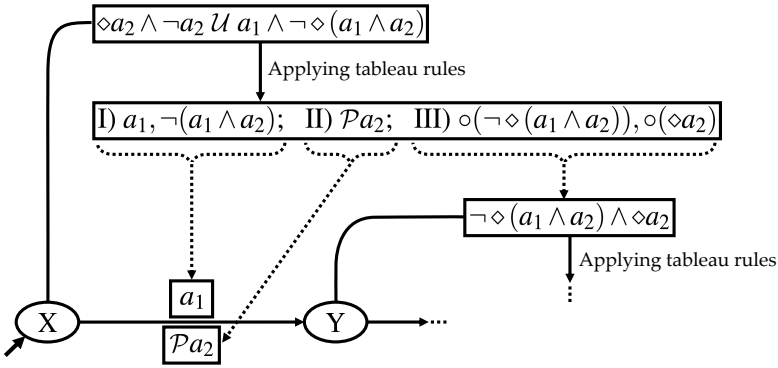


Figure 11.1.: One step of an exemplary expansion of the LTL formula $\diamond a_2 \wedge \neg a_2 \, \mathcal{U} \, a_1 \wedge \neg \diamond (a_1 \wedge a_2)$ according to [DLP04]

nodes. In Figure 11.1, there is the promise to eventually execute a_2, which means that the next automaton node Y needs to be expanded further. Finally, part III is attached to the next node of the automaton because it specifies what has to hold after the activities of the part I have been executed. In Figure 11.1, the formula of part III is associated with node Y. Applying the tableau techniques to this new automaton node, i.e., to the associated LTL formula, will expand this node to the following ones. This procedure of expansion repeats until the formula is fully expanded to the complete automaton. From the labels of the edges, all n activity sequences can be constructed by strategically going through the transitions of the automaton. This solves the generation problem.

Out of all n generated activity sequences, we need to select the set of size k that will achieve the highest availability during a concurrent execution of the selected sequences. We solve this by rating all activity sequences sets of size k using the availability metric and select the one with the highest rating.

Complexity

The proposed generation technique has a high complexity. In fact, LTL satisfiability is a PSPACE-complete problem [SC85] and, therefore, finding satisfying traces, i.e., activity sequences that conform to the workflow specification, is a

PSPACE-complete problem. As a consequence, the generation process might produce a large number of activity sequences.

The generated activity sequences are the input of the selection problem, which is to find the best rated set of size k from n activity sequences. The selection problem can be mapped to the *maximum edge-weighted clique problem* (MEWCP), which is NP-hard [AGKW07]. The MEWCP is a generalization of the maximum clique problem [AGKW07]: Given a graph $G = (V, E, f)$, where f is a function that assigns a weight to every edge, i.e., $f : E \to \mathbb{R}$, find the clique in which the sum of all of its edges is maximal and the number of nodes of the clique $\leq k | k \in \mathbb{N}$.

Lemma 1. *Selecting the best rated set of k activity sequences out of n activity sequences can be mapped to the maximum edge-weighted clique problem.*

Proof sketch. Given all n activity sequences, compute the availability ratings for every pair of activity sequences and write them into a 2D-matrix. Using this as an adjacency matrix, a complete graph with n vertices (representing the activity sequences) can be created, where the availability ratings are the edge weights between the vertices. The problem of finding the best rated set of size k is equal to solving the MEWCP with a clique of size k in the graph. □

11.3. Heuristics for the Generation and Selection Problems

In the following, we propose two techniques to tackle the high run-time and memory complexity of the presented generation and selection approach. Our first technique significantly reduces the run-time of the selection process. The second technique omits the generation of all activity sequences and, thereby, reduces the memory consumption of the generation problem.

11.3.1. Improving the Performance of the Selection Problem

As already described above, the selection problem is equal to solving the MEWCP. To solve the MEWCP, we use a binary quadratic programming formulation of

the problem [AGKW07]: Given a symmetric $n \times n$-matrix Q composed of all availability ratings (as presented above), find the vector $v \in \{0,1\}^n$, such that the outcome of Equation 11.3 is maximal, where the sum of all elements of vector v must be k, i.e., $|v| = k$.

$$f(v) = \frac{1}{2} v^T Q v \tag{11.3}$$

If the element i of vector v is equal to 1, the sequence of column i in the matrix is part of the set. If it is 0, it is not. To maximize the outcome of Equation 11.3, we apply simulated annealing, as it is an established strategy to produce almost optimal results within a short amount of time for binary quadratic programming problems [KN01, MF02].

11.3.2. Improving the Performance of the Generation Problem

Our evaluations show that simulated annealing drastically reduces the run-time of the selection problem, while producing almost optimal activity sequences with respect to availability (cf. Chapter 13). However, if the workflow specification has many activities, the generation of all activity sequences easily becomes unfeasible due to the run-time and the memory consumption. Thus, for formulas that lead to huge automata, the full generation has to be omitted and a different strategy has to be applied. To tackle this problem, we introduce the *availability prediction metric* (APM). The idea is to prune the automaton during expansion by only expanding those paths of the automaton that might lead to activity sequences that are part of sets with high availability ratings. The activity sequences that are created during the expansion and still have to fulfill promises (cf. Section 11.2), i.e., that do not satisfy the workflow specification yet, are called *intermediate activity sequences*.

In the following, we describe how to perform pruning during the expansion of the automaton. We expand all automaton nodes that currently need further expansion. This step is referred to as an *iteration* (cf. Figure 11.2 and Figure 11.3). After each iteration, all newly created (intermediate) activity sequences generated by the iteration are rated by the availability prediction metric. The difference between the availability prediction metric and the availability metric is twofold.

Firstly, we do not know the activity sequence with the fewest and most activities during the expansion of the automaton. Therefore, to determine the ratings of the (intermediate) activity sequences, the availability prediction metric estimates c_{max} and c_{min} based on the currently available satisfying and intermediate activity sequences. Secondly, to omit the selection problem, the availability prediction metric only calculates the rating for every pair of sequences (i.e., only sets of two sequences) and ranks the current (intermediate) activity sequences according to the pairwise ratings. Then, we select the f_{Flows} best rated sequences, where $f_{Flows} \geq k | f_{Flows} \in \mathbb{N}$. For instance, in Figure 11.2, the second iteration leads to eight intermediate activity sequences. Figure 11.3 depicts that all but three of the intermediate sequences are pruned before the third iteration. Only the paths that produce the selected intermediate sequences are expanded further.

Note that it is possible that a branch of the automaton is fully expanded but does not include an activity sequence that satisfies the workflow specification. These paths are called *dead ends*. If a selected activity sequence leads to a dead end through expansion, the dead end branch is replaced by another branch. In order to do so, the expansion algorithm goes backwards from the dead end to find a node that was not expanded due to pruning. This node, then, will be expanded compensating for the dead end.

In our evaluation, we show that the pruning strategy significantly improves the run-time, however, compared to simulated annealing, it produces results that have a lower but reasonable availability during execution (cf. Chapter 13).

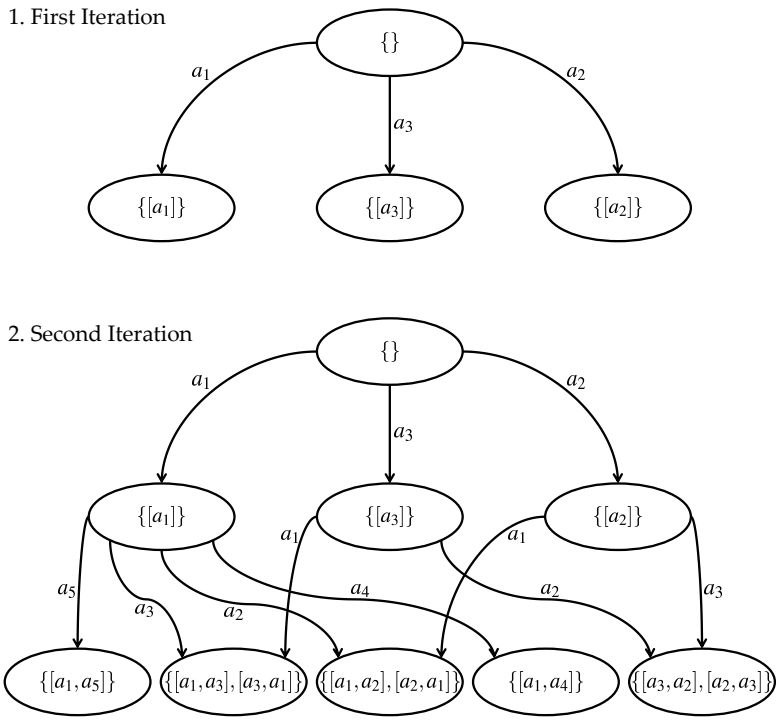
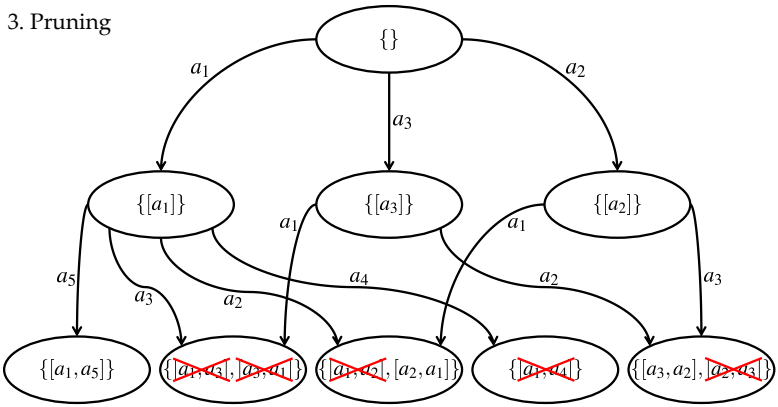


Figure 11.2.: This is an example that shows how pruning is applied while expanding the workflow specification of Figure 9.2. After each iteration, three activity sequences are selected, i.e., $f_{Flows} = 3$. The example is continued in Figure 11.3.



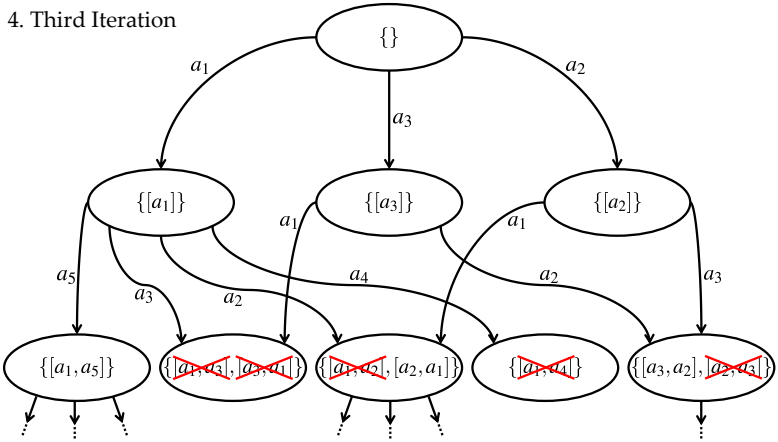


Figure 11.3.: Continuation of the example from Figure 11.2.

