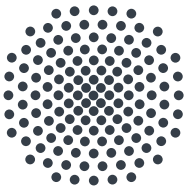

University of Stuttgart
Germany

IPVS

Technical Report 2016/02

Highly Available Process Executions

David Richard Schäfer
Muhammad Adnan Tariq
Kurt Rothermel

Institute of Parallel and Distributed Systems
University of Stuttgart
Universitätsstraße 38
70569 Stuttgart
Germany

March 2016

Highly Available Process Executions

David Richard Schäfer Muhammad Adnan Tariq Kurt Rothermel
Institute of Parallel and Distributed Systems, University of Stuttgart, Germany
Email: {david.schaefer, adnan.tariq, kurt.rothermel}@ipvs.uni-stuttgart.de

***Abstract*—Modern businesses operate globally with business locations and partners scattered across the globe. To handle the complexity arising with this distribution, automated business processes help managing the data and interactions between locations and partners. This migrates the execution of business processes, which previously ran in reliable back-end systems, to massively distributed environments. Due to heterogeneity and high dynamics, a key challenge in such environments is to ensure availability for these process executions in the presence of frequent communication and device failures.**

In this paper, we propose a protocol that provides high availability by replicating process executions. To ensure that a replicated execution produces the same outcome as a non-replicated execution, we formally define one-copy consistency for replicated process executions. Subsequently, we define rules that ensure one-copy consistency and present a replication protocol adhering to these rules even in the presence of failures. Our evaluations on Amazon EC2 and PlanetLab show that our proposed protocol significantly increases the availability of processes executing in a distributed environment while incurring very low overhead in terms of bandwidth utilization and execution time.

I. Introduction

Today's technology allows to easily bridge long distances and transmit information to virtually any location on this planet within milliseconds. Nowadays, even small businesses have several locations and partners all over the world. As a consequence, the business's data and business processes operating on that data are geo-distributed. These business processes consist of interrelated actions, where each action is an atomic unit of work, e.g., for exchanging, analyzing, or managing data. Today, business processes play a key role for automating and optimizing the management and operation of business locations. To keep locations operable, the processes frequently interact with processes of other locations and partners, e.g., to enable just-in-time production [1]. Consequently, businesses depend on highly available process executions meaning that the process executions should be able to start and progress at any point in time. Executions that are hindered or even stopped imply loss of money. When, for example, production breaks down because parts were not ordered in time or data analysis was not finished in time to provision enough cloud resources making a business's service unresponsive. Thus, availability is essential for staying competitive. Only one hour of unavailability implies a typical cost of up to $6.48 million [2].

Ensuring high availability in heterogeneous environments is, however, not trivial because of the frequent occurrences of device, network and communication failures. Even though fixed networks are usually known to be reliable, they are prone to frequent failures. A study of the IP Backbone network shows that connectivity failures in wide area networks occur at a median rate of 3000 s with a median duration of $2-1000$ s [3], [4]. Failures are even common in datacenters, which are widely known to provide high availability. Microsoft datacenters experience 40.8 network failures with end-user impact per day (on average), where each failure implies a median loss of $59,000$ packets [3], [5]. Large-scale operators like Google and Amazon confirm that network partitioning is an important design consideration to ensure high availability [3], [6].

Redundancy is a common approach for increasing the availability by replicating the same functionality on multiple machines [7]. With passive replication techniques only one replica executes the process at any time and the produced state is transferred to the other replicas [8]. This, however, implies the bandwidth overhead of transferring the state after the execution of each action to ensure a consistent state across all replicas. Active replication mechanisms, such as state machine replication [9]–[12], overcome this limitation by executing the processes in parallel on multiple replicas. State machine replication, however, assumes all actions of the process to be deterministic. Thus, after ensuring that the actions are executed in the same order on all replicas, the produced state is guaranteed to be identical. Processes in general, however, also comprise non-deterministic actions, rendering state machine replication unusable. Moreover, state machine replication assumes that all actions run in isolation on each replica. Actions accessing external functionality or interacting with other processes – which is very typical of business processes [13] – cannot be handled. Other approaches combine active and passive replication [14], [15]. The approaches, however, are also limited to deterministic actions.

Thus, the problem addressed in this paper has a broader scope. We target achieving high availability by the replicated execution of processes containing interactive and non-deterministic actions, which can access external functionality. In particular, our main contributions are: 1) We formally define consistency for replicated process executions, where a process is consistent if the outcome of a replicated execution is the same as for a non-replicated execution. 2) We specify rules for executing a process in a replicated manner such that the defined consistency is achieved. 3) We present a protocol that complies with the defined consistency rules while providing high availability. 4) Finally, we thoroughly evaluate the protocol on Amazon EC2 and PlanetLab with respect to scalability and performance in the presence of failures.[1]

[1] http://aws.amazon.com, https://www.planet-lab.eu

II. System Model

We consider a distributed system consisting of a network of computing nodes and services. Each computing node runs an *Execution Engine* which is able to execute processes. The services provide operations that the process uses during execution. Any component of the distributed system, i.e., computing node, service, or communication link of the network, might experience crash failures at any point in time. We, however, assume that any failed component eventually recovers.

We provide availability by replicating the execution such that multiple nodes execute a replica of the process (cf. Fig. 1). For understanding the challenges arising with the replicated execution, we define the process and the execution model.

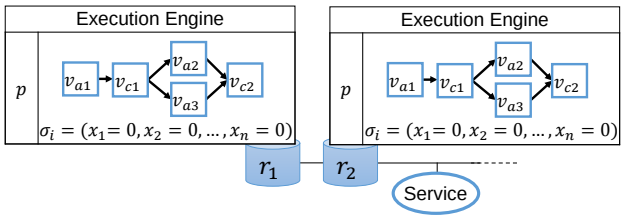

Fig. 1. Replicated process execution depicting two replicas and one service.

A. Process Model

The process model describes the logic of a process. It is defined by a directed acyclic graph $P = (V, E, \Sigma)$ consisting of process vertices V defining atomic blocks of the process logic called *actions*, directed edges $E : V \times V$ defining the execution order of the actions, and the internal state Σ containing the variables needed during process execution. A predominant number of process description languages is using graphs as the basis of modeling processes [13], [16]. This allows to transfer our concepts to most of the process description languages used today. Our graph model allows conditional branching (XOR), where only one of multiple branches is executed. Our model also supports concurrency (AND), where multiple branches are executed in parallel on disjoint subsets of the internal state Σ (which is a common assumption [13]). Therefore, the set of vertices V consists of two subsets $V = V_C \cup V_A$. The first subset is the set of control vertices V_C, where each control vertex $v_c \in V_C$ has a specific branching implementation, i.e., $v_c \in \{\textit{AND-Split}, \textit{AND-Join}, \textit{XOR-Split}, \textit{XOR-Join}\}$. The second subset is the set of actions V_A, where each action $v_a \in V_A$ might specify an operation s_{v_a} to execute. The action, however, does not implement the operation. Instead, it interacts with a service providing the operation s_{v_a} as defined in the execution model.

