Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab

Robbert van Renesse, Nicolas Schiper, Fred B. Schneider
Department of Computer Science, Cornell University

***Abstract*—Paxos, Viewstamped Replication, and Zab are replication protocols that ensure high-availability in asynchronous environments with crash failures. Various claims have been made about similarities and differences between these protocols. But how does one determine whether two protocols are the same, and if not, how significant the differences are?**

We propose to address these questions using refinement mappings, where protocols are expressed as succinct specifications that are progressively refined to an implementable protocol. Doing so enables a principled understanding of their correctness, and it provides clear guidelines to implement the protocols correctly. Additionally, comparing Paxos, Viewstamped Replication, and Zab using this approach allowed us to identify key differences that have a significant impact on performance.

***Index Terms*—C.0.f Systems specification methodology, C.2.4 Distributed Systems, D.4.5 Reliability**

I. Introduction

A protocol expressed in terms of a state transition specification Σ refines another specification Σ' if there exists a mapping of the state space of Σ to the state space of Σ' and each state transition in Σ can be mapped to a state transition in Σ' or to a no-op. This mapping between specifications is called *refinement* [1] or *backward simulation* [2], [3].[1] Two protocols are the same if they refine one another. But if they don't, how does one qualify the similarities and differences between two different protocols?

We became interested in this question while comparing three replication protocols for high availability in asynchronous environments with crash failures:

- *Paxos* [4] is a state machine replication protocol [5], [6]. We consider a version of Paxos that uses the multi-decree Synod consensus algorithm described in [4], sometimes called Multi-Paxos. Many implementations have been deployed, including in Google's Chubby service [7], [8], in Microsoft's Autopilot service [9] (used by Bing), and in the popular Ceph distributed file system [10], with interfaces now part of the standard Linux kernel.
- *Viewstamped Replication* (VSR) [11], [12] is a replication protocol originally targeted at replicating participants in a Two-Phase Commit (2PC) [13] protocol. VSR has also been used in the implementation of the Harp File System [14];
- Zab [15], [16] (ZooKeeper Atomic Broadcast) is a replication protocol used for the popular *ZooKeeper* [17] configuration service. ZooKeeper has been in active use at Yahoo! and is now a popular open source product distributed by Apache.

Many claims have been made about the similarities and differences of these protocols. For example, citations [18], [19], [20] claim that Paxos and Viewstamped Replication are "the same algorithm independently invented," "equivalent," or that "the view management protocols seem to be equivalent" (annotations in [4]).

In this paper, we approach the question of similarities and differences between these protocols using refinements, as illustrated in Fig. 1. Refinement mappings induce an ordering relation on specifications, and the figure shows a Hasse diagram of a set of eight specifications of interest ordered by refinement. In this figure, we write $\Sigma' \rightarrow \Sigma$ if Σ refines Σ', that is, if there exists a refinement mapping of Σ to Σ'.

At the same time, we have indicated informal levels of abstraction in this figure, ranging from a highly abstract specification of a linearizable service [21] to concrete, executable specifications. Active and passive replication are common approaches to replicate a service and ensure that behaviors are still linearizable. BEV-Replication protocols use a form of rounds in order to refine active and passive replication. Finally, we obtain protocols such as Paxos, VSR, and Zab.

Each refinement corresponds to a *design decision*, and as can be seen from the figure it is possible to arrive at the same specification following different paths of such design decisions. There is a qualitative difference between refinements that cross abstraction boundaries and those that do not. When crossing an abstraction

[1]State transition specifications may include supplementary liveness conditions.

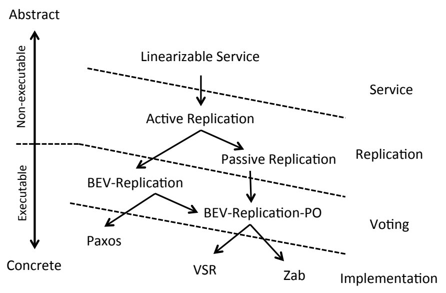


Fig. 1: An ordering of refinement mappings and informal levels of abstraction.

boundary, a refinement takes an abstract concept and replaces it with a more concrete one. For example, it may take an abstract *decision* and replace it by a *majority of votes*. When staying within the same abstraction, a refinement *restricts* behaviors. For example, one specification might decide commands out of order while a more restricted specification might decide them in order.

Using these refinements, we can identify and analyze commonalities and differences between the replication protocols. They also enable consideration of new variants of these protocols and what conditions they must satisfy to be correct refinements of a linearizable service. Other papers have analyzed the correctness of distributed protocols using stepwise refinement [22], [23]. To the best of our knowledge, this paper is the first to employ refinement for comparing replication protocols.

This paper is organized as follows: Section II introduces state transition specifications of linearizable replication as well as active and passive replication. Section III presents BEV-Replication, a canonical protocol that generalizes Paxos, VSR, and Zab and forms a basis for comparison. *Progress indicators*, a new class of invariants, enable constructive reasoning about why and how these protocols work. We show how passive replication protocols refine BEV-Replication by adding prefix-ordering with BEV-Replication-PO. Section IV presents implementation details of Paxos, VSR, and Zab that constitute the final refinement steps to executable protocols. Section V discusses the implications of identified differences on performance. Section VI gives a short overview of the history of concepts used in these replication protocols, and Section VII is a conclusion. The details of the refinement mappings in Fig. 1 appear in a technical report [24].

II. Masking Failures

To improve the availability of a service, a common technique is to replicate it onto a set of servers. A *consistency criterion* defines expected responses to clients for concurrent operations. Ideally, the replication protocol

Specification 1 Linearizable Service

```
var $inputs_\nu, outputs_\nu, appState, invoked_{clt}, received_{clt}$

initially: $appState = \bot \wedge inputs_\nu = outputs_\nu = \varnothing \wedge$
    $\forall clt : invoked_{clt} = received_{clt} = \varnothing$

interface transition invoke($clt, op$):
  precondition:
    $op \notin invoked_{clt}$
  action:
    $invoked_{clt} := invoked_{clt} \cup \{op\}$
    $inputs_\nu := inputs_\nu \cup \{(clt, op)\}$

internal transition execute($clt, op, result, newState$):
  precondition:
    $(clt, op) \in inputs_\nu \wedge$
    $(result, newState) = nextState(appState, (clt, op))$
  action:
    $appState := newState$
    $outputs_\nu := outputs_\nu \cup \{((clt, op), result)\}$

interface transition response($clt, op, result$):
  precondition:
    $((clt, op), result) \in outputs_\nu \wedge op \notin received_{clt}$
  action:
    $received_{clt} := received_{clt} \cup \{op\}$
```

ensures *linearizability* [21]—the execution of concurrent client operations is equivalent to a sequential execution, where each operation is atomically performed at some point in time between its invocation and response.

