Asynchronous Reconfiguration for Paxos State Machines

Leander Jehl and Hein Meling

Department of Electrical Engineering and Computer Science
University of Stavanger, Norway

Abstract. This paper addresses reconfiguration of a Replicated State Machine (RSM) in an asynchronous system. It is well known that consensus cannot be solved in an asynchronous system. Therefore an RSM providing strong consistency, cannot guarantee progress in an asynchronous system. However, we show that reconfiguring the RSM is possible in a purely asynchronous system. This differs from all existing reconfiguration methods which rely on consensus to choose a new configuration. Since a reconfiguration to a new set of machines or even a different datacenter can serve to restore synchrony between replicas, asynchronous reconfiguration can also serve to increase the availability of an RSM.

1 Introduction

State machine replication [1] is a common approach for building fault-tolerant 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, 3], 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. Therefore real systems need to be able to react to failures and include new replicas, before this bound is exceeded. Additionally, in todays cloud computing infrastructures, a fault tolerant, replicated service should be able to add, or move replicas based on the systems load and redistribute the replicas to new physical locations. Therefore real systems should include a reconfiguration method, ensuring a continuous correct service while enabling the changes above.

The initial descriptions of the state machine approach [1] and Paxos [2, 3] described a reconfiguration method. This method lets the RSM reconfigure itself by executing a special command, which specifies the new configuration. It was refined and implemented, e.g. for the Zookeeper service [4], enabling reconfiguration for a replicated system in production.

To use this method however, the old configuration (being changed) must be operational to complete the reconfiguration command. Thus the old configuration needs to have a single leader and a majority of correct servers, to use

M. Chatterjee et al. (Eds.): ICDCN 2014, LNCS 8314, pp. 119–133, 2014.

reconfiguration. Vertical Paxos [5] developed a different reconfiguration method, showing that reconfiguration is possible with only a minority of correct replicas, if it is coordinated by an abstract, fault tolerant configuration manager. In Vertical Paxos, synchrony is needed to guarantee a working configuration manager.

In this paper we target the synchrony assumption of traditional reconfiguration. We present ARec, an asynchronous reconfiguration protocol, showing that reconfiguration is possible in an asynchronous system. This means that, agreeing on a single leader is not necessary to change to a new set of replicas. Reconfiguring an RSM can therefore take place during a period of asynchrony, e.g. when clocks are not synchronized and the network introduces unpredictable delays. This is an important result, because it means that replicas could be moved to a set of machines or a datacenter where synchrony assumptions hold.

1.1 ARec: Reconfiguration without Consensus

A *configuration* is a set of replicas running on different machines. To change the set of replicas, a reconfiguration simply proposes a new configuration. We assume that any two configurations are disjoint. Thus every replica belongs to exactly one configuration. However we do not require replicas of different configurations to run on different machines. Thus reconfiguration also enables operations like adding or removing one replica. For example, to add a replica to the configuration $\{r_1, r_2, r_3\}$, running on machines m_1, m_2, and m_3, one simply proposes a new configuration $\{r'_1, r'_2, r'_3, r_4\}$ on machines m_1, m_2, m_3, and m_4. Thus our reconfiguration allows all possible changes both in number and placement of replicas.

The main challenges in reconfiguration are to ensure that when stopping the old configuration, exactly one new configuration starts and that all requests that have been committed by the old, stay committed in the new configuration. Assume for example two new configurations C_1 and C_2 are proposed, while an initial configuration C_0 is running the RSM. Classical reconfiguration uses consensus in one configuration to choose the next configuration. Thus replicas in C_0 would choose whether to change to C_1 or C_2. Thus, the new configuration that is chosen can then simply copy the state from C_0.

Since consensus is impossible in an asynchronous system [6], ARec uses eventual consistency to determine the next configuration. For this, we assume that new configurations are issued together with a unique timestamp. If several configurations are proposed concurrently, we let the one with the highest timestamp be the next configuration. Thus, if the replicas eventually become aware of all configurations, they will agree on which configuration should take over.

However this complicates state transfer, since it is difficult to determine which configuration can transfer a correct state to the new configuration. Assume for example C_1 and C_2 have timestamps 1 and 2. In this case, we can ignore C_1 and C_2 can receive the state from C_0. However if C_2 is only proposed after C_1 received the state from C_0 and started the RSM, C_2 has to copy its state from C_1. This problem is solved in Section 4.

