Exploring the Search Space between Active and Passive Workflow Replication

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

***Abstract*—Today, workflows are executed in heterogeneous environments, such as the Internet, to automate interactions with services of business partners and other third parties. In such heterogeneous environments, device and communication failures occur frequently. These failures might delay or even stop the workflow executions rendering the workflow unavailable. For ensuring availability, workflow executions are replicated on multiple devices allowing the workflow execution to continue even if some of the devices fail. So far, two workflow replication mechanisms have been proposed: active and passive replication. While active replication incurs high cost in terms of compensation even for failure free executions, passive replication provides lower availability than active replication. In this paper, we propose a new replication mechanism called partition-tolerant replication that combines the benefits of active and passive replication. Our evaluations on OpenStack show that partition-tolerant replication imposes no compensation cost for failure free executions like passive replication while providing availability close to active replication.**

1. Introduction

Workflows have manifested as the de facto standard for managing and optimizing business processes [1]. A workflow defines the process as a set of activities and specifies the order in which these activities have to be executed through an ordering relation. Thereby, workflows can be represented as graphs, allowing the intuitive management and optimization of the underlying processes.

The automation of complex business operations requires workflows to orchestrate services that are offered by business partners and other third parties located on servers anywhere on the planet. Consequently, these service orchestrations need to be executed in heterogeneous environments, where devices and communication failures occur frequently. A failure of the device on which the workflow is executed might delay or even terminate the workflow execution. Thus, the workflow execution becomes unavailable through failures, which incurs cost. A study revealed that a single hour of unavailability can induce a typical cost of up to $6.48 million [2].

Often, businesses assume that migrating their systems to datacenters or the cloud solves all availability concerns. However, this is far from reality. A study showed that Microsoft datacenters on average experience more than 40 failures with end-user impact per day [3], [4], [5]. Also, wired connections that are usually assumed to be highly reliable suffer from frequent failures. For instance, the IP Backbone experiences failures at a median rate of 3000 s [4], [6]. Thus, availability is an important design consideration for all networked systems irrespective where these are deployed.

Workflow replication places multiple replicas of a workflow instance on different devices that are able to execute the workflow [1], [7]. Thereby, a failure or partitioning of one of these devices does not harm availability as the replicas on the other devices can continue the workflow execution.

In particular, there are two replication techniques: passive and active replication. Passive replication [1], [8], [9] elects a *primary* that executes the workflow. The other replicas, called *backups*, receive state updates from the primary. In case of a primary failure, the backups decide a new primary using majority election. The new primary then continues the execution from the received state.

However, since failures occur frequently, a situation where no majority is available might arise quickly. Regardless of the number of replicas, one partitioning plus one crash failure can divide the network such that no partition contains a majority. Then, passive replication is unable to elect a new primary rendering the workflow execution unavailable.

Active replication [1] overcomes this problem because it executes the workflow on each replica independently. Thus, the failure of a replica does not affect the execution of the other replicas. Upon finishing the workflow execution, the replicas agree on one of the executions and compensate all others – causing significant compensation cost. Especially undesirable is that active replication causes the compensation cost even if no failure occurs during the workflow execution, where passive replication incurs no compensation cost at all.

In this paper, we propose *partition-tolerant replication*, which allows to exploit the search space between active and passive replication. In specific, our main contributions are: 1) We develop a partition-tolerant replication protocol, which allows continuing workflow executions without majorities, similar to active replication, while inducing significant lower compensation cost by mimicking the behavior of passive replication. 2) We integrate a mechanism that allows to control whether the protocol behaves closer to passive replication – allowing fewer primaries and decreasing the compensation cost – or closer to active replication – reducing the impact of failures on availability. 3) Our extensive evaluations on the OpenStack platform show that our protocol incurs no compensation cost for failure free executions while reaching nearly the availability of active replication.

