On Barriers and the Gap between Active and Passive Replication

(Extended Abstract)

Flavio P. Junqueira[1] and Marco Serafini[2]

[1] Microsoft Research, Cambridge, UK
fpj@apache.org
[2] Yahoo! Research, Barcelona, Spain
serafini@yahoo-inc.com

Abstract. Active replication is commonly built on top of the atomic broadcast primitive. Passive replication, which has been recently used in the popular ZooKeeper coordination system, can be naturally built on top of the primary-order atomic broadcast primitive. Passive replication differs from active replication in that it requires processes to cross a *barrier* before they become primaries and start broadcasting messages. In this paper, we propose a barrier function τ that explains and encapsulates the differences between existing primary-order atomic broadcast algorithms. We also show that implementing primary-order atomic broadcast on top of a generic consensus primitive and τ inherently results in higher time complexity than atomic broadcast, as witnessed by existing algorithms. We overcome this problem by presenting an alternative, primary-order atomic broadcast implementation that builds on top of a generic consensus primitive and uses consensus itself to form a barrier. This algorithm is modular and matches the time complexity of existing τ-based algorithms.

1 Introduction

Passive replication is a popular approach to achieve fault tolerance in practical systems [3]. Systems like ZooKeeper [8] or Megastore [1] use primary replicas to produce state updates or state mutations. Passive replication uses two types of replicas: primaries and backups. A primary replica executes client operations, without assuming that the execution is deterministic, and produces state updates. Backups apply state updates in the order generated by the primary. With active replication, by contrast, all replicas execute all client operations, assuming that the execution is deterministic. Replicas execute a sequence of consensus instances on client operations to agree on a single execution sequence using atomic broadcast (abcast). Passive replication has a few advantages such as simplifying the design of replicated systems with non-deterministic operations, *e.g.*, those depending on timeouts or interrupts.

It has been observed by Junqueira *et al.* [9] and Birman *et al.* [2] that using atomic broadcast for passive, instead of active, replication requires taking care of specific constraints. State updates must be applied in the exact sequence in which they have been generated: if a primary is in state A and executes an operation making it transition to

state update B, the resulting state update δ_{AB} must be applied to state A. Applying it to a different state $C \neq A$ is not safe because it might lead to an incorrect state, which is inconsistent with the history observed by the primary and potentially the clients. Because a state update is the difference between a new state and the previous, there is a causal dependency between state updates. Unfortunately, passive replication algorithms on top of atomic broadcast (*abcast*) do not necessarily preserve this dependency: if multiple primaries are concurrently present in the system, they may generate conflicting state updates that followers end up applying in the wrong order. Primary-order atomic broadcast (*POabcast*) algorithms, like Zab [9], have additional safety properties that solve this problem. In particular, it implements a *barrier*, the *isPrimary* predicate, which must be crossed by processes that want to broadcast messages.

Interestingly, the only existing passive replication algorithm using consensus as a communication primitive, the semi-passive replication algorithm of Defago *et al.* [7], has linear time complexity in the number of concurrently submitted requests. Recent algorithms for passive replication have constant complexity but they directly implement *POabcast* without building on top of consensus [2, 9].

During our work on the ZooKeeper coordination system [8] we have realized that it is still not clear how these algorithms relate, and whether this trade-off between modularity and time complexity is inherent. This paper shows that existing implementations of passive replication can be seen as instances of the same unified consensus-based *POabcast* algorithm, which is basically an atomic broadcast algorithm with a barrier predicate implemented through a *barrier function* τ we define in this work. The τ function outputs the identifier of the consensus instance a leader process must decide on before becoming a primary.

Existing algorithms constitute alternative implementations of τ; the discriminant is whether they consider the underlying consensus algorithm as a black-box whose internal state cannot be observed. Our τ-based algorithm exposes an inherent trade off. We show that if one implements τ while considering the consensus implementation as a black box, it is *necessary* to execute consensus instances sequentially, resulting in higher time complexity. This algorithm corresponds to semi-passive replication.

If the τ implementation can observe the internal state of the consensus primitive, we can avoid the impossibility and execute parallel instances. For example, Zab is similar to the instance of our unified algorithm that uses Paxos as the underlying consensus algorithm and implements the barrier by reading the internal state of the Paxos protocol. We experimentally evaluate that using parallel instances almost doubles the maximum throughput of passive replication in stable periods, even considering optimizations such as batching. Abstracting away these two alternatives and their inherent limitations regarding time complexity and modularity is one of the main observations of this paper.

Finally, we devise a *τ-free POabcast* algorithm that makes this trade off unnecessary, since it enables running parallel consensus instances using an unmodified consensus primitive as a black box. Unlike barrier-based algorithms, a process becomes a primary by proposing a special value in the next available consensus instances; this value marks the end of the sequence of accepted messages from old primaries. Table 1 compares the different PO abcast algorithms we discuss in our paper.