Another problem in asynchronous reconfiguration is to find a correct state for the new configuration to start. In classical reconfiguration, the old configuration again uses consensus to decide on the last request. Any replica can then tell this decision to the new configuration, which can choose new requests. In an asynchronous system, the old configuration cannot decide on a final request. Thus, some requests might be committed in the old configuration but remain unknown to some replicas. ARec therefore collects the state of a majority of a configuration's replicas, to determine the new configuration's state. Luckily leader change in Paxos already includes a mechanism to find any value, that might be committed.We reuse this mechanism in ARec.

2 Paxos

In this section we briefly present Paxos [2, 3] and how replicas agree on a single request. We then elaborate on how this is used to implement an RSM.

2.1 Paxos

Paxos is a consensus algorithm 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 may be chosen and every correct 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 correct 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. Therefore 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 PREPARE message to all replicas, starting a new round.

PROMISE **(1b)** Replicas return a PROMISE not to vote in lower rounds, and include their last vote.

ACCEPT **(2a)** After receiving a quorum of PROMISE messages, the leader determines a safe value and proposes it to all replicas in an ACCEPT message.

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

DECIDE Receiving a quorum of LEARNs for one value and round, a replica decides on that value.

A *quorum* is a majority of the replicas and a *quorum of messages* is a set of messages of the same type, sent by a quorum in the same round. A value v is *chosen* if a quorum of LEARN messages with value v was sent. The DECIDE step above ensures that only chosen values get learned.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*. Clearly to be able to report their votes in a later PROMISE message, replicas have to store their last vote (round and value) in variables *vrnd* and *vval*. For a replica to be able to promise not to vote in lower rounds, it must also store the last round it participated in. We refer to this as *rnd*. These variables are all that is needed to ensure *safety*. We call them the Paxos State of a replica. We also write Φ for the tuple $(rnd, vrnd, vval)$.

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 correct replicas, while no other leader starts a higher round.

2.2 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.

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. A replica only executes the ith request, if it has executed request $i-1$ and has learned the request chosen for the ith instance via the Paxos protocol. Thus, as long as Paxos is live, new request can be processed. These will eventually be learned and executed by all correct 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 $i+1$, without waiting for the ith request to be learned.

Note that, if concurrent Paxos instance have the same leader, round change can be common for all instances. Thus, PREPARE and PROMISE messages for concurrent Paxos instances can be sent in the same message. [3] explains how this is done, even for all future instances.

3 Liveness for a Dynamic RSM

In this section, we define the problem of asynchronous reconfiguration. That is, we both define the interface used to issue reconfigurations and specify the liveness conditions for a Dynamic RSM and ARec.

A reconfiguration is initialized by sending $\langle \text{RECONF}, C_i, i, C_l \rangle$ to the new configuration C_i. Here, i is the unique timestamp of the new configuration and C_l is some old configuration. The old configuration is needed, since the new configuration needs to contact the service to take over. We require configurations to be disjoint. Thus a replica only belongs to a single configuration. However replicas from different configurations can easily be placed on the same machine. Note that a RECONF request will typically be sent by a replica in configuration C_l that wants to move the service, e.g. in response to asynchrony or crashes. It can also be sent by an external configuration manager.

We say a configuration is *available*, if a majority of the configuration's replicas are correct. A static Paxos state machine is guaranteed to process incoming requests, if the initial configuration is available and eventually a single leader is elected. In a dynamic system, this condition becomes more complicated. While a reconfiguration might require several old and new configurations to be available, a dynamic system should not require an old configuration to remain available after a new configuration has taken over and copied all relevant state. We say that a replica in a new configuration is *stateful*, if it has copied a correct system state from a previous configuration. Note that this system state might differ from the individual states of single replicas in the old configuration. We define a configuration to be *stateful*, when a majority of the replicas in this configuration are stateful. Thus, if a configuration is stateful and available, there exists a correct and stateful replica that can disseminate a correct system state to the other correct replicas. Clearly, a Dynamic RSM always needs at least one available stateful configuration. Also, new configurations must stay available during reconfiguration. We therefore define that a Dynamic RSM, at any time in an execution, *depends on* the stateful configuration with highest timestamp, C_{max}, and all configurations with higher timestamps. We can now define liveness of reconfiguration as the following:

Definition 1 (ARec Liveness). *For any execution, if*
(a) at all times the configurations that the system depends on are available, and (b) $\langle \text{RECONF}, C_i, i, C_l \rangle$ *is sent by a correct client for a stateful configuration* C_l*, then the system will eventually no longer depend on* C_l*.*

Note that ARec Liveness does not guarantee that the state machine actually makes progress. That is because this always requires some synchrony assumption.