Published in Proceedings of the 10th IEEE International Conference on Service-Oriented Computing and Applications (SOCA 2017), Kanazawa, Japan, November 22-25, 2017
© IEEE 2017

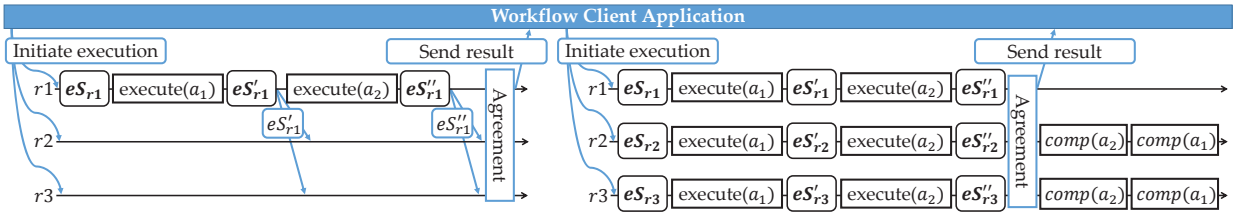

Figure 1. Passive replication (left) versus active replication (right)

2. System Model

We consider a distributed system of computing nodes and services connected by communication links. Each computing node is running a workflow engine capable of executing workflows. All resources, i.e., computing nodes, services, and communication links, might fail at any point in time. According to the crash recovery model [10], we assume that each resource eventually recovers from a failure. For enabling failure tolerance, each computing node participates in the replicated execution of a workflow. Thus, the computing nodes are called *replicas*, where each replica r from the set of all replicas R has a unique identifier.

A workflow is modeled as a directed acyclic graph $G = (mID, A, L, \Sigma)$, where mID is a unique workflow model identifier, A is the set of activities, L the set of links defining the execution order of the activities, and Σ specifies the variables that the workflow stores, called the *internal state*. The activities $a \in A$ are performed in the order specified by the links $l \in L$. The links are defined by $L : A \times A \times T$, where T is the set of all transition conditions. A link $l = (a_1, a_2, t)$ specifies that an activity a_2 is performed after the activity a_1 completed and the transition condition $t \in T$ is fulfilled.

For executing a workflow, a replica instantiates the workflow model, which initializes the variables of the internal state. The initial internal state σ_i created by instantiation is identical for any instance of a workflow G. Each activity of a workflow can read from or write to the internal state. Each activity might also interact with a service by sending a request to the service and receiving a reply. A service might write to external state, where the external state refers to any state that is not in direct control of the execution engine, e.g., the service's state in case of a stateful service.

We assume that each activity $a \in A$ has a compensation handler $comp(a)$. The execution of this compensation handler semantically reverses the effects that an activity execution had on the external state and rolls back the internal state. The compensation may incur cost (e.g., monetary cost) that we refer to as the *compensation cost*. An activity a can only be compensated after all causally succeeding activities [11] have been compensated, i.e., after all activities that used the internal state produced by a have been compensated. This compensation model originates from Sagas [12] and conforms to business processes [13]. Thereby, our model is compatible with workflow description languages such as the Web Service Business Process Execution Language (WS-BPEL), which is standardized by OASIS[1] and widely used in industry. Each workflow engine hosts a component called the Compensation Unit that is responsible for executing compensation handlers. For the purpose of this work, we assume that any compensation handler that is sent to the Compensation Unit is eventually executed, where all handlers are executed in the order in which they were received.

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

3. Passive versus Active Replication

Passive replication elects one *primary* replica that executes the workflow, for example, replica $r1$ in Fig. 1. The other replicas, called *backups*, save the execution state (or *eS* for short) that the primary sends after each activity execution. In case of a primary failure, the backups elect a new primary that uses the received execution state for continuing the workflow execution. To ensure that there is always only one valid primary, passive replication uses majority election [14]. Thus, out of any majority of replicas at least one knows the most recent, i.e., valid, primary. Upon finishing the workflow execution, the primary initiates a majority consensus [15] for agreeing on the result of the execution.

