Vertical Atomic Broadcast and Passive Replication

Manuel Bravo
Informal Systems, Madrid, Spain

Gregory Chockler
University of Surrey, Guildford, UK

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

Alejandro Naser-Pastoriza
IMDEA Software Institute, Madrid, Spain
Universidad Politécnica de Madrid, Spain

Christian Roldán
IMDEA Software Institute, Madrid, Spain

Abstract

Atomic broadcast is a reliable communication abstraction ensuring that all processes deliver the same set of messages in a common global order. It is a fundamental building block for implementing fault-tolerant services using either active (aka state-machine) or passive (aka primary-backup) replication. We consider the problem of implementing reconfigurable atomic broadcast, which further allows users to dynamically alter the set of participating processes, e.g., in response to failures or changes in the load. We give a complete safety and liveness specification of this communication abstraction and propose a new protocol implementing it, called Vertical Atomic Broadcast, which uses an auxiliary service to facilitate reconfiguration. In contrast to prior proposals, our protocol significantly reduces system downtime when reconfiguring from a functional configuration by allowing it to continue processing messages while agreement on the next configuration is in progress. Furthermore, we show that this advantage can be maintained even when our protocol is modified to support a stronger variant of atomic broadcast required for passive replication.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Reconfiguration, consensus, replication

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.10

Related Version *Extended Version*: `https://arxiv.org/abs/2408.08702` [4]

Funding This work was partially supported by the projects BYZANTIUM, DECO and PRODIGY funded by MCIN/AEI, and REDONDA funded by the CHIST-ERA network.

1 Introduction

Replication is a widely used technique for ensuring fault tolerance of distributed services. Two common replication approaches are *active* (aka state-machine) replication [33] and *passive* (aka primary-backup) replication [6]. In active replication, a service is defined by a deterministic state machine and is executed on several replicas, each maintaining a copy of the machine. The replicas are kept in sync using *atomic broadcast* [10], which ensures that client commands are delivered in the same order to all replicas; this can be implemented using, e.g., Multi-Paxos [21].

In contrast, in passive replication commands are executed by a single replica (the *leader* or *primary*), which propagates the state updates induced by the commands to the other replicas (*followers* or *backups*). This approach allows replicating services with non-deterministic operations, e.g., those depending on timeouts or interrupts. But as shown in [3, 17, 19],

38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 10; pp. 10:1–10:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

implementing it requires propagating updates from the leader to the followers using a stronger primitive than the classical atomic broadcast. This is because in passive replication, a state update is incremental with respect to the state it was generated in. Hence, to ensure consistency between replicas, each update must be applied by a follower to the same state in which it was generated by the leader. Junqueira et al. formalized the corresponding guarantees by the notion of *primary-order atomic broadcast (POabcast)* [18,19], which can be implemented by protocols such as Zab [18], viewstamped replication [30] or Raft [31].

The above implementations of atomic or primary-order atomic broadcast require replicating data among $2f + 1$ replicas to tolerate f failures. This is expensive: in principle, storing the data at $f + 1$ replicas is enough for it survive f failures. Since with only $f + 1$ replicas even a single replica failure will block the system, to recover we need to *reconfigure* it, i.e., change its membership to replace failed replicas with fresh ones. Unfortunately, processes concurrently deciding to reconfigure the system need to be able to agree on the next configuration; this reduces to solving consensus, which again requires $2f + 1$ replicas [22]. The way out of this conundrum is to use a separate *configuration service* with $2f + 1$ replicas to perform consensus on configurations. In this way we use $2f + 1$ replicas to only store configuration metadata and $f + 1$ replicas to store the actual data. This *vertical approach*, layering replication on top of a configuration service, was originally proposed in RAMBO [27] for atomic registers and in *Vertical Paxos* [23] for single-shot consensus. Since then it has been used by many practical storage systems [2,8,11,14]. These often use reconfiguration not only to deal with failures, but also to make changes to a functional configuration: e.g., to move replicas from highly loaded machines to lightly loaded ones, or to change the number of machines replicating the service [26,29,39].

Unfortunately, while the space of atomic broadcast protocols with $2f + 1$ replicas has been extensively explored, the design of such protocols in vertical settings is poorly understood. Even though one can obtain a vertical solution for atomic broadcast by reducing it to Vertical Paxos, this would make it hard to ensure the additional properties required for passive replication. Furthermore, both Vertical Paxos and similar protocols [3] stop the system as the very first step of reconfiguration, which increases the downtime when reconfiguring from a functional configuration. Due to the absence of a theoretically grounded and efficient atomic broadcast protocol for vertical settings, the designs used in industry are often ad hoc and buggy. For example, until recently the vertical-style protocol used in Kafka, a widely used streaming platform, contained a number of bugs in its failure handling [14]. In this paper we make several contributions to improve this situation.

First, we give a complete safety and liveness specification of *reconfigurable atomic broadcast*, sufficient for active replication (§3). We then propose its implementation in a vertical system with $f + 1$ replicas and an external configuration service, which we call *Vertical Atomic Broadcast (VAB)* (§4). In contrast to prior vertical protocols [3,23], our implementation allows the latest functional configuration to continue processing messages while agreement on the next configuration is in progress. This reduces the downtime when reconfiguring from a functional configuration from 4 message delays in the prior solutions to 0. We rigorously prove that the protocol correctly implements the reconfigurable atomic broadcast specification, including both safety and liveness.

We next consider the case of passive replication (which we review in §5). We propose *speculative primary-order atomic broadcast (SPOabcast)*, which we show to be sufficient for implementing passive replication in a reconfigurable system (§6). A key novel aspect of SPOabcast is that SPOabcast is able to completely eliminate the downtime induced by a *Primary Integrity* property of the existing POabcast [18,19]. This property requires the leader of a new configuration to suspend normal operation until an agreement is reached on which messages broadcast in the previous configurations should survive in the new one: in passive replication, these messages determine the initial service state at the leader. Instead, SPOabcast allows the leader to *speculatively* deliver a tentative set of past messages before the agreement on them has been reached, and then to immediately resume normal broadcasting. SPOabcast guarantees that, if a process delivers a message m_2 broadcast by the new leader, then prior to this the process will also deliver every message m_1 the leader speculatively delivered before broadcasting m_2. This helps ensure that the process applies the update in m_2 to the same state in which the leader generated it, as required for the correctness of passive replication.

We show that SPOabcast can be implemented by modifying our Vertical Atomic Broadcast protocol. The use of speculative delivery allows the resulting protocol to preserve VAB's downtime of 0 when reconfiguring from a functional configuration. It thus allows using Vertical Atomic Broadcast to replicate services with non-deterministic operations.

Overall, we believe that our specifications, protocols and correctness proofs provide insights into the principles underlying existing reconfigurable systems, and can serve as a blueprint for building future ones.

2 System Model

We consider an asynchronous message-passing system consisting of an (infinite) universe of processes $\mathcal{P}$ which may fail by *crashing*, i.e., permanently stopping execution. A process is *correct* if it never crashes, and *faulty* otherwise. Processes are connected by reliable FIFO channels: messages are delivered in FIFO order, and messages between non-faulty processes are guaranteed to be eventually delivered. The system moves through a sequence of *configurations*. A configuration C is a triple $\langle e, M, p_i \rangle$ that consists of an epoch $e \in \mathbb{N}$ identifying the configuration, a finite set of processes $M \subseteq \mathcal{P}$ that belong to the configuration, and a distinguished *leader* process $p_i \in M$. We denote the set of configurations by Config. In contrast to static systems, we do not impose a fixed global bound on the number of faulty processes, but formulate our availability assumptions relative to specific configurations (§3).

Reconfiguration is the process of changing the system configuration. We assume that configurations are stored in an external *configuration service (CS)*, which is reliable and wait-free. The configuration service provides three atomic operations. An operation $\texttt{compare\_and\_swap}(e, \langle e', M, p_l \rangle)$ succeeds iff the epoch of the last stored configuration is e; in this case it stores the provided configuration with a higher epoch $e' > e$. Operations $\texttt{get\_last\_epoch}()$ and $\texttt{get\_members}(e)$ respectively return the last epoch and the members associated with a given epoch e.