Table 1. Time complexity of *POabcast* algorithms presented in this paper - see Sect. 5.4 and 6 for detail. We consider the use of Paxos as the underlying consensus algorithm since it has optimal latency [12]. However, only the third solution *requires* the use of Paxos; the other algorithms can use any implementation of consensus. For the latency analysis only, we assume that message delays are equal to Δ. The *Stable periods* column reports the time, in a passive replication system, between the receipt of a client request and its delivery by a single broadcasting primary/leader (c is the number of clients). The *Leader change* column reports idle time after a new single leader is elected by Ω and before it can broadcast new messages.

	Stable periods	Leader change
Atomic broadcast [11]	2Δ	2Δ
τ-based *POabcast* (Sect. 5.1)	$2\Delta \cdot c$	4Δ
τ-based *POabcast* with white-box Paxos (Sect. 5.3)	2Δ	4Δ
τ-free *POabcast* (Sect. 6)	2Δ	4Δ

Our barrier-free algorithm shows that both active and passive replication can be implemented on top of a black-box consensus primitive with small and well understood changes and without compromising performance.

2 Related Work

Traditional work on passive replication and the primary-backup approach assumes synchronous links [3]. Group communication has been used to support primary-backup systems; it assumes a $\Diamond P$ failure detector for liveness [6]. Both atomic broadcast and *POabcast* can be implemented in a weaker system model, *i.e.*, an asynchronous system equipped with an Ω leader oracle [5]. For example, our algorithms do not need to agree on a new view every time a non-primary process crashes.

Some papers have addressed the problem of reconfiguration: dynamically changing the set of processes participating to the state machine replication group. Vertical Paxos supports reconfiguration by using an external master, which can be a replicated state machine [13]. This supports *primary-backup* systems, defined as replicated systems where write quorums consist of all processes and each single process is a read quorum. Vertical Paxos does not address the issues of passive replication and considers systems where commands, not state updates, are agreed upon by replicas. Virtually Synchronous Paxos (VS Paxos) aims at combining virtual synchrony and Paxos for reconfiguration [2]. Our work assumes a fixed set of processes and does not consider the problem of reconfiguring the set of processes participating to consensus. Shraer *et al.* have recently shown that reconfiguration can be implemented *on top* of a *POabcast* construction as the ones we present in this paper, making it an orthogonal topic [15].

While there has been a large body of work on group communication, only few algorithms implement passive replication in asynchronous systems with Ω failure detectors: semi-passive replication [7], Zab [9] and Virtually synchronous Paxos [2]. We relate these algorithms with our barrier-based algorithms in Sect. 5.5.

Pronto is an algorithm for database replication that shares several design choices with our τ-free algorithm and has the same time complexity in stable periods [14]. Both

algorithms elect a primary using an unreliable failure detector and have a similar notion of epochs, which are associated to a single primary. Epoch changes are determined using an agreement protocol, and values from old epochs that are agreed upon after a new epoch has been agreed upon are ignored. Pronto, however, is an active replication protocol: all replicas execute transactions, and non-determinism is handled by agreeing on a per-transaction log of non-deterministic choices that are application specific. Our work focuses on passive replication algorithms, their difference with active replication protocols, and on the notion of barriers in their implementation.

3 System Model and Primitives

Throughout the paper, we consider an asynchronous system composed of a set $\Pi = \{p_1, \ldots, p_n\}$ of processes that can fail by crashing. They implement a passive replication algorithm, executing requests obtained by an unbounded number of client processes, which can also fail by crashing. *Correct* processes are those that never crash. Processes are equipped with an Ω failure detector oracle.

Definition 1 (Leader election oracle). *A leader election oracle Ω operating on a set of processes Π outputs the identifier of some process $p \in \Pi$. Instances of the oracle running on different processes can return different outputs. Eventually, all instances of correct processes permanently output the same correct process.*

Our algorithms build on top of (uniform) consensus, which has the following properties.

Definition 2 (Consensus). *A consensus primitive consists of two operations:* propose*(v) and* decide*(v) of a value v. It satisfies the following properties:*

Termination. *If some correct process proposes a value, every correct process eventually decides some value.*
Validity. *If a processes decides a value, this value was proposed by some process.*
Integrity. *Every correct process decides at most one value.*
Agreement. *No two processes decide differently.*

Since our algorithms use multiple instances of consensus, *propose* and *decide* have an additional parameter denoting the identifier of the consensus instance.

Primary order atomic broadcast (*POabcast*) is an intermediate abstraction used by our unified passive replication algorithm. *POabcast* provides a broadcast primitive *POabcast* and a delivery primitive *POdeliver*. *POabcast* satisfies all safety properties of atomic broadcast.

Definition 3 (Atomic broadcast). *An atomic broadcast primitive consists of two operations:* broadcast *and* deliver *of a value. It satisfies the following properties:*

Integrity. *If some process delivers v then some process has broadcast v.*
Total Order. *If some process delivers v before v' then any process that delivers v' must deliver v before v'.*

Agreement. *If some process p_i delivers v and some other process p_j delivers v', then either p_i delivers v' or p_j delivers v.*[1]

POabcast extends atomic broadcast by introducing the concept of primary and a *barrier:* the additional *isPrimary*() primitive, which *POabcast* uses to signal when a process is ready to broadcast state updates. This predicate resembles $Prmy_s$ in the specification of Budhiraja *et al.* [3]. However, as failure detectors are unreliable in our model, primary election is also unreliable: there might be multiple concurrent primaries at any given time, unlike in [3].