12. Coordination Protocol

In the previous chapter, we presented methods to efficiently generate and select a set of activity sequences that provide high availability when being executed concurrently on different replicas. However, how the concurrent execution of these activity sequences needs to be coordinated such that the execution preserves the workflow specification has not been described so far. In specific, each activity shall only be executed as often as allowed by the cardinality constraint of this activity.

For preserving the cardinality constraints, we elect a coordinator that is responsible for managing the activity executions on the different replicas accordingly. Thus, when a workflow client application sends a request for executing a declarative workflow, the replicas elect one replica to be coordinator. If the elected coordinator fails, a new coordinator is elected out of the remaining replicas. We use majority election [vEVS02] meaning that (1) at any point in time there is at most one valid coordinator and (2) out of a majority always one replica knows the valid (i.e., most recently elected) coordinator. For being able to identify the most recent coordinator, we reuse the concept of views [LC12], where each view has at most one coordinator elected (cf. Chapter 5).

The coordinator basically is responsible for permitting replicas to execute activities. It only permits a replica the execution of the activity if an additional execution of that activity does not violate the cardinality constraint of that activity. However, the information on which replicas are allowed to execute the activities must not be lost through the failure of the coordinator. Thus, the information needs to be replicated on the other replicas.

When a replica has executed an activity, the result of that activity execution is provided to the other replicas. Thus, when a replica reaches the point in its activity sequence where it would also need to execute the activity, it can simply reuse the result that it received instead of executing the activity itself.

Consequently, the execution speeds up since any activity of which the result is already available can be skipped.

The execution of a workflow terminates when a replica has executed the last activity of its activity sequence. Since we coordinate the replicated execution such that no activity needs to be compensated (which is anyway not possible according to our declarative workflow model described in Chapter 9), the termination is rather simple. A finished replica sends the result of its workflow execution to the coordinator. The coordinator forwards the first result that it receives to the workflow client application that initiated this workflow execution. This prevents that the workflow client application receives duplicates. Afterwards, the coordinator interacts with all replicas to forget the workflow execution. Note that this forget phase requires all replicas to be available. However, this is no problem in terms of availability since this phase is totally decoupled from the workflow client application.

12.1. Data Structures

A replica r that participates in the replicated execution keeps a workflow record in the volatile memory. The workflow record contains the following data:

- eID_r: the unique execution identifier of the workflow execution.
- W_r: the declarative workflow specification.
- v_r: the current view.
- $next_r$: the next activity to execute within the activity sequence.
- σ_r: internal state of the workflow execution.
- E_r: the activity execution table. Each activity a of the workflow specification has an entry $(a, R_a) \in E_r$ that specifies the set of replicas R_a that have been permitted to execute a so far. For example, the entry $(a_1, \{r1\})$ specifies that the replica $r1$ is permitted to execute the activity a_1. Initially, all sets R_a of the entries $(a, R_a) \in E_r$ are empty because no replica has the permission to execute any activity yet. We define the norm of E to be

$|E| = \sum_{(a,R_a)\in E} |R_a|$, where the norm basically counts how many activity execution permissions were given to replicas so far.

- Ω_r: the set that contains all activity execution results that are received from other replicas. Each item (a, ω) in the set comprises the activity a that was executed as well as the result ω of that activity execution.

Note that the execution identifier eID and the workflow specification W are identical on all replicas. All other data might be different on each replica. The activity sequence s_r that a replica executes is saved on stable storage together with the execution identifier eID, i.e., (eID, s). This prevents that the sequence is lost through crash failures. Since the generation of the sequences is time intensive, it is more efficient to save the sequence on stable storage than to regenerate it after a crash failure (as described in Chapter 11).

12.2. Normal Operation

The execution is initiated when a workflow client application sends an execution request to the replicas. In specific, the execution request message EXEC(eID, W) is sent to all replicas and contains the unique workflow execution identifier and the workflow specification.

The replicas elect a coordinator using a majority election, e.g., as described in Section 5.1.1.3. The coordinator then generates the activity sequences as described in Chapter 11. Afterwards, the coordinator sends one activity sequence s to each replica with an ACT_SEQ(eID, s) message.

Upon receiving an ACT_SEQ message, a replica starts the execution of the received activity sequence (cf. Algorithm 16 line 1-10). Here, we differentiate between idempotent and cardinality constraint activities. If the activity that the replica strives to execute next is idempotent, the replica can directly start the execution. If the activity is cardinality constraint, the replica has to request the permission to execute the activity from the coordinator. The replica sends an ACT_REQ message to the coordinator, which contains the activity that the replica strives to execute. Figure 12.1 depicts an example, where the replica $r1$ requests the permission to execute the cardinality constrained activity a_1.

Algorithm 16: Executing an activity sequence

```
   // Replica - normal operation
1  while next_r ≠ null do
2      if ∃(a, ω_a) ∈ Ω_r, where a = next_r then
3          apply ω_a to σ_r;
4          next_r := succ(next_r, s_r);
5      else if next_r is idempotent then
6          ω := execute(next_r);
7          apply ω to σ_r;
8          next_r := succ(next_R, s_r);
9      else
10         send ACT_REQ(eID, next_r) to coordinator;
11 upon receive ACT_PERMIT(eID, a_x) from coordinator do
12     if a_x = next_r then
13         ω := execute(next_r);
14         apply ω to σ_r;
15         async send RESULT(eID, next_r, ω) to all replicas;
16         next_r := succ(next_r, s_r);
17 upon receive RESULT(eID, a_x, ω_x) from x do
18     add (a_x, ω_x) to Ω_r;
19     send ACK_RESULT(eID, a_x) to x;
   // Coordination - normal operation
20 upon receive ACT_REQ(eID, a_x) from x do
21     if (a, R_a) ∈ E_r, where a = a_x AND (|R_a| < θ(a) OR x ∈ R_a) then
22         add x to R_a;
23         send TABLE_UPDATE(eID, E_r, v_r) to all replicas;
24         wait for majority of replicas to send TABLE_ACK for E_r, v_x;
25         send ACT_PERMIT(eID, a) to x;
26 upon receive TABLE_UPDATE(eID, E_x, v_x) from x do
27     if v_x ≥ v_r AND |E_x| ≥ |E_r| then
28         E_r := E_x;
29         send TABLE_ACK(eID, E_x, v_x) to x;
```

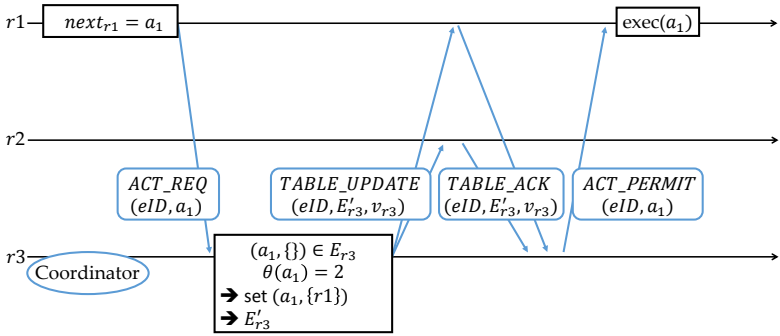


Figure 12.1.: Example of the coordination protocol, where $r1$ receives the permission to execute the cardinality constrained activity a_1.

Upon receiving an ACT_REQ message, the coordinator checks whether the activity can be executed without violating the cardinality constraints (cf. Algorithm 16 line 20-25). If the constraint is not violated, the coordinator adds the requestor to the replicas that are permitted to execute the activity and broadcasts the new activity execution table to all replicas by sending a TABLE_UPDATE message, which contains the new table and the current view of the coordinator. A TABLE_UPDATE message receiver (cf. Algorithm 16 line 26-29) only accepts the update if the update was sent by the current coordinator, which the receiver checks based on the view number. Only after a majority of replicas has acknowledged that they have received the new table with a TABLE_ACK message, it is ensured that the permission is not lost through failures. Requiring the coordinator to wait for the majority also prevents an old coordinator (which, for example, has been partitioned) from giving activity execution permissions because out of majority of replicas at least one replica will know about the newer view (and current coordinator). Thus, an old coordinator can never receive TABLE_ACK messages from a majority. In contrast, the current primary sends an ACT_PERMIT to the requestor, which includes the activity for which the coordinator gives the execution permission, after the current coordinator has received the TABLE_ACK message from a majority of replicas.

Upon receiving the ACT_PERMIT message (cf. Algorithm 16 line 11-16), the replica executes the activity for which it received the execution permission.

After executing the activity, the replica propagates the result of the activity execution with a RESULT message. Note that the result is sent asynchronously to the replicas, meaning that the workflow execution continues while the RESULT messages are being sent. However, the replica still repeats each RESULT message until the receiver has acknowledged the reception (cf. Algorithm 16 line 17-19). This ensures that for a non-idempotent activity, i.e., an activity which cannot be executed arbitrarily often, the result is made available to all replicas.

12.3. Election

The replicas monitor the coordinator by means of a heartbeat mechanism. Upon detecting the coordinator as failed, the remaining replicas start a new view, where they elect a new coordinator. As an election procedure was already described extensively in Section 5.1.1.3, we omit the description here and only focus on the aspects that are unique to our coordination protocol.

As the activity execution table is only guaranteed to be replicated on a majority of replicas, a newly elected coordinator might not have the most recent activity execution table. However, in order to preserve the cardinality constraints, the new coordinator must use the most recent activity execution table for checking new activity execution requests. Thus, all participants of an election include their activity execution table in the vote messages. The new coordinator will select the activity execution table that has permitted the most activity executions so far, i.e., the table E, where $|E|$ is maximal. Since a table is replicated on a majority before the according activity execution is permitted and a new coordinator needs a vote from a majority of replicas, it is ensured that no activity execution permission is lost through a failover.

12.4. Termination

When a replica has executed the last activity of its activity sequence, the workflow execution is finished. The replica sends a FINISHED message to the coordinator, where the message includes the result of the workflow execution. The coordinator forwards the result to the workflow client application. The coordinator

only forwards the first result that it receives to ensure that the workflow client application does not receive duplicates.

Basically, the workflow execution is now finished. However, some replicas might still be executing their activity sequence. To stop these replicas and to forget the workflow execution (i.e., remove all data from the volatile memory), the coordinator starts a 2PC protocol. In the first phase, the coordinator sends PREPARE(eID) messages. Upon receiving a PREPARE message, the replica stops any ongoing executions of eID and replies with an PREPARE_OK(eID) message. When the coordinator has received a PREPARE_OK from all replicas, it sends a FORGET(eID) message to all replicas and forgets the workflow execution. Upon receiving a FORGET message, a replica also forgets the execution, i.e., it deletes all data from volatile memory and also the activity sequence s from stable storage.