As passive replication uses majority election, the backups can only elect a new primary in case a majority of replicas is operational. In Fig. 2, we evaluate a replicated workflow execution with 5 replicas, where we inject one partitioning plus one crash failure such that none of the partitions contain a majority. The failure lasts for 30 s. We measured the time of unavailability (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 can observe that passive replication is unavailable for the complete failure time because it cannot elect a new primary.

Active replication overcomes this problem by executing the workflow on all replicas independently. Upon finishing the execution, the replicas agree on one of the workflow executions via majority consensus and compensate all other executions (cf. Fig. 1). Consequently, all but one replica might fail during the execution without impacting availability. Only the agreement in the end requires a majority to be operational.

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 Fig. 2, the availability of active replication is not impacted by the failures. 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.

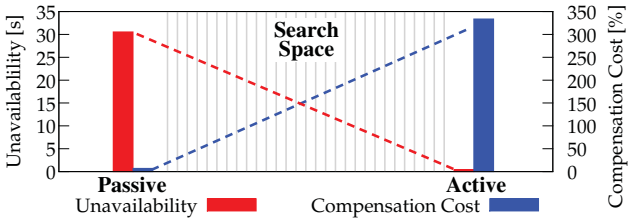

Figure 2. Comparison of passive and active replication with 5 replicas in the failure scenario of having no majority within a partition [lower is better]

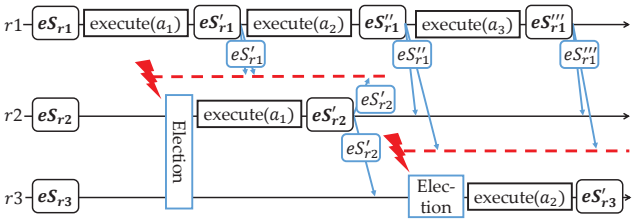

Figure 3. Example of partition-tolerant replication with 3 replicas and $t_v = 1$

In conclusion, Fig. 2 shows that passive and active replication constitute two opposing extremes leaving a huge search space in between. In this paper, we address this search space by proposing a protocol for obtaining the best of both worlds. Without any failures, one primary executes the workflow exactly as when using passive replication. However, in the case of a failure, we allow the replicas to elect a new primary *without* requiring a majority. In specific, each partition elects a primary. For example, assume 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. We call this replication mechanism *partition-tolerant replication*.

Of course, partition-tolerant replication increases the compensation cost in the presence of network partitioning because it elects multiple primaries. For controlling this compensation cost, we integrate the vote threshold t_v. A replica needs t_v votes for being elected as primary, where $1 \leq t_v \leq \lfloor \frac{|R|}{2} \rfloor + 1$. A workflow designer, as the domain expert, can decide the threshold depending on the desired availability. When setting the $t_v = 1$, each replica can vote for itself and become primary. Setting $t_v = \lfloor \frac{|R|}{2} \rfloor + 1$ leads to passive replication, where a majority is required to elect a primary. Any threshold between 1 and $f+1$ realizes a tradeoff.

But how can we realize such a partition-tolerant replication protocol? In specific, every partition should only elect a single primary even when $t_v = 1$. Otherwise, we again would induce unnecessary compensation cost. However, when a replica is in a partition on its own, it should quickly become primary. Moreover, 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.

4. Partition-tolerant Replication Protocol

A replicated workflow execution is started when the workflow client application [16] sends an execution request to all replicas. The request contains the unique identifier of the workflow model that shall be executed. Upon receiving the request, a replica loads the respective model from the workflow repository, which is running locally or on a remote server. The replica with the highest identifier is the initial primary saving the overhead of an election on start-up.