A. Specification

We characterize linearizability by giving a state transition specification (see Specification 1). A specification defines states and gives legal transitions between states. A state is defined by a collection of variables and their current values. Transitions can involve parameters (listed in parentheses) that are bound within the defined scope. A transition definition gives a precondition and an action. If the precondition holds in a given state, then the transition is *enabled* in that state. The action relates the state after the transition to the state before. A transition is performed indivisibly starting in a state satisfying the precondition. No two transitions are performed concurrently, and if multiple transitions are enabled simultaneously, then the choice of which transition to perform is unspecified.

There are *interface variables* and *internal variables*. Interface variables are subscripted with the location of the variable, which is either a process name or the network, ν. Internal variables have no subscripts, and their value will be determined by a function on the state of the underlying implementation. Specification Linearizable Service has the following variables:

- $inputs_\nu$: a set that contains (clt, op) messages sent by process *clt*. Here *op* is an operation invoked by *clt*;
- $outputs_\nu$: a set of $(clt, op, result)$ messages sent by the service, containing the results of client operations that have been executed;

- $appState$: an internal variable containing the state of the application;
- $invoked_{clt}$: the set of operations invoked by process clt. This is a variable maintained by clt itself;
- $received_{clt}$: the set of completed operations, also maintained by clt.

Similarly, there are *interface transitions* and *internal transitions*. Interface transitions model interactions with the environment, which consists of a collection of processes connected by a network. An interface transition is performed by the process that is identified by the first parameter to the transition. Interface transitions are not allowed to access internal variables. Internal transitions are performed by the service, and we will have to demonstrate how this is done by implementing those transitions. The transitions of Specification Linearizable Service are:

- Interface transition `invoke`(clt, op) is performed when clt invokes operation op. Each operation is uniquely identified and can be invoked at most once by a client (enforced by the precondition). Adding op to $inputs_\nu$ models clt sending a message containing op to the service. The client maintains what operations it has invoked in $invoked_{clt}$;
- Transition `execute`$(clt, op, result, newState)$ is an internal transition that is performed when the replicated service executes op for client clt. The application-dependent deterministic function $nextState$ relates an application state and an operation from a client to a new application state and a result. Adding $(clt, op, result)$ to $outputs_\nu$ models the service sending a response to clt.
- Interface transition `response`$(clt, op, result)$ is performed when clt receives the response. The client keeps track of which operations have been completed in $received_{clt}$ to prevent this transition being performed more than once per operation.

From Specification Linearizable Service it is clear that it is not possible for a client to receive a response to an operation before it has been invoked and executed. However, the specification does allow each client operation to be executed an unbounded number of times. In an implementation, multiple execution could happen if the response to the client operation got lost by the network and the client retransmits its operation to the service. The client will only learn about at most one of these executions. In practice, replicated services will try to reduce or eliminate the probability of a client operation being executed more than once by keeping state about which operations have been executed. For example, a service could keep track of all its clients and eliminate duplicate operations using sequence numbers on client operations.

We make the following assumptions about interface transitions:

- **Crash Failures**: A process follows its specification until it fails by crashing. Thereafter, it executes no transitions. Processes that never crash are called *correct*. A process that crashes and later recovers using state from stable storage is considered correct albeit, temporarily, slow. Processes are assumed to fail independently.
- **Failure Threshold**: There is a bound f on the maximum number of processes that may crash.
- **Fairness**: Except for transitions at a crashed process, a transition that becomes continuously enabled is eventually executed.
- **Asynchrony**: There is no bound on the time before a continuously enabled transition is executed.

B. Active and Passive Replication

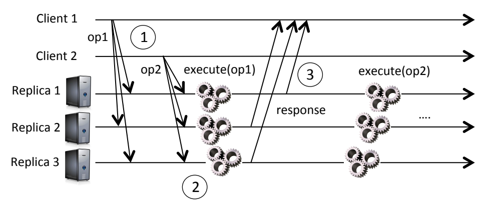
Specification 1 has internal variables and transitions that have to be implemented. There are two well-known approaches to replication:

- With *active replication*, also known as *state machine replication* [5], [6], each replica implements a deterministic state machine. All replicas process the same operations in the same order.
- With *passive replication*, also known as *primary backup* [25], [26], a primary replica runs a deterministic state machine, while backups only store states. The primary computes a sequence of new application states by processing operations and forwards these states to each backup in order of generation.

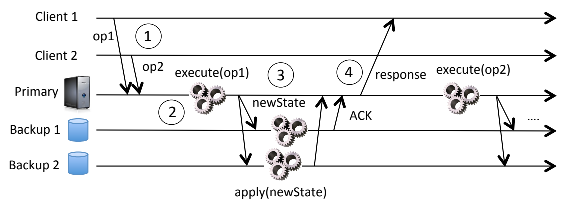
Fig. 2a illustrates a failure-free execution of active replication:

1) Clients submit operations to the service (op1 and op2 in Fig. 2a).
2) Replicas, starting out in the same state, apply received client operations in the same order.
3) Replicas send responses to the clients. Clients ignore all but the first response they receive.

The tricky part of active replication is ensuring that replicas execute operations in the same order, despite replica failures, message loss, and unpredictable delivery and processing delays. A fault-tolerant *consensus* protocol [27] is typically employed to agree on the i^{th} operation. Specifically, each replica proposes an operation that was received from one of the clients in instance i of the consensus protocol. Only one of the proposed operations can be decided. The service remains available as long as each instance of consensus eventually terminates.



(a) Active replication



(b) Passive replication

Fig. 2: A failure-free execution of active and passive replication.

Fig. 2b depicts a failure-free execution of passive replication:

1) Clients submit operations only to the primary.
2) The primary orders the operations and computes new states and responses.
3) The primary forwards the new states (so-called *state updates*) to each backup in the order generated.
4) The primary sends the response of an operation only after sufficiently many backups have acknowledged receipt of the corresponding state update (this is made precise in Section III).

Since the primary is replaced after a crash, it is important that the replicas receive and apply state updates in the order sent by a particular primary. That is, if a primary sends updates u_1 and u_2 in that order, then a replica is only allowed to execute u_2 after executing u_1. This is sometimes called the *prefix ordering* property.

For example, consider a replicated integer variable with initial value 3. One client wants to increment the variable, while the other wants to double it. One primary receives both operations and submits state updates 4 followed by 8. Another primary receives the operations in the opposite order and submits updates 6 followed by 7. Without prefix ordering, it may happen that the decided states are 4 followed by 7, not responding to any legal history of the two operations.