12.5. Recovery

When a replica recovers from a crash failure, all data of the workflow execution that was stored in volatile memory is lost. The data saved on stable storage is available, i.e., the execution identifier eID and the activity sequence of that replica. However, since we assume logging to be used by each workflow engine (cf. Chapter 9), the workflow execution could continue after recovery. Before continuing the execution, the replica has to receive data for the volatile memory from the other replicas. Thus, the recovering replica sends a RECOV_REQ message to all replicas, which contains the execution identifier.

Upon receiving a RECOV_REQ message, the replicas reply by sending a RECOV_REPLY message, which contains, the workflow specification W, the current view v, the activity execution table E, and the results of the activity executions Ω. Upon receiving a RECOV_REPLY message from a majority of replicas, it saves the workflow specification and the highest view number it receives. Moreover, it saves the most recent activity execution table, i.e., the table E, where $|E|$ is maximal, and the Ω that has saved the most activity execution results. Afterwards, the recovering replica returns to normal operation.

13. Evaluations

In this chapter, we evaluate our replication scheme for declarative workflows. We evaluate the speed up in terms of execution time when using the reordered activity sequences on the different replicas. In specific, we investigate the impact of cardinality constraints on the speed up. Afterwards, we evaluate the techniques for generating activity sequences from the workflow specification with respect to availability and run-time of the generation. Here, we compare the simulated annealing approach (SA) and the pruning-based availability prediction metric (APM) to finding the optimal solution. Finally, we evaluate the accuracy of our availability metric by comparing the predicted values to the actual availability when executing the activity sequences in the presence of failures using our coordination protocol.

13.1. Implementation

To evaluate our activity sequence generation techniques, we use the SPOT library [DLP04, DL17] for translating the LTL-based workflow specifications into automata. From the automata, we derive all activity sequences that fulfill the workflow specification. For evaluating our availability prediction metric, we have integrated the availability prediction metric in SPOT.

The coordination protocol is implemented in the peer to peer simulator, PeerSim[1], where each peer represents a replica that executes an individual activity sequence. Any type of failure delays the execution on a replica until the recovery from the failure. Since we assume logging (cf. Chapter 9), this leads to the delay of the execution on this replica. Thus, we model all failures as activity failures, where each activity has a failure probability of 5%. For the evaluation,

[1] http://peersim.sourceforge.net/

we generate activity failure traces, to expose the replicas, generated by the different techniques, to the same failure patterns. We also repeated all experiments with a varying execution speed of the replicas, where the execution speed was varied by $\pm 10\%$. We discovered that such variances do not influence the average availability.

13.2. Speed Up

In this evaluation, we use 10 replicas that execute activity sequences of the same workflow specification. In Figure 13.1a, we can observe that no replica is idle when all replicas execute identical activity sequences that contain only idempotent activities. All activities can be executed arbitrarily often without violating the workflow specification. Thus, all replicas execute all activities, where all activities are executed in the same order meaning that no activity execution result can be reused by other replicas. Thus, the execution time is same as it would be when using a non-replicated execution of this activity sequence. In other words, there is no speed up of the execution in this scenario.

When using identical activity sequences with cardinality constrained activities as shown in Figure 13.1c, the replicas are mostly idle because only one replica is allowed to execute the workflow – basically realizing a passive replication scheme. Moreover, since there is an execution time overhead for receiving the allowance to execute an activity, the workflow execution is even slowed down compared to a workflow consisting only of idempotent activities.

When using reordered activity sequences that consist only of idempotent activities (cf. Figure 13.1b), the replicas are never idle similar to the identical activity sequences of Figure 13.1a. However, here the replicas can reuse the activity execution results of the other replicas. The speed up that is achieved through reusing is significant. The execution of a workflow with 5 activities is reduced from 18 cycles to 6. For a workflow with 25 activities, the execution time is even reduced from 80 cycles to 16. The reduction is of course related to how freely the activities might be reordered and how many replica there are. With 5 activities, we cannot use the parallelism fully, while with 25 activities the 10 replicas can often execute different activities. Still due to the workflow

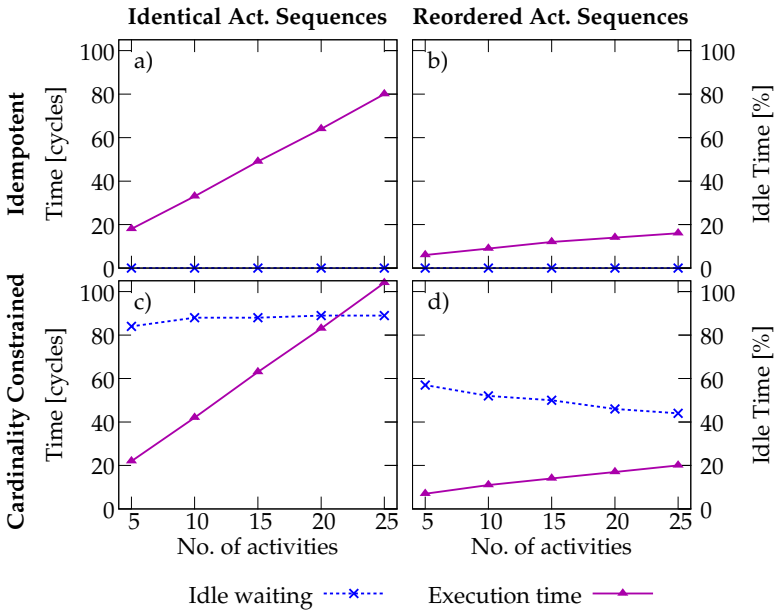


Figure 13.1.: Comparison of using reordered versus identical activity sequences running on 10 replicas. The activity sequences contain either only idempotent or only cardinality constrained activities. Here, we plot the number of cycles required for executing the activity sequences against the number of activities contained in that activity sequence. [Lower is better]

specification, we do not achieve a reduction down to 8 cycles, which would be the optimal case with 10 replicas.

When having a workflow that only contains cardinality constrained activities, reordering still achieves a tremendous speed up (cf. Figure 13.1d). However, due the overhead of receiving a permission for being allowed to execute an activity, the execution time is a bit increased in comparison to workflows that only contain idempotent activities. For example, workflows with 5 activities take 7 cycles, while 25 activities take 20 cycles. We also can observe that the number of replicas that are idle is drastically reduced when comparing the reordered activity sequences to the identical activity sequences of Figure 13.1c.

In specific, now through the reordering, many replicas can execute the activities such that the workflow is exploiting parallelism. Also, the number of idle replicas is linearly decreasing when the workflows contain more activities because with more activities in a specification, there are typically more allowed activity sequences allowing more parallelization.

In conclusion, we can observe that the speed up that is achieved through reordering is significant. Moreover, it does not matter whether the activities are idempotent or cardinality constrained, reordering always is beneficial in terms of execution time.

13.3. Activity Sequence Generation

We compare the strategies for generating the activity sequences from a workflow specification and selecting a set of sequences for the replicated execution. We use workflow specifications with eight activities by randomly combining LTL constraints [Pes08]. We select activity sequence sets of size $k = 3$.

We compare the following strategies: the optimal solution, the simulated annealing approach (SA), and the availability prediction metric (APM). The strategies are compared based on the availability rating (cf. Equation 11.2) of the selected set of activity sequences and the run-time needed for generating and selecting this set. However, since the rating might heavily deviate between the randomly generated workflow specifications, we normalize the ratings found by simulated annealing and the availability prediction metric by the rating of the optimal solution.[2] Similarly, we normalize the run-time for finding the activity sequence sets by the time needed to find the optimal solution. Note that both for finding the optimal solution and for simulated annealing the complete automaton is expanded from the workflow specification. We depict the shared intermediate step called the full expansion (FE), which the availability prediction metric avoids by pruning the automaton during expansion.

[2]The availability prediction metric estimates c_{min} and c_{max} (cf. Chapter 11). To make the availability rating of the set that the availability prediction metric generated and selected comparable, we use the correct values of c_{min} and c_{max}, determined when generating the optimal solution, for evaluating the selected set.

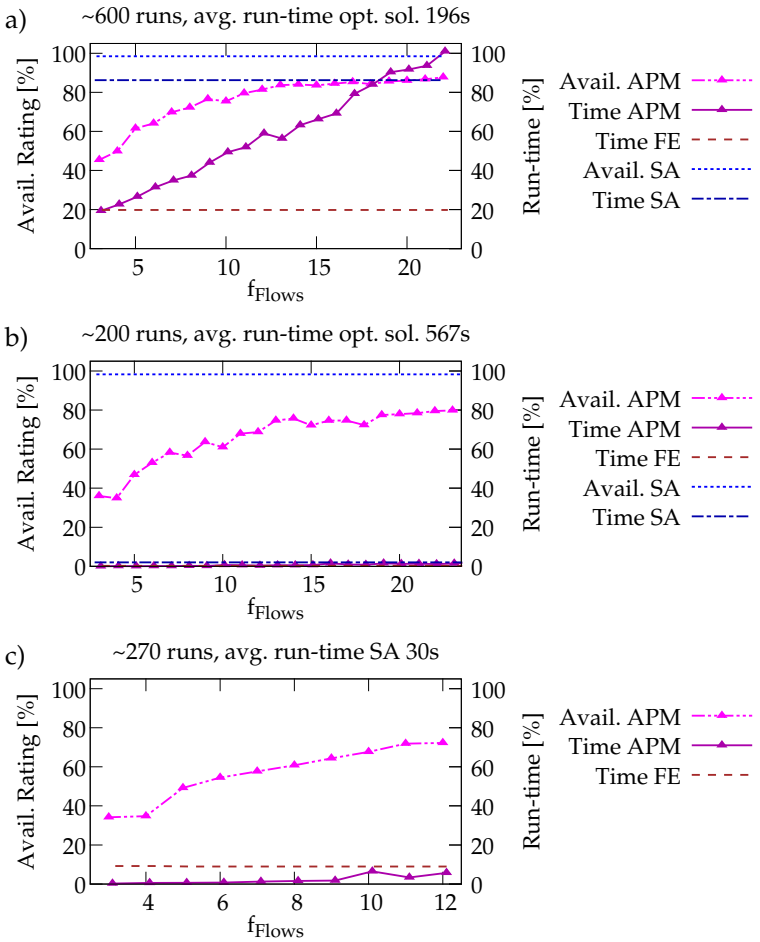


Figure 13.2.: Plotting the availability rating that the produced activity sequences have according to our availability metric (cf. Section 11.1) against f_{Flows}. Additionally, we plot the run-time for producing the activity sequences against f_{Flows}. Figure a shows the results of workflow specifications that produce 3 to 500 activity sequences, Figure b the results of workflow specifications that produce 200 to 500 activity sequences, and Figure c the results of workflow specifications that produce 500 and 2000 activity sequences. [Availability Rating: higher is better, Run-time: lower is better]

The results and the run-time of our availability prediction metric depend on the parameter f_{Flows}. Thus, we depict the results and run-time of the availability prediction metric in dependence of f_{Flows}, where we evaluate the range from $f_{Flows} = k$ up to $f_{Flows} = 22$. In the following, we grouped our measurements by the number of activity sequences produced by the full expansion of the automata.