B. Execution Model

The Execution Engine running on each computing node is able to interpret the above defined process model. Any device or client can issue the execution of a process model P. Then, the Execution Engine of the respective computing node instantiates the model resulting in a process instance p. This also sets the initial state $\sigma_i \in \Sigma$, which is identical for all instances $p \in P$. Now, the Execution Engine can start executing the vertices according to their execution order and the specified branching.

During execution, the process might interact with services to use the operations provided at the service's interface. The execution of an operation is triggered by sending a request message to the respective service. During the operation's execution the service might change its state or perform a write (e.g., on database entries). This state is not in direct control of the Execution Engine and we will summarize such state as the *external state*. After execution, the service sends a reply containing the result of the operation's execution.

Operations that possibly write to external state are called *write operations*. All other operations might only read external state. The requests to the operations are called *write requests* or *read request*, respectively. This allows to interact with any service being compatible with the request-reply interaction pattern, e.g., a database service for issuing read and write operations. For simplifying presentation, we delay discussing the support of arbitrary interaction patterns (cf. Sec. V-C).

Any execution of an action might transfer the internal state σ into a new state σ', which we denote by $\sigma \rightarrow_{v_a} \sigma' | \sigma, \sigma' \in \Sigma \wedge v_a \in V_A$. Each action execution is atomic and, thus, either commits or aborts. For now, we additionally assume that each action has a compensation handler. The execution of this handler reverses a committed execution including the effects on the external state. An action's execution $\sigma \rightarrow_{v_a} \sigma'$ can, however, only be compensated if all executions of causally succeeding actions (i.e., action executions using σ') are either aborted or compensated. For simplicity, we refer to a write operation issued by a process as an *effective write* iff it is performed successfully and is not compensated. We treat actions without compensation handlers in Sec. V-C.

Each replica also maintains a log on stable storage. The atomic execution of an action writes the respective compensation handler with all necessary information (e.g., variable values of σ) to the log. This model of atomic actions and compensation handlers conforms to open nested transactions [17], Sagas [18], and business processes [16]. In general, actions can be grouped into an atomic block, where all or none of the actions are executed [16], [18]. Although we support this by treating the block as a single action, we will not further discuss this concept.

III. Correctness of Replicated Executions

In the following, we will define a property called one-copy consistency that ensures the correctness of the replicated execution of process models. This property is independent of the type of consistency achieved by the underlying process model. It ensures that the replicated execution of a process model has the same effects on the external state as some non-replicated execution of that model.

Axiom 1: The non-replicated execution of a process model P 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 approaches for verifying that a process model is correct, e.g., [13], [19], and any of these can be applied on the process model.

The effects of a process execution on the external state are solely determined by the interactions of this execution with the services. In particular, we distinguish between input and output events of a process execution. For each request (read or write) sent by the process, a reply message is returned to the requestor. Clearly, received replies typically influence the future behavior of the process and, thus, need to be considered as input. On the other hand, the effective write requests of a process modify the external state and, thus, need to be considered as output.

Informally, the replicated execution of a process model P is considered to be one-copy consistent if there exists some non-replicated execution of P that takes the same input and generates the same output. Before we can define one-copy consistency more precisely, we have to take a closer look at the input dependencies of write requests. Fig. 2 shows a process

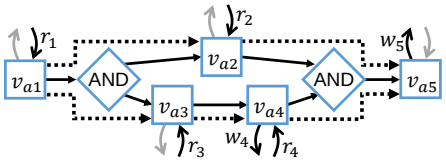

Fig. 2. Execution of an exemplary process.

execution consisting of actions v_{a1} to v_{a5}, where actions v_{a3} and v_{a4} are concurrent to action v_{a2} due to an AND-split vertex. The dotted arrows originating from v_{a1} indicate that the internal process state produced by v_{a1} is made available to activities v_{a2} and v_{a3}, while the dotted arrows originating from v_{a2} and v_{a4} indicate that the internal state produced by v_{a2} and v_{a4} is merged and made available to activity v_{a5}. Obviously, a write operation may be impacted only by input events in its causal history. For example, write request w_4 may be influenced by reply events r_3 and r_1, while w_5 depends on r_4, r_3, r_2, and r_1. We will use the term Causal Reply History (or *CRH* for short) to denote the reply events a given write request depends upon.

Definition 1: **Causal Reply History**
Let w be a write request issued by an execution of process model P. $CRH(w) = (I, \rightarrow)$ comprises the set of reply events I that causally precede w in the execution of P. The events in I are partially ordered according to their causal relationship $\rightarrow$ in the execution of P.

That is, $CRH(w)$ defines the input events on which w depends and their causal dependencies. A pair of concurrent events (i.e., with no defined ordering) can be consumed in any order without affecting w (because concurrent branches operate on disjoint subsets of the internal state, cf. Sec. II). In our example above, set I of $CRH(w_5)$ is $\{r_1, r_2, r_3, r_4\}$ with the partial ordering $r_1 \rightarrow r_2$ and $r_1 \rightarrow r_3 \rightarrow r_4$. For instance, r_2 and r_3 are concurrent and, thus, the order of their consumption does not affect w_5.

Definition 2: **One-copy consistency**
Let $E^R(P)$ be the replicated execution of process model P. $E^R(P)$ is one-copy consistent iff there exists some non-replicated execution $E^N(P)$ fulfilling the following conditions:

1) $O^R = O^N$, where O^R and O^N denote the sequence of effective write requests of $E^R(P)$ and $E^N(P)$, respectively.
2) $\forall w^N \in O^N, w^R \in O^R : w^N = w^R \Rightarrow I^N = I^R \wedge \rightarrow^N \subseteq \rightarrow^R$, where $CRH(w^N) = (I^N, \rightarrow^N) \wedge CRH(w^R) = (I^R, \rightarrow^R)$

The first condition requires that $E^R(P)$ generates the same sequence of effective writes as some non-replicated execution $E^N(P)$, i.e., the two executions are consistent in terms of their output. The second condition requires $E^R(P)$ and $E^N(P)$ to be also consistent in terms of their input dependencies. Assume a simple process that reads price information from a product server, computes a sales discount and requests the sales server to add the new offer. Without the second condition, a replicated execution generating any offer would be considered consistent.

