Towards Fast and Efficient Failure Handling for Paxos State Machines

Leander Jehl and Hein Meling
Department of Electrical Engineering and Computer Science, University of Stavanger, Norway
Email: leander.jehl@uis.no, hein.meling@uis.no

***Abstract*—We investigate methods for handling failures in a Paxos State Machine and introduce Live Replacement, which quickly repairs failures at a low cost. Live Replacement enables a failed or disconnected replica to be replaced with a new one, with minimal disruption. Replacement does not rely on a special state machine command, and need not be ordered with respect to other commands. It is therefore independent from the state machine's progress. This enables Live Replacement to guarantee fast reaction to failures with minimal effect on the state machines operation.**

I. Introduction

State machine replication [1] is a common approach for building fault-tolerant cloud services. In this approach, all service replicas of the state machine execute the same requests. To ensure that replicas remain consistent after multiple updates, the order of requests has to be synchronized across replicas. Typically this is accomplished using a consensus protocol, such as Paxos [2], [1], which is an essential component in a Replicated State Machine (RSM). Paxos helps to prevent the replicas from entering an inconsistent state, despite any number of replica failures. Moreover, the RSM can continue to process new requests, as long as *more than half of the replicas* remain operational. If this bound is violated, however, the current RSM is forced to stop making progress indefinitely. Such a situation would require that complicated manual repair procedures be initiated in order for the RSM to become operational again. To avoid scenarios in which the number of failures exceeds the bound, it is obviously beneficial to immediately instantiate failure handling, if this can be done without causing a significant disruption to request execution.

In this paper we investigate methods for *immediate failure handling* in RSMs. We are interested in methods enabling fast and efficient reaction to failure, restoring a system's initial fault tolerance as soon as possible, and minimally impacting throughput and latency of state machine requests. This is different from the traditional approach to this problem, as presented in [1] and later implemented in SMART [3] and Google's Chubby Service [4], among others. This approach tries to recover the failed replica's state from stable storage and restart the replica on the same physical machine. Only if a replica is unable to recover, e.g. due to a disk failure, is it replaced by a new replica, using Reconfiguration. With this approach, only infrequent reconfigurations are necessary. Thus reduced latencies during reconfiguration are of little concern. However, waiting for recovery leads to long periods between failure and reestablishing the initial level of fault tolerance.

We thoroughly review Reconfiguration with respect to immediate failure handling and identify causes for slow repair and reduced latencies. In response to these defects, we introduce Replacement, a new method for handling failures. Replacement can be used to substitute a faulty, disconnected, or slow replica with a new one. Compared to Reconfiguration, Replacement reduces the impact on non-faulty replicas, since they never have to stop to change to a new configuration. This differs from Reconfiguration, which is often abstracted as stopping the current state machine and restarting with a new set of replicas, e.g. [3], [5], [6].

Our main contribution is Live Replacement, a protocol for Replacement that provably cannot endanger the consistency of the replicas. Live Replacement has the same common case repair time as that of Reconfiguration, while avoiding several causes for degraded performance of the latter. This is achieved by decoupling Replacement from the state machine's progress.

In the next section, we define the scope and assumptions for our work, and introduce two metrics to evaluate whether a method is suitable for immediate failure handling. In §III we give brief overview of Paxos and state machine replication, followed by a review of existing methods for handling failures in §IV. In §V we derive our Live Replacement method from classic Reconfiguration, and conclude in §VI.

II. Scope and Assumptions

To start a replica on a new machine requires additional resources. Also if the replicated service has a large state, which has to be transferred to the new replica, this can be very costly. However, Paxos is frequently used to implement lightweight services such as locks or configuration management, used to coordinate distributed applications on a larger set of machines. Thus, we assume that

additional machines are available, and that application state is small and can be transmitted efficiently.

As mentioned, our approach of immediate failure handling requires reaction to a replica failure as soon as possible, and with the least possible impact on other replicas. We thus define the following metrics that an efficient failure handling method should minimize:

- *Delay* is the *time from detection of a failure until a new replica, replacing the failed one, is participating in Paxos*.
- *Disruption* is *the additional latency state machine requests experience during failure handling*.

We do not include detection time in the Delay, since it is a configurable parameter, depending on the specific system, not the method.

We distinguish between two approaches to replace a faulty replica, depending on how the new replica receives a consistent state.