In Figure 13.2a, we depict workflow specifications resulting in 3 to 500 activity sequences: simulated annealing on average needs only 85 % time compared to finding the optimal solution. The availability prediction metric is very fast for small f_{Flows}, however, always selects sets with a lower availability rating compared to the optimal solution and simulated annealing. For $f_{Flows} < 4$, the availability prediction metric is even a bit faster than only expanding the full automaton (FE) without starting any selection process. In contrast to that, for $f_{Flows} \geq 22$ the availability prediction metric is even slower than generating the optimal solution because of the overhead of deciding the pruning during the automaton expansion. However, we consider workflow specifications that have many constraints on the execution order and, thus, only few valid activity sequences. For example, when there are only 3 allowed sequences, we need to find all of these making a full expansion of the automaton necessary. In this case, pruning only adds overhead during the automaton expansion.

For filtering out these highly restrictive workflow specifications, we only depict specifications that result in 200 to 500 activity sequences in Figure 13.2b. In comparison to Figure 13.2a, simulated annealing is now significantly faster because we have to choose from at least 200 sequences making the speed up rather obvious. Simulated annealing finds an activity sequence set with an availability rating of $\sim 98\,\%$ while taking only about 2 % of the time of finding the optimal solution. The availability prediction metric only generates results with $\sim 38\,\%$ of availability when $f_{Flows} = k$. When, however, increasing f_{Flows} to follow more than 10 branches, the availability reaches between 70 % and 80 % percent of the optimal solution. However, the availability prediction metric needs only a fraction of computation time even when compared to simulated annealing.

Figure 13.2c depicts the results, where the full expansion of the automaton produced between 500 and 2000 activity sequences. Because of the high complexity of the selecting the optimal set (cf. Chapter 11), determining the optimal solution is simply unfeasible. Instead, we only use simulated annealing and the

availability prediction metric for finding a solution. Thus, we normalize the availability and run-time by the time needed for finding a set with simulated annealing. In other words, 100 % of availability is equal to the availability rating of the set found by simulated annealing. Similarly, 100 % of run-time is the time that simulated annealing needed for selecting the activity sequence set. The availability prediction metric again reaches an availability above 70 % for $f_{Flows} > 10$. It, however, is clearly faster than only the full expansion of the automaton without even starting simulated annealing for selecting the set.

In conclusion, we have shown that both simulated annealing and the availability prediction metric realize reasonable trade-offs between the time required for generating and selecting activity sequences and the availability achieved by the selected activity sequences. Moreover, for workflow specifications with more than 10 activities, our availability prediction metric might be the only feasibly strategy in terms of execution time.

13.4. Accuracy of the Availability Rating

So far, we only showed that the set of activity sequences that we generate and select have a good theoretical availability since we used our availability rating (cf. Equation 11.2) for evaluating the generated sets. Now, we will execute the selected activity sequences in the presence of failures for showing that our availability metric correctly predicts the actual availability during execution. The availability is represented by the execution time because a faster execution is congruent to a highly available execution. In other words, when the execution is unavailable, it does not proceed. For this evaluation, we use the activity sequences generated in Section 13.3. We normalized the execution time of these activity sequences sets with the concurrent execution of a set of identical activity sequences, i.e., 100 % of execution time represents the execution with identical activity sequences. As described in Section 13.1, we inject failures based on the failure traces such that the sets, selected by the different approaches, are exposed to the same failures.

Figure 13.3 shows the execution time of the activity sequence sets generated in Section 13.3. In specific, Figure 13.3a shows the execution of the sequence sets from Figure 13.2a, where 3 to 500 activity sequences are generated by the

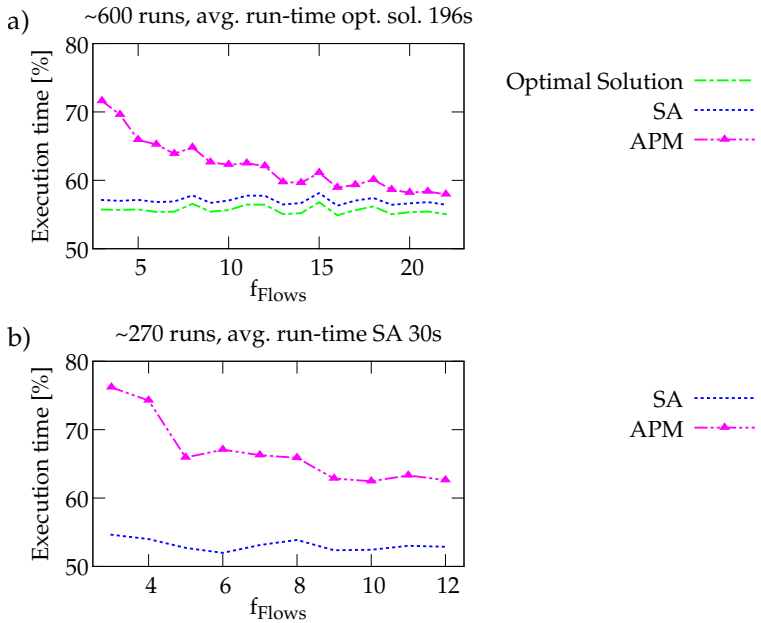


Figure 13.3.: Plotting the execution time against f_{Flows}. Here, we compare the execution time of the optimal solution, the simulated annealing approach (SA), and the availability prediction metric (APM), where all execution times are normalized by the execution time of using identical activity sequences on all replicas. [Lower is better]

full automaton expansion. Figure 13.3b shows the execution of the sets that are selected in Figure 13.2c, where the workflow specifications resulted 500 to 2000 activity sequences.

We can observe that the optimal solution has the highest availability, followed by simulated annealing, the availability prediction metric, and, finally, the set of identical activity sequences (represented by 100 %). Also the execution time decreases when f_{Flows} is increased, which reflects the increased rating of our availability metric. The graph verifies the correctness of our availability rating. The replicated execution of the optimal solution sets take around 55% of execution time compared to identical replicas. Simulated annealing results take around

56% of the time, followed by the availability prediction metric with around 65% ($f_{Flows} = 5$). When increasing f_{Flows}, the availability prediction metric can even reach 60% showing its usefulness for efficiently generating and selecting replicas, accepting a slightly decreased availability. It, however, should only be used with a small or minimal f_{Flows}. If availability is very critical, simulated annealing can be used, which, on the downside, implies a higher generation overhead.

In addition, we generated sets of activity sequences using the availability prediction metric for large workflows, where simulated annealing is too time consuming due to the full generation of the automaton. We generated activity sequences sets of size $k = 5$ for workflows with 15 different activities. The generation of those sequences (with $f_{Flows} = 5$) took an average time of $75s$. The execution took around 83% of the execution time of sets with identical replicas. This shows the benefits of the availability prediction metric for cases where the full generation is no longer practical.

14. Related Work

Service invocation as described in Section 2.4 decouples the availability of a workflow execution from the availability of a specific service. Here, multiple alternative services are called simultaneously or staggered over time [Dob06, SJP06,SF07,LV07,GHL+07,SPJ07,ZL08a,ZL08c,ZL08b,ZL09,TLHL09,SPJ11, MSM15, BTKR15]. The result of the service that replies first is used while all other service executions are stopped or compensated. With our structurally different activity sequences, we can mask transient service failures without requiring the compensation. Moreover, the service invocation techniques cannot tolerate failures of a workflow engine.

Other approaches automatically compose a service orchestration with a desired functionality – described in a declarative languague – from existing web services using QoS values [LPC+11, LHG+16, EKEF16]. This requires the QoS values to be available in order to compose a highly reliable activity sequence. In contrast, we do not require such information. Moreover, when executing such a automatically composed service orchestration without replication, a single failure will still lead to outages. To overcome this problem, some work proposes to automatically re-compose the orchestration at run time when a failure occurs [LPC+11, EKEF16]. However, both the techniques with and without automatic re-composition only work if the workflow engine can detect the failures. When the engine in which the workflow is running fails, the techniques do not provide availability.

Part C.

Conclusion

15. Summary

THE automation of business operations through workflows is of major importance not only to save costs and optimize the underlying processes, but also to enable paradigms that are impossible without automation, such as on-demand cloud resource allocation. Businesses nowadays often have multiple business locations and business partners scattered across the globe. For automating the interactions between the locations and partners, the workflows are required to run in a heterogeneous and distributed environment, where failures occur frequently. Such a failure can delay or stop the workflow executions, which, in turn, might also degrade or stop the business operations.

For ensuring availability in the presence of failures, we presented mechanisms for replicating workflow executions. The key idea is to ensure that the outcome of a replicated execution is identical to a non-replicated execution. We formally defined this property called *Single-Execution-Equivalence*. We presented a majority-based replication scheme that adheres to this definition, which elects one replica as primary that executes the workflow. The other replicas are backups that receive and save execution state updates sent by the primary. In case that the primary fails, the backups elect a new primary that continues the workflow execution using a majority election.

However, since any replication incurs overhead, there are trade-offs to consider. The basic majority-based replication scheme is based on a strict synchronous update mechanism, which keeps the compensation overhead at a minimum. In specific, the primary has to verify after each activity execution that it still is primary. In other words, an old primary executes at most one activity after a new primary was elected. Since the new primary is responsible for the execution of the activity, the old primary has to compensate its activity execution.

In terms of execution time, this strict synchronous update mechanism is very costly because the primary is paused after each activity execution until a majority

of replicas has acknowledged that it is still primary. For this reason, we have developed the relaxed majority-based replication scheme for controlling the implied overhead. Here, we introduced synchronization groups, where a primary sends only asynchronous updates for all activities within the groups. Only after the complete groups has been executed, the primary has to send a synchronous update, where the primary verifies that it is still primary. When containing many activities in one synchronization group, this saves a lot of execution time overhead, where our evaluations have shown to save up to $\frac{2}{3}$ of the overhead of the basic majority-based replication scheme. In turn, an old primary might – in the worst case – notice that it is not primary anymore after executing the complete synchronization group, which increases the compensation cost up to four times of the basic majority-based replication scheme. However, without any failures there is no need for electing a new primary and, thus, none of the approaches require any compensation. As a consequence, large synchronization groups are preferable when failures are rare.