Similar, ARec Liveness does not guarantee, that $\langle \text{RECONF}, C_i, i, C_l \rangle$ actually results in C_i running Paxos. That makes sense, since in the case of concurrent reconfigurations, we don't want several configurations to start. However it guarantees that C_l will stop running Paxos, and replicas in C_l need not stay available. Clearly implementing ARec Liveness makes sense only if also the state machine is live, under reasonable assumptions. We therefore define the following:

Definition 2 (Dynamic RSM Liveness). *For any execution, if*
(a) at all times the configurations that the system depends on are available,
(b) only finitely many configurations are initialized, and
(c) eventually a single leader is elected in C_{max},
then requests submitted to this leader will be processed.

Note that synchrony between processes in C_{max} is needed to guarantee *(c)*. *(b)* says that eventually no more reconfigurations are issued. This guarantees that eventually, one configuration will be C_{max} forever. This configuration can run Paxos without being interrupted by reconfigurations. Since it is impossible to agree on group membership in an asynchronous system [7], it can be theoretically challenging to guarantee condition *(b)*. For example, if reconfiguration is used to replace faulty replicas with new ones, false detections might cause infinitely many reconfigurations, even with only finitely many crashes. This is because in an asynchronous system, a failure detector might falsely detect a crash, causing a reconfiguration, in which a correct replica is replaced by a new one. In practice however, the number of reconfigurations are typically bounded because of their cost. That is, reconfiguration usually requires new machines, possibly transferring large amounts of state between replicas, and may also disrupt service delivery.

4 Asynchronous Reconfiguration

We now explain our reconfiguration protocol ARec and argue for its correctness using examples. See Algorithm 1 on Page 125 for details. We assume reliable communication but specify later how to reduce retransmissions. Proof of correctness is given in the following sections. We first assume that only a single instance of Paxos is reconfigured and later explain how this can be extended to an RSM. ARec maintains *safety* of Paxos by ensuring that at any time, at most one configuration is running Paxos, and that this configuration copies the Paxos State from a previous configuration, upon start-up. To achieve this, the Paxos messages are tagged with the senders configuration, and a receiver only delivers Paxos messages from its own configuration. We write C_i for the configuration with timestamp i. We define a quorum of messages from C_j as a set of messages from a majority of the replicas in configuration C_j.

A Single Reconfiguration: As specified before, a new configuration C_i is initialized by sending $\langle \text{RECONF}, C_i, i, C_l \rangle$, where C_l is some earlier configuration. The new replicas broadcast this message to their configuration to make sure

Algorithm 1. Asynchronous Reconfiguration

```
 1: State:
 2: MyC                                              {This replica's configuration}
 3: MyC.ts                                           {My configuration's timestamp}
 4: Older                                            {Older configurations}
 5: Newer                                            {Newer configurations}
 6: P := ∅                                           {Promises}
 7: Φ := ⊥                                           {Paxos State: (rnd, vrnd, vval)}
 8: stateful := False                                {stateful = True ⇔ Φ ≠ ⊥}
 9: valid := True                                    {valid = True ⇔ Newer = ∅}

10: upon ⟨Reconf, C_i, i, C_l⟩ with l < i and C_l stateful   {On all replicas in C_i}
11:    MyC := C_i                                   {Start this replica in C_i}
12:    MyC.ts := i
13:    send ⟨Reconf, C_i, i, C_l⟩ to MyC            {Make sure all received Reconf }
14:    Older := {C_l}
15:    send ⟨NewConf, C_i⟩ to C_l

16: upon ⟨NewConf, C_j⟩ with j > MyC.ts             {A newer configuration exists}
17:    if valid and stateful then
18:       stop Paxos
19:       Φ := (rnd, vrnd, vval)                    {Get state from Paxos}
20:    valid := False
21:    Newer := Newer ∪ {C_j}
22:    send ⟨CPromise, Φ, Newer, MyC⟩ to C_j

23: upon ⟨CPromise, Φ', Confs, ID, C⟩ when not stateful
24:    P := P ∪ {Φ', Confs, ID, C}
25:    oldC := {C_j ∈ Confs | j < MyC.ts}
26:    newC := {C_j ∈ Confs | j > MyC.ts}
27:    if newC ⊄ Newer then                         {newC already known?}
28:       Newer := Newer ∪ newC
29:       valid := False
30:    if ∃Q ⊂ P; Q is stateful valid Quorum then   {See Definition 3}
31:       Φ := findState(Q)                         {See Algorithm 3}
32:       stateful := True
33:       send ⟨Activation, Φ, Newer⟩ to MyC        {Optimization: See Algorithm 2}
34:       if valid then
35:          start Paxos with Φ in MyC
36:       else
37:          send ⟨CPromise, Φ, Newer, MyC⟩ to C_t ∈ newC
38:    else if Φ' ≠ ⊥ and oldC ⊄ Older then        {CPromise stateful and not valid?}
39:       Older := Older ∪ oldC
40:       send ⟨NewConf, MyC⟩ to oldC               {Ask other configurations}
```

that all correct replicas in C_i receive it. (See Lines 10 and 13 in Algorithm 1.) A replica from C_i then informs the replicas in C_l about the reconfiguration, sending $\langle \text{NewConf}, C_i \rangle$ (Line 15). Upon receiving this message, replicas in C_l stop running Paxos, retrieve their local Paxos State Φ and send it in a