VSR and Zab employ passive replication; Paxos employs active replication. However, it is possible to implement one style on the other. The Harp file system [14], for example, uses VSR to implement a replicated message queue containing client operations—the Harp primary proposes state updates that backups apply to the state of the message queue. Replicas, running deterministic NFS state machines, then execute NFS operations in queue order. In other words, Harp uses an active replication protocol built using a message queue that is passively replicated using VSR.

C. Refinement

Below, we present a refinement of a linearizable service (Specification 1) using the active replication approach. We then further refine active replication to obtain passive replication. The refinement of a linearizable service to passive replication follows transitively.

1) Active Replication: We omit interface transitions $\texttt{invoke}(\mathit{clt}, \mathit{op})$ and $\texttt{response}(\mathit{clt}, \mathit{op}, \mathit{result})$, which are the same as in Specification 1. Hereafter, a command cmd denotes a tuple $(\mathit{clt}, \mathit{op})$.

Specification 2 uses a sequence of *slots*. A replica executes transition $\texttt{propose}(\mathit{replica}, \mathit{slot}, \mathit{cmd})$ to propose a command cmd for the given slot. We call the command a *proposal*. Transition $\texttt{decide}(\mathit{slot}, \mathit{cmd})$ guarantees that at most one proposal is decided for each slot. Transition $\texttt{learn}(\mathit{replica}, \mathit{slot})$ models a replica learning a decision and assigning the decision to the corresponding slot of the $\mathit{learned}_{\mathit{replica}}$ array. Replicas update their state by executing a learned operation in increasing order of slot number with transition $\texttt{update}(\mathit{replica}, \mathit{cmd}, \mathit{res}, \mathit{newState})$. The slot of the next operation to execute is denoted by $\mathit{version}_{\mathit{replica}}$.

2) Passive Replication: Passive replication (Specification 3) also uses slots, and proposals are tuples $(\pi, (\mathit{cmd}, \mathit{res}, \mathit{newState}))$ consisting of a lambda expression π, a command, the output of executing the command, and a new state that results from applying the command to the state of the proposing replica.

Any replica can act as primary and propose to apply operations. Primaries act *speculatively*, computing a sequence of states before they are decided. Because of this, primaries have to maintain a separate version of the application state, which we call the *shadow state*. Different primaries may propose to apply different state updates for the same slot.

A primary proposes to apply cmd to *oldState* in a certain slot, resulting in output *res* using transition $\texttt{propose}(\mathit{replica}, \mathit{slot}, \mathit{cmd}, \mathit{res}, \mathit{newState}, \mathit{oldState})$. State *oldState* is what the primary calculated for the previous slot (even though that state is not necessarily decided as of yet, and may never be decided). The lambda expression provides a way to check later that the new state is applied on the proper old state.

Transition $\texttt{decide}(\mathit{slot}, \mathit{cmd}, \mathit{res}, \mathit{newState})$ specifies that only one of the proposed state updates can be

Specification 2 Specification Active Replication

```
var  $proposals_{replica}[1...]$, $decisions[1...]$, $learned_{replica}[1...]$
     $appState_{replica}$, $version_{replica}$, $inputs_{\nu}$, $outputs_{\nu}$

initially:
  $\forall s \in \mathbb{N}^+ : decisions[s] = \bot \wedge$
  $\forall replica :$
    $appState_{replica} = \bot \wedge version_{replica} = 1 \wedge$
    $\forall s \in \mathbb{N}^+ :$
      $proposals_{replica}[s] = \varnothing \wedge learned_{replica}[s] = \bot$

interface transition propose($replica$, $slot$, $cmd$):
  precondition:
    $cmd \in inputs_{\nu}$
  action:
    $proposals_{replica}[slot] := proposals_{replica} \cup \{cmd\}$

internal transition decide($slot$, $cmd$):
  precondition:
    $decisions[slot] = \bot \ \wedge \ \exists r : cmd \in proposals_r[slot]$
  action:
    $decisions[slot] := cmd$

internal transition learn($replica$, $slot$):
  precondition:
    $learned_{replica}[slot] = \bot \wedge decisions[slot] \neq \bot$
  action:
    $learned_{replica}[slot] := decisions[slot]$

interface transition update($replica$, $cmd$, $res$, $newState$):
  precondition:
    $cmd = learned_{replica}[version_{replica}] \wedge cmd \neq \bot \wedge$
    $(res, newState) = nextState(appState_{replica}, cmd)$
  action:
    $outputs_{\nu} := outputs_{\nu} \cup \{(cmd, res)\}$
    $appState_{replica} := newState$
    $version_{replica} := version_{replica} + 1$
```

Specification 3 Specification Passive Replication

```
var  $proposals_{replica}[1...]$, $decisions[1...]$, $learned_{replica}[1...]$
     $appState_{replica}$, $version_{replica}$, $inputs_{\nu}$, $outputs_{\nu}$,
     $shadowState_{replica}$, $shadowVersion_{replica}$

initially:
  $\forall s \in \mathbb{N}^+ : decisions[s] = \bot$
  $\forall replica :$
    $appState_{replica} = \bot \wedge version_{replica} = 1 \wedge$
    $shadowState_{replica} = \bot \wedge shadowVersion_{replica} = 1 \wedge$
    $\forall s \in \mathbb{N}^+ :$
      $proposals_{replica}[s] = \varnothing \wedge learned_{replica}[s] = \bot$

interface transition propose($replica$, $slot$, $cmd$, $res$,
                                             $newState$, $oldState$):
  precondition:
    $cmd \in inputs_{\nu} \wedge slot = shadowVersion_{replica} \wedge$
    $oldState = shadowState_{replica} \wedge$
    $(res, newState) = nextState(oldState, cmd)$
  action:
    $proposals_{replica}[slot] := proposals_{replica}[slot] \cup$
        $\{((\lambda(c, r, s).s = oldState), (cmd, res, newState))\}$
    $shadowState_{replica} := newState$
    $shadowVersion_{replica} := slot + 1$

internal transition decide($slot$, $cmd$, $res$, $newState$):
  precondition:
    $decisions[slot] = \bot \wedge$
    $\exists r, \pi : (\pi, (cmd, res, newState)) \in proposals_r[slot] \wedge$
          $(slot > 1 \Rightarrow \pi(decisions[slot-1]))$
  action:
    $decisions[slot] := (cmd, res, newState)$

internal transition learn($replica$, $slot$):
  precondition:
    $learned_{replica}[slot] = \bot \wedge decisions[slot] \neq \bot$
  action:
    $learned_{replica}[slot] := decisions[slot]$

interface transition update($replica$, $cmd$, $res$, $newState$):
  precondition:
    $(cmd, res, newState) = learned_{replica}[version_{replica}]$
  action:
    $outputs_{\nu} := outputs_{\nu} \cup \{(cmd, res)\}$
    $appState_{replica} := newState$
    $version_{replica} := version_{replica} + 1$

interface transition resetShadow($replica$, $version$, $state$):
  precondition:
    $version \geq version_{replica} \wedge$
    $(version = version_{replica} \Rightarrow state = appState_{replica})$
  action:
    $shadowState_{replica} := state$
    $shadowVersion_{replica} := version$
```

decided. Because cmd was performed speculatively, the `decide` transition checks that the state decided in the prior slot matches the state $oldState$ on which cmd was executed, thus ensuring prefix ordering. This is expressed as $(slot > 1 \Rightarrow \pi(decisions[slot-1]))$, where π is the proposal's lambda expression $\pi(c, r, s).s = oldState$.