In practice, a configuration service can be implemented under partial synchrony using Paxos-like replication over $2f + 1$ processes out of which at most f can fail [22] (as is done in systems such as Zookeeper [17]). Our protocols use the service as a black box, and as a result, do not require any further environment assumptions about timeliness [12] or failure detection [7].

3 Specification

In this section we introduce *reconfigurable atomic broadcast*, a variant of atomic broadcast [10] that allows reconfiguration. The broadcast service allows a process to send an *application message* m from a set Msg using a call $\texttt{broadcast}(m)$. Messages are delivered using a notification $\texttt{deliver}(m)$. Any process may initiate system reconfiguration using a call $\texttt{reconfigure}()$. If successful, this returns the new configuration C arising as a result; otherwise it returns $\bot$. Each process participating in the new configuration then gets a notification $\texttt{conf\_changed}(C)$, informing it about C. In practice, `reconfigure` would take as a parameter a description of the desired reconfiguration. For simplicity we abstract from this in our specification, which states broadcast correctness for any results of reconfigurations.

We record the interactions between the broadcast and its users via *histories* h – sequences of *actions* a of one of the following forms:

$$\begin{aligned}&\texttt{broadcast}_i(m), \quad \texttt{deliver}_i(m), \quad \texttt{conf\_changed}_i(C),\\ &\texttt{reconfig\_req}_i, \quad \texttt{reconfig\_resp}_i(C), \quad \texttt{introduction}_i(C),\end{aligned}$$

where $p_i \in \mathcal{P}$, $m \in \mathsf{Msg}$ and $C \in \mathsf{Config}$. Each action is parameterized by a process p_i where it occurs (omitted when irrelevant). The first three actions respectively record invocations of `broadcast`, `deliver` and `conf_changed`. The next pair of actions record calls to and returns from the `reconfigure` function. Finally, the `introduction` action records the moment when this function stores the new configuration in the configuration service.

For a history h we let h_k be the k-th action in h, and we write $a \in h$ if a occurs in h. We also write _ for an irrelevant value. We only consider histories where calls to and returns from `reconfigure` match, and a process may perform at most one `introduction` action during the execution of `reconfigure`. For simplicity we assume that all application messages broadcast in a single execution are unique:

$$\forall m, k, l.\ h_k = \texttt{broadcast}(m) \ \wedge\ h_l = \texttt{broadcast}(m) \implies k = l.$$

For a history h, a partial function $\mathrm{epochOf} : \mathbb{N} \rightharpoonup \mathbb{N}$ returns the epoch of the action in h with a given index. This is the epoch of the latest preceding `conf_changed` at the same process:

$$\begin{aligned}\mathrm{epochOf}(k) = e \iff (\exists i, l, a.\, h_k = a_i \ \wedge\ h_l = \texttt{conf\_changed}_i(\langle e, \_, \_\rangle) \ \wedge\ l < k \ \wedge\\ \forall l'.\ l < l' < k \implies h_{l'} \neq \texttt{conf\_changed}_i(\langle \_, \_, \_\rangle)).\end{aligned}$$

When $\mathrm{epochOf}(k) = e$, we say that the action h_k occurs in e.

Reconfigurable atomic broadcast is defined by the properties over histories h listed in Figure 1. Properties 1 and 2 are self-explanatory. Property 3 ensures that processes cannot deliver messages in contradictory orders. Property 4 disallows executions where sequences of messages delivered at different processes diverge.

The liveness requirements of reconfigurable atomic broadcast are given by Property 5. Property 5a asserts a termination guarantee for reconfiguration requests. As shown by Spiegelman and Keidar [36], wait-free termination is impossible to support even for reconfigurable read/write registers, which are weaker than atomic broadcast. Hence, the guarantee given by Property 5a is similar to obstruction-freedom [15]. Let us say that a configuration C is *activated* when all its members get $\texttt{conf\_changed}(C)$ notifications. Property 5a asserts that, in a run with finitely many reconfigurations, the last reconfiguration request invoked by a correct process and executing in isolation must eventually succeed to introduce a configuration C, which must then become activated if all its members are correct. Properties 5b-c state liveness guarantees for the configuration C similar to those of the classical atomic broadcast. Property 5b asserts that any message broadcast by a (correct) member of C eventually gets delivered to all members of C. Property 5c additionally ensures that the members of C eventually deliver all messages delivered by any process in any configuration.

As in prior work [1, 3, 37], the liveness of our protocols is premised on the following assumption, which limits the power of the environment to crash configuration members.

1. **Basic Configuration Change Properties.**
   a. Any epoch e is associated with unique membership and leader:

   $$\forall e, i, j, M_1, M_2.\ \texttt{conf\_changed}(\langle e, M_1, p_i\rangle) \in h \ \wedge\ \texttt{conf\_changed}(\langle e, M_2, p_j\rangle) \in h \implies p_i = p_j \ \wedge\ M_1 = M_2$$

   b. If a process p_i joins a configuration $C = \langle \_, M, \_\rangle$, then p_i is a member of M:

   $$\forall i, M.\ \texttt{conf\_changed}_i(\_, M, \_) \in h \implies p_i \in M$$

   c. Processes join configurations with monotonically increasing epochs:

   $$\forall e_1, e_2, i, k, l.\ h_k = \texttt{conf\_changed}_i(e_1, \_, \_) \ \wedge\ h_l = \texttt{conf\_changed}_i(e_2, \_, \_) \ \wedge\ k < l \implies e_1 < e_2$$

   d. Any configuration a process joins is introduced; a configuration is introduced at most once:

   $$\forall C.\ (\texttt{conf\_changed}(C) \in h \implies \texttt{introduction}(C) \in h) \ \wedge\ (\forall k, l.\ h_k = \texttt{introduction}(C) \ \wedge\ h_l = \texttt{introduction}(C) \implies k = l)$$

2. **Integrity.** A process delivers a given application message m at most once, and only if m was previously broadcast:

   $$\forall m, i, k, l.\ h_k = \texttt{deliver}_i(m) \ \wedge\ h_l = \texttt{deliver}_i(m) \implies k = l \ \wedge\ \exists j.\ h_j = \texttt{broadcast}(m) \ \wedge\ j < k$$

3. **Total Order.** If some process delivers m_1 before m_2, then any process that delivers m_2 must also deliver m_1 before this:

   $$\forall m_1, m_2, i, j, k, l, l'.\ h_k = \texttt{deliver}_i(m_1) \ \wedge\ h_l = \texttt{deliver}_i(m_2) \ \wedge\ k < l \ \wedge\ h_{l'} = \texttt{deliver}_j(m_2) \implies \exists k'.\ h_{k'} = \texttt{deliver}_j(m_1) \ \wedge\ k' < l'$$

4. **Agreement.** If p_i delivers m_1 and p_j delivers m_2, then either p_i delivers m_2 or p_j delivers m_1:

   $$\forall m_1, m_2, i, j.\ \texttt{deliver}_i(m_1) \in h \ \wedge\ \texttt{deliver}_j(m_2) \in h \implies (\texttt{deliver}_i(m_2) \in h \ \vee\ \texttt{deliver}_j(m_1) \in h)$$

5. **Liveness.** Consider an execution with finitely many reconfiguration requests (`reconfig_req`), and let r be the last reconfiguration request to be invoked. Suppose that r is invoked by a correct process and no other reconfiguration call takes steps after r is invoked. Then r terminates, having introduced a configuration $C = \langle e, M, p_i\rangle$: $\texttt{reconfig\_resp}(C)$. Furthermore, if all processes in M are correct, then:
   a. all processes in M deliver $\texttt{conf\_changed}(C)$;
   b. if $p_i \in M$ broadcasts m while in e, then all processes in M eventually deliver m;
   c. if a process delivers m, then all processes in M eventually deliver m.

Figure 1 Properties of reconfigurable atomic broadcast over a history h.

▶ **Assumption 1** (Availability). *Let $C = \langle e, M, \_\rangle$ be an introduced configuration, i.e., such that $\texttt{introduction}(C) \in h$. Then at least one member of M does not crash before another configuration $C' = \langle e', \_, \_\rangle$ with $e' > e$ is activated.*

Our protocols use the period of time when some member of M is guaranteed not to crash to copy its state to the members of a new configuration.