Algorithm 2. Asynchronous Reconfiguration (Continued)

```
40: upon ⟨ACTIVATION, Φ', Confs⟩ when not stateful        {From replica in MyC}
41:   Φ := Φ'
42:   stateful := TRUE
43:   Newer := Newer ∪ Confs
44:   send ⟨ACTIVATION, Φ, Newer⟩ to MyC                  {Activate other replicas in MyC}
45:   if Newer = ∅ then                                   {I'm the newest conf.}
46:     start Paxos with Φ in MyC
47:   else                                                {There's a newer conf.}
48:     valid := FALSE
49:     send ⟨CPROMISE, Φ, Newer, MyC⟩ to Newer
```

Configuration-Promise $\langle$CPromise, Φ, *Confs*, $C_l\rangle$ to the replicas in C_i, including all known configurations with timestamp larger than l in *Confs* (Lines 18-22). The replicas in C_l store the new configuration C_i in the set *Newer* and include it in all future CPromises (Line 21).

Upon receiving a quorum Q of CPromises from C_l with empty *Confs* fields, the replicas in C_i can determine a Paxos State as explained in Algorithm 3. This corresponds to the processing of Promise messages in Paxos. *rnd* is set to a higher value than any one reported in Q, *vrnd* is set to the highest value reported in Q and *vval* to the value, reported in *vrnd*. Paxos is then started with a new round, where the leader sends out Prepare messages. We say that a replica is *stateful* , after it has determined a Paxos State $(rnd, vrnd, vval)$ (See Lines 31,32). This matches the notion of statefulness, introduced in Section 3.

Concurrent Reconfigurations: When several configurations are initialized concurrently, we have to ensure that exactly one of them starts running Paxos. To do this, a replica keeps track of the new configurations it has seen, using the set *Newer*, and includes it in CPromises (Lines 21, 22). Thus, for any two concurrent reconfigurations, at least one will know about the other. To prevent several new configurations from starting Paxos simultaneously, a new replica remembers the configurations included in CPromises (Lines 28, 39). If a replica knows about a configuration with higher timestamp, it will consider itself *not valid* (Lines 9, 29), and therefore will not start running Paxos (Lines 34, 35). Further, if a new replica receives a CPromise, informing it about another configuration with lower timestamp, it will also ask this configuration for CPromises (Lines 25 and 38-40). To make sure a new replica receives its state from the latest stateful configuration, we require a stateful valid quorum to determine a correct state (Lines 30-31). A stateful valid quorum is defined as follows:

Definition 3. *A set Q of* CPromises *sent from a majority of the replicas in C_i to a replica in C_j, is a* stateful valid quorum from C_i, *if*

(1) all messages contain a Paxos State, that is $\Phi \neq \perp$,
(2) and if for any configuration C_t, included in the Confs field of a CPromise *in Q, $t \geq j$ holds.*

Algorithm 3. Procedure to find the highest Paxos state in Q

```
1: Input:
2: MyC.ts                                   {This configurations timestamp}
3: Q = {(Φ, Confs, ID, C), ...}             {CPromises with Φ = (rnd, vrnd, vval)}
                                            {and ∀C_i ∈ Confs : i ≥ MyC.ts}
4: procedure findState(Q)
5:     rnd := max rnd ∈ Q + 1               {Highest rnd from Paxos states, Φ, in Q}
6:     vrnd := max vrnd ∈ Q                 {Highest vrnd from Paxos states, Φ, in Q}
7:     vval := vval(vrnd)                   {vval reported with vrnd}
8:     return (rnd, vrnd, vval)
```

Note that a CPromise is stateful, if and only if it was sent by a stateful replica. The following example shows how our algorithm ensures that only one of two configurations starts running Paxos.

Example 1. Assume an initial stateful configuration C_0 with replicas x, y, and z. Assume two new configurations C_1 and C_2 are initialized. If C_2 receives a stateful valid quorum of CPromises from x and y, and starts Paxos, then x and y received the NewConf message from C_2 before the one from C_1. They will include C_2 in their CPromises to C_1. C_1 will therefore never start running Paxos.