A *primary epoch* for a process p is a continuous period of time during which *isPrimary*() is true at p and therefore p is a *primary*. Multiple primaries can be present at any given time: the *isPrimary*() predicate is local to a single process and multiple primary epochs can overlap in time. Let P be the set of primaries such that at least one value they propose is ever delivered by some process. A *primary mapping* Λ is a function that maps each primary epoch in P to a unique *primary identifier* λ, which we also use to denote the process executing the primary role. We consider primaries as logical processes: saying that event ϵ occurs at primary λ is equivalent to saying that ϵ occurs at some process p during a primary epoch for p having primary identifier λ.

Definition 4 (Primary order atomic broadcast). *A primary order atomic broadcast primitive consists of two operations* broadcast(v) *and* deliver(v)*, and of a binary* isPrimary$()$ *predicate, which indicates whether a process is a primary and is allowed to broadcast a value. Let Λ be a primary mapping and $\prec_\Lambda$ a total order relation among primary identifiers. Primary order broadcast satisfies the Integrity, Total order, and Agreement properties of atomic broadcast; furthermore, it also satisfies the following additional properties:*

Local Primary Order. *If λ broadcasts v before v', then a process that delivers v' delivers v before v'.*

Global Primary Order. *If λ broadcasts v, λ' broadcasts v', $\lambda \prec_\Lambda \lambda'$, and some process p delivers v and v', then p delivers v before v'.*

Primary Integrity. *If λ broadcasts v, λ' broadcasts v', $\lambda \prec_\Lambda \lambda'$, and some process delivers v, then λ' delivers v before it broadcasts v'.*

These properties are partially overlapping, as we show in the full version of the paper [10]. For example, Global primary order is very useful in reasoning about the behaviour of *POabcast*, but it can be implied from the other *POabcast* properties. It is also worth noting that Local primary order is weaker than the single-sender FIFO property, since it only holds within a single primary epoch.

The above properties focus on safety. For liveness, it is sufficient to require the following:

Definition 5 (Eventual Single Primary). *There exists a correct process such that eventually it is elected primary infinitely often and all messages it broadcasts are delivered by some process.*

Definition 6 (Delivery Liveness). *If a process delivers v then eventually every correct process delivers v.*

[1] We modified the traditional formulation of agreement to state it as a safety property only.

```
initially
    dec ← 0;
    prop ← 0;
upon POabcast(v) ∧ isPrimary()
    prop ← max(prop + 1, dec + 1);
    propose(v, prop);
upon decide(v, dec + 1)
    dec ← dec + 1;
    POdeliver(v);
function isPrimary()
    return (dec ≥ τ) ∧ (Ω = p);
```

Algorithm 1: *POabcast* based on the barrier function and consensus - process p

4 Unified *POabcast* Algorithm Using the Barrier Function

In passive replication, a primary replica is responsible for executing client operations and for broadcasting state updates. All replicas apply the state updates they deliver to their local state. As we argued in the introduction, atomic broadcast is not sufficient to preserve a correct ordering of state updates. In the full version of this paper, we give an example of incorrect ordering with atomic broadcast and show that using *POabcast* is sufficient to guarantee correctness [10]. In addition to the *POabcast* and *POdeliver* primitives, replicas use the *isPrimary*() predicate to determine whether they should take the primary role.

We now introduce our unified τ-based *POabcast* algorithm (Algorithm 1). It uses three underlying primitives: consensus, the Ω leader oracle, and a new *barrier function* τ we will define shortly.

Like typical atomic broadcast algorithms, our *POabcast* algorithm runs a sequence of consensus instances, each associated with an instance identifier [4]. Broadcast values are proposed using increasing consensus instance identifiers, tracked using the $prop$ counter. Values are decided and delivered following the consensus instance order: if the last decided instance was dec, only the event *decide*$(v, dec + 1)$ can be activated, resulting in an invocation of *POdeliver*. This abstracts the buffering of out-of-order decisions between the consensus primitive and our algorithm.

The most important difference between our algorithm and an implementation of atomic broadcast is that it imposes an additional barrier condition for broadcasting messages: it must hold *isPrimary*. In particular, it is necessary for safety that $dec \geq \tau$. The barrier function τ returns an integer and is defined as follows.

Definition 7 (Barrier function). *Let σ be an infinite execution, Λ a primary mapping in σ, $\prec_\Lambda$ a total order among the primary identifiers, and λ a primary such that at least one value it proposes is delivered in σ. A barrier function τ for λ returns:*[2]

$$\tau = \max\{i : \exists v, p, \lambda' \; s.t. \; decide_p(v, i) \in \sigma \wedge propose_{\lambda'}(v, i) \in \sigma \wedge \lambda' \prec_\Lambda \lambda\}$$

[2] Subscripts denote the process that executes the *propose* or *decide* steps.