The primary executes the activities, where each activity execution produces an output internal state. The primary stores this internal state as well as the activity to be executed next in the *execution state*. Additionally, each execution state contains a *state number* that is incremented upon every activity execution. Thereby, the state number indicates the progress of the workflow execution. The primary sends updates containing the produced execution state to all other replicas, called *backups*, which save the received state.

In case that the primary crashes or is partitioned, the backups elect a new primary. The new primary has to receive a vote from t_v replicas (including itself) for being elected. Each vote message includes the execution state of the voting replica. The new primary chooses one of the received execution states as the state to continue the workflow execution from. We call this state the *take-over state*. Because the state number reflects the progress of the workflow execution, the new primary chooses the state with the highest state number as take-over state.

In case of partitioning, multiple partitions might elect a new primary – depending on t_v. For example, with $t_v = 1$ each partition will elect a primary. The primaries of the different partitions are not aware of each other. Upon reconnecting, 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 correct [1]. For example in Fig. 3, 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 are stopped to reduce 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 example, in Fig. 3, $r2$ becomes primary after partitioning from $r1$. After reconnecting, $r2$ stops its execution because it receives an update from $r1$, which is already progressed further.

For completely resolving the conflict, $r2$ has to compensate its execution of a_1. However, r_2 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 Fig. 3.

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) were compensated (if any). For this purpose, each replica keeps a history of its activity executions, where each activity includes the information which execution state was used as input and which execution state was produced through the activity execution. Thus, each execution state needs a unique identifier.

An execution state identifier consists of three parts: a replica's 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 as already described.

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. Consequently, 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 agree on one finished workflow execution. The agreement requires a majority of replicas to be operational. 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 operational while the agreement requires a majority of replicas. The last two termination phases require all replicas to be operational. However, this is not critical as this phase is completely decoupled from the workflow client application.

Note that our protocol supports XOR- and AND-branching because we identify the logged activity executions based on the input execution state and the produced execution state, where the execution state identifiers are decoupled from the workflow model. Only one small extension is required for AND-branching: the execution state contains not a single activity but a set of activities that have to be executed next, i.e., one activity per branch. The primary can selects any of the activities in the set for execution.

4.1. Data Structures

The protocol has to maintain data in volatile memory as well as on stable storage for the workflow execution. In specific, the volatile memory of a replica r contains the following information:

- eID: the unique identifier of the replicated execution.
- G: the workflow model that is executed.
- t_v: the vote threshold required for becoming primary.
- $prim_r$: the indicator on whether r is currently primary.
- eS_r: the 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:

- $next$: the next activity to execute (or set of activities during AND-branching).
- σ: the internal state that is input for the next activity execution.
- sID: the execution state's unique identifier.

As described above, the identifier of an execution state sID consists of the following variables:

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

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: the current failover counter of r. Each time r starts a failover, the counter is incremented. Initially, f_r is 0.
- H_r: the 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 $eS_{prod}.ID$.

4.2. 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 replica, say r, that is primary executes the workflow as follows (cf. Alg. 1 line 1-10): First, r writes an execution record for the activity that is going to be executed into the execution history H_r. Then, r executes the activity and updates $eS_r.next$. Finally, r sends an update containing the