Finally, we note that in the case of a single static configuration, our specification in Figure 1 corresponds to the classical notion of atomic broadcast [10].

```
1  epoch $\leftarrow 0 \in \mathbb{Z}$
2  new_epoch $\leftarrow 0 \in \mathbb{Z}$
3  next $\leftarrow 0 \in \mathbb{Z}$
4  init_len $\leftarrow -1 \in \mathbb{Z}$
5  last_delivered $\leftarrow -1 \in \mathbb{Z}$
6  members $\in 2^{\mathcal{P}}$
7  leader $\in \mathcal{P}$
8  msg[] $\in \mathbb{N} \to$ Msg $\cup \{\perp\}$
9  status $\in \{$LEADER, FOLLOWER, FRESH$\}$
10 function broadcast($m$):
11     send FORWARD($m$) to leader
12 when received FORWARD($m$) from $p_j$
13 // function broadcast($m$):
14     pre: $p_i$ = leader
15     msg[next] $\leftarrow m$
16     send ACCEPT(epoch, next, $m$)
         to members $\setminus \{p_i\}$
17     next $\leftarrow$ next + 1
18 when received ACCEPT($e, k, m$) from $p_j$
19     pre: status = FOLLOWER $\wedge$ epoch = $e$
20     msg[$k$] $\leftarrow m$
21     send ACCEPT_ACK($e, k$) to $p_j$
22 when received ACCEPT_ACK($e, k$)
     from all members $\setminus \{p_i\}$
23     pre: status = LEADER $\wedge$ epoch = $e$
24     send COMMIT($e, k$) to members
25 when received COMMIT($e, k$)
26     pre: status $\in \{$LEADER, FOLLOWER$\} \wedge$
           epoch = $e \wedge k$ = last_delivered + 1
27     last_delivered $\leftarrow k$
28     deliver(msg[$k$])
```

Figure 2 Vertical Atomic Broadcast at a process p_i: normal operation.

4 The Vertical Atomic Broadcast Protocol

In Figures 2 and 3 we present a protocol implementing the specification of §3, which we call *Vertical Atomic Broadcast (VAB)* by analogy with Vertical Paxos [23]. For now the reader should ignore the code in blue. At any given time, a process executing the protocol participates in a single configuration, whose epoch is stored in a variable epoch. The membership of the configuration is stored in a variable members. Every member of a given configuration is either the leader or a *follower*. A status variable at a process records whether it is a LEADER, a FOLLOWER, or is in a special FRESH state used for new processes. A leader variable stores the leader of the current configuration. We assume that the system starts in an initial active configuration with epoch 0.

Normal operation. When a process receives a call broadcast(m), it forwards m to the leader of its current configuration (line 10). Upon receiving m (line 12), the leader adds it to an array msg; a next variable points to the first free slot in the array (initially 0). The leader then sends m to the followers in an ACCEPT(e, k, m) message, which carries the leader's epoch e, the position k of m in the msg array, and the message m itself.

A process acts on the ACCEPT message (line 19) only if it participates in the corresponding epoch. It stores m in its local copy of the msg array and sends an ACCEPT_ACK(e, k) message to the leader of e. The application message at position k is *committed* if the leader of e receives ACCEPT_ACK messages for epoch e and position k from all followers of its configuration (line 22). In this case the leader notifies all the members of its configuration that the application message can be safely delivered via a COMMIT message. A process delivers application messages in the order in which they appear in its msg array, with last_delivered storing the last delivered position (line 25).

Reconfiguration: probing. Any process can initiate a reconfiguration, e.g., to add new processes or to replace failed ones. Reconfiguration aims to preserve the following invariant, key to proving the protocol correctness.

```
29 function reconfigure():
30     var $e, M, e_{\text{new}}, M_{\text{new}}$
31     $e \leftarrow$ get_last_epoch() at CS
32     $e_{\text{new}} \leftarrow e + 1$
33     repeat
34         if $e \geq 0$ then
35             $M \leftarrow$ get_members($e$) at CS
36             send PROBE($e_{\text{new}}, e$) to $M$
37             wait until received
                   PROBE_ACK(_, $e_{\text{new}}$)
                   from a process in $M$
38             $e \leftarrow e - 1$
39     until received PROBE_ACK(TRUE, $e_{\text{new}}$)
        from some $p_j$
40     $M_{\text{new}} \leftarrow$ compute_membership()
41     if compare_and_swap($e_{\text{new}}-1, \langle e_{\text{new}}, M_{\text{new}}, p_j\rangle$)
        at CS /* introduction($\langle e_{\text{new}}, M_{\text{new}}, p_j\rangle$) */
        then
42         send NEW_CONFIG($e_{\text{new}}, M_{\text{new}}$) to $p_j$
43         return $\langle e_{\text{new}}, M_{\text{new}}, p_j\rangle$
44     else
45         return $\perp$

46 when received PROBE($e_{\text{new}}, e$) from $p_j$
47     pre: $e_{\text{new}} \geq$ new_epoch
48     new_epoch $\leftarrow e_{\text{new}}$
49     if epoch $\geq e$ then
50         send PROBE_ACK(TRUE, $e_{\text{new}}$) to $p_j$
51     else
52         send PROBE_ACK(FALSE, $e_{\text{new}}$) to $p_j$

53 when received NEW_CONFIG($e, M$)
    from $p_j$
54     pre: new_epoch $= e$
55     status $\leftarrow$ LEADER
56     epoch $\leftarrow e$
57     members $\leftarrow M$
58     leader $\leftarrow p_i$
59     next $\leftarrow \max\{k \mid$ msg$[k] \neq \perp\} + 1$
60     init_len $\leftarrow$ next $- 1$
61     conf_changed($e, M, p_i$)
62     // conf_changed($e, M, p_i$,
           msg[last_delivered+1..init_len])
63     send NEW_STATE($e$, msg, $M$)
        to members $\setminus \{p_i\}$

64 when received NEW_STATE($e, msg, M$)
    from $p_j$
65     pre: new_epoch $\leq e$
66     status $\leftarrow$ FOLLOWER
67     epoch $\leftarrow e$
68     new_epoch $\leftarrow e$
69     msg $\leftarrow msg$
70     leader $\leftarrow p_j$
71     conf_changed($e, M, p_j$)
72     // conf_changed($e, M, p_j, \perp$)
73     send NEW_STATE_ACK($e$) to $p_j$

74 when received NEW_STATE_ACK($e$)
    from all members $\setminus \{p_i\}$
75     pre: new_epoch $=$ epoch $= e$
76     for $k = 1$..init_len do
        send COMMIT($e, k$) to members
```

Figure 3 Vertical Atomic Broadcast at a process p_i: reconfiguration.

▶ **Invariant 1.** *Assume that the leader of an epoch e sends* COMMIT(e, k) *while having* msg$[k] = m$*. Whenever any process p_i has* epoch $= e' > e$*, it also has* msg$[k] = m$*.*

The invariant ensures that any application message committed in an epoch e will persist at the same position in all future epochs e'. This is used to establish that the protocol delivers application messages in the same order at all processes.

To ensure Invariant 1, a process performing a reconfiguration first *probes* the previous configurations to find a process whose state contains all messages that could have been committed in previous epochs, which will serve as the new leader. The new leader then transfers its state to the followers of the new configuration. We say that a process is *initialized* at an epoch e when it completes the state transfer from the leader of e; it is at this moment that the process assigns e to its epoch variable, used to guard the transitions at lines 18, 22, 25. Our protocol guarantees that a configuration with epoch e can become activated only after all its members have been initialized at e. Probing is complicated by the fact that there may be a series of failed reconfiguration attempts, where the new leader fails before initializing all its followers. For this reason, probing may require traversing epochs from the current one down, skipping epochs that have not been activated.