Theorem 1: A replicated execution of a process model P that is one-copy consistent is correct.

The proof of the theorem follows directly from the one-copy consistency property and the assumption that any non-replicated execution of a process is correct.

IV. Rules for Replicated Executions

In the following, we define rules that ensure one-copy consistency for the replicated execution of processes. In general, we can use two replication mechanisms: *passive* and *active replication* (cf. Fig. 3). Passive replication means that each action is only executed by one replica, while the other replicas wait for the newly produced state to be replicated. When using

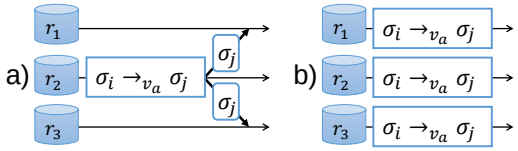

Fig. 3. a) Passive replication b) Active replication

active replication, all replicas actively execute the actions of a process rendering a state transfer unnecessary. However, which replication mechanism should be used depends on the class of action that is executed. Therefore, we will provide the criteria for classifying actions before specifying the execution rules.

In specific, the actions are classified according to the effect on the external and internal state. The first criteria specifies whether an action writes to external state. An action writes to external state if it calls a write operation of a service. When actively replicating a *write action*, all but one execution of the action need to be compensated to ensure the same effective writes as a non-replicated execution. We can avoid the high compensation overhead by using passive replication in the first place. For all the actions that do not call write operations, there is no effect on the external state. According to one-copy consistency (cf. Def. 2) actively replicating *non-write actions* does not require any execution to be compensated. If, however, a causally preceding write action has to be compensated, the non-write action still needs to be compensated before (cf. Sec. II). Even if that is the case, compensating a non-write actions is cheap because we only need to rollback the internal state.

The second criteria specifies whether the action execution changes the internal state deterministically. Consider an operation for booking a flight ticket. Even when sending multiple identical requests, the replies might vary because the plane has a limited number of seats. From the process's view, the action is *non-deterministic*. If we want to ensure that the replicas

have the same internal state, we need to use passive replication. In contrast, consider an operation offering a function, where the request includes all input parameters. Here, the operation's reply will always be the same when sending identical requests. The behavior appears *deterministic* to the process execution. Hence, actively replicating a deterministic action results in the same internal state on all replicas.

In conclusion, both passive and active replication are used depending on the class of the respective action. Thus, we will define rules ensuring one-copy consistency for passive and active replication. To define these rules, we first need to take a closer look on how a replicated execution takes place. In Fig. 4b, a process is executed using passive replication. Here, only one replica is executing the process at any point in time. The replica executing v_{a2} fails and another replica restarts the execution of v_{a2} to provide availability. Thus, the first execution of v_{a2} needs to be compensated later if v_{a2} is a write action.

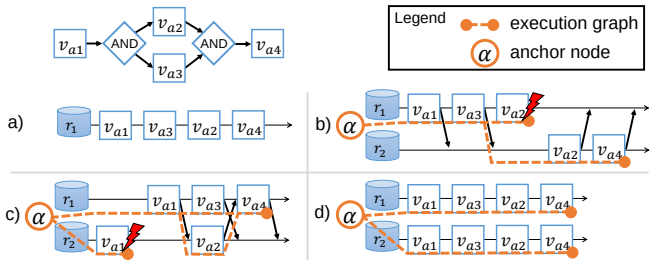

Fig. 4. a) Non-replicated execution, b) passive replication, c) passive replication exploiting concurrency, and d) active replication. Here, the execution of an action $\sigma \rightarrow_{v_a} \sigma'$ is abbreviated by v_a.

In Fig. 4b, the chain of action executions and state transfers define a causal execution order. An action v_a is directly causally preceded by the action that produced the internal state that v_a is using as input. We capture this causal order as a directed acyclic graph $G = (N, L)$ called *execution graph*. The set of nodes N represent the executed vertices, where the graph is connected through an additional anchor node. The set of edges $L : N \times N$ captures the execution order and defines the *executed before* $<_G$ relation. As depicted in Fig. 4c, AND vertices might also split and join branches of the execution graph.

Informally, a replicated execution is one-copy consistent if the execution graph contains a subgraph that conforms to the process model and all actions not contained in the subgraph are compensated. We will now define this more formally.

Definition 3: **Execution Rules**
Let $G = (N, L)$ be an execution graph of a replicated execution of process $P = (V, E, \Sigma)$. There is exactly one subgraph $G_s \subseteq G$, where $G_s = (N_s, L_s)$, such that:

1) The causal order of the actions in G adheres to the specified execution order and branching of P.
2) The subgraph G_s is a completed process execution of P.
3) None of the actions in G_s are compensated.
4) All write actions v_a in the execution graph G, which are not in the subgraph G_s (i.e., $v_a \in N \wedge v_a \notin N_s$), are compensated. However, an action v_a is only compensated after all succeeding actions in the execution graph G (i.e., $\forall v_x \in N : v_a <_G v_x$) were already compensated (cf. Sec. II).

Now, consider an actively replicated execution as depicted in Fig. 4d. All replicas execute independently and, thus, the execution graph contains one branch for each replica. Even though active replication saves the need of transferring state, it implies the overhead of compensating all but one execution (if the process comprises write actions). Then, the execution that is not compensated is the subgraph G_s. Consequently, the defined execution rules apply to active replication as well.

Theorem 2: A replicated execution that adheres to the Execution Rules (cf. Def. 3) is one-copy consistent (cf. Def 2).

Proof sketch: Since only the write actions of G_s are not compensated (Def. 3.3 & 3.4), only G_s issues effective writes. Def. 3.1 & 3.2 require G_s to be a valid execution of P, which implies the action execution order is the same as some non-replicated execution. Thereby, the same sequence of effective writes is already ensured. The process produces the content of the writes from the input. Thus, the content of the write request is the same as some non-replicated execution if the *CRH*s of the writes is also same. For showing the latter, consider the execution graph. When replicating state, the state receivers overwrite their own internal state by which they inherit the *CRH* that produced this internal state. Concurrent AND-branches execute on disjoint subsets of the internal state (cf. Sec. II) and only replicate this subset. Thus, any AND-branch has its own *CRH* and these *CRH*s are only merged by AND-joins. Through the *CRH* inheritance, the *CRH*s of the writes in G_s have to be the same as in some non-replicated execution.