Similarly to active replication, transition `learn` models a replica learning the decision of a slot. With the `update` transition, a replica updates its state based on what was learned for the slot. With active replication, each replica performs each client operation, while in passive replication only the primary performs client operations and backups simply obtain the resulting states.

Replicas wishing to act as primary perform transition `resetShadow` to update their speculative state and version, respectively denoted by variables $shadowState_{replica}$ and $shadowVersion_{replica}$. The new shadow state may itself be speculative and must be set to a version at least as recent as the latest learned state.

III. A Generic Protocol

Specifications 2 and 3 contain internal variables and transitions that need to be refined for an executable implementation. We start with refining active replication. BEV-Replication (Specification 4) refines active replication and contains no internal variables or transitions. As previously, the `invoke` and `response` transitions (and corresponding variables) have been omitted—they are the same as in Specification 1. Transitions `propose` and `update` of Specification 2 have been omitted for the same reason. In this section we explain how and why BEV-Replication works. The refinement mappings appear in [24].

Our term	Paxos [4]	VSR [11]	Zab [16]	meaning
replica	learner	cohort	server	stores copy of application state
certifier	acceptor	cohort	server	maintains consensus state
sequencer	leader	primary	leader	certifier that proposes orderings
BEV	ballot	view	epoch	round of certification
BEV-id	ballot number	view-id	epoch number	uniquely identifies a BEV
coordinator	leader	view manager	leader	certifier that coordinates recovery
normal case	phase 2	normal case	normal case	processing in the absence of failures
recovery	phase 1	view change	recovery	protocol to establish a new BEV
command	proposal	event record	transaction	a pair of a client id and an operation to be performed
BEV-stamp	N/A	viewstamp	zxid	uniquely identifies a sequence of proposals

TABLE I: Translation between terms used in this paper and in the various replication protocols under consideration.

A. *Certifiers and BEVs*

BEV-Replication has two basic building blocks:

- A static set of n processes called *certifiers*. A minority of these may crash. So for tolerating at most f failures, we require that $n \geq 2f + 1$ holds.
- An unbounded number of *BEV*s.

A BEV is called a "Ballot" in Paxos, an "Epoch" in Zab, and a "View" in VSR, hence the name. For ease of reference, Table I contains a translation between terminology used in this paper and those found in the papers describing the protocols under consideration.

At most one of the certifiers in a BEV is made its *sequencer*. Certifiers certify the commands proposed by the sequencer. The sequencer of a BEV can propose at most one command per slot. Note that if two certifiers certify a command in the same slot and the same BEV, it must be the same command. Moreover, a certifier cannot retract a certification. Once a majority of certifiers certify the command within a BEV, the command is *decided* (and because certifications cannot be retracted the command will remain decided thereafter). In Section III-D we show why two BEVs cannot decide different commands for the same slot.

Each BEV has a BEV-id that uniquely identifies the BEV. BEVs are totally ordered by their BEV-ids. A BEV is in one of three modes: *pending*, *operational*, or *wedged*. One BEV is the first BEV (it has the smallest BEV-id), and initially only that BEV is operational. Other BEVs start out pending. The two possible transitions on the mode of a BEV are as follows:

1) A pending BEV can become operational only if all BEVs with lower BEV-id are wedged;
2) A pending or operational BEV can become wedged under any circumstance.

This implies that at any time at most one BEV is operational and that wedged BEVs can never become unwedged. An operational BEV has exactly one certifier that is designated as *coordinator*—this certifier handles the recovery part of the protocol (Section III-D).

B. *Tracking Progress*

In Specification 4, each certifier $cert$ maintains a *progress indicator* $progress_{cert}[slot]$ for each $slot$, defined as:

Progress Indicator: A progress indicator is a pair $\langle bevId, cmd \rangle$ where $bevId$ is the identifier of a BEV and cmd is a proposed command or $\perp$, satisfying:

- If $cmd = \perp$, then the progress indicator guarantees that no BEV with an id less than $bevId$ can ever decide, or have decided, a proposal for the slot.
- If $cmd \neq \perp$, then the progress indicator guarantees that if a BEV with id $bevId'$ such that $bevId' \leq bevId$ decides (or has decided) a proposal cmd' for the slot, then $cmd = cmd'$.
- Given two progress indicators $\langle bevId, cmd \rangle$ and $\langle bevId, cmd' \rangle$ for the same slot, if neither cmd nor cmd' equals $\perp$, then $cmd = cmd'$.

We define a total ordering $\succ$ on progress indicators for the same slot as follows: $\langle bevId', cmd' \rangle \succ \langle bevId, cmd \rangle$ iff

- $bevId' > bevId$; or
- $bevId' = bevId \wedge cmd' \neq \perp \wedge cmd = \perp$.

At any certifier, the progress indicator for a slot is monotonically non-decreasing.

C. *Normal case processing*

Each certifier $cert$ *supports* exactly one BEV-id $bevId_{cert}$, initially 0, the BEV-id of the first BEV. The *normal case* holds when a majority of certifiers support the same BEV-id, and one of these certifiers is sequencer, signified by having its $isSeq_{cert}$ flag set to `true`.

Transition `certifySeq`$(cert, slot, \langle bevId, cmd \rangle)$ is performed when sequencer $cert$ proposes command cmd for the given slot and BEV. The condition $progress_{cert}[slot] = \langle bevId, \perp \rangle$ holds only if no command can be decided in this slot by a BEV with an id lower than $bevId_{cert}$. The transition requires that $slot$ is the lowest empty slot. If the transition is performed, $cert$ updates $progress_{cert}[slot]$ to reflect that if a command is decided in its BEV, then it must be command cmd.