In more detail, a process p_r initiates a reconfiguration by calling `reconfigure` (line 29). The process picks an epoch number e_{new} higher than the current epoch stored in the configuration service and then starts the probing phase. The process p_r keeps track of the epoch being probed in e and the membership of this epoch in M. The process initializes these variables when it obtains the information about the current epoch from the configuration service. To probe an epoch e, the process sends a `PROBE`(e_{new}, e) message to the members of its configuration, asking them to join the new epoch e_{new} (line 36). Upon receiving this message (line 46), a process first checks that the proposed epoch e_{new} is $\geq$ the highest epoch it has ever been asked to join, which is stored in new_epoch (we always have epoch $\leq$ new_epoch). In this case, the process sets new_epoch to e_{new}. Then, if the process was initialized at an epoch $\geq$ the epoch e being probed, it replies with `PROBE_ACK`$(\text{TRUE}, e_{\text{new}})$; otherwise, it replies with `PROBE_ACK`$(\text{FALSE}, e_{\text{new}})$.

If p_r receives at least one `PROBE_ACK`$(\text{FALSE}, e_{\text{new}})$ from a member of e (line 37), p_r can conclude that e has not been activated, since one of its processes was not initialized by the leader of this epoch. The process p_r can also be sure that e will never become activated, since it has switched at least one of its members to the new epoch. In this case, p_r starts probing the preceding epoch $e - 1$. Since no application message could have been committed in e, picking a new leader from an earlier epoch will not lose any committed messages and thus will not violate Invariant 1. If p_r receives some `PROBE_ACK`$(\text{TRUE}, e_{\text{new}})$ messages, then it ends probing: any process p_j that replied in this way can be selected as the new leader (in particular, p_r is free to maintain the old leader if this is one of the processes that replied).

Reconfiguration: initialization. Once the probing finds a new leader p_j (line 39), the process p_r computes the membership of the new configuration using a function `compute_membership` (line 40). We do not prescribe any particular implementation for this function, except that the new membership must contain the new leader p_j. In practice, the function would take into account the desired changes to be made by the reconfiguration. Once the new configuration is computed, p_r attempts to store it in the configuration service using a `compare_and_swap` operation. This succeeds if and only if the current epoch in the configuration service is still the epoch from which p_r started probing, which implies that no concurrent reconfiguration occurred during probing. In this case p_r sends a `NEW_CONFIG` message with the new configuration to the new leader and returns the new configuration to the caller of `reconfigure`; otherwise, it returns $\perp$. A successful `compare_and_swap` also generates an `introduction`$_r$ action for the new configuration, which is used in the broadcast specification (§3).

When the new leader receives the `NEW_CONFIG` message (line 53), it sets status to LEADER, epoch to the new epoch, and stores the information about the new configuration in members and leader. The leader also sets next to the first free slot in the msg array and saves its initial length in a variable init_len. The leader then invokes `conf_changed` for the new configuration. In order to finish the reconfiguration, the leader needs to transfer its state to the other members of the configuration. To this end, the leader sends a `NEW_STATE` message to them, which contains the new epoch and a copy of its msg array (line 63; a practical implementation would optimize this by sending to each process only the state it is missing). Upon receiving a `NEW_STATE` message (line 64), a process overwrites its msg array with the one provided by the leader, sets its status to FOLLOWER, epoch to the new epoch, and leader to the new leader. The process also invokes `conf_changed` for the new configuration. It then acknowledges its initialization to the leader with an `NEW_STATE_ACK` message. Upon receiving `NEW_STATE_ACK` messages from all followers (line 74), the new leader sends `COMMIT`s for all application messages from the previous epoch, delimited by init_len. These messages can be safely delivered, since they are now stored by all members of epoch e.

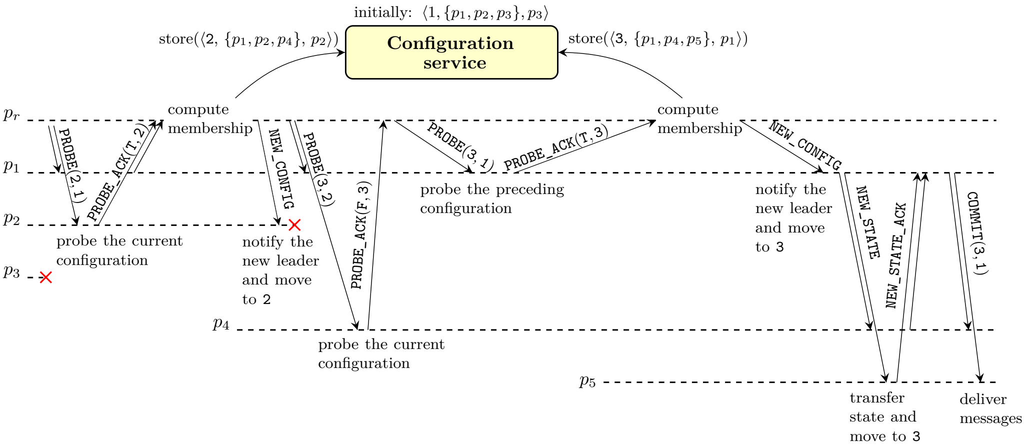
initially: ⟨1, {p_1, p_2, p_3}, p_3⟩
store(⟨2, {p_1, p_2, p_4}, p_2⟩)
Configuration service
store(⟨3, {p_1, p_4, p_5}, p_1⟩)
p_r
p_1
p_2
p_3
p_4
p_5
PROBE(2, 1)
PROBE_ACK(T, 2)
compute membership
probe the current configuration
NEW_CONFIG
PROBE(3, 2)
notify the new leader and move to 2
PROBE_ACK(F, 3)
probe the current configuration
PROBE(3, 1)
PROBE_ACK(T, 3)
probe the preceding configuration
compute membership
NEW_CONFIG
notify the new leader and move to 3
NEW_STATE
NEW_STATE_ACK
COMMIT(3, 1)
transfer state and move to 3
deliver messages

Figure 4 The behavior of the protocol during reconfiguration.

Example. Figure 4 gives an example illustrating the message flow of reconfiguration. Assume that the initial configuration 1 consists of processes p_1, p_2 and p_3. Following a failure of p_3, a process p_r initiates reconfiguration to move the system to a new configuration 2. To this end, p_r sends PROBE(2, 1) to the members of configuration 1. Both processes p_1 and p_2 respond to p_r with PROBE_ACK(TRUE, 2). The process p_r computes the membership of the new configuration, replacing p_3 by a fresh process p_4, and stores the new configuration in the configuration service, with p_2 as the new leader. Next, p_r sends a NEW_CONFIG message to p_2.

Assume that after receiving this message p_2 fails, prompting p_r to initiate yet another reconfiguration to move the system to a configuration 3. To this end, p_r sends PROBE(3, 2) to the members of configuration 2, and p_4 responds with PROBE_ACK(FALSE, 3). The process p_r concludes that epoch 2 has not been activated and starts probing the preceding epoch 1: it sends PROBE(3, 1) and gets a reply PROBE_ACK(TRUE, 3) from p_1, which is selected as the new leader. The process p_r computes the new set of members, replacing p_3 by a fresh process p_5, stores the new configuration in the configuration service, and sends a NEW_CONFIG message to the new leader p_1. This process invokes the conf_changed upcall for the new configuration and sends its state to the followers in a NEW_STATE message. The followers store the state, invoke conf_changed upcalls and reply with NEW_STATE_ACKs. Upon receiving these, p_1 sends COMMITs for all application messages in its state.

Steady-state latency and reconfiguration downtime. A configuration is *functional* if it was activated and all its members are correct. A configuration is *stable* if it is functional and no configuration with a higher epoch is introduced. The *steady-state latency* is the maximum number of message delays it takes from the moment the leader p_i of a stable configuration receives a broadcast request for a message m and until m is delivered by p_i. It is easy to see that our protocol has the steady-state latency of 2 (assuming self-addressed messages are received instantaneously), which is optimal [22].

The system may be reconfigured not only in response to a failure, but also to make changes to a functional configuration: e.g., to move replicas from highly loaded machines to lightly loaded ones, or to change the number of machines replicating the service [26,29,39]. As modern online services have stringent availability requirements, it is important to minimize the period of time when a service is unavailable due to an ongoing reconfiguration. More precisely, suppose the system is being reconfigured from a functional configuration C to a stable configuration C'. The reconfiguration *downtime* is the maximum number of message delays it takes from the moment C is disabled and until the leader of C' is ready to broadcast application messages in the new configuration.