Algorithm 1: Protocol 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     async send UPDATE(eID, eS_r);
11 upon receive UPDATE(eID, eS_x) do
12   if eS_x.s > eS_r.s then
13     prim_r := false;
14     eS_r := eS_x;
   // failover
15 upon detected primary failure do
16   f_r := f_r + 1;
17   send REQ_VOTE(eID, r, f_r) to all replicas;
18   wait for votes;
19 upon receive REQ_VOTE(eID, x, f_x) from x do
20   if prim_r = true then // r is primary
21     send REJECT(eID, x, f_x) to x;
22   else if x < r then // r has a higher ID
23     if r is not waiting for votes then
24       trigger event detected primary failure;
25     send REJECT(eID, x, f_x) to x;
26   else
27     send VOTE(eID, x, f_x, eS_r) to x;
28 upon receive VOTE(eID, x, f_x, eS_x) from x do
29   if received VOTE from ≥ t_v replicas for (x, f_x)
30    AND waited ≥ t_t then
31     eS_r := execStateWithHighestStateNumber(VOTEs);
32     prim_r := true;
```

produced execution state to all replicas. Note that r sends the update asynchronously, i.e., r 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. Alg. 1 line 11-14).

4.3. 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. Alg. 1 line 15-18), a backup increments its failover counter indicating it now starts a failover. Then, it requests VOTE messages from all replicas. All replicas with lower IDs reply with a VOTE that includes the execution state of the voter (cf. Alg. 1 line 19-27). Any replica with a higher ID 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 ID in the same partition. It, however, might be the case that a replica receives enough VOTE messages before receiving a REJECT. Thus, every replica that requested VOTE messages waits for the time threshold t_t 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. Alg. 1 line 28-32). 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. Alg. 1 line 13). Thereby, competing executions are stopped without introducing additional coordination overhead.

4.4. Termination

When a primary has executed the last activity of the workflow, it initiates the termination (cf. Alg. 2 line 1-2). The termination consists of three phases. In the first phase, the replicas agree on one workflow execution. 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.

The first phase can be realized by any consensus protocol, such as the Paxos protocol [17]. 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. Alg. 2 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. Alg. 2 line 6-13), a replica, say r, uses the included execution state identifier for checking whether r used the state as input

Algorithm 2: Protocol on replica r (Part II)