Specification 4 BEV-Replication

```
var $bevId_{cert}, isSeq_{cert}, progress_{cert}[1...]$
  $certifics_{\nu}, snapshots_{\nu}$

initially: $certifics_{\nu} = snapshots_{\nu} = \varnothing \wedge$
  $\forall cert : bevId_{cert} = 0 \wedge isSeq_{cert} = \texttt{false} \wedge$
            $\forall slot \in \mathbb{N}^{+} : progress_{cert}[slot] = \langle 0, \perp \rangle$

interface transition certifySeq($cert, slot, \langle bevId, cmd \rangle$):
  precondition:
    $isSeq_{cert} \wedge bevId = bevId_{cert} \wedge cmd \in proposals_{replica} \wedge$
    $progress_{cert}[slot] = \langle bevId, \perp \rangle \wedge$
    $(\forall s \in \mathbb{N}^{+} : progress_{cert}[s] = \langle bevId, \perp \rangle \Rightarrow s \geq slot)$
  action:
    $progress_{cert}[slot] := \langle bevId, cmd \rangle$
    $certifics_{\nu} := certifics_{\nu} \cup \{(cert, slot, \langle bevId, cmd \rangle)\}$

interface transition certify($cert, slot, \langle bevId, cmd \rangle$):
  precondition:
    $\exists cert' : (cert', slot, \langle bevId, cmd \rangle) \in certifics_{\nu} \wedge$
    $bevId_{cert} = bevId \wedge \langle bevId, cmd \rangle \succ progress_{cert}[slot]$
  action:
    $progress_{cert}[slot] := \langle bevId, cmd \rangle$
    $certifics_{\nu} := certifics_{\nu} \cup \{(cert, slot, \langle bevId, v \rangle)\}$

interface transition observeDecision($replica, slot, cmd$):
  precondition:
    $\exists bevId :$
      $|\{cert \mid (cert, slot, \langle bevId, cmd \rangle) \in certifics_{\nu}\}| > \frac{n}{2} \wedge$
    $learned_{replica}[slot] = \perp$
  action:
    $learned_{replica}[slot] := cmd$

interface transition supportBEV($cert, bevId, coord$):
  precondition:
    $bevId > bevId_{cert}$
  action:
    $bevId_{cert} := bevId \wedge isSeq_{cert} := \texttt{false}$
    $snapshots_{\nu} :=$
      $snapshots_{\nu} \cup \{(cert, bevId, coord, progress_{cert})\}$

interface transition recover($cert, bevId, \mathcal{S}, follow$):
  precondition:
    $bevId_{cert} = bevId \wedge \neg isSeq_{cert} \wedge |\mathcal{S}| > \frac{n}{2} \wedge$
    $\mathcal{S} \subseteq \{(id, prog) \mid (id, bevId, cert, prog) \in snapshots_{\nu}\}$
  action:
    $\forall s \in \mathbb{N}^{+} :$
      $\langle b, cmd \rangle := \max_{\succ}\{prog[s] \mid (id, prog) \in \mathcal{S}\}$
      $progress_{cert}[s] := \langle bevId, cmd \rangle$
      if $follow \vee cmd \neq \perp$ then
        $certifics_{\nu} := certifics_{\nu} \cup \{(cert, s, \langle bevId_{cert}, cmd \rangle)\}$
    $isSeq_{cert} := \texttt{true}$
```

Sequencer *cert* also notifies all other certifiers by adding $(cert, slot, \langle bevId, cmd \rangle)$ to set $certifics_{\nu}$ (modeling the sequencer broadcasting to the certifiers).

A certifier that receives such a message checks to see if the message contains the same BEV-id that it is currently supporting and that the progress indicator in the message exceeds its own progress indicator for the same slot. If so, then the certifier updates its own progress indicator and certifies the proposed command (transition `certify`$(cert, slot, \langle bevId, cmd \rangle)$).

The `observeDecision`$(replica, slot, cmd)$ transition at some replica *replica* is enabled when a majority of certifiers in the same BEV have certified a command (clt, op). At this point, the command is decided and all replicas that undergo the observeDecision transition for this slot will decide on the same command.

D. Recovery

In this section, we show how BEV-Replication deals with failures. The reason for having an unbounded number of BEVs is to achieve liveness. When an operational BEV is no longer certifying proposals, for example, because its sequencer has crashed or is slow, it can become wedged and a BEV with a higher BEV-id can become operational.

BEV modes are implemented as follows: A certifier *cert* can transition to supporting a new BEV-id *bevId* and coordinator *coord* (transition `supportBEV`$(cert, bevId, coord)$). This transition can only increase *bevId*. The transition also sends the certifier's *snapshot* by adding it to the set *snapshots*. A snapshot is a four-tuple $(cert, bevId, coord, progress_{cert})$ containing the certifier's identifier, its current BEV-id, the identifier of *coord*, and the certifier's list of progress indicators. Note that a certifier can send at most one snapshot for each BEV.

A BEV *bevId* with coordinator *coord* is operational, by definition, if a majority of certifiers support *bevId* and added $(cert, bevId, coord, progress_{cert})$ to the set *snapshots*. Clearly, the majority requirement guarantees that there cannot be two BEVs that are simultaneously operational, nor can there be operational BEVs that do not have exactly one coordinator. Certifiers that support *bevId* can no longer certify commands in BEVs prior to *bevId*. Consequently, if a majority of certifiers support a BEV-id larger than x, then all BEVs with an id of x or lower are wedged.

The role of the coordinator of BEV *bevId* is to guide the BEV to normal case operation, with one certifier acting as sequencer and assigning commands to slots. The coordinator also helps to ensure that the BEV does not decide commands inconsistent with prior BEVs by collecting enough information (in the form of progress indicators) about prior BEVs.

After coordinator *cert* starts supporting *bevId* (transition `supportBEV`$(cert, bevId, cert)$), it starts waiting for snapshots from the other certifiers that support the same *bevId* and coordinator. Transition `recover`$(cert, bevId, \mathcal{S}, follow)$ is enabled at *cert* if the set $\mathcal{S}$ contains snapshots for *bevId* and coordinator *cert* from a majority of certifiers.

In the action of `recover`$(cert, bevId, \mathcal{S}, follow)$, coordinator *cert* determines for each slot the maximum progress indicator $\langle b, cmd\rangle$ for the slot in the snapshots contained in $\mathcal{S}$. It then sets its own progress indicator to $\langle bevId, cmd\rangle$. It is easy to see that $bevId \geq b$. We argue that $\langle bevId, cmd\rangle$ satisfies the definition of progress indicator in Section III-B. All certifiers in $\mathcal{S}$ support $bevId$ and form a majority. Thus, it is not possible for any BEV between b and $bevId$ to decide a command because none of these certifiers can certify a command in those BEVs. There are two cases:

- If $cmd = \bot$, no command can be decided before b, so no command can be decided before $bevId$. Hence, $\langle bevId, \bot\rangle$ is a correct progress indicator.
- If $cmd \neq \bot$, then if a command is decided by b or a BEV prior to b, it must be cmd. Since no command can be decided by BEVs between b and $bevId$, $\langle bevId, cmd\rangle$ is a correct progress indicator.