As we argue in §7, existing vertical solutions for atomic broadcast stop the system as the first step of reconfiguration [3], resulting in the reconfiguration downtime of at least 4 (2 message delays to disable the latest functional configuration plus at least 2 message delays to reach consensus on the next configuration and propagate the decision). In contrast, our protocol achieves the downtime of 0 by keeping the latest functional configuration active while the probing of past configurations and agreement on a new one is in progress.

▶ **Theorem 1.** *The VAB protocol reconfigures a functional configuration with* 0 *downtime.*

Proof. Suppose that the current configuration C with an epoch e is functional. Note that the normal path of our protocol is guarded by preconditions $\mathsf{epoch} = e$, so that C can broadcast and deliver application messages as long as this holds at all its members (lines 19, 23 and 26). Assume now that a process p_r starts reconfiguring the system to a new configuration C' with epoch $e + 1$. The process p_r will send `PROBE` messages to the members of C and, since C is functional, p_r will only get replies $\texttt{PROBE\_ACK}(\textsc{true}, e + 1)$. Handling a `PROBE` message only modifies the `new_epoch` variable, not `epoch`. Therefore, C can continue processing broadcasts while p_r is probing its members, storing C' in the configuration service, and sending $\texttt{NEW\_CONFIG}(e + 1, \_)$ to the leader p_i of C'. When the new leader p_i handles $\texttt{NEW\_CONFIG}(e + 1, \_)$, it will set $\mathsf{epoch} = e + 1$, disabling the old configuration. However, the leader will at once be ready to broadcast messages in the new configuration, as required. ◀

Correctness. Our protocol achieves the above 0-downtime guarantee without violating correctness. Informally, this is because it always chooses the leader of the new configuration from among the members of the latest activated configuration, and a message can only be delivered in this configuration after having been replicated to all its members. Hence, the new leader will immediately know about all previously delivered messages, including those delivered during preliminary reconfiguration steps. The following theorem (proved in [4, §A]) states the correctness of our protocol.

▶ **Theorem 2.** *The VAB protocol correctly implements reconfigurable atomic broadcast as defined in Figure 1.*

5 Passive Replication

The protocol presented in the previous section can be used to build reconfigurable fault-tolerant services via *active* (aka state-machine) replication [33]. Here a service is defined by a deterministic state machine and is executed on several replicas, each maintaining a copy of the machine. All replicas execute all client commands, which they receive via atomic broadcast. Together with the state machine's determinism, this ensures that each command yields the same result at all replicas, thus maintaining an illusion of a centralized fault-tolerant service.

In the rest of the paper, we focus on an alternative approach of building reconfigurable fault-tolerant services via *passive* (aka primary-backup) replication [6]. Here commands are only executed by the leader, which propagates the state changes induced by the commands to the other replicas. This allows replicating services with non-deterministic operations, e.g., those depending on timeouts or interrupts.

Formally, we consider services with a set of states $\mathcal{S}$ that accept a set of commands $\mathcal{C}$. A command $c \in \mathcal{C}$ can be executed using a call $\texttt{execute}(c)$, which produces its return value. Command execution may be non-deterministic. To deal with this, the effect of executing a command c on a state $\Sigma \in \mathcal{S}$ is defined by transition relation $\Sigma \xrightarrow{c} \langle r, \delta \rangle$, which produces a possible return value r of c and a *state update* δ performed by the command. The latter can be applied to any state Σ' using a function $\mathit{apply}(\Sigma', \delta)$, which produces a new state. For example, a command **if** $x = 0$ **then** $y \leftarrow 1$ **else** $y \leftarrow 0$ produces a state update $y \leftarrow 1$ when executed in a state with $x = 0$. A command assigning x to a random number may produce an update $x \leftarrow 42$ if the random generator returned 42 when the leader executed the command.

We would like to implement a service over a set of fault-prone replicas that is linearizable [16] with respect to a service that atomically executes commands on a single non-failing copy of the state machine. The latter applies each state update to the machine state Σ immediately after generating it, as shown in Figure 5. Informally, this means that commands appear to clients as if produced by a single copy of the state machine in Figure 5 in an order consistent with the *real-time order*, i.e., the order of non-overlapping command invocations.

5.1 Passive Replication vs Atomic Broadcast

As observed in [3, 17, 19], implementing passive replication requires propagating updates from the leader to the followers using a stronger primitive than atomic broadcast. To illustrate why, Figure 6 gives an incorrect attempt to simulate the specification in Figure 5 using our reconfigurable atomic broadcast (ignore the code in blue for now). This attempt serves as a strawman for a correct solution we present later. Each process keeps track of the epoch it belongs to in `cur_epoch` and the leader of this epoch in `cur_leader`. To execute a command (line 5), a process sends the command, tagged by a unique identifier, to the leader. It then waits until it hears back about the result.

A process keeps two copies of the service state – a *committed* state Σ and a *speculative* state Θ; the latter is only used when the process is the leader. When the leader receives a command c (line 10), it executes c on its speculative state Θ, producing a return value r and a state update δ. The leader immediately applies δ to Θ and distributes the triple of the command identifier, its return value and the state update via atomic broadcast. When a process (including the leader) delivers such a triple (line 15), it applies the update to its committed state Σ and sends the return value to the process the command originated at, determined from the command identifier. When a process receives a `conf_changed` upcall (line 18), it stores the information received in `cur_epoch` and `cur_leader`. If the process is the leader of the new epoch, it also initializes its speculative state Θ to the committed state Σ.

In passive replication, a state update is incremental with respect to the state it was generated in. Thus, to simulate the specification in Figure 5, it is crucial that the committed state Σ at a process delivering a state update (line 16) be the same as the speculative state Θ from which this state update was originally derived (line 12). This is captured by the following invariant. Let $\Sigma_i(k)$ denote the value of Σ at process p_i before the k-th action in the history (and similarly for Θ).

▶ **Invariant 2.** *Let h be a history of the algorithm in Figure 6. If $h_k = \mathtt{deliver}_i(m)$, then there exist j and $l < k$ such that $h_l = \mathtt{broadcast}_j(m)$ and $\Sigma_i(k) = \Theta_j(l)$.*

Unfortunately, if we use atomic broadcast to disseminate state updates in Figure 6, we may violate Invariant 2. We next present two examples showing how this happens and how this leads to violating linearizability. The examples consider a replicated counter x with two commands – an increment ($x \leftarrow x + 1$) and a read (**return** x). Initially $x = 0$, and then two clients execute two increments.

```
$\Sigma \leftarrow \Sigma_0 \in \mathcal{S}$
function execute($c$):
    $\Sigma \xrightarrow{c} \langle r, \delta \rangle$
    $\Sigma \leftarrow apply(\Sigma, \delta)$
    return $r$
```

Figure 5 Passive replication specification.

```
cur_epoch $\in \mathbb{N}$
cur_leader $\in \mathcal{P}$
$\Sigma \leftarrow \Sigma_0 \in \mathcal{S}$  // committed state
$\Theta \leftarrow \Theta_0 \in \mathcal{S}$  // speculative state
function execute($c$):
    $id \leftarrow$ get_unique_id()
    send EXECUTE($id, c$) to cur_leader
    wait until receive RESULT($id, r$)
    return $r$

when received EXECUTE($id, c$)
    pre: cur_leader $= p_i$
    $\Theta \xrightarrow{c} \langle r, \delta \rangle$
    $\Theta \leftarrow apply(\Theta, \delta)$
    broadcast($\langle id, r, \delta \rangle$)

upon deliver($\langle id, r, \delta \rangle$)
    $\Sigma \leftarrow apply(\Sigma, \delta)$
    send RESULT($id, r$) to $origin(id)$

upon conf_changed($\langle e, M, p_j \rangle$)
    // conf_changed($\langle e, M, p_j \rangle, \sigma$)
    cur_epoch $\leftarrow e$
    cur_leader $\leftarrow p_j$
    if $p_i = p_j$ then
        $\Theta \leftarrow \Sigma$
        // $\langle \_, \_, \delta_1 \rangle \dots \langle \_, \_, \delta_k \rangle \leftarrow \sigma$
        // forall $l = 1..k$ do $\Theta \leftarrow apply(\Theta, \delta_l)$
```

Figure 6 Passive replication on top of broadcast: code at process p_i.