An actual implementation of the τ function can only observe the finite prefix of σ preceding its invocation; however, it must make sure that its outputs are valid in any infinite extension of the current execution. If none of the values proposed by a primary during a primary epoch are ever delivered, τ can return arbitrary values.

We show in the full paper [10] that this definition of τ is sufficient to guarantee the additional properties of *POabcast* compared to atomic broadcast. In particular, it is key to guarantee that the primary integrity property is respected. Local primary order is obtained by delivering elements in the order in which they are proposed and decided.

The key to defining a barrier function is identifying a primary mapping Λ and a total order of primary identifiers $\prec_\Lambda$ that satisfy the barrier property, as we will show in the following section. There are some important observations to do here. First, we use the same primary mapping Λ and total order $\prec_\Lambda$ for the barrier function and for *POabcast*. Note also that a primary might not know its identifier λ: this is only needed for the correctness argument.

5 Implementations of the Barrier Function τ

5.1 Barriers with Black-Box Consensus

We first show how to implement τ using the consensus primitive as a black box. This solution is modular but imposes the use of sequential consensus instances: a primary is allowed to have at most one outstanding broadcast at a time. This corresponds to the semi-passive replication algorithm [7].

Let $prop$ and dec be the variables used in Algorithm 1, and let τ_{seq} be equal to $\max(prop, dec)$. We have the following result:

Theorem 1. *The function τ_{seq} is a barrier function.*

Proof. We define Λ as follows: if a leader process p proposes a value $v_{i,p}$ for consensus instance i and $v_{i,p}$ is decided, p has primary identifier $\lambda = i$. A primary has only one identifier: after $v_{i,p}$ is broadcast, it holds $prop > dec$ and $dec < \tau_{seq}$, so *isPrimary*() stops evaluating to true at p. The order $\prec_\Lambda$ is defined by ordering primary identifiers as regular integers.

If a process p proposes a value v for instance $i = \max(prop + 1, dec + 1)$ in Algorithm 1, it observes $\tau_{seq} = \max(prop, dec) = i - 1$ when it becomes a primary. If v is decided, p has primary identifier $\lambda = i$. All primaries preceding λ in $\prec_\Lambda$have proposed values for instances preceding i, so τ_{seq} meets the requirements of barrier functions. □

5.2 Impossibility

One might wonder if this limitation of sequential instances is inherent or not. Indeed, this is the case as we show in the following.

Theorem 2. *Let Π be a set of two or more processes executing the τ-based POabcast algorithm with an underlying consensus implementation C that can only be accessed*

through its propose and decide calls. There is no local implementation of τ for C allowing a primary p to propose a value for instance i before p reaches a decision for instance $i-1$.

Proof. The proof is by contradiction: we assume that a barrier function τ_c allowing primaries to propose values for multiple concurrent consensus instances exists.

Run σ_1: The oracle Ω outputs some process p as the only leader in the system from the beginning of the run. Assume that p broadcasts two values v_1 and v_2 at the beginning of the run. For liveness of *POabcast*, p must eventually propose values for consensus instances 1 and 2. By assumption, τ_c allows p to start consensus instance 2 before a decision for instance 1 is reached. Therefore p observes $\tau_c = 0$ when it proposes v_1 and v_2. The output of τ_c must be independent from the internal events of the underlying consensus implementation C, since τ_c cannot observe them. We can therefore assume that no process receives any message before p proposes v_2.

Run σ_1': The prefix of σ_1 that finishes immediately after p proposes v_2. No process receives any message.

Run σ_2: Similar to σ_1, but the only leader is $p' \neq p$ and the proposed values are v_1' and v_2'. Process p' observes $\tau_c = 0$ when it proposes v_1' and v_2'.

Run σ_2': The prefix of σ_2 that finishes immediately after p' proposes v_2'. No process receives any message.

Run σ_3: The beginning of this run is the union of all events in the runs σ_1' and σ_2'. No process receives any message until the end of the union of σ_1' and σ_2'. The Ω oracle is allowed to elect two distinct leaders for a finite time. Process p (resp. p') cannot distinguish between run σ_1' (resp. σ_2') and the corresponding local prefix of σ_3 based on the outputs of the consensus primitive and of the leader oracle. After the events of σ_1' and σ_2' have occurred, some process decides v_1' for consensus instance 1 and v_2 for consensus instance 2.

Regardless of the definition of Λ and $\prec_\Lambda$, the output of τ_c in σ_3 is incorrect. Let p and p' have primary identifiers λ and λ' when they proposed v_2 and v_1', respectively. If $\lambda \prec_\Lambda \lambda'$, τ_c should have returned 2 instead of 0 when p' became primary. If $\lambda' \prec_\Lambda \lambda$, τ_c should have returned 1 instead of 0 when p became primary. □

5.3 Barriers with White-Box Paxos

An alternative, corresponding to Zab [9], to avoid the aforementioned impossibility is to consider the internal states of the underlying consensus algorithm. We exemplify this approach considering the popular Paxos algorithm [11]. A detailed discussion of Paxos is out of the scope of this work and we only present a summary for completeness.