If $cmd \neq \bot$, then it is important that the same command is re-proposed in the slot for the current BEV in order to achieve liveness. If $cmd = \bot$, liveness is not in jeopardy. However, it does not hurt to tell the other certifiers that no command was or can be decided by BEVs before $bevId$. The *follow* flag makes this optional. In the case of Paxos, this flag is **false**. However, it turns out that for protocols that support prefix ordering, the other certifiers need to learn the most current progress indicator (see Section III-E).

The coordinator sets its $isSeq_{cert}$ flag upon recovery. As a result it becomes enabled to propose new commands, and normal case for the BEV begins.

E. Passive Replication

Section II-C showed how in passive replication a state update from a particular primary can only be decided in a slot if all state updates for earlier slots proposed by the same primary have been decided as well. We called this property prefix ordering. However, Specification 4 does not satisfy prefix ordering because slots can be decided in any order and there is no requirement on who proposed a particular command. Thus BEV-Replication does not refine Passive Replication. However, it is possible to refine BEV-Replication to obtain a specification that also refines Passive Replication and satisfies prefix ordering. We call this specification BEV-Replication-PO.

One way of implementing prefix ordering would be for the sequencer (the primary) to wait with proposing a command for a slot until it knows the decisions for all prior slots. Doing so would be slow. Instead, a sequencer speculatively proposes commands for new slots, but the BEV-Replication protocol is refined to guarantee that such commands cannot be decided until the sequencer's prior speculative commands have been decided. In particular, we add the following requirements at all certifiers *cert*:

- For all slots s_1 and s_2 such that $s_1 < s_2$, if $progress_{cert}[s_1] = \langle b_1, c_1\rangle$ and $progress_{cert}[s_2] = \langle b_2, c_2\rangle$, then $b_1 \geq b_2$. Informally, progress indicators for earlier slots are always at least as "current" as progress indicators for later slots.
- For all slots s, if $progress_{cert}[s] = \langle b, c_1\rangle$, $progress_{cert}[s+1] = \langle b, c_2\rangle$ and $c_2 \neq \bot$, then $c_1 \neq \bot$. That is, if a certifier certifies a command in slot $s+1$, then it must have certified a command in slot s in the same BEV b.

With these requirements, if a state update proposed by a sequencer is decided in a slot s, then all prior slots are decided as reflected in the progress indicators of that sequencer. After all, if all certifiers in a majority have certified $\langle b, c'\rangle$ in some slot $s+1$ (and thus c' is the decided command for slot $s+1$) and the sequencer has certified $\langle b, c\rangle$ in slot s, then it must also be the case that that same majority of certifiers have certified $\langle b, c\rangle$ in slot s.

These constraints can be implemented as follows. Note that the first requirement already holds at the sequencer. In order to ensure that all progress indicators have the same BEV-id at all certifiers, we take the following two steps:

1) We add the constraint *follow* = **true** to the precondition of the transition `recover`, such that the coordinator updates progress indicators for all slots.
2) We add the constraint $(slot > 1 \Rightarrow \exists c \neq \bot : progress_{cert}[slot - 1] = \langle bevId, c\rangle)$ to transition `certify`, thus enforcing that certifiers certify slots in order.

Also, BEV-Replication-PO inherits from transitions `invoke` and `response` of Specification 1 as well as transitions `propose`, `update`, and `resetShadow` of Specification 3. Naturally, the variables contained in these transitions are inherited as well.

The passive replication protocols that we consider in this paper, VSR and Zab, share the following design decision in the recovery procedure: The sequencer broadcasts a single message containing its entire snapshot rather than sending separate certifications for each slot. Certifiers wait for this comprehensive snapshot, and overwrite their own snapshot with it, before they certify new commands in this BEV. As a result, at a certifier all $progress_{cert}$ slots have the same BEV identifier, and can thus be maintained as a separate variable. Additionally, the *sequencer* of a passive replication protocol does not need to send the entire application state in the `certifySeq` transition; a state update suffices. We

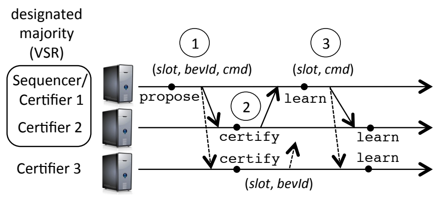


Fig. 3: Normal case processing at three certifiers. Dots indicate transitions, and arrows between certifiers are messages.

shall also see that with VSR, the coordinator of a BEV is not necessarily its sequencer.

IV. Implementation

Specifications BEV-Replication and BEV-Replication-PO do not contain internal variables or transitions, and are in some sense an executable implementation of active and passive replication. However, Paxos, VSR, and Zab each make further refinements to BEV-Replication and BEV-Replication-PO to make the protocol pragmatic. We show, informally, the final refinements of these specifications required to obtain those protocols.

A. Normal Case

We first turn to implementing the state and transitions of BEV-Replication and BEV-Replication-PO. The first question is how to implement the variables. Variables $inputs_\nu$, $outputs_\nu$, $certifics_\nu$, and $snapshots_\nu$ are not per-process but global. They model messages that have been sent. In actual protocols, these are implemented by the network: a value in either set is implemented by a message on the network tagged with the appropriate type, such as `snapshot`.

The remaining variables are all local to a process such as a client, a replica, or a certifier, and can be implemented as ordinary program variables. In Zab, the $progress_{cert}$ variable is implemented by a queue of commands. In VSR, the $progress_{cert}$ variable is replaced by the application state and a counter that counts the number of updates made to the state in this BEV.