- In *Reconfiguration* the replicas agree on a consistent system state and which replicas to replace. Then all replicas, including the new one, change to that state.
- In *Replacement* the replicas agree on a new replica and a state that guarantees consistency of the whole system. Only the new replica adopts this state, while the other replicas continue as before.

Since we are interested in immediate failure handling, we do not consider the possibility for a replica to recover from failure by rebooting from stable storage.

III. Paxos and State Machine Replication

RSM-based systems using Paxos have been intently studied over the last decade and numerous industrial-strength systems have been implemented, e.g. [4], [3], [7]. In this section we briefly present Paxos [2], [1] and how replicas agree on a single request. We then elaborate on how this is used to implement a RSM.

A. Paxos

Paxos is a consensus algorithm and can be used by several participants (our replicas) to decide on exactly one value (our request). To get chosen, a value has to be proposed by a leader and accepted by a majority of the replicas. Only one value should be chosen and every non-faulty replica should be able to learn what was decided on. Paxos assumes a partially synchronous network, tolerating asynchrony, but relying on eventual synchrony to guarantee progress. That means that single messages can be lost or arbitrarily delayed (asynchrony). However, eventually all messages between non-faulty processes arrive within some fixed delay (synchrony). Since an initial leader might fail and a single leader cannot be chosen during periods of asynchrony, multiple leaders can try concurrently to get their values decided. To coordinate concurrent proposals, leaders use preassigned round numbers for their proposals. Replicas cooperate only with the leader using the highest round. However, after leader failure it might be necessary for replicas to cooperate with several leaders. Paxos solves this problem by enforcing that, once a value has been decided, leaders of higher rounds will also propose this locked-in value. Thus in the first phase of a round, the leader determines if it is safe to propose any value or if there is another value that was already voted for. In the second phase, the leader then tries to get a safe value accepted. If not interrupted a round proceeds as follows:

PREPARE **(1a)** The leader sends a message to all replicas, starting a new round.

PROMISE **(1b)** Replicas reply with a PROMISE not to vote in lower rounds, and inform the leader of their last vote.

ACCEPT **(2a)** After receiving a quorum of PROMISEs, the leader determines a safe value and proposes it to all replicas.

LEARN **(2b)** Replicas vote for the proposal, by sending the value and round number to all other replicas, if no higher round was started in the meantime.

A set of more than f replicas is called a *quorum*. By abuse of notation, we also call a set of messages a *quorum*, when sent by a quorum of replicas. A value v is *chosen* in round r if a quorum of LEARN messages with value v was sent in round r. A replica learns and accepts this decision only if it received a quorum of LEARN messages. The above can be repeated in new rounds, until all replicas have learned the value. From a quorum of PROMISE messages, safe values can be determined following these rules:

- If no replica reports a vote, all values are safe.
- If some replica reports a vote in round r for value v and no replica reports a vote in a round higher than r, then v is safe.

These rules ensure, that once a value is chosen, only this value is safe in higher rounds. Thus all replicas learn the same value, even if they learn it in different rounds. We call this property *Safety*.

We say that an execution of Paxos is *live* if a value can get learned. For Paxos to be live, a leader has to finish a round, communicating with a quorum of non-faulty replicas, while no other leader starts a higher round.

B. The Paxos State Machine

A state machine is an application that, given a state and a request, deterministically computes a new state and possibly a reply to the request. When this state machine is replicated for fault tolerance, it is important that the replicas execute the same requests to maintain a consistent state among themselves. Since several requests might interfere with each other, it is also important that replicas execute requests in the same order.

TABLE I
COMPARISON OF FAILURE HANDLING METHODS.

	Live Replacement	Reconfiguration at $i+\alpha$	Reconfiguration at $i+1$
Disruption:			
Paxos Instances Occupied	0	1	1
Requests Discarded	0	0	$< \alpha$
Delay:			
Waiting for Instances	–	$< 2\alpha$	$\leq \alpha$
Communication Steps	2	2/4[c]	2/4[c]

[c] After leader failure, 4 communication steps are required.

The replicas in a Paxos State Machine achieve this by using Paxos to choose requests. For every new request, a new Paxos instance is started. Its messages are tagged with a request number i, saying that this instance is choosing the ith request. Thus, as long as Paxos is live, new requests can be processed. These will eventually be learned and executed by all non-faulty replicas.