When, however, comparing the majority-based replication schemes to active replication, there is still room for improvement in terms of availability. When a majority of replicas is failed, the replicas are unable to elect a new primary. Thus, we proposed a flexible failover replication scheme, where the workflow designer or user can set the threshold on how many replicas are required for electing a new primary. This allows to tolerate temporal failures of more than a majority of replicas. Of course, this also allows multiple partitions to elect a primary when the threshold is set low, e.g., to only one required vote. Since all but one workflow execution have to be compensated, flexible failover replication increases the induced compensation cost compared to majority-based replication. However, we reach nearly the availability of active replication while inducing no compensation cost in the failure free case – like our majority-based replication scheme. Thus, flexible failover replication is the mechanism of choice for workflows, where availability is the main concern and failures are rare. However, in case that a failure occurs, the user or provider of the workflow should be willing to pay the high compensation cost.

Declarative workflow languages provide the possibility to reorder the activities of the workflow at run-time. We exploited this property to increase availability further and speed up replicated executions by executing different activity

sequences of the same workflow on the different replicas. For generating the sequences, we developed a metric that allows to select the best activity sequences of a workflow in terms of availability. To reduce the time and space requirements of the generation of the sequences, we presented a pruning strategy that generates activity sequences that have up to 80% of the availability (according to our metric) compared to the optimal solution while requiring only a fraction of the time that the generation of the optimal solution requires. In some cases pruning saves over 99% of the generation time. Due to the speed up, the reordered replicas execute within 55% of the time required for executing structurally identical replicas – even in the presence of failures.

In conclusion, we provided the means for ensuring the availability of workflows specified in imperative or declarative workflow languages. The workflow replication schemes allow a user, workflow designer, or workflow provider to choose exactly the replication scheme that conforms to their requirements.

15.1. Outlook

As we have shown that workflow replication ensures availability in the presence of failures. When increasing the replication degree, i.e., the number of used replicas, the execution can tolerate more failures. In other words, increasing the replication degree also increases availability. However, using more replicas also increases the replication overhead, e.g., in terms of the compensation cost or execution time. Additionally, more replicas require the workflow provider or user to deploy and operate more computing nodes. It might not be obvious to a workflow designer and especially to a user, which replication degree should be chosen in a specific environment.

For overcoming this limitation, an in-depth study on the required parameters, such as the failure rates of the computing nodes and the communication links, for making the decision on the replication degree might be necessary. From this study, a utility formula for choosing the replication degree under user specified constraints should be developed.

Furthermore, our workflow replication schemes for imperative workflow languages support AND- and XOR-gateways as well as synchronous and asynchronous service calls. Even though these are probably the most widely used

patterns for modeling workflows, there exist many control flow and exception patterns (e.g., [RvdAtH06]). When implementing an industry-grade workflow replication system, each pattern has to be considered individually in order to ensure that Single-Execution-Equivalence is always ensured before supporting the replicated execution of this pattern.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AAA+96] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Günthör, and C. Mohan. Advanced transaction models in workflow contexts. In *Proceedings of the 12th IEEE International Conference on Data Engineering (ICDE)*, pages 574–581. IEEE, February 1996.

[AD76] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, pages 562–570, Los Alamitos, CA, USA, October 1976. IEEE Computer Society Press.

[Ady99] Atul Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[AFB+06] Thara Angskun, Graham E. Fagg, George Bosilca, Jelena Pješivac-Grbović, and Jack J. Dongarra. Scalable fault tolerant protocol for parallel runtime environments. In Bernd Mohr, Jesper Larsson Träff, Joachim Worringen, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science (LNCS)*, pages 141–149. Springer Berlin Heidelberg, 2006.

[AGK04] I. Assayad, A. Girault, and H. Kalla. A bi-criteria scheduling heuristic for distributed embedded systems under reliability and real-time constraints. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN)*, pages 347–356. IEEE, June 2004.

[AGKW07] Bahram Alidaee, Fred Glover, Gary Kochenberger, and Haibo Wang. Solving the maximum edge weight clique problem via

unconstrained quadratic programming. *European Journal of Operational Research*, 181(2):592–597, September 2007.

[ALL] Allow ensembles: Description of work. Unpublished.

[AMG+95] G. Alonso, C. Mohan, R. Günthör, D. Agrawal, A. El Abbadi, and M. Kamath. Exotica/fmqm: A persistent message-based architecture for distributed workflow management. In Arne Sölvberg, John Krogstie, and Anne Helga Seltveit, editors, *Information Systems Development for Decentralized Organizations: Proceedings of the IFIP working conference on information systems development for decentralized organizations, 1995*, pages 1–18, Boston, MA, 1995. Springer US.

[AMM08] Eyhab Al-Masri and Qusay H. Mahmoud. Investigating web services on the world wide web. In *Proceedings of the 17th International Conference on World Wide Web (WWW)*, pages 795–804, New York, NY, USA, April 2008. ACM.

[Avi85] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering (TSE)*, SE-11(12):1491–1501, December 1985.

[Bac13] Thomas Bach. Methods to coordinate the execution of workflow replicas in a distributed environment. Diplomarbeit, Institute of Parallel and Distributed Systems, University of Stuttgart, April 2013.

[BCH+05] A. Bouteiller, B. Collin, T. Herault, P. Lemarinier, and F. Cappello. Impact of event logger on causal message logging protocols for fault tolerant mpi. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 97–97. IEEE, April 2005.

[BCPR04] M. Brambilla, S. Ceri, M. Passamani, and A. Riccio. Managing asynchronous web services interactions. In *Proceedings of the 2004 IEEE International Conference on Web Services (ICWS)*, pages 80–87. IEEE, July 2004.

[BD97] T. Bauer and P. Dadam. A distributed execution environment for large-scale workflow management systems with subnets and server migration. In *Proceedings of the 2nd IFCIS International Conference on Cooperative Information Systems (CoopIS)*, pages 99–108. IEEE, June 1997.

[BDF+13] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment*, 7(3):181–192, November 2013.

[BDH+12] Johannes Behl, Tobias Distler, Florian Heisig, Rüdiger Kapitza, and Matthias Schunter. Providing fault-tolerant execution of web-service-based workflows within clouds. In *Proceedings of the 2nd International Workshop on Cloud Computing Platforms (CloudCP)*, CloudCP '12, pages 7:1–7:6, New York, NY, USA, April 2012. ACM.

[BFF+14] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment*, 8(3):185–196, November 2014.

[BFHD12] K. Birman, D. Freedman, Q. Huang, and P. Dowell. Overcoming cap with consistent soft-state replication. *Computer*, 45(2):50–58, February 2012.

[BGS07] D. Bianculli, C. Ghezzi, and P. Spoletini. A model checking approach to verify bpel4ws workflows. In *Proceedings of the 2007 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 13–20. IEEE, June 2007.

[BHK+06] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. Mpich-v project: A multiprotocol automatic fault-tolerant mpi. *International Journal of High Performance Computing Applications (IJHPCA)*, 20(3):319–333, August 2006.

[BHR08] A. Benoit, M. Hakem, and Y. Robert. Fault tolerant scheduling of precedence task graphs on heterogeneous platforms. In *Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–8. IEEE, April 2008.

[BHR09a] A. Benoit, M. Hakem, and Y. Robert. Optimizing the latency of streaming applications under throughput and reliability constraints. In *Proceedings of the 2009 International Conference on Parallel Processing (ICPP)*, pages 325–332. IEEE, September 2009.

[BHR09b] Anne Benoit, Mourad Hakem, and Yves Robert. Contention awareness and fault-tolerant scheduling for precedence constrained tasks in heterogeneous systems. *Parallel Computing*, 35(2):83–108, February 2009.

[BK14] Peter Bailis and Kyle Kingsbury. The network is reliable. *Queue*, 12(7):20:20–20:32, July 2014.

[BLKC03] B. Bouteiller, P. Lemarinier, K. Krawezik, and F. Capello. Coordinated checkpoint versus message log for fault tolerant mpi. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, pages 242–250. IEEE, December 2003.

[BMM06] Luciano Baresi, Andrea Maurino, and Stefano Modafferi. Towards distributed bpel orchestrations. In *Proceedings of the Third Workshop on Software Evolution through Transformations: Embracing the Change (SeTra 2006)*, volume 3 of *Electronic Communications of the EASST*. EASST, 2006.

[BMST93] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. In Sape Mullender, editor, *Distributed Systems (2nd Ed.)*, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[BMV10] Aruna Balasubramanian, Ratul Mahajan, and Arun Venkataramani. Augmenting mobile 3g using wifi. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and*

Services (MobiSys), pages 209–222, New York, NY, USA, June 2010. ACM.

[Bre00] Eric A. Brewer. Towards robust distributed systems. In *Keynote Talk at the 19th ACM Symposium on Principles of Distributed Computing (PODC)*, July 2000.

[Bre12] Eric Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, February 2012.

[Bre17] Eric Brewer. Spanner, truetime & the cap theorem. Technical report, Google, February 2017.

[BRSR08] A. Benoit, V. Rehn-Sonigo, and Y. Robert. Optimizing latency and reliability of pipeline workflow applications. In *Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–10. IEEE, April 2008.

[Bru09] Jake Brutlag. Speed matters for google web search. Technical report, Google, Inc., June 2009.

[BT83] Joseph A. Bannister and Kishor S. Trivedi. Task allocation in fault-tolerant distributed systems. *Acta Informatica*, 20(3):261–281, May 1983.

[BTKR15] T. Bach, M. A. Tariq, B. Koldehofe, and K. Rothermel. A cost efficient scheduling strategy to guarantee probabilistic workflow deadlines. In *Proceedings of the 2015 International Conference and Workshops on Networked Systems (NetSys)*, pages 1–8. IEEE, March 2015.

[BWDL08] J. Bian, C. Weng, J. Du, and M. Li. A qos-aware and fault-tolerant workflow composition for grid. In *Proceedings of the 7th International Conference on Grid and Cooperative Computing (GCC)*, pages 510–516. IEEE, October 2008.

[CB14] R.N. Calheiros and R. Buyya. Meeting deadlines of scientific workflows in public clouds with tasks replication. *IEEE Trans-*

actions on Parallel and Distributed Systems (TPDS), 25(7):1787–1796, September 2014.

[CBPS10] Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science (LNCS)*. Springer Berlin Heidelberg, 2010.

[CCE03] T. Chahed, A. F. Canton, and S. E. Elayoubi. End-to-end tcp performance in w-cdma / umts. In *Proceedings of the 38th IEEE International Conference on Communications (ICC)*, pages 71–75. IEEE, May 2003.

[CDK05] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Pearson Education Limited, Edinburgh Gate, Harlow, Essex CM20 2JE, England, fourth edition, 2005.

[CGH+06] David Churches, Gabor Gombas, Andrew Harrison, Jason Maassen, Craig Robinson, Matthew Shields, Ian Taylor, and Ian Wang. Programming scientific and distributed workflow with triana services. *Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, 2006.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, February 1985.

[CL12] James Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC)*, pages 223–235, Boston, MA, June 2012. USENIX.