Fig. 3 illustrates the steps of normal case processing in the protocols. The figure shows three certifiers ($f = 1$). Upon receiving an operation from a client (not shown):

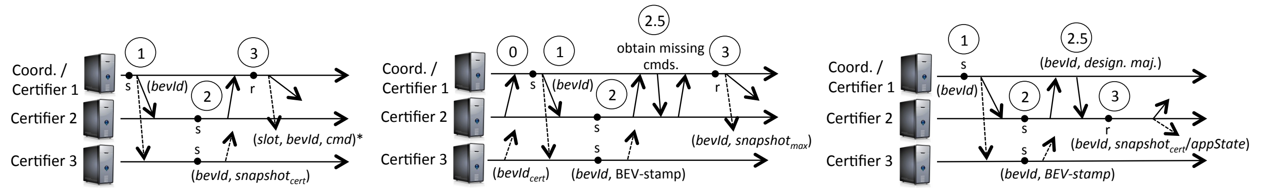
1) The sequencer $cert$ proposes a command for the next open slot and sends a message to the other certifiers (maps to $\texttt{certifySeq}(cert, slot, \langle bevId, cmd\rangle)$). In the case of VSR and Zab, the command is a state update that results from executing the client operation; in the case of Paxos, the command is the operation itself.
2) Upon receipt by a certifier $cert$, if $cert$ supports the BEV-id in the message, then $cert$ updates its slot and replies to the sequencer (transition $\texttt{certify}(cert, slot, \langle bevId, cmd\rangle)$). With VSR and Zab, prefix-ordering must be ensured, and $cert$ only replies to the sequencer if its progress indicator for $slot - 1$ contains a non-empty command for the same BEV-id.
3) If the sequencer receives successful responses from a majority of certifiers (transition observeDecision), then the sequencer learns the decision and broadcasts a `decide` message for the command to the replicas (resulting in `learn` transitions that update the replicas (see Specifications 2 and 3).

There are more specific design decisions:

- In the case of VSR, a specific majority of certifiers is determined a priori and fixed for each BEV. We call this a *designated majority*. In Paxos and Zab, any certifier can certify proposals.
- In VSR, replicas are co-located with certifiers, and certifiers speculatively update their local replica as part of certification. A replica may well be updated before some proposed command is decided, so if another command is decided the state of the replica must be rolled back, as we shall see later. Upon learning that the command has been decided (Step 3), the sequencer responds to the client.
- Paxos, being an active replication protocol, does not require replicas to be co-located with certifiers. Also, these protocols use leases [28], [4] for read-only operations (optional in Paxos). Leases have the advantage that read-only operations can be served at a single replica while still guaranteeing linearizability. This method assumes synchronized clocks (or clocks with bounded drift) and has the sequencer obtain a lease for a certain time period. A sequencer that is holding a lease can thus forward read-only operations to any replica, inserting the operation in the ordered stream of commands sent to that replica.
- Zab (or rather ZooKeeper, the service) has any replica handle read-only operations individually, circumventing Zab. This is efficient, but a replica may not have learned the latest decided proposals and its clients receive results based on stale state. Read-only operations in ZooKeeper are thus not linearizable.

For replicas to learn about decisions, two options exist that vary in network latency and the number of messages exchanged:

- Certifiers can respond back to the sequencer. The sequencer learns that its proposed command has been decided if the sequencer receives responses from a majority (counting itself). The sequencer then notifies the replicas.



(a) Paxos (b) Zab (c) VSR

Fig. 4: Depiction of the recovery phase. Dots represent transitions and are labeled with `s` and `r`, respectively denoting a `supportBEV` and a `recovery` transition. Messages of the form (x, y)* contain multiple (x,y) tuples.

- Certifiers can broadcast notifications to all replicas, and each replica can individually determine if a majority of the certifiers have certified a particular command.

There is a trade-off between the two options: with n certifiers and m replicas, the first approach requires $n+m$ messages and two network latencies. The second approach requires $n \times m$ messages but involves only one network latency. All implementations we know of use the first approach.

B. Recovery

Fig. 4 illustrates the recovery steps in the protocols.

1) Recovery in Paxos: In Paxos, the coordinator and sequencer of an operational BEV are the same certifier. If unhappy with progress, a certifier c starts the following process to try to become coordinator itself (see Fig. 4a):

- Step 1: Certifier c supports a new BEV $bevId$ proposing itself as coordinator (transition $\texttt{supportBEV}(c, bevId, c)$), and queries at least a majority of certifiers.
- Step 2: Upon receipt, a certifier $cert$ that transitions to supporting BEV $bevId$ and certifier c as coordinator ($\texttt{supportBEV}(cert, bevId, c)$) responds with its snapshot.
- Step 3: Upon receiving responses from a majority, certifier c learns that it is coordinator of BEV $bevId$ and becomes the sequencer (transition $\texttt{recover}(c, bevId, \mathcal{S}, follow)$). Normal case operation resumes after c broadcasts the command with the highest $bevId$ for each slot not known to be decided.

2) Recovery in Zab: Coordinators in Zab are determined by a weak leader election protocol Ω [29]. When Ω determines that a coordinator has become unresponsive, it initiates a four-step protocol (see Fig. 4b) in which a new coordinator is elected:

- Step 0: Ω proposes a prospective coordinator c and notifies the certifiers. Upon receipt a certifier sends a message containing the BEV-id it supports to c.
- Step 1: Upon receiving such messages from a majority of certifiers, prospective coordinator c selects a BEV-id $bevId$ that is one larger than the maximum it received, transitions to supporting it (transition $\texttt{supportBEV}(c, bevId, c)$), and broadcasts this to the other certifiers for approval.
- Step 2: Upon receipt, if certifier $cert$ can support $bevId$ and has not agreed to a certifier other than c becoming coordinator of the BEV, it performs transition $\texttt{supportBEV}(cert, bevId, c)$. Zab exploits prefix ordering to optimize the recovery protocol. Instead of sending its entire snapshot to the coordinator of the BEV, a certifier that transitions to supporting BEV $bevId$ sends a ***BEV-stamp***. A BEV-stamp is a lexicographically ordered pair consisting of the BEV-id in the snapshot (the same for all slots) and the number of slots in the BEV for which it has certified a command.
- Step 2.5: If c receives responses from a majority, it learns that it is now the coordinator and (sequencer) of $bevId$. Coordinator c computes the maximum BEV-stamp and determines if it is missing commands. If it is the case, c retrieves them from the certifier $cert_{max}$ with the highest received BEV-stamp. If c is missing too many commands (e.g. if c did not participate in the last BEV $cert_{max}$ participated in), $cert_{max}$ sends its entire snapshot to c.
- Step 3: After receiving the missing commands, c broadcasts its snapshot (transition $\texttt{recover}(c, bevId, \mathcal{S}, follow)$).

3) Recovery in VSR: In VSR, the coordinator of a BEV is a deterministic function of the BEV-id. In the case of VSR, a BEV-id is a lexicographically ordered pair comprising a number and the process identifier of the coordinator; the coordinator is easily extracted from the BEV-id.