V. Replication Protocol

So far, we defined consistency of replicated process executions but did not discuss availability. For providing high availability in the presence of f concurrent failures, the protocol has to fulfill two complementary properties, called *decidability* and *progress*, and a third property called *compensation independence*. Decidability requires that the protocol is able to decide which execution to keep and which to compensate if multiple replicas executed the same part of a process. Regarding the execution rules (cf. Def. 3), decidability correlates to deciding whether an execution is part of the subgraph G_s. In other words, decidability ensures the safety of our protocol.

Liveness is ensured by the progress property, which requires the ability to always progress the process execution. This includes that the state produced by an execution which is decided to be part of G_s has to be available when f failures occur because the state is needed for continuing the execution.

Now, assume a part of the process was executed twice, e.g., due to network partitioning. Thus, the protocol decides that one execution is not part of G_s meaning this execution has to be compensated. The compensation independence property requires that the replica can compensate its execution independent of all other replicas. A dependency arises if an execution, e.g., of replica r_1, has to be compensated but the produced state is used for proceeding the execution on another replica. Replica r_1 can only compensate after all executions building on the produced state are aborted or compensated because compensation has to be performed in the reverse

causal execution order (cf. Def. 3.4). If a single replica might have used the produced state and is failed, r_1 is blocked from compensating its execution. Allowing compensation dependencies might lead to cascades of blocked compensations, which is obviously undesirable. In other words, compensation independence requires that a replicated state is only used for continuing the execution after the protocol decided that the execution that produced this state is part of G_s (because then the execution will never be compensated).

Additional to the availability properties, we have performance goals, which are to minimize failover time, compensation cost, and bandwidth overhead. However, for simplifying the presentation of our protocol, we will first describe a passive replication protocol without supporting AND- and XOR-branching, where we focus on fulfilling the execution rules (cf. Def. 3) and the availability properties. Subsequently, we present a hybrid protocol combining active and passive replication, where we discuss the performance goals in depth. Finally, we extend the protocol to support non-compensable operations, complex interactions, and AND/XOR-branching.

A. Passive Replication

The goal of passive replication is that the actions of a process are executed only once if no failures occur (cf. Sec. IV). In terms of the execution rules (cf. Def. 3) that means in the failure free case the execution graph G only consists of G_s. For ensuring decidablity and progress, the protocol frequently replicates the state on at least $f+1$ replicas. When these replicas acknowledge the reception, the corresponding execution is decided to be in G_s manifesting the progress of the process execution. Using $2f+1$ replicas, we request $f+1$ replicas for their state to recover the most recent replicated state after a failure. Then, there is always at least one replica knowing about the latest progress. In the following, we will discuss the protocol in further detail.

The protocol is started when a client requests the execution of a process P. The computing node receiving the request, selects $2f+1$ nodes (including itself) that become replicas for the execution of P. The replica that received the client's request is the initial *planner*, which decides the first *master*. In general a planner is responsible for deciding the next master. The master is in turn responsible for executing the actions of the process and replicating the produced state to the *followers*.

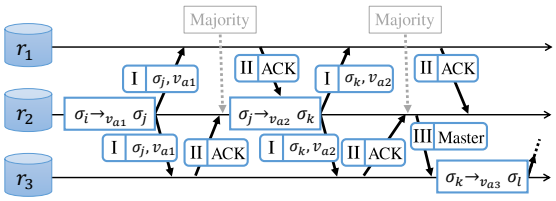

Fig. 5. Passive replication

The decided master executes the first action, where the atomic execution of the action writes the respective compensation handler including all necessary information (e.g., variable values of the internal state) to the log on stable storage. This ensures the replica loses neither the knowledge that it executed the action nor the information for a potentially needed compensation. However, we do not require the internal state to be kept on stable storage. Instead, the master replicates the produced state by sending it to all replicas. The state replication message includes the identifier of the last executed action to inform the followers about the execution progress, as depicted in Fig. 5(I). A follower can either reject or acknowledge the received state (II). A state might be rejected if there is already another execution of the same action that is decided to be in G_s, which might happen in case of failures. The master waits for a majority of ACKs before continuing the execution for three reasons. Firstly, the state might be rejected as mentioned above. This means the respective execution is not in G_s and needs to be compensated. Continuing this execution with a rejected state means that this continued execution also has to be compensated – increasing the compensation overhead. Secondly, when not enforcing the replication on $f+1$ replicas before continuing, the state might reach less than $f+1$ replicas. In case of a failure, a majority might restart the execution without this state, losing the execution progress. This not only increases the compensation overhead but also contradicts the progress property. Finally, the master switches to the planner role after executing the action, deciding either itself or another replica to be the next master by sending a message (III).[2] When not waiting for the majority of ACKs, the execution can continue on another replica building on an execution, which might need to be compensated later. This creates compensation dependencies, which contradict the compensation independence property. In conclusion, the availability properties require waiting for a majority of ACKs before continuing the execution.

However, replicating the state after each action execution implies a high overhead. It might, therefore, be beneficial to group consecutive actions. Then, the state is only replicated after the whole group has been executed, which reduces the overhead. On the other hand, we might lose more execution progress when a master fails. However, before we discuss the impact of grouping in further detail, we will first present the complete protocol. For now, assume that the grouping of the process is predefined and coupled to the process model.

1) Handling Failures: To provide availability, we need to handle master, follower, and abort failures. In case a *master fails*, we elect a planner that determines a new master. To invalidate the old master and the respective execution, we use the concept of views [10], where the initial view of all replicas is 0. The master annotates the logged compensation handlers with the view number in which the respective action was executed. The master also annotates the replicated state with this view number. The followers only accept a state if their local view is smaller or equal, i.e., outdated master executions are rejected (ensuring decidability). When accepting, the followers log the view for which they received a state together with an identifier of the action which produced the state.

[2]There are many methods for deciding the next master. For example, the master could be chosen based on the load or based on the latency to the service operation needed by the next action. We support any arbitrary method.

For electing a planner, we can use any arbitrary election protocol. The one we describe in the following is based on the election protocol of Raft [11]. The master continuously sends heartbeats and if a follower does not receive a heartbeat within time t_h (plus a random back-off time t_b to prevent concurrent time outs), the follower increments its view and broadcasts a view change request as depicted in Fig. 6(I). Receivers change to this view and vote for the sender if the proposed view is higher than their local view (II). The view change initiator becomes planner when it receives $f+1$ votes ensuring that there is only one planner for the view change. Additionally, the votes include the most recent group that was completely executed, the view in which it was executed, and the respective internal state of the voter. The planner can determine the most recent group execution from the vote messages (III), i.e., the most recent group executed in the highest view. The according internal state has to be replicated on a majority of replicas before the state can be used for continuing the execution for the same reasons as during normal operation. Thus, the planner replicates the state and then decides a new master (IV-VI).