```
   // termination
1  upon eS_r.next = null do
2      start Agreement;
3  upon Successful Agreement(eS_x) do
4      forall the (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(eS_r.next)) 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;
   // recovery
22 upon Recover in ACTIVE state do
23     send RECOV_REQ(eID) to all replicas;
24 upon Receive RECOV_REQ(eID) from x do
25     if Terminated and already reached agreement then
26         send DECISION(eID, eS_r) to x;
27     else
28         send RECOV_RPLY(eID, mID, t_v, eS_r) to x;
```

for any of its activity executions. If not, then r can allow the compensation by replying with an ALLOW 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 an ALLOW message (cf. Alg. 2 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. Alg. 2 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. Basically, the workflow execution is 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.

To forget the execution, i.e., to remove all the data of the execution from the volatile memory – and stable storage if desired – the termination has a third phase. A replica that finishes the second phase starts a 2PC protocol [18]. The action carried out by the 2PC protocol 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.

4.5. Recovery

Through crash failures, a replica loses the data kept in its volatile memory. Upon recovery, a replica reads its execution history. Here, we differentiate the following cases:

UNKNOWN A replica that has either no record of the execution *eID* in the execution history or both a begin and an end record for *eID* in its execution history.

ACTIVE A replica that has a begin but no end record for the execution *eID* in its execution history.

Obviously, a replica that recovers in the UNKNOWN state cannot initiate any recovery steps for *eID*.

When recovering in the ACTIVE state (cf. Alg. 2 line 22-23), 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. Alg. 2 line 24-28), a replica reacts differently depending on whether 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. Evaluations

We implemented a prototype of our partition-tolerant replication protocol using the Apache MINA framework for realizing the communication.[2] We deploy the prototype

2. https://mina.apache.org/, https://www.openstack.org/

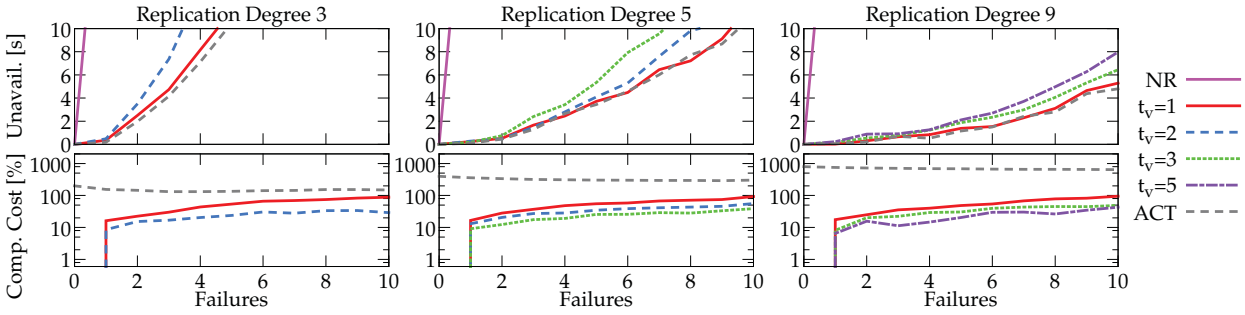

Figure 4. Execution time and compensation cost of replicated workflow executions in the presence of failures [lower is better].

on OpenStack, where up to 9 replicas are running each in a dedicated virtual machine (VM) with 1 vCPU and 2 GB RAM. For the measurements, we generate workflows with 100 activities, which is in the range of typical workflow lengths [19]. A study has shown that for clients with a reliable connection, most web services have a response time below 500 ms while few services have a response time of several seconds [20]. We reflect this by setting the execution time of the activities using random non-negative values of a Gaussian distribution with a mean of 0 ms and a standard deviation of 500 ms. The compensation cost of activities is domain specific and, thus, cannot be generalized. However, this is no problem as a user or domain expert can set the vote threshold t_v through which the compensation cost incurred by our protocol can be controlled, as we will see below. In our measurements, we set the compensation cost of each activity using the values of a random function with a uniform distribution ranging from \$0 to \$100.

Overall, we measured over 40,000 workflow executions in terms of availability and compensation cost. For making the compensation cost of the randomly generated workflows comparable, we normalize the compensation cost by the cost for compensating the complete workflow once. Here, 100 % is equal to compensating all activities of the workflow once. The optimal value is 0 %. Moreover, we inject failures at random points in time, where each failure 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 [21]. Both partition and crash failures have a mean-time to recovery of 30 s.

For comparison, we evaluate active replication (ACT) and a non-replicated execution (NR). Additionally, our partition-tolerant replication protocol is identical to passive replication when setting the vote threshold t_v to $\lfloor \frac{|R|}{2} \rfloor + 1$, where R is the set of all replicas.

In Fig. 4, we evaluated replicated workflow executions with 3, 5, and 9 replicas as well as a non-replicated execution. The unavailability of the non-replicated execution (NR) is steeply increasing with failures. Actually, each failure delays the non-replicated execution by the mean-time-to-recovery, i.e., 30 s. Any replicated execution easily outperforms the non-replicated execution.

With only one failure, the availability of active replication and all partition-tolerant replication approaches is basically unaffected. This is not surprising because all approaches can tolerate one partitioning or crash failure. 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 passive 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, active replication performs substantially better because it continues the execution even if all but one replica is failed. Thus, active replication provides the baseline of what can be achieved in the best case. With replication degree 3, our partition-tolerant replication with $t_v = 1$ reaches nearly the availability of active replication. With replication degree 5 and 9, the availability of active replication and our partition-tolerant replication with $t_v = 1$ are almost identical.

When considering the compensation cost of the replication approaches in Fig. 4, 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.