Overview of Paxos. In Paxos, each process keeps, for every consensus instance, an *accepted value*, which is the most current value it is aware of that might have been decided. A process p elected leader must first read, for each instance, the value that may have been decided upon for this instance, if any. To obtain this value, the leader selects a unique *ballot number* b and executes a *read phase* by sending a read message to all other processes. Processes that have not yet received messages from a leader with a higher ballot number b reply by sending their current accepted value for the instance.

Each accepted value is sent attached to the ballot number of the previous leader that proposed that value. The other processes also promise not to accept any message from leaders with ballots lower than b. When p receives accepted values from a majority of processes, it *picks* for each instance the accepted value with the highest attached ballot. Gaps in the sequence instance with picked values are filled with empty *no op* values.

After completing the read phase, the new leader proposes the values it picked as well as its own values for the instances for which no value was decided. The leader proposes values in a *write* phase: it sends them to all processes together with the current ballot number b. Processes accept proposed values only if they have not already received messages from a leader with a ballot number $b' > b$. After they accept a proposed value, they send an acknowledgement to the leader proposing it. When a value has been written with the same ballot at a majority of processes, it is decided.

In a nutshell, the correctness argument of Paxos boils down to the following argument. If a value v has been decided, a majority of processes have accepted it with a given ballot number b; we say that the proposal $\langle v, b\rangle$ is *chosen*. If the proposal is chosen, no process in the majority will accept a value from a leader with a ballot number lower than b. At the same time, every leader with a ballot number higher than b will read the chosen proposal in the read phase, and will also propose the v.

Integrating the Barrier Function. We modify Paxos to incorporate the barrier function. If a process is not a leader, there is no reason for evaluating τ. Whenever a process is elected leader, it executes the read phase. Given a process p such that $\Omega = p$, let $read(p)$ be the maximum consensus instance for which any value is picked in the last read phase executed by p. The barrier function is implemented as follows:

$$\tau_{\text{Paxos}} = \begin{cases} \top & \text{iff } \Omega \neq p \ \vee p \text{ is in read phase} \\ read(p) & \text{iff } \Omega = p \ \wedge p \text{ is in write phase} \end{cases}$$

The output value $\top$ is such that $dec \geq \tau_{Paxos}$ never holds for any value of dec. This prevents leaders from becoming primaries until a correct output for τ_{Paxos} is determined.

We now show that this τ implementation is correct. The proof relies on the correctness argument of Paxos.

Theorem 3. *The function τ_{Paxos} is a barrier function.*

Proof. By the definition of τ_{Paxos}, a process becomes a primary if it is a leader and has completed the read phase. Let Λ associate a primary with the unique ballot number it uses in the Paxos read phase and let $\prec_\Lambda$ be the ballot number order.

Paxos guarantees that if any process ever decides a value v proposed by a leader with ballot number smaller than the one of λ, then v is picked by λ in the read phase [11]. This is sufficient to meet the requirements of τ. □

5.4 Time Complexity of τ-Based *POabcast* with Different Barrier Functions

We now explain the second and third row of Table 1. Just for the analysis, we assume that there are c clients in the system, the communication delay is Δ, and Paxos is used as underlying consensus protocol since it is optimal [12].

We first consider the barrier function of Sect. 5.1. If a primary receives requests from all clients at the same time, it will broadcast and deliver the corresponding state updates sequentially. Delivering a message requires 2Δ, the latency of the write phase of Paxos. Since each message will take 2Δ time to be delivered, the last message will be delivered in $2\Delta \cdot c$ time. During leader change, Paxos takes 2Δ time to execute the read phase and 2Δ to execute the write phase if a proposal by the old primary has been chosen and potentially decided in the last consensus instance.

With the barrier function of Sect. 5.3, consensus instances are executed in parallel with a latency of 2Δ. The complexity for leader changes is the same, since the write phase is executed in parallel for all instances up to τ.

Note that the longer leader change time of *POabcast* algorithms compared to atomic broadcast (see Table 1) is due to the barrier: before it becomes a primary, a process must *decide* on all values that have been proposed by the previous primaries and potentially decided (chosen). This is equivalent to executing read and write phases that require 4Δ time. In atomic broadcast, it is sufficient that a new leader *proposes* chosen values from previous leaders.

5.5 Relationship between τ Functions and Existing *POabcast* Algorithms

The *POabcast* algorithm with the barrier function of Sect. 5.1 is similar to semi-passive replication [7] since both enforce the same constraint: primaries only keep one outstanding consensus instance at a time. The time complexity of the two protocols using Paxos as the underlying consensus protocol is the same (Table 1, second row).