This next example shows, how stateful valid quorums guarantee that a new configuration gets its Paxos State from the last stateful configuration.

Example 2. Assume as above an initial configuration C_0 and concurrently initialized configurations C_1 and C_2. Assume C_1 received an stateful valid quorum of CPromises from C_0 and started running Paxos.

Thus, the replicas in C_0 will inform C_2 about C_1 in there CPromises, and because of condition (2), the CPromises from C_0 to C_2 will not make a stateful valid quorum. Therefore C_2 will ask C_1 for CPromises. The replicas in C_1 will either directly reply to C_2 with a statefulCPromise, or send it, after they determined a Paxos State (Lines 22, 37 and 49).

Optimizations: Algorithm 1 assumes that all replicas in a new configuration are initialized with a Reconf message. And all send NewConf messages and receive CPromises. However, if one replica has determined a Paxos State, the others can simply copy it, instead of waiting for a quorum of CPromises. Thus, after a replica has determined a correct Paxos State using Algorithm 3, it sends this state to all replicas in its configuration, using an Activation message (Line 33). Other replicas can just copy this state, instead of waiting for a quorum of CPromises as explained in Algorithm 2. To reduce the message load, processes in the new configuration can further run a weak leader election algorithm and only the leader would send NewConf messages and receive CPromises. The other replicas can then be started by an Activation message. Note that

we do not require a single leader to emerge. It is only needed that eventually one or several correct leaders are elected. [1] Finally, since we assume reliable communication, messages have to be resent until an acknowledgement is received. In an asynchronous system, this means that messages to faulty processes have to be sent infinitely often. However, NewConf and CPromise messages must only be resent until the new configuration is stateful. The rest can be done by Activation messages.

4.1 The ARec State Machine

Just as Prepare and Promise messages in Paxos, we can send NewConf and CPromise messages for infinitely many instances of Paxos at the same time, since the Paxos State will be the initial state ($\Phi = (0, 0, \text{nil})$), for all except finitely many instances. However it is unpractical to replay all requests of an execution history on the new configuration. A more practical solution is to provide a snapshot of the replicated application to the new replicas when invoking Reconf. If this snapshot incorporates all requests chosen in the first k Paxos instances, it is enough to exchange NewConf and CPromise messages for the Paxos instances starting with $k + 1$.

While in undecided instances a new configuration needs to receive a quorum of Paxos States, in a decided instance, it would suffice to receive the decided value from one replica. This can be used to further reduce the size of CPromise messages in an implementation.

5 Safety of the RSM

We now prove that RSM *safety* cannot be compromised by ARec. That is, only a single value can get chosen in a Paxos instance, even if the instance runs over several configurations. Proving *safety* for Paxos usually relies on the condition that any two quorums intersect [3]. However, this condition does not hold in a dynamic system. We derive a substitute for this condition in Corollary 1.

Recall that a replica is *stateful* if it has Paxos State and *valid* if it does not know of any configuration higher than its own (Lines 8 and 9). A message is stateful or valid *iff* its sender is.

According to Algorithm 1, a replica only starts Paxos when it is stateful and valid (Lines 32-35 and 42-46), and stops Paxos when it becomes invalid (Lines 16-18). A stateful replica always stays stateful. Thus, the sender of a Paxos message is always stateful and valid. We can therefore define all quorums of Promise or Learn messages to be stateful valid quorums, analogue to Definition 3.

Since two quorums do not necessarily intersect, we define the following property to describe that one quorum knows about another quorum.

[1] This leader election is strictly weaker than the one required for Paxos [8]. In this case, a failure detector suspecting everybody forever, and therefore electing all replicas as leaders gives an inefficient but correct solution.

Definition 4. *For two stateful valid quorums of messages, Q and Q', we say that Q' knows Q, writing $Q \mapsto Q'$, if*

(1) Q is a quorum of Paxos messages, and some replica first sent a message in Q and then a message in Q', or
(2) Q is a stateful valid quorum of CPromises *and the Paxos State of some replica, sending a message in Q' was derived using Algorithm 3 on Q.*

We say that Q' knows about Q, writing $Q \dashrightarrow Q'$, if there exist k stateful valid quorums $Q_1, \ldots, Q_k$, such that $Q \mapsto Q_1 \mapsto \ldots \mapsto Q_k \mapsto Q'$.

Note that, condition (2) applies both, if a replica in Q' called findState(Q), or if it received a Activation message including $\Phi = \text{findState}(Q)$.