[Cou99] Jean-Michel Couvreur. On-the-fly verification of linear temporal logic. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems Toulouse,*

France, September 20–24, 1999 Proceedings, Volume I, volume 1708 of *Lecture Notes in Computer Science (LNCS)*, pages 253–271. Springer Berlin Heidelberg, 1999.

[CRCR09] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2stm: Dependable distributed software transactional memory. In *Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 307–313. IEEE, November 2009.

[CTX09] K. T. Chen, C. C. Tu, and W. C. Xiao. Oneclick: A framework for measuring network quality of experience. In *Proceedings of the 28th IEEE International Conference on Computer Communications (INFOCOM)*, pages 702–710. IEEE, April 2009.

[DDGB09] Remco Dijkman, Marlon Dumas, and Luciano García-Bañuelos. Graph matching algorithms for business process model similarity search. In Umeshwar Dayal, Johann Eder, Jana Koehler, and Hajo A. Reijers, editors, *Business Process Management: 7th International Conference, BPM 2009, Ulm, Germany, September 8-10, 2009. Proceedings*, volume 5701 of *Lecture Notes in Computer Science (LNCS)*, pages 48–63. Springer Berlin Heidelberg, 2009.

[DDGJ01] W. Derks, J. Dehnert, P. Grefen, and W. Jonker. Customized atomicity specification for transactional workflows. In *Proceedings of the 3rd International Symposium on Cooperative Database Systems for Advanced Applications. (CODAS)*, pages 140–147. IEEE, April 2001.

[Dea09] Jeff Dean. Designs, lessons and advice from building large distributed systems. In *Keynote at the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, October 2009.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo:

Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review (OSR)*, 41(6):205–220, October 2007.

[DL17] Alexandre Duret-Lutz. *Contributions to LTL and ω-Automata for Model Checking*. Habilitation thesis, Université Pierre et Marie Curie, 2017.

[DLP04] A. Duret-Lutz and D. Poitrenaud. Spot: an extensible model checking library using transition-based generalized b uuml;chi automata. In *Proceedings of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS 2004)*, pages 76–83. IEEE, October 2004.

[DO02] A. Dogan and F. Ozguner. Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 13(3):308–323, March 2002.

[Dob06] G. Dobson. Using ws-bpel to implement software fault tolerance for web services. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 126–133. IEEE, August 2006.

[DPEV+06] Massimiliano Di Penta, Raffaele Esposito, Maria Luisa Villani, Roberto Codato, Massimiliano Colombo, and Elisabetta Di Nitto. Ws binder: A framework to enable dynamic binding of composite web services. In *Proceedings of the 2006 International Workshop on Service-oriented Software Engineering (IW-SOSE)*, pages 74–80, New York, NY, USA, May 2006. ACM.

[dSdSeSL07] S. Araujo de Sousa, F. Jose da Silva e Silva, and R. F. Lopes. A flexible fault-tolerance mechanism for the integrade grid middleware. In *Proceedings of the 3rd International Conference on Networking and Services (ICNS)*, pages 26–26. IEEE, June 2007.

[DSS+05] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi,

G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, December 2005.

[DW01] John R. Douceur and Roger P. Wattenhofer. Competitive hill-climbing strategies for replica placement in a distributed file system. In Jennifer Welch, editor, *Distributed Computing: 15th International Conference, DISC 2001 Lisbon, Portugal, October 3–5, 2001 Proceedings*, volume 2180 of *Lecture Notes in Computer Science (LNCS)*, pages 48–62. Springer Berlin Heidelberg, 2001.

[EAWJ02] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, September 2002.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1):151–178, January 1986.

[EKEF16] Mohamed El Kholy and Ahmed El Fatatry. Frwsc: a framework for robust web service composition. *Service Oriented Computing and Applications (SOCA)*, 10(4):413–435, December 2016.

[EMSU11] Faramarz Safi Esfahani, Masrah Azrifah Azmi Murad, Md. Nasir B. Sulaiman, and Nur Izura Udzir. Adaptable decentralized service oriented architecture. *Journal of Systems and Software*, 84(10):1591–1617, October 2011.

[ES06] Onyeka Ezenwoye and S. Masoud Sadjadi. Robustbpel-2: Transparent autonomization in aggregate web services using dynamic proxies. Technical report, School of Computing and Information Sciences, Florida International University, 11200 SW 8th St., Miami, FL 33199, 2006.

[ES07] Onyeka Ezenwoye and S. Masoud Sadjadi. Trap/bpel: A framework for dynamic adaptation of composite services. In *In Proceedings of the International Conference on Web Information Systems and Technologies (WEBIST 2007*, 2007.

[FABK03] Nick Feamster, David G. Andersen, Hari Balakrishnan, and M. Frans Kaashoek. Measuring the effects of internet path faults on reactive routing. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):126–137, June 2003.

[FB99] A. Fox and E. A. Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 174–178. IEEE, March 1999.

[FLLL07] Chen-Liang Fang, Deron Liang, Fengyi Lin, and Chien-Cheng Lin. Fault tolerant web services. *Journal of Systems Architecture (JSA)*, 53(1):21–38, January 2007.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, April 1985.

[FWM03] Roberto Silveira Silva Filho, Jacques Wainer, and Edmundo R. M. Madeira. A fully distributed architecture for large scale workflow enactment. *International Journal of Cooperative Information Systems (IJCIS)*, 12(04):411–440, December 2003.

[FYG09] W. Fdhila, U. Yildiz, and C. Godart. A flexible approach for automatic process decentralization using dependency tables. In *Proceedings of the 2009 IEEE International Conference on Web Services (ICWS)*, pages 847–855. IEEE, July 2009.

[GAP+12] Aaron Gember, Aditya Akella, Jeffrey Pang, Alexander Varshavsky, and Ramon Caceres. Obtaining in-context measurements of cellular network performance. In *Proceedings of the 2012 Internet Measurement Conference (IMC)*, pages 287–300, New York, NY, USA, November 2012. ACM.

[GCB+14] Felipe Pontes Guimaraes, Pedro Célestin, Daniel Macedo Batista, Genaína Nunes Rodrigues, and Alba Cristina Magalhaes Alves de Melo. A framework for adaptive fault-tolerant execution of workflows in the grid: Empirical and theoretical analysis. *Journal of Grid Computing*, 12(1):127–151, March 2014.

[GdM11] F. P. Guimaraes and A. C. Magalhaes Alves de Melo. User-defined adaptive fault-tolerant execution of workflows in the grid. In *Proceedings of the 11th IEEE International Conference on Computer and Information Technology (CIT)*, pages 356–362. IEEE, August 2011.

[GEST09] Alain Girault, Érik Saule, and Denis Trystram. Reliability versus performance for critical applications. *Journal of Parallel and Distributed Computing*, 69(3):326–336, March 2009.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GHL+07] H. Guo, J. Huai, H. Li, T. Deng, Y. Li, and Z. Du. Angel: Optimal configuration for high available service composition. In *Proceedings of the 2007 IEEE International Conference on Web Services (ICWS)*, pages 280–287. IEEE, July 2007.

[GJN11] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. *ACM SIGCOMM Computer Communication Review*, 41(4):350–361, August 2011.

[GMGK+91] Hector Garcia-Molina, Dieter Gawlick, Johannes Klein, Karl Kleissner, and Kenneth Salem. Modeling long-running activities as nested sagas. *Data Engineering*, 14(1):14–18, March 1991.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. *ACM SIGMOD Record*, 16(3):249–259, December 1987.

[Gra85] Jim Gray. Why do computers stop and what can be done about it? Tandem Technical report 85.7, June 1985.

[Gre02] Paul Grefen. Transactional workflows or workflow transactions? In Abdelkader Hameurlain, Rosine Cicchetti, and Roland Traunmüller, editors, *Database and Expert Systems Applications: 13th International Conference, DEXA 2002 Aix-en-Provence, France, September 2–6, 2002 Proceedings*, volume 2453 of *Lecture Notes in Computer Science (LNCS)*, pages 60–69. Springer Berlin Heidelberg, September 2002.

[GWLY11] Y. Gu, Q. Wu, X. Liu, and D. Yu. Improving throughput and reliability of distributed scientific workflows for streaming data processing. In *Proceedings of the 13th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 347–354. IEEE, September 2011.

[HB07] M. Hakem and F. Butelle. Reliability and scheduling on systems subject to failures. In *Proceedings of the 2007 International Conference on Parallel Processing (ICPP)*, pages 38–38. IEEE, September 2007.

[HCP03] Jiun-Long Huang, Ming-Syan Chen, and Wen-Chih Peng. Exploring group mobility for replica data allocation in a mobile environment. In *Proceedings of the 12th International Conference on Information and Knowledge Management (CIKM)*, pages 161–168, New York, NY, USA, November 2003. ACM.

[HCTS09] M. K. Hedayat, W. Cai, S. J. Turner, and S. Shahand. Distributed execution of workflow using parallel partitioning. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 106–112. IEEE, August 2009.

[HK03] Soonwook Hwang and Carl Kesselman. A flexible framework for fault tolerance in the grid. *Journal of Grid Computing*, 1(3):251–272, September 2003.

[Hol95] David Hollingsworth. Workflow management coalition: The workflow reference model. The Workflow Management Coalition Specification Document Number TC00-1003, The Workflow Management Coalition, Winchester, Hampshire, UK, January 1995.

[JDF09] E. Juhnke, T. Dörnemann, and B. Freisleben. Fault-tolerant bpel workflow execution via cloud-aware recovery policies. In *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 31–38. IEEE, August 2009.

[KAGM96] M. Kamath, G. Alonso, R. Günthör, and C. Mohan. Providing high availability in very large workflow management systems. In Peter Apers, Mokrane Bouzeghoub, and Georges Gardarin, editors, *Advances in Database Technology — EDBT '96: 5th International Conference on Extending Database Technology Avignon, France, March 25–29, 1996 Proceedings*, volume 1057 of *Lecture Notes in Computer Science (LNCS)*, pages 425–442. Springer Berlin Heidelberg, March 1996.

[KHC+05] D. Karastoyanova, A. Houspanossian, M. Cilia, F. Leymann, and A. Buchmann. Extending bpel for run time adaptability. In *Proceedings of the 9th IEEE International EDOC Enterprise Computing Conference (EDOC)*, pages 15–26. IEEE, September 2005.

[KIH10] Adrian Klein, Fuyuki Ishikawa, and Shinichi Honiden. Efficient qos-aware service composition with a probabilistic service selection policy. In Paul P. Maglio, Mathias Weske, Jian Yang, and Marcelo Fantinato, editors, *Service-Oriented Computing: 8th International Conference, ICSOC 2010, San Francisco, CA, USA, December 7-10, 2010. Proceedings*, volume 6470 of *Lecture Notes in Computer Science (LNCS)*, pages 182–196. Springer Berlin Heidelberg, August 2010.

[KKW13] T. Kobus, M. Kokocinski, and P.T. Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication

schemes combined. In *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 286–296. IEEE, July 2013.