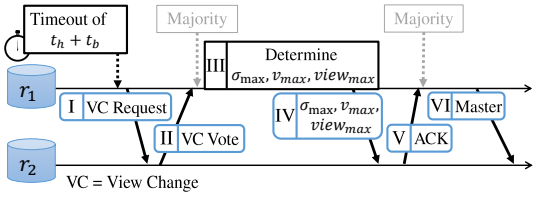

Fig. 6. Interaction pattern for a successful view change.

When the outdated master recovers, it requests the process and the grouping from the other replicas. It also reads its log by which it knows which actions it already executed. The master needs to verify if the state it produced before failing survived the failure, i.e., if the state was used for continuing the execution after the view change. Therefore, it requests if at least a majority of replicas has the according execution in their logs (identified by action identifier and view). If this is not the case, the respective group is already re-executed in a higher view and the outdated master compensates its execution. Here, the compensations are also logged for knowing the compensation progress in case of another failure. Afterwards, the recovery is finished. In case a *follower fails*, the recovery is exactly the same except that no compensation is necessary.

Finally, to handle *abort failures*, a master, which for example receives an abort from a called service operation, first will restart executing the respective action. In case of another failure, it compensates the whole group and triggers a view change after which a new master will re-execute the group.

B. Hybrid Approach

When using passive replication, a master failure implies changing the view and re-executing the respective group. In the worst case, the failure occurs right before the state is replicated leading to a high failover time. Moreover, the concept of grouping, which is used to reduce the state replication overhead, increases failover time further when increasing the number of actions that are encapsulated within a group.

Active replication overcomes this problem. Here, all replicas are actively executing the process. On the one hand, we do not need to re-execute in case of a failure because all replicas are executing already – proactively providing re-executions. On the other hand, this implies high compensation cost because after the whole process was executed, the replicas agree on one of the executions, while all other executions are compensated.

In conclusion, active replication is the mechanism of choice for parts of the process, where the compensation cost is low or non-existent. Other parts of the process are executed using passive replication. To enable this combination of active and passive replication in one process execution, we extend the previously defined passive replication protocol. The protocol now can switch the replication mechanism per group, i.e., any group uses either active or passive replication. We also determine a master for actively replicated groups, making a replica's execution of a group uniquely identifiable by the view number (as in the case of passive replication). The executions of the other replicas are proactive backup executions, where the compensation handlers are logged without a view number (identifying these as backup executions). In the case of a master failure, the new master uses its backup execution for the new view, where the view number is added to the corresponding log entry. Now, we can exploit the benefits of both replication mechanisms when grouping the process accordingly. The process designer, as a domain expert, could specify the grouping at design time. We, however, cannot assume that grouping is provided for every process. Thus, we present a method for automatic grouping.

We aim to keep compensation cost and state replication overhead low, while ensuring that the failover time is below a threshold t_t, which is specified by process designer or user. Let $c : V \to \mathbb{N}$ be the function specifying the compensation cost of a vertex and $t : V \to \mathbb{N}$ specifying the execution time in *ms*, where these values are either provided by the process designer or monitored and learned from past executions.

A group g has to fulfill the following rules: 1) if g is actively replicated: $\sum_{v \in g} c(v) = 0$ and 2) if g is passively replicated: $t_v + \sum_{v \in g} t(v) \leq t_t$, where t_v is the time needed for a view change. The first rule ensures that groups are only actively replicated if the compensation cost of the group is zero. This is especially the case for deterministic, non-write actions, which do not require compensation or state replication because they produce the same internal state on all replicas (cf. Sec. IV). In case of passive replication, the second rule ensures that the sum of the time needed for changing the view and re-executing the group is below t_t. Thus, when a master fails, the failover time is below t_t. Note that the provided method of grouping is simplistic and can, for example, be improved by considering the failure rates of the replicas. Providing optimal grouping is, however, not the scope of this paper and left for future work.

C. Extending the Protocol

The above described protocol can only handle a sequence of actions calling compensable operations. In the following, we

extend the protocol to incorporate non-compensable operations, more complex interaction patterns, and all control vertices.

1) Incorporating Non-compensable Operations: So far, we assumed all write operations to be compensable. This, however, might not be the case. For example, some cheap air lines do not allow to cancel bookings. Thus, we subdivide write actions further into *compensable* and *non-compensable*. Non-compensable actions need to execute the called operation exactly once. Therefore, the preceding group ends directly before the non-compensable action. This enforces that the state used as input for the non-compensable action was produced by an execution of G_s and is never lost. To make the requests to the service operation repeatable, the master announces the used service and an unique interaction ID (IID). Only after a majority agreed, the master starts the execution. In case of f failures both the specific service and the IID are available for repeating the request. We then require the service to filter duplicates using the IID and simply repeat the reply. Alternatively, we could integrate a middleware layer that is responsible for filtering.

2) Incorporating All Interaction Patterns: Until now, we only considered interactions realized by a request-reply pattern within a single action. However, there might be interaction partners (e.g., a service, an application, or a user) requiring more complex interaction patterns. Therefore, we allow actions that only receive or only send a message and construct the required pattern with these actions. The replicas again agree on an IID that is used to identify the complex interaction allowing any replica to resume the interaction. For not losing messages that an interaction partner sends to a failed replica, the interaction partner broadcasts messages to all replicas.

3) Incorporating Control Vertices: To support XOR-branching, we require the XOR-split logic to be deterministic. If a non-deterministic decision is desired, this has to be realized by an preceding action. This allows to handle XOR-splits like deterministic, non-write actions. Otherwise, XOR-branching is congruent to a sequential execution requiring no additional mechanisms. For AND-branching, we require that all concurrent branches are executed on the same replica at any point in time. Then, a group spans across all branches and specifies which actions of which branch shall be executed. It is also possible to execute the branches on different replicas. We, however, omit the description due to space constraints.