In Lemma 1 we show how this notion is related to Paxos variables *rnd* and *vrnd*. We therefore first define the *rnd* and *vrnd* of a stateful valid quorum. Note that a stateful valid quorum from C_i, can either be a quorum of Learn or Promise messages sent in the same round, or a quorum of stateful CPromises from C_i.

Definition 5. *For a stateful valid quorum of messages Q we define:*

$$rnd(Q) = \begin{cases} rnd \text{ msgs were send} & \text{for Learns and Promises} \\ \max rnd & \text{for CPromises} \end{cases}$$

$$vrnd(Q) = \begin{cases} rnd \text{ msgs were send} & \text{for Learns} \\ \max vrnd & \text{for Promises} \\ \max vrnd & \text{for CPromises} \end{cases}$$

Where max*rnd and* max*vrnd refer to the highest rnd and vrnd reported in the messages.*

Lemma 1. *Let Q and Q' be stateful valid quorums from C_i and C_j.*
(1) If $Q \mapsto Q'$ and $i = j$, then $rnd(Q) \leq rnd(Q')$.
(2) If $Q \mapsto Q'$ and $i < j$, then $rnd(Q) < rnd(Q')$ and $vrnd(Q) \leq vrnd(Q')$.
(3) If $Q \mapsto Q'$, and Q are Learns*, then $vrnd(Q) \leq vrnd(Q')$ holds.*

Proof. Using the following facts, all cases easily follow from the definitions. In cases (1) and (3) of the Lemma, $Q \mapsto Q'$ is caused by case (1) in Definition 4. The claims easily follow, since Q and Q' have a sender in common. In case (2) of the Lemma, Q is a quorum of CPromises and case (2) of the Definition 4 holds. The claim follows, since $(rnd, vrnd, vval) = \text{findState}(Q)$ implies $rnd = rnd(Q) + 1$ and $vrnd = vrnd(Q)$. (Compare Algorithm 3 and Definition 5.) □

Lemma 2. *For two quorums of Paxos messages Q in C_i and Q' in C_j, if $i < j$ then $Q \dashrightarrow Q'$. Further, we can choose all quorums $Q_1, \ldots, Q_k$ to be quorums of* CPromises *and $Q \mapsto Q_1 \mapsto \ldots \mapsto Q_k \mapsto Q'$ still holds.*

Proof. Clearly there exist sequences $Q_0 \mapsto Q_1 \mapsto \ldots \mapsto Q$ and $Q'_0 \mapsto Q'_1 \mapsto \ldots \mapsto Q'$ with Q_0 and Q'_0 in the initial configuration C_0. We want to show, that

there exists a Q'_i in C_i. Choose maximal t and t', s.t. Q_t and $Q'_{t'}$ are in the same configuration C_u. It follows that $u \leq i$, and Q_t and $Q'_{t'}$ have at least one sender in common. For $u < i$ maximality implies that both Q_t and $Q'_{t'}$ are CPromises.

Assume first $Q_t \mapsto Q'_{t'}$ and $u < i$. Let messages in Q_t be sent to C_v. Thus C_v will be included in at least one CPromise from $Q'_{t'}$. Since $Q'_{t'}$ is a stateful valid quorum it follows from Definition 3, that $Q'_{t'}$ was sent to a configuration $C_{v'}$ with $u < v' \leq v$. Maximality of u implies that $v' < v$. It follows similar, that all quorums $Q'_{t''}$ for $t'' > t'$ are stateful valid quorums of CPromises send to configurations $C_{v''}$ with $v'' < v$. This contradicts that Q' is a valid quorum of Paxos messages.

If we assume that $Q'_{t'} \mapsto Q_t$ and $u < i$, we similarly reach a contradiction, to the fact that Q is a quorum of Paxos messages. It therefore follows, that $u = i$ and $Q'_{t'}$ is from C_i. Since $Q'_{t'}$ is still a quorum of CPromises and no replica sends a Paxos message, after sending a CPromise, it follows that $Q \mapsto Q'_{t'}$. This proofs the Lemma. □

Corollary 1. *For two quorums of Paxos messages, Q and P, either $Q \dashrightarrow P$ or $P \dashrightarrow Q$ holds.*

Proof. This follows from Lemma 2 if Q and P do not happen in the same configuration. Otherwise, it is clear since two quorums of messages in one configuration have at least one sender in common. □

The following corollary and theorem show that ARec preserves safety:

Corollary 2. *Only a single value can be chosen in one round.*

Proof. It is clear that only one value can be chosen in round r in configuration C_i. Assume another value was chosen in round s in configuration C_j, $j \neq i$. Let Q_r be the quorum of Learns in round r in C_i and Q_s the quorum of Learns in round s in C_j. From Corollary 1 it follows that $Q_s \dashrightarrow Q_r$ or vice versa. Since $j \neq i$, $Q_s \mapsto Q_1 \mapsto \ldots \mapsto Q_k \mapsto Q_r$ with $k > 0$ and at least one quorum of CPromises. From Lemma 1 part (1) and (2) it follows that $rnd(Q_s) < rnd(Q_r)$ or vice versa. Therefore $s \neq r$.