If the barrier function implementation selects a specific consensus protocol and assumes that it can access its internal state, as discussed in Sect. 5.1, our barrier-based *POabcast* algorithm can broadcast state updates in the presence of multiple outstanding consensus instances. This is the same approach as Zab, and indeed there are many parallelisms with this algorithm. The time complexity in stable periods is the same (see Table 1, third row). A closer look shows that also the leader change complexity is equal, apart from specific optimizations of the Zab protocol. In Zab, the read phase of Paxos corresponds to the *discovery phase*; the CEPOCH message is used to implement leader election and to speed up the selection of a unique ballot (or epoch, in Zab terms) number that is higher than any previous epoch numbers [9]. After the read phase is completed, the leader decides on all consensus instances until the instance identifier returned by τ_{Paxos} - this is the *synchronization phase*, which corresponds to a write phase in Paxos; in our implementation, the barrier function returns and the leader waits until enough consensus instances are decided. At this point, the necessary condition $dec \geq \tau_{Paxos}$ of our generic *POabcast* construction is fulfilled, so the leader crosses the barrier, becomes a primary, and can proceed with proposing values for new instances. In Zab, this corresponds to the *broadcast phase*.

Virtually-synchronous Paxos is also a modified version of Paxos that implements *POabcast* and the τ_{Paxos} barrier function, but it has the additional property of making the set of participating processes dynamic [2]. It has the same time complexity during stable periods and leader changes as in Table 1.

6 *POabcast* Using Consensus Instead of τ for the Barrier

The previous section shows an inherent tradeoff in τ implementations between modularity, which can be achieved by using sequential consensus instances and using consensus as a black box, and performance, which can be increased by integrating the implementation of the barrier function in a specific consensus protocol. In this section, we show that this tradeoff can be avoided through the use of an alternative *POabcast* algorithm.

Algorithm. Our τ-free algorithm (see Algorithm 2) implements *POabcast*, so it is an alternative to Algorithm 1. The algorithm is built upon a leader election oracle Ω and consensus. The main difference with Algorithm 1 is that the barrier predicate *isPrimary* is implemented using consensus instead of τ: consensus instances are used to agree not only on values, but also on primary election information. Another difference is that some decided value may not be delivered. This requires the use of additional buffering, which slightly increases the complexity of the implementation.

When a process p becomes leader, it picks a unique epoch number *tent-epoch* and proposes a $\langle$NEW-EPOCH, *tent-epoch*$\rangle$ value in the smallest consensus instance *dec* where p has not yet reached a decision (lines 5-9). Like in Algorithm 1, we use multiple consensus instances. All replicas keep a decision counter *dec*, which indicates the current instance where a consensus decision is awaited, and a proposal counter *prop*, which indicates the next available instance for proposing a value. Another similarity with Algorithm 1 is that decision events are processed following the order of consensus instances, tracked using the variable *dec* (see lines 10 and 29). Out-of-order decision events are buffered, although this is omitted in the pseudocode.

Every time a NEW-EPOCH tuple is decided, the sender of the message is elected primary and its epoch *tent-epoch* is *established* (lines 10-23). When a new epoch is established, processes set their current epoch counter *epoch* to *tent-epoch*. If the process delivering the NEW-EPOCH tuple is a leader, it checks whether the epoch that has been just established is its own tentative epoch. If this is the case, the process considers itself as a primary and sets *primary* to true; else, it tries to become a primary again.

When p becomes a primary, it can start to broadcast values by proposing VAL tuples in the next consensus instances, in parallel (lines 24-28). Ensuring that followers are in a state consistent with the new primary does not require using barriers: all processes establishing *tent-epoch* in consensus instance i have decided and delivered the same sequence of values in the instances preceding i. This guarantees that the primary integrity property of *POabcast* is respected.

Processes only POdeliver VAL tuples of the last established epoch until a different epoch is established (lines 29-33, see in particular condition $epoch_m = epoch$). The algorithm establishes the following total order $\prec_\Lambda$ of primary identifiers: given two different primaries λ and λ' which picked epoch numbers e and e' respectively, we say that $\lambda \prec_\Lambda \lambda'$ if and only if a tuple $\langle$NEW-EPOCH, $e\rangle$ is decided for a consensus instance n, a tuple $\langle$NEW-EPOCH, $e'\rangle$ is decided for a consensus instance m, and $n < m$. Suppose that p is the primary λ with epoch number e_λ elected in consensus instance dec_λ. All processes set their current epoch variable e to e_λ after deciding in instance dec_λ. From consensus instance number $dec_\lambda + 1$ to the next consensus instance where a NEW-EPOCH tuple is decided, processes decide and deliver only values that are sent from λ and included in VAL tuples with $epoch_m = e_\lambda$. Replicas thus deliver messages

```
1  initially
2      tent-epoch, dec, decseq, prop,
       seqno ← 0;
3      epoch ← ⊥;
4      primary ← false;
5  upon Ω changes from q ≠ p to p
6      try-primary();
7  procedure try-primary()
8      tent-epoch ← new unique epoch
       number;
9      propose(⟨NEW-EPOCH,
       tent-epoch⟩, dec);
10 upon decide(⟨NEW-EPOCH,
   tent-epoch_m⟩, dec)
11     dec ← dec+1;
12     epoch ← tent-epoch_m;
13     da ← empty array;
14     pa ← empty array;
15     decseq ← dec;
16     if Ω = p then
17         if tent-epoch = tent-epoch_m
           then
18             prop ← dec;
19             seqno ← dec;
20             primary ← true;
21         else
22             primary ← false;
23             try-primary();
24 upon POabcast( v)
25     propose(⟨VAL, v, epoch,
       seqno⟩, prop);
26     pa[prop] ← ⟨ v, seqno⟩;
27     prop← prop+1;
28     seqno ← seqno+1;
29 upon decide(⟨VAL, v, epoch_m,
   seqno_m⟩, dec)
30     if epoch_m = epoch then
31         da[seqno_m] ← v;
32         while da[decseq] ≠ ⊥ do
33             POdeliver(da[decseq]);
34             decseq← decseq+1;
35     if primary ∧ epoch_m≠ epoch
36     ∧ prop ≥ dec then
37         ⟨ v′, seqno′⟩ ← pa[dec];
38         pa[prop] ← pa[dec];
39         propose(⟨VAL, v′, epoch,
           seqno′⟩, prop);
40         prop← prop+1;
41     if ¬ primary ∧ Ω = p then
42         try-primary();
43     dec← dec+1;
44 upon Ω changes from p to q ≠ p
45     primary← false;
46 function isPrimary()
47     return primary;
```