All other replication approaches, i.e., passive and our partition-tolerant replication with the different vote thresholds, do not cause any compensation cost in the failure free case. With more than one failure, partition-tolerant 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.

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 $t_v = 1$ and tolerating the higher compensation cost increases availability significantly.

6. Related Work

Ensuring availability is an important consideration for any network-related application. Replication is commonly used to ensure availability in the presence of failures for such applications [10]. So far, there only exist two workflow replication techniques: passive and active replication [8], [9], [22]. However, both techniques have their shortcomings:

active replication implies high compensation cost even in failure free cases, while passive replication provides lower availability than active replication. Our proposed partition-tolerant replication protocol overcomes the limitations by providing the availability of active replication while imposing no compensation cost in the failure free case and even significantly lower compensation cost in the presence of failures compared to active replication.

Some existing approaches propose hybrid active-passive replication scheme, where they switch between active and passive replication at run-time [1]. Thus, the approaches use either active or passive replication at each point in time, which means that the approaches still have the shortcomings of the respective technique. In contrast, our protocol explores the search space between active and passive replication.

Other approaches for tolerating failures are integrating fault handlers into the workflow model [23]. If failures occur, the workflow model will react as defined by the workflow designer. However, these approaches cannot mask failures of the device on which the workflow is running.

For ensuring availability in the presence of service failures, there are mechanisms that allow to call an alternative service if the currently called service fails [24]. Even though this decouples the availability of the workflow execution from the availability of a specific service, the technique cannot tolerate the failure of the device on which the workflow is being executed. Interestingly, we could integrate these techniques into our protocol for further improving availability. Another possibility would be to execute the services (if these are internally modeled as workflows) with our protocol for making the services themselves highly available nullifying the need of calling alternative services.

7. Conclusion

In this paper, we proposed a partition-tolerant replicated workflow execution protocol, which combines the strengths of active and passive replication protocols. Our protocol provides a mechanism for letting a domain expert decide whether the execution should behave closer to active replication – tolerating more concurrent failures during the workflow execution – or to passive replication – minimizing the compensation cost in the presence of failures. Our evaluations show that our protocol successfully avoids to incur compensation cost for failure free executions like passive replication while ensuring high availability in the presence of multiple failures like active replication.

Acknowledgments

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

References

[1] D. R. Schäfer, A. Weiß, M. A. Tariq, V. Andrikopoulos, S. G. Sáez, L. Krawczyk, and K. Rothermel, "Hawks: A system for highly available executions of workflows," in *IEEE SCC'16*, pp. 130–137.

[2] V. Solutions, "Assessing the financial impact of downtime," White Paper, 2008.

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

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

[5] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *ACM SIGCOMM'11*, pp. 350–361.

[6] 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. Netw.*, vol. 16, no. 4, pp. 749–762, 2008.

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

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

[9] N. Stojnić and H. Schuldt, "Osiris-sr: A scalable yet reliable distributed workflow execution engine," in *ACM SIGMOD SWEET'13*, pp. 3:1–3:12.

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

[11] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[12] H. Garcia-Molina and K. Salem, "Sagas," in *ACM SIGMOD '87*, pp. 249–259.

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

[14] M. van Erp, L. Vuurpijl, and L. Schomaker, "An overview and comparison of voting methods for pattern recognition," in *ICFHR'02*. IEEE, pp. 195–200.

[15] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Trans. Database Syst.*, vol. 4, no. 2, pp. 180–209, 1979.

[16] D. Hollingsworth, "Workflow management coalition: The workflow reference model," The Workflow Management Coalition Specification Document Number TC00-1003, 1995.

[17] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[18] B. Lampson and H. E. Sturgis, "Crash recovery in a distributed data storage system," 1979.

[19] M. Brambilla, S. Ceri, M. Passamani, and A. Riccio, "Managing asynchronous web services interactions," in *IEEE ICWS'04*, pp. 80–87.

[20] Z. Zheng and M. R. Lyu, "A distributed replication strategy evaluation and selection framework for fault tolerant web services," in *IEEE ICWS'08*, pp. 145–152.

[21] E. Brewer, "Spanner, truetime and the cap theorem," Google, Tech. Rep., 2017.

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

[23] N. Russell, W. Aalst, and A. Hofstede, "Workflow exception patterns," in *Advanced Information Systems Engineering*, E. Dubois and K. Pohl, Eds. Springer, 2006.

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