Though execution has to be done in correct order, the Paxos instances can be performed concurrently. Thus it is possible to propose and choose a request in instance $i+1$, before the ith request is learned. To avoid that request scheduling (Paxos) advances too far ahead of execution, and to limit the state required for Paxos, typically only α concurrent Paxos instances are allowed.

Note that, if a leader of one Paxos instance is faulty, he is faulty in all instances. Therefore, round change should be common for all Paxos instances. Thus, PREPARE and PROMISE messages for concurrent Paxos instances can be sent in the same message. [1] explains how this is done, even for all future instances.

IV. VARIANTS OF RECONFIGURATION

In this section, we explain Reconfiguration and survey different implementations. We also discuss how the different methods cause Disruption and Delay.

Abstractly, Reconfiguration works by replacing the whole set of replicas with a new set of replicas. Each such set is called a *configuration*. The old configuration has to decide on the new set of replicas and the initial state of the new configuration. All new replicas then start with the same state. When used to handle failures, the new configuration will contain all the non-faulty replicas and new ones instead of those considered faulty. A non-faulty replica that is part of both the new and the old configuration, will also adjust its state to the starting state. Reconfiguration can also be applied without apparent failures, e.g. to realize hardware upgrades or other optimizations. These changes are infrequent and can be scheduled in advance. Therefore Delay and Disruption are of no concern in these cases.

Before Reconfiguration can take effect, the new configuration has to be known to a majority of the old replicas, to avoid splitting the state machine. An easy way to achieve this is to issue a special *reconfiguration command* as request to the state machine, specifying the new set of replicas. The new configuration can start after this request was chosen. However, it is not that easy to decide on a starting state for the new configuration, especially if several Paxos instances are running concurrently.

Assume first, that only one request ($\alpha = 1$) is chosen at a time. If a reconfiguration command is proposed as request number i, all earlier requests have been chosen and executed, and no request higher than i has been proposed. Therefore the new configuration should start with the state obtained after executing request $i - 1$.

However, with $\alpha > 1$, several requests are chosen concurrently. In this case there might be both undecided instance before the reconfiguration command, and already decided instances after it. A conservative approach, proposed in [1] is to wait until all possibly decided requests have been learned and executed, and only then change to the new configuration. Thus, if the reconfiguration command was chosen as request number i, the new configuration only starts running the instances with number $i + \alpha$ and higher. This can cause a significant Delay, depending on the $\alpha - 1$ instances after the reconfiguration command. This solution is summarized as Reconfiguration at $i + \alpha$ in Table I.

In [3] a similar method is used. As described above, the new configuration only starts from instance $i + \alpha$. But where possible a no-op command is proposed. This clearly reduces Delay, but leads to higher Disruption, since application requests might have to wait while no-op commands are being chosen.

A different solution was developed in [5], stopping the old configuration directly after deciding on the reconfiguration command (with number i) and discarding possibly decided requests with numbers higher than i. This method reduces Delay, but discarding decisions causes Disruption since they have to be chosen again. Also, the possibility to discard already decided requests makes it more difficult to determine when requests are irrevocably chosen. This solution is referred to as Reconfiguration at $i + 1$ in Table I. Note that in both solutions, all Paxos instances with lower numbers than the reconfiguration command have to be decided before the old configuration can stop. There can be up to α proposed, but undecided requests before the reconfiguration command. This can delay a reconfiguration command issued concurrently with other state machine requests.

Some Paxos implementations achieve high throughput despite a very small α parameter, by batching many requests together and accepting all of them in the same instance [8]. To minimize Disruption, a reconfiguration command should be sent as part of a full batch. However that causes Delay, when waiting for the batch to be filled up. Also it is unclear, whether the old or the new configuration should execute the requests in the same

batch as a reconfiguration command. The alternative of delaying the batch of requests while deciding on the configuration reduces Delay, but introduces Disruption.

V. From Reconfiguration to Replacement

We now derive Live Replacement in five steps starting from a protocol called Fault-Masking Virtually Synchronous Paxos presented in [9].