Algorithm 2: Barrier-free *POabcast* using black-box consensus - process p

following the order $\prec_\Lambda$ of the primaries that sent them, fulfilling the global primary order property of *POabcast*.

The additional complexity in handling VAL tuples is necessary to guarantee the local primary order property of *POabcast*. VAL tuples of an epoch are not necessarily decided in the same order as they are proposed. This is why primaries include a sequence number *seqno* in VAL tuples. In some consensus instance, the tuples proposed by the current primary might not be the ones decided. This can happen in the presence of concurrent primaries, since primaries send proposals for multiple overlapping consensus instances without waiting for decisions. If a primary is demoted, values from old and new primaries could be interleaved in the sequence of decided values for a finite number of instances. All processes agree on the current epoch of every instance, so they do not deliver messages from other primaries with different epoch numbers. However, it is necessary to buffer out-of-order values from the current primary to deliver them later. That is why processes store decided values from the current primary in the *da*

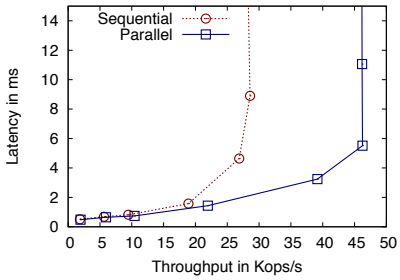

Fig. 1. Latency and throughput with micro benchmarks. Request and state update sizes were set to 1kb, which is the typical size observed in ZooKeeper. Both protocols use batching.

array (line 31), and deliver them only if a continuous sequence of sequence numbers, tracked by *decseq*, can be delivered (lines 32-34).

Primaries also need to resend VAL tuples that could not be decided in the correct order. When values are proposed, they are stored in the *pa* following the sequence number order; this buffer is reset to the next ongoing consensus instance every time a new primary is elected. Primaries resend VAL tuples in lines 35-40. Primaries keep a proposal instance counter *prop*, indicating the next consensus instance where values can be proposed. If an established primary has outstanding proposals for the currently decided instance *dec*, it holds $prop \geq dec$. In this case, if the decided VAL tuple is not one such outstanding proposal but has instead been sent by a previous primary, it holds that $epoch_m \neq epoch$. If all the previous conditions hold, the established primary must resend the value that has been skipped, $pa[dec].v'$, using the same original sequence number $pa[dec].seqno$' in the next available consensus instance, which is *prop*.

The arrays *da* and *pa* do not need to grow indefinitely. Elements of *da* (resp. *pa*) with position smaller than *decseq* (resp. *dec*) can be garbage-collected.

For liveness, a leader which is not a primary keeps trying to become a primary by sending a NEW-EPOCH tuple for every consensus instance (lines 22-23). The *primary* variable is true if a leader is an established primary. It stops being true if the primary is not a leader any longer (lines 44-45).

Algorithm 2 correctly implements *POabcast*, as shown in our full paper [10].

Time Complexity. As before, we use Paxos for the consensus algorithm and assume a communication delay of Δ. During stable periods, the time to deliver a value is 2Δ, which is the time needed to execute a Paxos write phase. When a new leader is elected, it first executes the read phase, which takes 2Δ. Next, it executes the write phase for all instances in which values have been read but not yet decided, and for one additional instance for its NEW-EPOCH tuple. All these instances are executed in parallel, so they finish within 2Δ time. After this time, the new leader crosses the barrier, becomes a primary, and starts broadcasting new values.

7 Experimental Evaluation

Our τ-free algorithm combines modularity with constant time complexity. Since our work was motivated by our work on systems like ZooKeeper, one might wonder whether this improvement has a practical impact. Current implementations of replicated systems can reach some degree of parallelism even if they execute consensus instances sequentially. This is achieved through an optimization called *batching*: multiple clients requests are aggregated in a batch and agreed upon together using a single instance. Even in presence of batching, we found that there is a substantial advantage of running multiple consensus instances in parallel.