[KL05] Tevfik Kosar and Miron Livny. A framework for reliable and efficient data placement in distributed computing systems. *Journal of Parallel and Distributed Computing*, 65(10):1146–1157, October 2005.

[KL12] Rania Khalaf and Frank Leymann. Coordination for fragmented loops and scopes in a distributed business process. *Information Systems*, 37(6):593–610, September 2012.

[KLL09] Ryan K.L. Ko, Stephen S.G. Lee, and Eng Wah Lee. Business process management (bpm) standards: a survey. *Business Process Management Journal*, 15(5):744–791, 2009.

[KM97] S. Kartik and C.S.R. Murthy. Task allocation algorithms for maximizing reliability of distributed computing systems. *IEEE Transactions on Computers (TC)*, 46(6):719–724, June 1997.

[KMR08] G. Kandaswamy, A. Mandal, and D. A. Reed. Fault tolerance and recovery of scientific workflows on computational grids. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 777–782. IEEE, May 2008.

[KN01] Kengo Katayama and Hiroyuki Narihisa. Performance of simulated annealing-based heuristic for the unconstrained binary quadratic programming problem. *European Journal of Operational Research*, 134(1):103–119, October 2001.

[Kra16] Lukas Krawczyk. Implementierung einer bestehenden architektur zur ausführung replizierter workflows. Unpublished Study Thesis, April 2016.

[LAJ99] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental study of internet stability and backbone failures. In *Digest of Papers.*

Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, pages 278–285. IEEE, June 1999.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):51–58, December 2001.

[LC12] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA, July 2012.

[LCZ05] Qiao Lian, Wei Chen, and Zheng Zhang. On the impact of replica placement to the reliability of distributed brick storage systems. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 187–196. IEEE, June 2005.

[LFCL03] Deron Liang, Chen-Liang Fang, Chyouhnwa Chen, and Fengyi Lin. Fault tolerant web service. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference (APSEC)*, pages 310–319. IEEE, December 2003.

[LFKA11] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 401–416, New York, NY, USA, October 2011. ACM.

[LGG+91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the harp

file system. *ACM SIGOPS Operating Systems Review (OSR)*, 25(5):226–238, October 1991.

[LHG+16] Aida Lahouij, Lazhar Hamel, Mohamed Graiet, Abir Elkhalfa, and Walid Gaaloul. A global sla-aware approach for aggregating services in the cloud. In Christophe Debruyne, Hervé Panetto, Robert Meersman, Tharam Dillon, eva Kühn, Declan O'Sullivan, and Claudio Agostino Ardagna, editors, *On the Move to Meaningful Internet Systems: OTM 2016 Conferences: Confederated International Conferences: CoopIS, C&TC, and ODBASE 2016, Rhodes, Greece, October 24-28, 2016, Proceedings*, volume 10033 of *Lecture Notes in Computer Science (LNCS)*, pages 363–380, Cham, 2016. Springer International Publishing.

[LL06] Yawei Li and Zhiling Lan. Exploit failure prediction for adaptive fault-tolerance in cluster computing. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 531–538, Washington, DC, USA, May 2006. IEEE Computer Society.

[LLdSFV08] Jim Lau, Lau Cheuk Lung, Joni da S. Fraga, and Giuliana Santos Veronese. Designing fault tolerant web services using bpel. In *Proceedings of the 7th IEEE/ACIS International Conference on Computer and Information Science (ICIS)*, pages 618–623. IEEE, May 2008.

[LLSG92] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 10(4):360–391, November 1992.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review (OSR)*, 44(2):35–40, April 2010.

[LMJ10] Guoli Li, Vinod Muthusamy, and Hans-Arno Jacobsen. A distributed service-oriented architecture for business process exe-

cution. *ACM Transactions on the Web (TWEB)*, 4(1):2:1–2:33, January 2010.

[Loh12] Steve Lohr. For impatient web users, an eye blink is just too long to wait. New York Times Online Article, February 2012.

[LPC+11] F. Lopes, T. Pereira, E. Cavalcante, T. Batista, F. C. Delicato, P. F. Pires, and P. Ferreira. Adaptubiflow: Selection and adaptation in workflows for ubiquitous computing. In *Proceedings of the 9th IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 63–71. IEEE, October 2011.

[LR00] Frank Leymann and Dieter Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[LS79] Butler Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. 1979.

[LV07] Nuno Laranjeiro and Marco Vieira. Towards fault tolerance in web services compositions. In *Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems (EFTS)*, New York, NY, USA, September 2007. ACM.

[MF02] Peter Merz and Bernd Freisleben. Greedy and local search heuristics for unconstrained binary quadratic programming. *Journal of Heuristics*, 8(2):197–213, March 2002.

[MIB+08] 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 Transactions on Networking (TON)*, 16(4):749–762, August 2008.

[ML83] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 76–88, New York, NY, USA, August 1983. ACM.

[MM05] F. Montagut and R. Molva. Enabling pervasive execution of workflows. In *Proceedings of the 2005 International Conference on Collaborative Computing: Networking, Applications and Worksharing (COLLABORATECOM)*, page 10 pp. IEEE, July 2005.

[MSB08] Z. Maamar, Q. Z. Sheng, and D. Benslimane. Sustaining web services high-availability using communities. In *Proceedigns of the 3rd International Conference on Availability, Reliability and Security (ARES)*, pages 834–841. IEEE, March 2008.

[MSM15] P. M. Melliar-Smith and L. E. Moser. Conversion infrastructure for maintaining high availability of web services using multiple service providers. In *Proceedings of the 2015 IEEE International Conference on Web Services (ICWS)*, pages 759–764. IEEE, June 2015.

[MWW+98] Peter Muth, Dirk Wodtke, Jeanine Weissenfels, Angelika Kotz Dittrich, and Gerhard Weikum. From centralized workflow specification to distributed workflow execution. *Journal of Intelligent Information Systems (JIIS)*, 10(2):159–184, March 1998.

[NDO11] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, cells and platters: An empirical analysisof hardware failures on a million consumer pcs. In *Proceedings of the 6th ACM European Conference on Computer Systems (EuroSys)*, pages 343–356, New York, NY, USA, April 2011. ACM.

[Oki88] Brian M. Oki. *Viewstamped Replication for Highly Available Distributed Systems*. PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA, USA, August 1988.

[OL88] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 8–17, New York, NY, USA, August 1988. ACM.

[Ong14] Diego Ongaro. *Consensus: bridging theory and practice*. PhD thesis, Stanford University, Department of Computer Science, Stanford, CA, USA, August 2014.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC)*, pages 305–319, Philadelphia, PA, USA, June 2014. USENIX Association.

[PA13] J.D. Power and Associates. 2013 u.s. wireless network quality performance study - vol. 1. Press Release, August 2013.

[PCD91] David Powell, Marc Chérèque, and David Drackley. Fault-tolerance in delta-4. *ACM SIGOPS Operating Systems Review (OSR)*, 25(2):122–125, April 1991.

[Pes08] Maja Pesic. *Constraint-based workflow management systems: shifting control to users*. PhD thesis, Technische Universiteit Eindhoven, 2008.

[PGS03] Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[PLM+10] Carlos Pedrinaci, Dong Liu, Maria Maleshkova, David Lambert, Jacek Kopecky, and John Domingue. iserve: a linked services publishing platform. In *CEUR Workshop Proceedings*, volume 596, 2010.

[PPF09] K. Plankensteiner, R. Prodan, and T. Fahringer. A new fault tolerance heuristic for scientific workflows in highly distributed environments based on resubmission impact. In *Proceedings of the 5th IEEE International Conference on e-Science (eScience)*, pages 313–320. IEEE, December 2009.

[PSvdA07] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst. Declare: Full support for loosely-structured processes. In *Proceedings of the*

11th IEEE International Enterprise Distributed Object Computing Conference (EDOC), pages 287–287. IEEE, October 2007.

[PW10] R. Prodan and M. Wieczorek. Bi-criteria scheduling of scientific grid workflows. *IEEE Transactions on Automation Science and Engineering (T-ASE)*, 7(2):364–376, April 2010.

[Qua07] Dang Minh Quan. Error recovery mechanism for grid-based workflow within sla context. *International Journal of High Performance Computing and Networking (IJHPCN)*, 5(1–2):110–121, November 2007.

[RS95] Marek Rusinkiewicz and Amit Sheth. Specification and execution of transactional workflow. In Won Kim, editor, *Modern database systems: the object model, interoperability, and beyond*, pages 592–620. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[Rus80] D.L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering (TSE)*, SE-6(2):183–194, March 1980.

[RvdAtH06] Nick Russell, Wil van der Aalst, and Arthur ter Hofstede. Workflow exception patterns. In Eric Dubois and Klaus Pohl, editors, *Advanced Information Systems Engineering: 18th International Conference, CAiSE 2006, Luxembourg, Luxembourg, June 5-9, 2006. Proceedings*, volume 4001 of *Lecture Notes in Computer Science (LNCS)*, pages 288–302. Springer Berlin Heidelberg, 2006.

[SABS02] Heiko Schuldt, Gustavo Alonso, Catriel Beeri, and Hans-Jörg Schek. Atomicity and isolation for transactional processes. *ACM Transactions on Database Systems (TODS)*, 27(1):63–116, March 2002.

[SBT15] David Richard Schäfer, Thomas Bach, and Muhammad Adnan Tariq. Allow ensembles deliverable 6.2: Robustness models and algorithms. Project Deliverable, 2015.

[SBTR14] David Richard Schäfer, Thomas Bach, Muhammad Adnan Tariq, and Kurt Rothermel. Increasing availability of workflows executing in a pervasive environment. In *Proceedings of the 2014 IEEE International Conference on Services Computing (SCC)*, pages 717–724. IEEE, June 2014.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM (JACM)*, 32(3):733–749, July 1985.

[SCGM14] Ankit Singla, Balakrishnan Chandrasekaran, P. Brighten Godfrey, and Bruce Maggs. The internet at the speed of light. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets)*, pages 1:1–1:7, New York, NY, USA, October 2014. ACM.

[Sch13] David Richard Schäfer. Robust execution of workflows in a distributed environment. Diplomarbeit, Institute of Parallel and Distributed Systems, University of Stuttgart, June 2013.

[Seg13] Sascha Segan. Fastest mobile network 2013, June 2013.

[SF07] N. Salatge and J. C. Fabre. Fault tolerance connectors for unreliable web services. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 51–60. IEEE, June 2007.

[SJP06] Sebastian Stein, Nicholas R. Jennings, and Terry R. Payne. Flexible provisioning of service workflows. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI)*, pages 295–299, Amsterdam, The Netherlands, August 2006. IOS Press.

[SK11] Mirko Sonntag and Dimka Karastoyanova. Enforcing the repeated execution of logic in workflows. In *Proceedings of the 1st International Conference on Business Intelligence and Technology (BUSTECH)*, pages 20–25. IARIA, September 2011.