Step 1: Separating Reconfiguration from the Replicated State Machine

In Reconfiguration as described earlier, the consensus engine used to choose requests for the RSM is also used to decide on reconfigurations. Birman et al. [9] proposed a different approach, separating the protocol to choose requests from the one used to decide on reconfigurations. Here, we shall call the protocol for choosing requests *State Machine Consensus* (SMC), and the one deciding on reconfigurations *Reconfiguration Consensus* (RC). Assuming the Paxos protocol is used in RC, we further distinguish the two protocols by naming messages in RC: RPrepare, RPromise, RAccept, and RLearn. We call rounds in RC for *epochs*. The leader (proposer) of an epoch in RC is called the RC-leader. The SMC and RC interact in the following way:

- A replica stops participating in SMC upon receiving a RPrepare, and includes the last SMC state in its RPromise message.
- If, after receiving a quorum of RPromises, the RC-leader can propose any value, a new configuration and a consistent SMC starting state is chosen and proposed in a RAccept message.
- RLearn messages are sent to replicas in both the old and the new configuration.
- On receiving a quorum of RLearns, the replicas in the new configuration can start executing SMC with the starting state determined by the RC-leader.

This protocol is depicted in Figure 1(a). For clarity, the RC-leader is shown as an external process for all the protocols in Figure 1. Normally the RC-leader will be co-located with one of the replicas.

Step 2: Vertical Paxos

Separating RC from SMC clearly serves to reduce Delay. Thus, RC never has to wait for SMC. However, the problem with this protocol is that the RPromise messages also carry the SMC state. Thus, the first phase cannot be skipped in the first epoch, or executed in advance as is usually done in Paxos.

To overcome these problems, we consider a variation of the above RC protocol, depicted in Figure 1(b). In this protocol, replicas can continue SMC in the old epoch until receiving a RAccept message. They then include their SMC state in a RLearn message sent only to the RC-leader, who collects a quorum of these messages, and decides on a starting state for the new configuration. This is then forwarded as a RDecided message to the replicas. Since SMC is only interrupted on receiving a RAccept, RPrepare messages can be sent in advance or omitted for the first epoch.

In essence, this protocol is the same as Vertical Paxos [10]. However, Vertical Paxos does not require the exchange of RPrepare and RPromise since it relies on a centralized RC-leader to keep track of reconfigurations. Vertical Paxos also makes use of a few other optimizations that are not compatible with the following steps, hence we do not discuss them here.

Step 3: Starting State Machine Consensus with Possibly Different Values

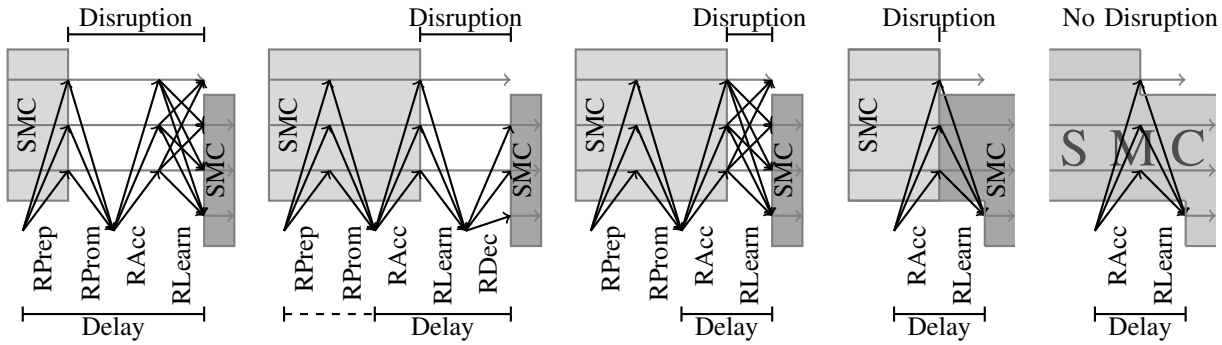
For Step 3, note that it is not necessary for every replica in the new reconfiguration to start SMC with the same state, as long as no previously chosen values are forgotten. Thus, every replica can determine its SMC state from a quorum of RLearn messages. Upon learning about the new configuration, the replicas can determine a consistent state for restarting SMC. This reduces the average Disruption during reconfiguration to one message delay, as depicted in Figure 1(c).

Step 4: Lightweight Reconfiguration Consensus

In this step, Reconfiguration Consensus is further reduced. When choosing state machine requests, it is important that every replica executes all chosen requests in the correct order. Thus, a chosen request can only be executed after all preceding requests have been executed. For reconfigurations this strict order is not necessary. To install a chosen configuration, it is not necessary to install all preceding configurations. Especially, since no new requests can get chosen in these configurations, after the replicas decided to reconfigure. We can therefore remove the first phase of RC without risking safety.