We implemented two variants of the Paxos algorithm, one with sequential consensus instances and one with parallel ones, and measured the performance of running our *POabcast* algorithms on top of it. We consider fault-free runs where the leader election oracle outputs the same leader to all processes from the beginning. We used three replicas and additional dedicated machines for the clients; all servers are quad-core 2.5 GHz CPU servers with 16 GB of RAM connected through a Gigabit network.

The experiments consist of micro-benchmarks where the replicated object does nothing. These benchmarks are commonly used in the evaluation of replication algorithms because they reproduce a scenario in which the replication protocol, rather than execution, is the bottleneck of the system so its performance is critical.

We used batching in all our experiments. With sequential consensus instances, we batch all requests received while a previous instance is ongoing. In the pipelined version, we start a new consensus instance when either the previous instance is completed or b requests have been batched. We found $b = 50$ to be optimal. Every measurement was repeated five times at steady state, and variances were negligible.

Figure 1 reports the performance of the two variants with a growing number of clients. Messages (requests and state updates) have size 1 kB, which is a common state update size for ZooKeeper and Zab [9].

The peak throughput with the parallel consensus instances is almost two times the one with sequential instances. The same holds with messages of size 4 kB. The difference decreases with smaller updates than the ones we observe in practical systems like ZooKeeper. In the extreme case of empty requests and state updates, the two approaches have virtually the same request latency and throughput: they both achieve a maximum throughput of more than 110 kops/sec and a minimum latency of less than 0.5 ms.

These results show that low time complexity (see Table 1) is very important for high-performance passive replication. When there is little load in the system, the difference in latency between the two variants is negligible. In fact, due to the use of batching, running parallel consensus instances is not needed. As the number of clients (c in Table 1) increases, latency grows faster in the sequential case, as predicted by our analysis. With sequential consensus instances, a larger latency also results in significantly worse throughput compared to the parallel variant due to lower network and CPU utilization.

8 Conclusions

Some popular systems such as ZooKeeper have used passive replication to mask crash faults. We extracted a unified algorithm for implementing *POabcast* using the barrier

function that abstracts existing passive replication approaches. The barrier function is a simple way to understand the difference between passive and active replication, as well as the characteristics of existing *POabcast* algorithms, but it imposes a tradeoff between parallelism and modularity. We have proposed an algorithm that avoids such a limitation by not relying upon a barrier function. This algorithm is different from existing ones in its use of consensus, instead of barrier functions, for primary election.

Acknowledgement. We would like to express our gratitude to Alex Shraer and Benjamin Reed for the insightful feedback on previous versions of the paper, and to Daniel Gómez Ferro for helping out with the experiments.

References

1. Baker, J., Bond, C., Corbett, J., Furman, J.J., Khorlin, A., Larson, J., Léon, J.-M., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing scalable, highly available storage for interactive services. In: CIDR, vol. 11, pp. 223–234 (2011)
2. Birman, K., Malkhi, D., Van Renesse, R.: Virtually synchronous methodology for dynamic service replication. Technical Report MSR-TR-2010-151, Microsoft Research (2010)
3. Budhiraja, N., Marzullo, K., Schneider, F.B., Toueg, S.: The primary-backup approach, pp. 199–216. ACM Press/Addison-Wesley (1993)
4. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
5. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM 43(4), 685–722 (1996)
6. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Compututing Surveys 33(4), 427–469 (2001)
7. Défago, X., Schiper, A.: Semi-passive replication and lazy consensus. Journal of Parallel and Distributed Computing 64(12), 1380–1398 (2004)
8. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: Wait-free coordination for Internet-scale systems. In: USENIX Annual Technical Conference, pp. 145–158 (2010)
9. Junqueira, F.P., Reed, B., Serafini, M.: Zab: High-performance broadcast for primary-backup systems. In: IEEE Conference on Dependable Systems and Networks, pp. 245–256 (2011)
10. Junqueira, F.P., Serafini, M.: On barriers and the gap between active and passive replication (full version). arXiv:1308.2979 [cs.DC] (2013)
11. Lamport, L.: The part-time parliament. ACM Transactions on Computing Systems (TOCS) 16(2), 133–169 (1998)
12. lamport, l.: lower bounds for asynchronous consensus. distributed computing 19(2), 79–103 (2006)
13. Lamport, L., Malkhi, D., Zhou, L.: Vertical paxos and primary-backup replication. In: ACM Symposium on Principles of Distributed Computing, pp. 312–313 (2009)
14. Pedone, F., Frolund, S.: Pronto: A fast failover protocol for off-the-shelf commercial databases. In: IEEE Symposium on Reliable Distributed Systems, pp. 176–185 (2000)
15. Shraer, A., Reed, B., Malkhi, D., Junqueira, F.: Dynamic reconfiguration of primary/backup clusters. In: USENIX Annual Technical Conference, pp. 425–438 (2012)