D. Correctness

To show that the hybrid approach fulfills one-copy consistency, we show that it adheres to the execution rules (cf. Def. 3). Any execution builds an execution graph G, where G is constructed through executing the groups $g_1, g_2, ..., g_n$ into which we divided the process P. Def. 3.1 is trivially fulfilled because the group and action execution order is based on P. After the execution of a group g, the replicas decide which execution of g is in G_s. This is always the execution of g being executed in the highest view. The state of the execution that was decided to be in G_s is the one that is used for executing the next group. By induction, this constructs an execution path defining G_s of which no action is compensated, satisfying Def. 3.3. Since our protocol only terminates once g_n is executed, Def. 3.2 is also fulfilled. Additionally, the highest view of execution of any group g is written to the log of at least $f+1$ replicas because a majority of replicas have to agree (ACK) for making the decision that an execution is part of G_s. Because all executions are logged with the corresponding view number (or no view number in case of backup executions), a faulty replica will, upon recovery, always be able to identify an execution it needs to compensate by having executed in a lower view than the one that was decided (or no view), which satisfies Def. 3.4.

VI. Evaluations

In this section, we evaluate our replication protocol in terms of replication overhead, scalability, and performance in the presence of failures. Therefore, we implemented a prototypical Execution Engine including our replication protocol, where we used Apache MINA[3] for realizing the communication between the replicas. To show the protocol's benefits, we deploy the prototype in two setups. First, we evaluate the protocol in a cloud environment (Amazon EC2), where we have dedicated resources and low latencies. Then, we evaluate the protocol in a heterogeneous environment (PlanetLab Europe), where we have shared resources and varying latency and bandwidth.

A. Amazon EC2

In Amazon EC2, we evaluate up to a replication degree of 17 using 17 t2.micro instances (1vCPU, 1GB RAM) each running our prototype. We generate random processes consisting of 100 actions, which is in the range of typical process lengths [20]. The actions' execution time is set using a Gaussian distribution with a standard deviation of 500 ms.[4] The compensation cost is set based on the class of action. Any non-write action has a compensation cost of 0 and occurs with a probability of $\frac{1}{4}$. All other actions' compensation cost is set by using a uniform distribution between 0 and 100. We evaluate our protocol including our cost model, where we set the time threshold t_t to 500 ms and 10000 ms. For comparison, we also evaluate active and passive replication as well as a non-replicated execution. Overall, we measured over 13,000 process executions.

First, we evaluate the bandwidth overhead during normal operation, which is incurred by replicating the state. Thus,

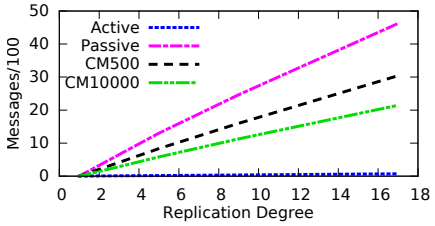

Fig. 8. State replication message count of the replication approaches

we measure the needed state replication messages against

[3]https://mina.apache.org

[4]For circumventing monitoring service execution times including latencies to all replicas, we simulate service calls. In PlanetLab, we actually call prototypical services but, in turn, do not evaluate different cost model settings.

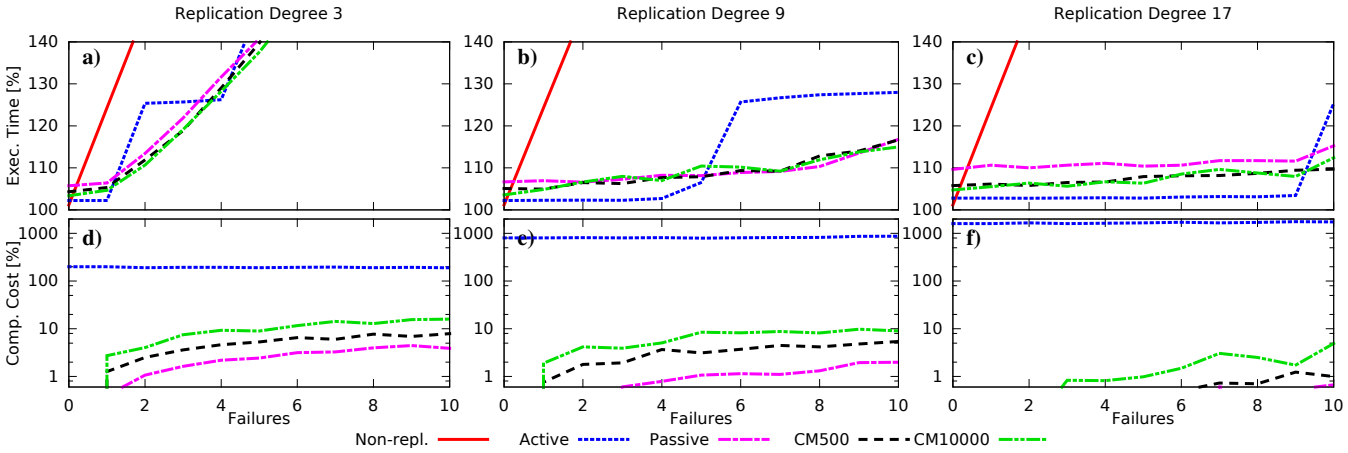

Fig. 7. Average execution time and compensation cost of the different replication approaches in the presence of failures in Amazon EC2

the replication degree. Fig. 8 shows that all approaches are scalable because the message count increases linearly with the replication degree. However, passive replication imposes a big overhead compared to active replication. Grouping actions with our cost model (abbreviated by *CM*) reduces this overhead. With $t_t = 500\,\text{ms}$ (i.e., *CM*500), we save around $\frac{1}{3}$ of the bandwidth compared to passive replication. When increasing t_t (and, thus, the groups size) to 10000 ms, we even save more than half of the bandwidth. However, solely considering bandwidth, active replication seems to be the best approach.

In Fig. 7, we evaluate execution time and compensation overhead in the presence of failures, where we randomly inject failures during execution. With a mean time to recovery of 10 s, failures are short compared to typical failures, which even last up to multiple days [21]. However, replication is obviously advantageous for long lived failures. In contrast, we show that our protocol is still worth its overhead when failures are short lived. To make different processes comparable, we normalize the measured execution time by the minimum time for executing the whole process (i.e., by the sum of all action execution times). An execution time above 100 % in the failure free case is due to replicating the state and waiting for a majority of ACKs. The increase of the execution time with an increasing number of failures shows the failover time of the different approaches. Respectively, we normalized the compensation cost, where a compensation cost of 100 % is equal to compensating all actions of the process once.

When using active replication, the execution time with up to f failures is almost as low as a non-replicated execution (cf. Fig. 7a-c). The sudden increase of execution time with $f+1$ failures is due to the implementation of active replication. A replica always finishes its process execution before participating in the decision which execution to use and which to compensate. This decision requires $f+1$ replicas. Thus, when $f+1$ failures occur on different replicas, the decision is delayed by the mean time to recovery (i.e., 10 s). Although this could be improved implementation-wise, the compensation cost anyways renders active replication infeasible. Even in the absence of failures the compensation cost of active replication is tremendous, i.e., 200 % for replication degree 3, 800 % for replication degree 9, and 1600 % for replication degree 17 (cf. Fig. 7e-f).