[SK12] Mirko Sonntag and Dimka Karastoyanova. Ad hoc iteration and re-execution of activities in workflows. *International Journal On Advances in Software*, 5(1&2):91–109, 2012.

[SK13] Mirko Sonntag and Dimka Karastoyanova. Model-as-you-go: An approach for an advanced infrastructure for scientific workflows. *Journal of Grid Computing*, 11(3):553–583, September 2013.

[SLM05] G.T. Santos, Lau Cheuk Lung, and C. Montez. Ftweb: a fault tolerant infrastructure for web services. In *Proceedings of the 9th IEEE International EDOC Enterprise Computing Conference*, pages 95–105. IEEE, September 2005.

[Sol08] Vision 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.

[SPJ07] Sebastian Stein, Terry R. Payne, and Nicholas R. Jennings. An effective strategy for the flexible provisioning of service workflows. In Jingshan Huang, Ryszard Kowalczyk, Zakaria Maamar, David Martin, Ingo Müller, Suzette Stoutenburg, and Katia P. Sycara, editors, *Service-Oriented Computing: Agents, Semantics, and Engineering. SOCASE 2007*, volume 4504 of *Lecture Notes in Computer Science (LNCS)*, pages 16–30. Springer Berlin Heidelberg, 2007.

[SPJ11] S. Stein, T. R. Payne, and N. R. Jennings. Robust execution of service workflows using redundancy and advance reservations. *IEEE Transactions on Services Computing (TSC)*, 4(2):125–139, April 2011.

[SRT17] David Richard Schäfer, Kurt Rothermel, and Muhammad Adnan Tariq. Exploring the search space between active and passive workflow replication. In *Proceedings of the 10th IEEE International Conference on Service Oriented Computing and Applications (SOCA)*. IEEE, November 2017.

[SRT18] D. R. Schäfer, K. Rothermel, and M. A. Tariq. Replication schemes for highly available workflow engines. *IEEE Transactions on Services Computing (TSC)*, PP(99):1–1, March 2018.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys (CSUR)*, 37(1):42–81, March 2005.

[SS12] Nenad Stojnić and Heiko Schuldt. Osiris-sr: A safety ring for self-healing distributed composite service execution. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, SEAMS '12, pages 21–26, Piscataway, NJ, USA, June 2012. IEEE Press.

[SS13] Nenad Stojnić and Heiko Schuldt. Osiris-sr: A scalable yet reliable distributed workflow execution engine. In *Proceedings of the 2nd ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies (SWEET)*, pages 3:1–3:12, New York, NY, USA, June 2013. ACM.

[SSB+15] David Richard Schäfer, Santiago Gómez Sáez, Thomas Bach, Vasilios Andrikopoulos, and Muhammad Adnan Tariq. Towards ensuring high availability in collective adaptive systems. In Fabiana Fournier and Jan Mendling, editors, *Business Process Management Workshops: BPM 2014 International Workshops, Eindhoven, The Netherlands, September 7-8, 2014, Revised Papers*, pages 165–171, Cham, September 2015. Springer International Publishing.

[SSR+10] Kunwadee Sripanidkulchai, Sambit Sahu, Yaoping Ruan, Anees Shaikh, and Chitra Dorai. Are clouds ready for large distributed applications? *ACM SIGOPS Operating Systems Review (OSR)*, 44(2):18–23, April 2010.

[STR16a] David Richard Schäfer, Muhammad Adnan Tariq, and Kurt Rothermel. Allow ensembles deliverable 6.3: Ensemble robustness prototype. Project Deliverable, 2016.

[STR16b] David Richard Schäfer, Muhammad Adnan Tariq, and Kurt Rothermel. Highly availabile process executions. Technical Report

2016/02, Institute of Parallel and Distributed Systems, University of Stuttgart, March 2016.

[SWG92] S.M. Shatz, J.-P. Wang, and M. Goto. Task allocation for maximizing reliability of distributed computer systems. *IEEE Transactions on Computers (TC)*, 41(9):1156–1168, September 1992.

[SWSS04] C. Schuler, R. Weber, H. Schuldt, and H. J. Schek. Scalable peer-to-peer process management — the osiris approach. In *Proceedings of the 2004 IEEE International Conference on Web Services (ICWS)*, pages 26–34. IEEE, July 2004.

[SWT+16] David Richard Schäfer, Andreas Weiß, Muhammad Adnan Tariq, Vasilios Andrikopoulos, Santiago Gómez Sáez, Lukas Krawczyk, and Kurt Rothermel. Hawks: A system for highly available executions of workflows. In *Proceedings of the 2016 IEEE International Conference on Services Computing (SCC)*, pages 130–137. IEEE, June 2016.

[TBV15] Lewis Tseng, Alec Benzer, and Nitin H. Vaidya. Application-aware consistency: An application to social network. *Computing Research Repository (CoRR)*, pages 1–18, February 2015.

[TF07] Wei Tan and Yushun Fan. Dynamic workflow model fragmentation for distributed execution. *Computers in Industry*, 58(5):381–391, June 2007.

[Tho79] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 4(2):180–209, June 1979.

[TLHL09] C. Tang, Q. Li, B. Hua, and A. Liu. Developing reliable web services using independent replicas. In *Proceedings of the 4th International Conference on Semantics, Knowledge and Grid (SKG)*, pages 330–333. IEEE, October 2009.

[TLL08] W. L. Tan, F. Lam, and W. C. Lau. An empirical study on the capacity and performance of 3g networks. *IEEE Transactions on Mobile Computing (TMC)*, 7(6):737–750, June 2008.

[TLM+12] Daniel Turner, Kirill Levchenko, Jeffrey C. Mogul, Stefan Savage, Alex C. Snoeren, Daniel Turner, Kirill Levchenko, Jeffrey C. Mogul, Stefan Savage, and Alex C. Snoeren. On failure in managed enterprise networks. Technical Report HPL-2012-101, HP Laboratories, May 2012.

[TLSS10] Daniel Turner, Kirill Levchenko, Alex C. Snoeren, and Stefan Savage. California fault lines: Understanding the causes and impact of network failures. *ACM SIGCOMM Computer Communication Review*, 40(4):315–326, October 2010.

[TLX+06] M. Tu, P. Li, L. Xiao, I. L. Yen, and F. B. Bastani. Replica placement algorithms for mobile transaction systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 18(7):954–970, July 2006.

[TSB+13] Muhammad Adnan Tariq, David Richard Schäfer, Thomas Bach, Santiago Gómez Sáez, Dimka Karastoyanova, and Vasilios Andrikopoulos. Allow ensembles deliverable 6.1: Failure models and robustness approaches. Project Deliverable, 2013.

[TX05] Xueyan Tang and Jianliang Xu. Qos-aware replica placement for content distribution. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 16(10):921–932, October 2005.

[vdA97] W. M. P. van der Aalst. Verification of workflow nets. In Pierre Azéma and Gianfranco Balbo, editors, *Application and Theory of Petri Nets 1997: 18th International Conference, ICATPN'97 Toulouse, France, June 23–27, 1997 Proceedings*, volume 1248 of *Lecture Notes in Computer Science (LNCS)*, pages 407–426. Springer Berlin Heidelberg, 1997.

[vEVS02] M. van Erp, L. Vuurpijl, and L. Schomaker. An overview and comparison of voting methods for pattern recognition. In *Proceedings*

of the 8th International Workshop on Frontiers in Handwriting Recognition, pages 195–200. IEEE, August 2002.

[VRA15] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):42:1–42:36, April 2015.

[Wes07] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer Berlin Heidelberg New York, 1 edition, 2007.

[WHP09] Marek Wieczorek, Andreas Hoheisel, and Radu Prodan. Towards a general model of the multi-criteria workflow scheduling on the grid. *Future Generation Computer Systems*, 25(3):237–256, March 2009.

[WKK+10] Branimir Wetzstein, Dimka Karastoyanova, Oliver Kopp, Frank Leymann, and Daniel Zwink. Cross-organizational process monitoring based on service choreographies. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, pages 2485–2490, New York, NY, USA, March 2010. ACM.

[WKK12] P. T. Wojciechowski, T. Kobus, and M. Kokocinski. Model-driven comparison of state-machine-based and deferred-update replication schemes. In *Proceedings of the 31st IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 101–110. IEEE, October 2012.

[Woo95] A. Wood. Predicting client/server availability. *Computer*, 28(4):41–48, Apr 1995.

[WPS+00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 464–474. IEEE, April 2000.

[YB05] Jia Yu and Rajkumar Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3):171–200, September 2005.

[ZL08a] Z. Zheng and M. R. Lyu. A distributed replication strategy evaluation and selection framework for fault tolerant web services. In *Proceedings of the 6th IEEE International Conference on Web Services (ICWS)*, pages 145–152. IEEE, September 2008.

[ZL08b] Zibin Zheng and Michael R. Lyu. Ws-dream: A distributed reliability assessment mechanism for web services. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 392–397. IEEE, June 2008.

[ZL08c] Zibin Zheng and M.R. Lyu. A qos-aware middleware for fault tolerant web services. In *Proceedings of the 19th International Symposium on Software Reliability Engineering (ISSRE)*, pages 97–106. IEEE, November 2008.

[ZL09] Z. Zheng and M. R. Lyu. A qos-aware fault tolerant middleware for dependable service composition. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 239–248. IEEE, June 2009.

[ZMKC09] Y. Zhang, A. Mandal, C. Koelbel, and K. Cooper. Combined fault tolerance and scheduling techniques for workflow applications on computational grids. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 244–251. IEEE, May 2009.

[ZP08] Vaidė Zuikevičiūtė and Fernando Pedone. Correctness criteria for database replication: Theoretical and practical aspects. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems: OTM 2008: OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008, Monterrey, Mexico, November 9-14, 2008, Proceedings, Part I*,

volume 5331 of *Lecture Notes in Computer Science (LNCS)*, pages 639–656. Springer Berlin Heidelberg, 2008.

[ZRXS10] L. Zhao, Y. Ren, Y. Xiang, and K. Sakurai. Fault-tolerant scheduling with dynamic number of replicas in heterogeneous systems. In *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 434–441. IEEE, September 2010.

[ZZL10] Z. Zheng, Y. Zhang, and M. R. Lyu. Distributed qos evaluation for real-world web services. In *Proceedings of the 8th IEEE International Conference on Web Services (ICWS)*, pages 83–90. IEEE, July 2010.

[ZZL14] Z. Zheng, Y. Zhang, and M. R. Lyu. Investigating qos of real-world web services. *IEEE Transactions on Services Computing (TSC)*, 7(1):32–39, January 2014.

Erklärung

Ich erkläre hiermit, dass ich, abgesehen von den ausdrücklich bezeichneten Hilfsmitteln und den Ratschlägen von jeweils namentlich aufgeführten Personen, die Dissertation selbstständig verfasst habe.

(David Richard Schäfer)