Theorem 1. *If a value is safe in round r, no other value was chosen in a lower round.*

Proof. We prove this by induction over r. If $r = 0$, there is nothing to show. Assume $r > 0$ and that the Theorem holds for all rounds smaller than r. If some value v is safe in round r, then a quorum P of Promises showing this was send in round r. Therefore v was also safe in round $vrnd(P)$. Assume some value v' was chosen in round $s < r$, then a quorum Q of Learns was send in round s. From Corollary 1 and Lemma 1 part (1) it follows that $Q \dashrightarrow P$. If we show that $vrnd(P) > s$, we can apply the induction hypothesis on $vrnd(P) < r$, and see that $v' = v$. If $vrnd(P) = s$, we can apply Corollary 2 and get that $v = v'$.

Since $Q \dashrightarrow P$ there exist quorums $Q_1, \ldots, Q_k$, such that $Q \mapsto Q_1 \mapsto \ldots \mapsto Q_k \mapsto P$. According to Lemma 2 we can choose all Q_i to be quorums

of CPromises. Now $s = vrnd(Q) \leq vrnd(Q_1)$ follows from Lemma 1, part (3). And $vrnd(Q_1) \leq \ldots \leq vrnd(P)$ follows from Lemma 1, part (2). Thus $s = vrnd(Q) \leq vrnd(P)$ □

6 Liveness for Reconfiguration

We now proof that ARec is live, that is that ARec Liveness and Dynamic RSM Liveness define in Section 3 hold. We assume throughout this section, that at any point during an execution, all configurations our system depends on are available. Proofs of both liveness properties rely on the following Lemma:

Lemma 3. *If configuration C_i is stateful and some correct replica in C_i knows about a configuration C_j, $j > i$, then the system will eventually no longer depend on C_i.*

Proof. Assume the system depends on C_i. According to the definition in of the depends on relation, the system also depends on all configurations with higher timestamp than i. It is enough, to show that one of these configurations will eventually become stateful. Assume therefore, that C_j is not stateful. Let $c \in C_i$ be a replica that knows about configuration C_j. c learned about C_j, either by receiving a NewConf message from C_j (See Line 16 in Algorithm 1), or by receiving a CPromise or Activation message including C_j (Lines 23,40). In the later cases, c will itself send a CPromise to C_j (Lines 37,49). Since the system depends on C_j, it is available, and will send NewConf messages to C_i either upon initialization (Line 15), or on receiving the CPromise from c (Line 40). Thus eventually the majority of correct replicas in C_i will send CPromises to C_j. If these build an stateful valid quorum, C_j will become stateful and the system will no longer depend on C_i. If the CPromises are no stateful valid quourum, one of the senders knows about a configuration C_k with $i < k < j$. We can now repeat the above arguement with C_k instead of C_j. Clearly we will eventually arrive at a configuration C_l, such that no configuration with timestamp between i and l exists. This configuration will become stateful and the system will no longer depend on C_i. □

Theorem 2. *ARec implements ARec Liveness.*

Proof. Assume there is a stateful configuration C_l and some correct client initialized a reconfiguration $\langle \text{Reconf}, C_i, i, C_l \rangle$. If the system does not depend on C_l, there is nothing to show. Otherwise, both C_l and C_i are available. Since the client is correct, eventually a correct replica in C_i will receive the Reconf message and send a NewConf message. On recieving this message, some correct replica in C_l will know about C_i. Lemma 3 says, that the system will eventually no longer depend on C_l. □

Theorem 3. *ARec, together with the Paxos implement Dynamic RSM Liveness.*

Proof. If, in an execution only finitely many configurations are initialized, eventually one configuration will forever be the stateful configuration with highest timestamp C_{max} and the system will depend on this configuration forever. Thus Lemma 3 implies, that no correct replica in C_{max} knows about a configuration with higher timestamp. Thus all corret replicas in C_{max} are valid (Line 9). Since at least one correct replica in C_{max} is stateful, it will send ACTIVATION messages, until all correct replicas in C_{max} are stateful. Since C_{max} is available by assumption (a) from Dynamic RSM Liveness, it eventually holds a majority of correct, stateful and valid replicas that will run Paxos. If additionally a single replica is elected as a leader, Liveness of Paxos implies that submitted requests can get learned and executed. □