Example 1. The two increments are executed by the same leader. The first one generates an update $\delta_1 = (x \leftarrow 1)$ and a speculative state $\Theta = 1$. Then the second generates $\delta_2 = (x \leftarrow 2)$. Atomic broadcast allows processes to deliver the updates in the reverse order, with δ_1 applied to a committed state $\Sigma = 2$. This violates Invariant 2. Assume now that after the increments complete we change the configuration to move the leader to a different process. This process will initialize its speculative state Θ to the committed state $\Sigma = 1$. If the new leader now receives a read command, it will return 1, violating the linearizability with respect to Figure 5.

Example 2. The first increment is executed by the leader of an epoch e, which generates $\delta_1 = (x \leftarrow 1)$. The second increment is executed by the leader of an epoch $e' > e$ before it delivers δ_1 and, thus, in a speculative state $\Theta = 0$. This generates $\delta_2 = (x \leftarrow 1)$. Finally, the leader of e' delivers δ_1 and then δ_2, with the latter applied to a committed state $\Sigma = 1$. This is allowed by atomic broadcast yet violates Invariant 2. It also violates linearizability similarly to Example 1: if now the leader of e' receives a read, it will incorrectly return 1.

5.2 Primary-Order Atomic Broadcast

To address the above problem, Junqueira et al. proposed *primary-order atomic broadcast (POabcast)* [18, 19], which strengthens the classical atomic broadcast. We now briefly review POabcast and highlight its drawbacks, which motivates an alternative proposal we present in the next section. In our framework we can define POabcast by adding the properties over histories h in Figure 7 to those of Figure 1. This yields a reconfigurable variant of POabcast that we call *reconfigurable primary-order atomic broadcast (RPOabcast)*. RPOabcast also modifies the interface of reconfigurable atomic broadcast (§3) by only allowing a process to call `broadcast` if it is the leader of its current configuration.

Property 6 (Local Order) restricts the delivery order of messages broadcast in the same epoch: they must be delivered in the order the leader broadcast them. Property 7 (Global Order) restricts the delivery order of messages broadcast in different epochs: they must be delivered in the order of the epochs they were broadcast in. Finally, Property 8 (Primary Integrity) ensures that the leader of an epoch e' does not miss relevant messages from previous epochs: each message broadcast in an epoch $e < e'$ either has to be delivered by the leader before entering e', or can never be delivered at all. Local and Global Order trivially imply Property 3 (Total Order), so we could omit it from the specification. POabcast is stronger than plain atomic broadcast: the latter can be implemented from the former if each process forwards messages to be broadcast to the leader of its configuration.

▶ **Proposition 3.** *Reconfigurable atomic broadcast can be implemented from RPOabcast.*

When the passive replication protocol in Figure 6 is used with POabcast instead of plain atomic broadcast, Invariant 2 holds, and the protocol yields a service linearizable with respect to the specification in Figure 5 [19]. In particular, Local Order disallows Example 1 from §5.1, and Primary Integrity disallows Example 2 (which does not violate either Local or Global Order). POabcast can be obtained from our Vertical Atomic Broadcast (VAB) algorithm in §4 as follows. First, VAB already guarantees both Local and Global Order: e.g., this is the case for Local Order because processes are connected by reliable FIFO channels.

▶ **Theorem 4.** *VAB guarantees Local and Global order.*

Second, to ensure Primary Integrity, neither the new leader nor the followers invoke `conf_changed` upon receiving `NEW_CONFIG` (line 61) or `NEW_STATE` (line 71). Instead, the leader first waits until it receives `NEW_STATE_ACK` messages from all followers (line 74) and tells the processes to deliver all application messages from the previous epoch via `COMMIT` messages. Only once a process delivers all these application messages does it invoke `conf_changed` for the new configuration (and if the process is the leader, starts broadcasting).

Deferring the invocation of `conf_changed` at the leader is the key to guarantee Primary Integrity. On the one hand, it ensures that, before the newly elected leader of an epoch e' generates `conf_changed`, it has delivered all application messages that could have been delivered in previous epochs: Invariant 1 from §4 guarantees that the leader's initial log includes all such messages. On the other hand, the leader can also be sure that any message broadcast in an epoch $< e'$ but not yet delivered can *never* be delivered by any process. This is because, by the time the leader generates `conf_changed`, all followers in e' have overwritten their log with that of the new leader.

Since deferring `conf_changed` results in deferring the start of broadcasting by the leader, the modified VAB protocol has a reconfiguration downtime of 2 messages delays. This cost is inherent: the lower bound of Friedman and van Renesse [13] on the latency of Strong Virtually Synchronous broadcast (a variant of POabcast) implies that any solution must have a non-zero downtime. In the next section we circumvent this limitation by introducing a weaker variant of POabcast, which we show sufficient for passive replication.

6 Speculative Primary-Order Atomic Broadcast

We now introduce *speculative primary-order atomic broadcast* (SPOabcast), a weaker variant of POabcast that allows implementing passive replication with minimal downtime. During reconfiguration, SPOabcast allows the new leader to deliver messages from previous epochs *speculatively* – without waiting for them to become durable – and start broadcast right away.

6. **Local Order.** If the leader of some epoch e receives $\texttt{broadcast}(m_1)$ before receiving $\texttt{broadcast}(m_2)$, then any process that delivers m_2 must also deliver m_1 before m_2:

$$\begin{aligned}&\forall m_1, m_2, i, j, k, l, l'.\ h_k = \texttt{broadcast}_i(m_1) \ \wedge\ h_l = \texttt{broadcast}_i(m_2) \ \wedge\ k < l \ \wedge\\ &\text{epochOf}(k) = \text{epochOf}(l) \ \wedge\ h_{l'} = \texttt{deliver}_j(m_2) \implies \exists k'.\ h_{k'} = \texttt{deliver}_j(m_1) \ \wedge\ k' < l'\end{aligned}$$

7. **Global Order.** Assume the leaders of e and $e' > e$ receive $\texttt{broadcast}(m_1)$ and $\texttt{broadcast}(m_2)$ respectively. If a process p_i delivers m_1 and m_2, then it must deliver m_1 before m_2:

$$\begin{aligned}&\forall m_1, m_2, i, k, k', l, l'.\ h_k = \texttt{broadcast}(m_1) \ \wedge\ h_l = \texttt{broadcast}(m_2) \ \wedge\\ &\text{epochOf}(k) < \text{epochOf}(l) \ \wedge\ h_{k'} = \texttt{deliver}_i(m_1) \ \wedge\ h_{l'} = \texttt{deliver}_i(m_2) \implies k' < l'\end{aligned}$$

8. **Primary Integrity.** Assume some process delivers an application message m originally broadcast in an epoch e. If any process p_i joins an epoch $e' > e$, then p_i must deliver m before joining e':

$$\begin{aligned}&\forall m, i, k, l, l', e, e'.\ h_k = \texttt{broadcast}(m) \ \wedge\ \text{epochOf}(k) = e \ \wedge\ h_l = \texttt{deliver}(m) \ \wedge\\ &h_{l'} = \texttt{conf\_changed}_i(\langle e', \_, \_\rangle) \ \wedge\ e < e' \implies \exists k'.\ h_{k'} = \texttt{deliver}_i(m) \ \wedge\ k' < l'\end{aligned}$$

Figure 7 Properties of reconfigurable primary-order atomic broadcast over a history h.