Passive replication does not imply compensation cost in the failure free case. However, the execution time overhead implied by the replicating the state and waiting for a majority of ACKs is noticeable. For replication degree 3, the execution time is delayed by 5 % compared to a non-replicated execution. For 17 replicas, the delay even increases to 10 %. Our cost model, reduces the overhead considerably, i.e., $t_t = 500\,\text{ms}$ saves $\frac{1}{3}$ and $t_t = 10000\,\text{ms}$ saves more than a half in the failure free case. However, this means that groups contain more actions, which – in the case of a failure – need to be re-executed. Thus, the failover-time increases when increasing t_t, which can especially be observed for the replication degree 9. Here, the execution time advantage of the failure-free case gets smaller or even vanishes with an increasing amount of failures. Surprisingly, the execution times of the cost model executions rarely increase beyond passive replication – even for $t_t = 10000\,\text{ms}$. This means that even though the failover time increases (because more actions need to be re-executed), it is mostly more efficient than replicating the state more often. This, however, is also due to our method of grouping, where all actions that do not imply compensation cost are grouped (independent of t_t). At the end of any group the state is replicated on $f+1$ replicas, which means the progress is never lost. With these groups already given (before additionally grouping using t_t), it is then more efficient (in terms of the execution time) to chose a high value for t_t, i.e., to re-execute more actions than replicate more often. When, however, considering the compensation cost, the group size has a big impact. Here, passive replication implies the smallest compensation cost because the fewest actions need to be re-executed and, hence, compensated. When increasing the group size (i.e., t_t), the compensation cost – especially with an increasing number of failures – increases significantly. However, with an increasing replication degree, the average compensation cost for the same number of failures decreases. The reason is that a higher replication degree makes it less probable that the same number of failures will effect the current group master. Instead, the failures mostly effect followers.

In conclusion, any replication method is worth its overhead in the presence of a single failure when striving for high availability. We showed that our protocol is scalable and implies low overhead. Additionally, setting t_t in the cost model allows to minimize the compensation overhead or the bandwidth and execution time overhead, or to achieve any trade-off in between.

B. PlanetLab

For evaluating our protocol in a geo-distributed setting, we perform measurements on PlanetLab Europe. In PlanetLab, resources are shared, which means performance, latency, and bandwidth vary. The goal is to show that the protocol is also beneficial in such a challenging environment.

First, we evaluate the time overhead the state replication implies for a replication degree up to 33. Fig. 9 depicts the time from sending the state replication message until receiving a majority of ACKs and continuing the execution. In PlanetLab,

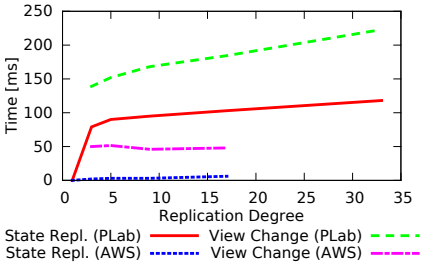

Fig. 9. The time overhead of state replication and the time needed for a view change in PlanetLab (PLab) and Amazon EC2 (AWS)

one state replication with replication degree 3 on median takes around 80 ms. From replication degree 5 to 33 the time increases linearly, staying below 120 ms, which shows the scalability with respect to the replication degree. The time needed for a view change starts at around 140 ms for replication degree 3 and increases linearly to 220 ms for a replication degree of 33. In Amazon EC2, both the overhead for replicating state (below 10 ms for replicaiton degree 17) and view changes (around 50 ms for replication degree 17) are significantly lower. However, when considering the distances between the geo-distributed replicas in PlanetLab compared to the Amazon EC2 set-up, where all replicas are placed in one data center, the overhead seems suprisingly low.

For evaluating the protocol in the presence of failures in PlanetLab, we generate a process containing 100 actions. The actions call prototypical services also placed in PlanetLab. We only evaluate the worst-case scenario in terms of execution time overhead, where the cost model threshold (t_t) is set so low that each group contains a single action. Fig. 10 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 need to replicate state, it performs better in the failure-free case. However, even with a single failure, the replicated executions perform better. A replication degree of 3 can already tolerate one failure. For

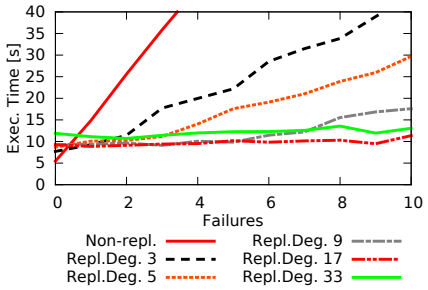

Fig. 10. Execution time in the presence of failures in PlanetLab

more failures, the execution time linearly increases. However, more slowly than in the non-replicated case. 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 becoming obvious for 33 replicas. On the other hand, the high replication degrees of 17 and 33 remain completely unaffected even in the presence of 10 failures.

In conclusion, the failure measurements show that the replication protocol allows the process execution to mask $f+1$ failures. Additionally, we showed that the execution time overhead implied by the protocol is negligible when only a single failure occurs even with worst-case grouping (in terms of execution time overhead) and a geo-distributed setup like PlanetLab with varying latency and bandwidth.

VII. Related Work

Providing availability is an important requirement for any network related system. Replication is a widely used mechanism increasing availability by redundancy [7]. Data replication ensures availability, for example, by using eventual consistency. Here, working on a single replicas is possible, while consistency across all replicas is ensured eventually [22]. However, data replication targets one level below process executions because processes operate on that data. Thus, we developed a new consistency definition and concepts for process replication.

Another approach is passive replication, where one master replica executes the process and transfers the execution state to the backup replicas [8]. This, however, implies overhead in terms of bandwidth for transferring the state, as well as a delay until the transfer is complete. Active replication approaches overcome these limitations by actively execution all actions on all replicas. One such active replication approach is state machine replication [9]–[12]. Here, a set of $2f+1$ replicas resemble a service, where the idea for making the service f resilient is to keep the internal state of all replicas consistent. In this respect, state machine replication is similar to our protocol. There, however, are two fundamental differences. Firstly, state machine replication assumes all replicas to be state machines (i.e., actions are assumed to be deterministic). State machine replication ensures that the client requests (i.e., the actions) are serialized, leading to the same execution order and, thus, to the same state across all replicas. In contrast, we consider a process, which might include calling non-deterministic actions, invalidating the state machine replication

assumption. Additionally, state machine replication assumes that the actions are executed in isolation on all replicas. This, for example, means that state can be recovered by replaying the requests. Process executions, however, trigger write operations of services. We, therefore, need to ensure that such operations are executed only once (or compensated otherwise). Thus, we solve a completely different problem, where we re-use some concepts which help solving our problem, such as views [10] and time-out based election [11].