7 Related Work

As mentioned in the introduction, the classical method to reconfigure an RSM was already explained in [1] and [2, 3]. Variations of this method where presented in [9] and it was refined and implemented in works like [10] and [4]. All these works use consensus to decide on the next configuration, resulting in a unique sequence of configurations, where each reconfiguration only involves one old and one new configuration. For this, a configuration must be available and have a single correct leader before it can be changed by reconfiguration.

The work most closely related to ARec is Vertical Paxos [5]. Vertical Paxos also changes configuration between rounds rather than between instances, and uses Paxos' first phase to deduce a starting state for the new configuration, similar to ARec. Also similar to ARec, a new configuration C_i knows about one past configuration C_l that is stateful and must receive a quorum from all configurations with sequence numbers between l and i. Different from ARec however, Vertical Paxos assumes that upon receiving a reconfiguration request, configuration C_i knows about all configurations C_j with $l \leq j < i$. Finally, Vertical Paxos relies on an abstract, fault tolerant configuration manager to order new configurations. Instead of relying on synchrony in the old configuration, Vertical Paxos relies on synchrony in the configuration manager, which must be implemented by another RSM to be fault tolerant. Note also that the configuration manager RSM cannot use Vertical Paxos to reconfigure itself, but has to rely on classical reconfiguration. ARec does not rely on a separate configuration manager.

That asynchronous reconfiguration is possible has previously only been proven for atomic R/W registers [11]. In their protocol, DynaStore, a reconfiguration does not specify a new configuration as a set of replicas, but as a set of operations, each adding or removing one replica. When several reconfigurations are issued concurrently, DynaStore takes the union of these reconfigurations as the configuration that the replicas eventually agree upon. If reconfigurations are issued by several parties, this interface has some disadvantages. For example, if several parties try to reduce the number of replicas, but disagree on the replica to be removed, they could by accident remove all replicas. Thus it is possible to violate DynaStore's Liveness property by issuing an "unlucky" reconfiguration. This is not possible with ARec.

As described in [9], a Dynamic RSM has similarities to group communication systems that provide totally ordered message delivery [12]. However, while view synchronous communication requires that a message received by one process in view v is received by all processes in view v, therefore consensus is required to end a view. A Dynamic RSM ensures that requests learned by one replica in a configuration, are learned by all replicas in some future configuration. Therefore ARec only requires consensus in some future configuration.

8 Conclusion

We have presented ARec a reconfiguration protocol for a Paxos State Machine that enables reconfiguration in a completely asynchronous system. To our knowledge this is the first protocol for asynchronous reconfiguration of an RSM. We have precisely specified the liveness conditions for asynchronous reconfiguration, improving on previous specifications, which allow unlucky reconfigurations to break liveness. Using ARec, availability of RSM based systems can be improved, since reconfiguration can be used to restore synchrony, and thus enable the RSM to make progress. However, since our work mainly focused on proving the possibility of asynchronous reconfiguration, we expect that ARec can be significantly optimized when adjusting it to a specific system.

References

[1] Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. 22(4), 299–319 (1990)
[2] Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. 16(2), 133–169 (1998)
[3] Lamport, L.: Paxos made simple. ACM SIGACT News (December 2001)
[4] Shraer, A., Reed, B., Malkhi, D., Junqueira, F.: Dynamic reconfiguration of primary/backup clusters. USENIX ATC (2011)
[5] Lamport, L., Malkhi, D., Zhou, L.: Vertical paxos and primary-backup replication. In: PODC (2009)
[6] Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)
[7] Chandra, T.D., Hadzilacos, V., Toueg, S., Charron-Bost, B.: On the impossibility of group membership. In: PODC (1996)
[8] Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43, 225–267 (1996)
[9] Lamport, L., Malkhi, D., Zhou, L.: Reconfiguring a state machine. SIGACT News 41(1), 63–73 (2010)
[10] Lorch, J.R., Adya, A., Bolosky, W.J., Chaiken, R., Douceur, J.R., Howell, J.: The smart way to migrate replicated stateful services. In: EuroSys (2006)
[11] Aguilera, M.K., Keidar, I., Malkhi, D., Shraer, A.: Dynamic atomic storage without consensus. J. ACM 58(2), 7 (2011)
[12] Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Comput. Surv. 33(4), 427–469 (2001)