9. **Basic Speculative Delivery Properties.** A process p_i can speculatively deliver a given application message m at most once in a given epoch and only if p_i is the leader of the epoch, m has previously been broadcast, and m has not yet been delivered by p_i:

$$\begin{aligned}&\forall i, j, k, \sigma.\ h_k = \texttt{conf\_changed}_i(\langle \_, \_, p_j\rangle, \sigma) \implies (\sigma \neq \bot \implies p_i = p_j) \ \wedge\ (\forall m_1, m_2 \in \sigma.\ m_1 \neq m_2)\\ &\wedge\ (\forall m \in \sigma.\ (\exists l.\ h_l = \texttt{broadcast}(m) \ \wedge\ l < k) \ \wedge\ (\neg \exists l.\ h_l = \texttt{deliver}_i(m) \ \wedge\ l < k))\end{aligned}$$

10. **Prefix Consistency.**
    a. Consider m_1 and m_2 broadcast in different epochs. Assume that a process p_i delivers m_2, and a process p_j broadcasts m_2 in an epoch e'. Then p_i delivers m_1 before m_2 iff p_j delivers m_1 before joining e' or speculatively delivers m_1 when joining e':

$$\begin{aligned}&\forall m_1, m_2, i, j, k_0, l_0, k, l, l', \sigma, e'.\ h_{k_0} = \texttt{broadcast}(m_1) \ \wedge\ h_{l_0} = \texttt{broadcast}(m_2) \ \wedge\\ &\text{epochOf}(k_0) \neq \text{epochOf}(l_0) \ \wedge\ h_k = \texttt{deliver}_i(m_2) \ \wedge\ h_l = \texttt{broadcast}_j(m_2) \ \wedge\\ &h_{l'} = \texttt{conf\_changed}_j(\langle e', \_, p_j\rangle, \sigma) \ \wedge\ \text{epochOf}(l) = e' \implies\\ &((\exists k'.\ h_{k'} = \texttt{deliver}_i(m_1) \ \wedge\ k' < k) \iff ((\exists l''.\ h_{l''} = \texttt{deliver}_j(m_1) \ \wedge\ l'' < l') \ \vee\ m_1 \in \sigma)\end{aligned}$$

    b. Consider m_1 and m_2 broadcast in different epochs. Assume that a process p_i delivers m_2, and a process p_j speculatively delivers m_2 when joining an epoch e'. Then p_i delivers m_1 before m_2 iff p_j delivers m_1 before joining e' or speculatively delivers m_1 before m_2 when joining e':

$$\begin{aligned}&\forall m_1, m_2, i, j, k_0, l_0, k, l, \sigma, e'.\ h_{k_0} = \texttt{broadcast}(m_1) \ \wedge\ h_{l_0} = \texttt{broadcast}(m_2) \ \wedge\\ &\text{epochOf}(k_0) \neq \text{epochOf}(l_0) \ \wedge\ h_k = \texttt{deliver}_i(m_2) \ \wedge\ h_l = \texttt{conf\_changed}_j(\langle e', \_, p_j\rangle, \sigma) \ \wedge\\ &m_2 \in \sigma \implies ((\exists k'.\ h_{k'} = \texttt{deliver}_i(m_1) \ \wedge\ k' < k) \iff\\ &((\exists l'.\ h_{l'} = \texttt{deliver}_j(m_1) \ \wedge\ l' < l) \ \vee\ \sigma = \_m_1\_m_2\_))\end{aligned}$$

p_i	p_j
deliver m_1	(speculatively) deliver m_1
deliver m_2	broadcast m_2

(a)

time ↓

p_i	p_j
deliver m_1	(speculatively) deliver m_1
deliver m_2	speculatively deliver m_2

(b)

Figure 8 Properties of speculative primary-order atomic broadcast over a history h. Property 10 replaces Property 8 from Figure 7. The tables summarize its action orderings: the actions at the top happen before the actions at the bottom.

SPOabcast specification. SPOabcast modifies the interface of reconfigurable atomic broadcast (§3) in two ways. First, like in POabcast, a process can call `broadcast` only if it is the leader. Second, the `conf_changed` upcall for a configuration C carries an additional argument σ: `conf_changed`(C, σ). When the upcall is invoked at the leader of C, σ is a sequence of messages *speculatively delivered* to the leader (σ is not used at followers). SPOabcast is defined by replacing Primary Integrity in the definition of POabcast by the properties in Figure 8. Property 9 is self-explanatory. Property 10 (Prefix Consistency) constrains how speculative deliveries are ordered with respect to ordinary deliveries and broadcasts. For the ease of understanding, in Figure 8 we summarize these orderings in tables.

Part (a)/"only if" of Prefix Consistency is a weaker form of Primary Integrity. Assume that the leader p_j of an epoch e' broadcasts a message m_2. The property ensures that for any message m_1 delivered before m_2 at some process p_i, the leader p_j has to either deliver m_1 before joining e' or *speculatively deliver* m_1 *when joining* e'. As we demonstrate shortly, the latter option, absent in Primary Integrity, allows our implementation of SPOabcast to avoid extra downtime during reconfiguration. Part (a)/"if" conversely ensures that, if the leader p_j speculatively delivers m_1 before broadcasting m_2, then m_1 must always be delivered before m_2. This ensures that the speculation performed by the leader p_j is correct if any of the messages it broadcasts (e.g., m_2) are ever delivered at any process. Part (b) of Prefix Consistency ensures that the order of messages in a sequence speculatively delivered at a `conf_changed` upcall cannot contradict the order of ordinary delivery.

Speculative delivery provides weaker guarantees than ordinary delivery, since it does not imply durability. In particular, we allow a message to be speculatively delivered at a process p but never delivered anywhere, e.g., because p crashed. However, in this case Part (a)/"if" of Prefix Consistency ensures that all messages p broadcast after such a non-durable speculative delivery will also be lost. As we show next, this allows us to use SPOabcast to correctly implement passive replication without undermining its durability guarantees.

Passive replication using SPOabcast. The passive replication protocol in Figure 6 requires minimal changes to be used with SPOabcast, highlighted in blue. When the leader of an epoch e receives a `conf_changed` upcall for e (line 19), in addition to setting the speculative state Θ to the committed state Σ, the leader also applies the state updates speculatively delivered via `conf_changed` to Θ (lines 24-25). The leader can then immediately use the resulting speculative state to execute new commands (line 10). We prove the following in [4, §B].

▶ **Theorem 5.** *The version of the protocol in Figure 6 that uses SPOabcast satisfies Invariant 2 and implements a service linearizable with respect to the specification in Figure 5.*

In particular, part (a)/"only if" of Prefix Consistency disallows Example 2 from §5.1: it ensures that the leader broadcasting δ_2 will be aware of δ_1, either via ordinary or speculative delivery. More generally, part (a) ensures that, if a process p_i delivers a state update δ_2 broadcast by a leader p_j, then at the corresponding points in the execution, p_i and p_j are aware of the same set of updates (cf. the table in Figure 8). Part (b) of Prefix Consistency furthermore ensures that the two processes apply these updates in the same order. This contributes to validating Invariant 2 and, thus, the specification in Figure 5.

Implementing SPOabcast. To implement SPOabcast we modify the Vertical Broadcast Protocol in Figures 2-3 as follows. First, since `broadcast` can only be called at the leader, we replace lines 10-12 by line 13. Thus, the leader handles `broadcast` calls in the same way it previously handled `FORWARD` messages. Second, we augment `conf_changed` upcalls with speculative deliveries, replacing line 61 by line 62, and line 71 by line 72. Thus, the `conf_changed` upcall at the leader speculatively delivers all application messages in its log that have not yet been (non-speculatively) delivered. It is easy to check that these modifications do not change the 0-downtime guarantee of Vertical Atomic Broadcast.

▶ **Theorem 6.** *The primary-order version of the Vertical Atomic Broadcast protocol is a correct implementation of speculative primary-order atomic broadcast.*

Thus, Theorems 5 and 6 allow us to use VAB to replicate even non-deterministic services while minimizing the downtime from routine reconfigurations, e.g., those for load balancing.

We prove Theorem 6 in [4, §C]. Here we informally explain why the above protocol validates the key part (a)/“only if” of Prefix Consistency, weakening Primary Integrity (cf. the explanations we gave regarding the latter at the end of §5.2). On the one hand, as in the ordinary VAB, Invariant 1 from §4 guarantees that the log of a newly elected leader of an epoch e' contains all application messages m_1 that could have been delivered in epochs $< e'$. The new leader will either deliver or speculatively deliver all such messages before broadcasting anything (line 62). On the other hand, if the leader broadcasts a message m_2, then a follower will only accept it after having overwritten its log with the leader’s initial one, received in `NEW_STATE` (line 64). This can be used to show that, if m_2 is ever delivered, then any message broadcast in an epoch $< e'$ that was not in `NEW_STATE` will never get delivered.