There are also techniques combining active and passive replication to minimize the computation overhead of active replication and still achieve high availability [14], [15]. Heinze et al., for example, estimate recovery time to decide whether active or passive replication is needed to ensure a user specified latency in case of a failure [14]. This is related to our cost model used for grouping, where we, however, use grouping for reducing the state replication overhead and also take into account compensation cost. Moreover, all these approaches again assume deterministic actions for active replication.

Transaction management techniques consider also clients sending requests (transactions) to the system [23]–[26]. However, the changes that a transaction performs on the system's state are only made permanent once the transaction commits and a reply is send to the requestor. In contrast, our process model is based open-nested transactions [16]–[18], where executing an write action of the process directly changes the external state. Thus, we have to consider the consistency of the external state when replicating. Additionally, systems like transaction management systems consider any message to external state (i.e., replies to clients) to be non-compensable [23]–[27]. Assuming all communication to be non-compensable is, however, too restrictive for processes, where compensation handlers are usually available [16]–[18].

Business processes usually provide fault-tolerance by integrating fault handlers and recovery mechanisms in the process model [28]. This, however, means that faults of the infrastructure (e.g., computing node failures) are not handled. Other approaches specifically tailored to business processes are only targeting service failures [29] or require a special kind of language for process modeling [30] (e.g., Declare [31]). In contrast, the protocol presented in this work incorporates all kinds of infrastructure failures and is applicable to any graph based process description language, which is the predominant method for modeling processes [13], [16].

VIII. Conclusion

We presented a novel replication protocol to ensure high availability for process executions in a heterogeneous and distributed environment. The key idea is to manage several replicas of a process such that the outcome is identical to a non-replicated execution. We formally defined this *one-copy consistency* and presented a protocol adhering to this definition while ensuring high availability. In our evaluations on Amazon EC2 and PlanetLab, we showed that our protocol significantly increases availability in the presence of failures, while introducing low overhead. Moreover, the protocol proves to be scalable because the bandwidth overhead as well as the time overhead grow linearly with higher replication degrees.

Acknowledgment

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

References

[1] P. Ward and H. Zhou, "Impact of Information Technology Integration and Lean/Just-In-Time Practices on Lead-Time Performance*," *Decision Sciences*, vol. 37, pp. 177–203, 2006.

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

[3] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 181–192, 2013.

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

[5] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 350–361, 2011.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proc. ACM SOSP*, 2007, pp. 205–220.

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

[8] J. Lau, L. C. Lung, J. da Fraga, and G. Santos Veronese, "Designing fault tolerant web services using bpel," in *Proc. ICIS '08*, 2008, pp. 618–623.

[9] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 51–58, Dec. 2001.

[10] B. Liskov and J. Cowling, "Viewstamped replication revisited," MIT, Tech. Rep. MIT-CSAIL-TR-2012-021, 2012.

[11] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX ATC*, 2014, pp. 305–320.

[12] P. Marandi and F. Pedone, "Optimistic parallel state-machine replication," in *Proc. IEEE SRDS*, Oct 2014, pp. 57–66.

[13] M. Weske, *Business Process Management: Concepts, Languages, Architectures*, 1st ed. Springer, 2007.

[14] T. Heinze, M. Zia, R. Krahn, Z. Jerzak, and C. Fetzer, "An adaptive replication scheme for elastic data stream processing systems," in *Proc. ACM DEBS*, 2015, pp. 150–161.

[15] Z. Zhang, Y. Gu, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu, "A hybrid approach to high availability in stream processing systems," in *Proc. IEEE ICDCS*, 2010, pp. 138–148.

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

[17] G. Weikum and H.-J. Schek, "Concepts and applications of multilevel transactions and open nested transactions," in *Database Trans. Models for Advanced Applications*. Morgan Kaufmann, 1992, pp. 515–553.

[18] H. Garcia-Molina and K. Salem, "Sagas," in *Proc. ACM SIGMOD*, 1987, pp. 249–259.

[19] D. Bianculli, C. Ghezzi, and P. Spoletini, "A model checking approach to verify bpel4ws workflows," in *Proc. IEEE SOCA*, 2007, pp. 13–20.

[20] R. Dijkman, M. Dumas, and L. Garca-Bauelos, "Graph matching algorithms for business process model similarity search," in *Business Process Management*, ser. LNCS, U. Dayal, J. Eder, J. Koehler, and H. Reijers, Eds. Springer, 2009, vol. 5701, pp. 48–63.

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

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

[23] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distr. and Parallel Databases*, vol. 14, no. 1, pp. 71–98, 2003.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *Proc. ACM SOSP*, 2011, pp. 401–416.

[25] T. Kobus, M. Kokocinski, and P. Wojciechowski, "Hybrid replication: State-machine-based and deferred-update replication schemes combined," in *IEEE ICDCS*, July 2013, pp. 286–296.

[26] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2stm: Dependable distributed software transactional memory," in *Proc. IEEE PRDC*, 2009, pp. 307–313.

[27] D. Powell, M. Chérèque, and D. Drackley, "Fault-tolerance in delta-4," *SIGOPS Oper. Syst. Rev.*, vol. 25, no. 2, pp. 122–125, 1991.

[28] N. Russell, W. van der Aalst, and A. ter Hofstede, "Workflow exception patterns," in *Advanced Information Systems Engineering*, ser. LNCS, E. Dubois and K. Pohl, Eds. Springer, 2006, vol. 4001, pp. 288–302.

[29] T. Bach, M. A. Tariq, B. Koldehofe, and K. Rothermel, "A cost efficient scheduling strategy to guarantee probabilistic workflow deadlines," in *Proc. NetSys'15*. IEEE, 2015, pp. 1–8.

[30] D. R. Schäfer, T. Bach, M. A. Tariq, and K. Rothermel, "Increasing availability of workflows executing in a pervasive environment," in *Proc. IEEE SCC '14*, 2014, pp. 717–724.

[31] M. Pesic, "Constraint-based workflow management systems: shifting control to users," Ph.D. dissertation, Eindhoven Univ. of Techn., 2008.