This reduction leads to a protocol that no longer solves classic consensus. However, as explained above, it is sufficient to eventually agree on a configuration, irrespective of intermediate configurations.

To further reduce RC, we note that, for a chosen request to remain chosen after reconfiguration, it has to be present in a majority of the replicas in the new configuration. It is therefore sufficient, if no replica forgets this request during reconfiguration, and all new replicas include this request in their SMC starting state. Thus, we only need to send RLearn messages to the new replicas, and have the other replicas restart SMC in the new configuration immediately after receiving the RAccept, with the same state as before. As shown in Figure 1(d), the number of RLearn messages is significantly reduced.



(a) Step 1: Separating RC from SMC.

(b) Step 2: Vertical Paxos.

(c) Step 3: Starting with possibly different values.

(d) Step 4: Starting immediately.

(e) Step 5: Live Replacement.

Fig. 1. Steps towards Live Replacement.

The changes introduced to RC in this step do not endanger safety, however liveness may be violated. That is because in the presence of multiple RC-leaders, a leader could send its RACCEPT to an outdated configuration, and two leaders could even separate the replicas into two configurations, both unable to make further decisions. We have designed extensions to our protocol to deal with these problems. Since these are special cases and irrelevant for the average Delay and Disruption, we do not describe them here.

Step 5: Live Replacement

Even though replicas that transition from the old to the new configuration can restart SMC immediately, state machine commands must be chosen in one of the configurations. Thus, reconfiguration still stops SMC, causing Disruption.

We therefore introduce new identifiers for our configurations, called *epoch vectors*. These are vector clocks, showing for each replica, the epoch when it first joined SMC. Thus, epoch vectors can be used to identify which replicas have been replaced by new replicas and which are stable across configurations. By attaching these epoch vectors to SMC LEARN messages, replicas in the new configuration can form a quorum and decide, even though some of the replicas have not yet installed the new configuration.

As depicted in Figure 1(e), SMC now suffers no Disruption from replacing a failed replica. However, we note that Live Replacement can only replace replicas. It is not applicable for changing the number of replicas in a configuration. The proof, showing that Live Replacement does not compromise safety, was omitted to meet the page limit. Also, in an asynchronous system, we cannot be sure that a replica that was detected as faulty, has really failed. We therefore also have to consider the case, when the replaced replica is actually non-faulty.

VI. CONCLUSIONS

Live Replacement takes a new approach to handle failures in a Paxos State Machine. It is particularly adept for immediate failure handling. On average, Live Replacement and Reconfiguration both take two message delays to decide on a new configuration, but as shown in §IV, implementing Reconfiguration enforces several decisions, favoring either Delay or Disruption. Since Live Replacement is decoupled from SMC, it never has to wait for any unfinished instances in SMC. Live Replacement avoids Disruption by allowing replicas to continue running SMC during replacement. Finally, Live Replacement is controlled by a RC-leader, who's task can be assigned to a different replica than the Paxos leader. Thus, replacement will not impose additional overhead for the already heavily loaded Paxos leader. It thus guarantees fast reaction to failures with minimal effect on the state machines operation.

REFERENCES

[1] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, December 2001.

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

[3] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, "The smart way to migrate replicated stateful services," in *EuroSys*, 2006.

[4] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *OSDI*, 2006.

[5] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," *SIGACT News*, vol. 41, no. 1, pp. 63–73, Mar. 2010.

[6] V. Bortnikov, G. Chockler, D. Perelman, A. Roytman, S. Shachor, and I. Shnayderman, "Brief announcement: reconfigurable state machine replication from non-reconfigurable building blocks," in *PODC*, 2012.

[7] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *USENIX ATC*, 2010.

[8] N. Santos and A. Schiper, "Tuning paxos for high-throughput with batching and pipelining," in *ICDCN*, 2012.

[9] K. Birman, D. Malkhi, and R. van Renesse, "Virtually synchronous methodology for dynamic service replication," MSR, Tech. Rep., 2010.

[10] L. Lamport, D. Malkhi, and L. Zhou, "Vertical paxos and primary-backup replication," MSR, Tech. Rep., 2009.