7 Related Work

The *vertical* paradigm of implementing reconfigurable services by delegating agreement on configuration changes to a separate component was first introduced by Lynch and Shvartsman [27] for emulating dynamic atomic registers. It was further applied by Lamport et al. [23] to solve reconfigurable single-shot consensus, yielding the Vertical Paxos family of protocols. Vertical Paxos and its follow-ups [3, 5, 11, 25, 28] require prior configurations to be disabled (“wedged”) at the start of reconfiguration. In contrast, our VAB protocol allows the latest functional configuration to continue processing messages while the agreement on the next configuration is in progress. This results in the downtime of 0 when reconfiguring from a functional configuration. This feature is particularly desirable for atomic broadcast, where we want to keep producing new decisions when reconfiguration is triggered for load balancing rather than to handle failures.

To achieve the minimal downtime, the VAB protocol uses different epoch variables to guard the normal operation (epoch) and reconfiguration (new_epoch). By not modifying the epoch variable during the preliminary reconfiguration steps, the protocol allows the old configuration to operate normally while the reconfiguration is in progress (cf. the proof of Theorem 1 in §4). In contrast, Vertical Paxos uses a single epoch variable (maxBallot) for both purposes, thus disabling the current configuration at the start of reconfiguration. Our protocol for SPOabcast further extends the minimal downtime guarantee to the case of passive replication.

Both our VAB and SPOabcast protocols achieve an optimal steady-state latency of two message delays [22]. Although Junqueira et al. [19] show that no POabcast protocol can guarantee optimal steady-state latency if it relies on black-box consensus to order messages, our SPOabcast implementation is not subject to this impossibility result, as it does not use consensus in this manner.

Although the vertical approach has been widely used in practice [2, 8, 11, 32, 38], prior systems have mainly focused on engineering aspects of directly implementing a replicated state machine for a desired service rather than basing it on a generic atomic broadcast layer.

Our treatment of Vertical Atomic Broadcast develops a formal foundation that sheds light on the algorithmic core of these systems. This can be reused for designing future solutions that are provably correct and efficient.

Most reconfiguration algorithms that do not rely on an auxiliary configuration service can be traced back to the original technique of Paxos [21], which intersperses reconfigurations within the stream of normal command agreement instances. The examples of practical systems that follow this approach include SMART [26], Raft [31], and Zookeeper [35]. Other non-vertical algorithms [24] implement reconfiguration by spawning a separate non-reconfigurable state machine for each newly introduced configuration. In the absence of an auxiliary configuration service, these protocols require at least $2f + 1$ processes in each configuration [22], in contrast to $f + 1$ in our atomic broadcast protocols.

The fault-masking protocols of Birman et al. [3] and a recently proposed MongoDB reconfiguration protocol [34] separate the message log from the configuration state, but nevertheless replicate them at the same set of processes. As in non-vertical solutions, these algorithms require $2f+1$ replicas. They also follow the Vertical Paxos approach to implement reconfiguration, and as a result, may wedge the system prematurely as we explain above.

A variant of Primary Integrity, known as Strong Virtual Synchrony (or Sending View Delivery [9]), was originally proposed by Friedman and van Renesse [13] who also studied its inherent costs. Our SPOabcast abstraction is a relaxation of Strong Virtually Synchrony and primary-order atomic broadcast (POabcast) of Junqueira et al. [18,19]. Keidar and Dolev [20] proposed Consistent Object Replication Layer (COReL) in which every delivered message is assigned a color such that a message is "yellow" if it was received and acknowledged by a member of an operational quorum, and "green" if it was acknowledged by all members of an operational quorum. While the COReL's yellow messages are similar to our speculative messages, Keidar and Dolev did not consider their potential applications, in particular, their utility for minimizing the latency of passive replication.

References

1 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2), 2011. `doi:10.1145/1944345.1944348`.

2 Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. CORFU: A distributed shared log. *ACM Trans. Comput. Syst.*, 31(4), 2013. `doi:10.1145/2535930`.

3 Kenneth Birman, Dahlia Malkhi, and Robbert van Renesse. Virtually synchronous methodology for building dynamic reliable services. In *Guide to Reliable Distributed Systems - Building High-Assurance Applications and Cloud-Hosted Services*, chapter 22. Springer, 2012.

4 Manuel Bravo, Gregory Chockler, Alexey Gotsman, Alejandro Naser-Pastoriza, and Christian Roldán. Vertical atomic broadcast and passive replication (extended version). *arXiv*, abs/2408.08702, 2024. URL: `https://arxiv.org/abs/2408.08702`.

5 Manuel Bravo and Alexey Gotsman. Reconfigurable atomic transaction commit. In *Symposium on Principles of Distributed Computing (PODC)*, 2019.

6 Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. In *Distributed Systems (2nd Ed.)*. ACM Press/Addison-Wesley, 1993.

7 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2), 1996. `doi:10.1145/226643.226647`.

8 Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

9 Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv.*, 33(4), 2001. doi:10.1145/503112.503113.

10 Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4), 2004. doi:10.1145/1041680.1041682.

11 Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Symposium on Operating Systems Principles (SOSP)*, 2015.

12 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2), 1988. doi:10.1145/42282.42283.

13 Roy Friedman and Robbert van Renesse. Strong and weak virtual synchrony in Horus. In *Symposium on Reliable Distributed Systems (SRDS)*, 1996.

14 Jason Gustafson. Hardening Kafka replication. Talk at Kafka Summit San Francisco, 2018. URL: https://www.confluent.io/kafka-summit-sf18/hardening-kafka-replication/.

15 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *International Conference on Distributed Computing Systems (ICDCS)*, 2003.

16 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990. doi:10.1145/78969.78972.

17 Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference (USENIX ATC)*, 2010.

18 Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Conference on Dependable Systems and Networks (DSN)*, 2011.

19 Flavio Paiva Junqueira and Marco Serafini. On barriers and the gap between active and passive replication. In *Symposium on Distributed Computing (DISC)*, 2013.

20 Idit Keidar and Danny Dolev. Efficient message ordering in dynamic networks. In *Symposium on Principles of Distributed Computing (PODC)*, 1996.

21 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998. doi:10.1145/279227.279229.

22 Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2), 2006. doi:10.1007/S00446-006-0155-X.

23 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and primary-backup replication. In *Symposium on Principles of Distributed Computing (PODC)*, 2009.

24 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1), 2010. doi:10.1145/1753171.1753191.

25 Leslie Lamport and Mike Massa. Cheap Paxos. In *Conference on Dependable Systems and Networks (DSN)*, 2004.

26 Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The SMART way to migrate replicated stateful services. In *European Conference on Computer Systems (EuroSys)*, 2006.

27 Nancy Lynch and Alex A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Symposium on Distributed Computing (DISC)*, 2002.

28 John MacCormick, Chandramohan A. Thekkath, Marcus Jager, Kristof Roomp, Lidong Zhou, and Ryan Peterson. Niobe: A practical replication protocol. *ACM Trans. Storage*, 3(4), 2008. doi:10.1145/1326542.1326543.

29 Neha Narkhede, Gwen Shapira, and Todd Palino. *Kafka: The Definitive Guide.* O'Reilly Media, 2017.

30 Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*, 1988.

31 Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

32 Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using Paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, 4(4), 2011. `doi:10.14778/1938545.1938549`.

33 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4), 1990. `doi:10.1145/98163.98167`.

34 William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis. Design and analysis of a logless dynamic reconfiguration protocol. In *Conference on Principles of Distributed Systems (OPODIS)*, 2021.

35 Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Paiva Junqueira. Dynamic reconfiguration of primary/backup clusters. In *USENIX Annual Technical Conference (USENIX ATC)*, 2012.

36 Alexander Spiegelman and Idit Keidar. On liveness of dynamic storage. In *Colloquium on Structural Information and Communication Complexity (SIROCCO)*, 2017.

37 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: Abstraction and optimal asynchronous solution. In *Symposium on Distributed Computing (DISC)*, 2017.

38 Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

39 Michael J. Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M. Hellerstein, Heidi Howard, Faisal Nawab, and Ion Stoica. Matchmaker Paxos: A reconfigurable consensus protocol. *J. Syst. Res.*, 1(1), 2021.