If unhappy with progress, a certifier c starts the following recovery process (see Fig. 4c):

- Step 1: c starts supporting BEV $bevId$ (transition $\texttt{supportBEV}(c, bevId, c)$), and queries at least a majority of certifiers.
- Step 2: Upon receipt of such a query, a certifier $cert$ starts supporting $bevId$ (transition $\texttt{supportBEV}(cert, bevId, c)$). Similarly to Zab, $cert$ sends its BEV-stamp to the coordinator.
- Step 2.5: Certifier c, upon receiving BEV-stamps from a majority of certifiers, learns that it has become

coordinator. Coordinator c uses the set of certifiers that respond as the designated majority for the BEV and assigns the sequencer role to the certifier p that reported the highest BEV-stamp. The coordinator then notifies certifier p, requesting it to become sequencer.
- Step 3: sequencer p, having the latest state, broadcasts its snapshot (transition $\texttt{recover}(p, \mathit{bevId}, \mathcal{S}, \mathit{follow})$). In the case of VSR, the state that the new sequencer sends consists of its application state rather than a snapshot.

C. Garbage Collection

BEV-Replication has each certifier building up state about all slots, which does not scale. Below, we describe approaches to trim this state at run-time.

1) VSR: In VSR, no garbage collection is required. Certifiers and replicas are co-located, and only store the most recent BEV-id they adopted and the application state that is updated upon certification of a command. During recovery the sequencer simply sends the application state to the replicas, and consequently, there is no need to replay any decided commands.

2) Paxos & Zab: Paxos and Zab do not cover the issue of garbage collection. The ZooKeeper service creates checkpoints of the application state and requires Zab to only replay decisions since the last checkpoint [17], but it is not described how Zab implements this.

D. Liveness

All of the protocols require—in order to make progress—that at most a minority of certifiers experience crash failures. If the current BEV is no longer making progress, a new BEV must become operational. If certifiers are slow at this in the face of an actual failure, then performance may suffer. However, if certifiers are too aggressive, BEVs will become wedged before being able to decide commands, even in the absence of failures.

To guarantee progress, some BEV with a correct sequencer must eventually not get preempted by a higher BEV [30], [31]. Such a guarantee is difficult or even impossible to make [32], but with careful failure detection a good trade-off can be achieved between rapid failure recovery and spurious wedging of BEVs [33].

In this section, we will look at how the various protocols try optimizing progress.

1) Partial Memory Loss: If certifiers keep their state on stable storage (say, a disk), then a crash followed by a recovery is not treated as a failure but instead as the affected certifier being slow. Stable storage allows protocols like Paxos, VSR, and Zab to deal with such transients. Even if all machines crash, as long as a majority eventually recovers their state from before the crash, the service can continue operating.

However, writing state to stable storage at every update can slow down performance by orders of magnitude.[2] VSR has an option that stores the BEV-id of a certifier on disk (only updated on recovery), but keeps the rest of the state in memory. In the case of a crash followed by a recovery, the certifier can determine that it has lost its application state and progress indicators, but the certifier still remembers its latest BEV-id. This can improve the chances of a recovery assuming enough state is present in correct certifiers. It does not allow recovery from a case where all certifiers temporarily lose power and lose the application state.

2) Total Memory Loss: In Section 5.1 of "Paxos Made Live" [8], the developers of Google's Chubby service describe a way for Paxos to deal with permanent memory loss of a certifier (due to disk corruption). The memory loss is total, so the recovering certifier starts in an initial state. It copies its state from another certifier and then waits until it has seen one decision before starting to participate fully in the Paxos protocol again. While this sounds reasonable, it is flawed (confirmed by the authors of [8]). It breaks the invariant that a certifier's BEV-id can only increase over time. By copying the state from another certifier, it may, as it were, go back in time, which can cause divergence.

Nonetheless, total memory loss can be tolerated by extending the protocols. The original Paxos paper [4] shows how the set of certifiers can be reconfigured, and this has been worked out in greater detail in Microsoft's SMART project [34] and later for Viewstamped Replication as well [12]. Conditions for liveness for such reconfigurable protocols, as well as improved reconfiguration techniques, can be found in [35].

V. Discussion

Table II summarizes differences between Paxos, VSR, and Zab. We believe that these differences are important because they both demonstrate that the protocols do not refine one another (and are thus different), and the differences have pragmatic consequences as discussed below. The comparisons are based on published algorithms; actual implementations may vary.

The various techniques described below can be mixed-and-matched in order to materialize other refinements of BEV-Replication. This also holds for additional techniques used in other papers that describe these protocols [8], [33], [36], [37], as long as the conditions described in Section III continue to hold.

[2]Harp assumes UPS to mask the overhead of writing to disk.

What	Section	Paxos	VSR	Zab
replication style	II-B	active	passive	passive
read-only operations	IV-A	leasing	certification	read any replica
designated majority	IV-A, IV-B	no	yes	no
time of execution	IV-A	upon decision	upon certification	depends on role
coordinator selection	IV-B	majority vote or deterministic	deterministic	majority vote
sequencer selection	IV-B	same as coordinator	coordinator assigned	same as coordinator
recovery direction	IV-B	two-way	from sequencer	two-way
recovery granularity	IV-B	slot-at-a-time	application state	command prefix
tolerates memory loss	IV-D1, IV-D2	reconfigure	partial	no

TABLE II: Overview of important differences between the various protocols.

The discussion in this section is organized around normal case processing and recovery overheads.

A. Normal Case

a) Passive vs. Active Replication: In active replication, there are at least $f+1$ replicas that each have to execute operations, albeit in parallel. In passive replication, only the sequencer executes operations, but has to propagate state updates to the backups. Depending on the overheads of executing operations and the size of state update messages, one or the other may perform better. Passive replication has the advantage that execution at the sequencer does not have to be deterministic and can take advantage of parallel processing on multiple cores.

b) Read-only Optimizations: Paxos supports leasing for read-only operations, but there is no reason why leasing could not be added to VSR and Zab as well. Indeed, Harp (built on VSR) uses leasing as well. A lease improves latency of read-only operations in the normal case, but delays recovery in case of a failure. ZooKeeper allows clients to read any replica at any time. Doing so compromises the observed consistency since replicas may have stale state.

c) Designated Majority: VSR uses designated majorities. This has the advantage that the other (typically f) certifiers and replicas are not employed during normal operation, and they play only a small role during recovery, saving almost half of overhead. Those machine can be used for other purposes. There are two disadvantages: (1) if the designated majority contains the slowest certifier the protocol will run at the rate of that slowest certifier, as opposed to the "median" certifier; and (2) if one of the certifiers in the designated majority crashes or becomes unresponsive or slow, then a recovery is necessary. In Paxos and Zab, recovery is necessary only if the sequencer crashes. A middle ground can be achieved by using $2f+1$ certifiers and $f+1$ replicas.

d) Time of Command Execution: In VSR, a sequencer pre-processes operations to produce state updates. Replicas apply state updates speculatively at the same time that they are certified, possibly before they are decided. Commands are forgotten as soon as they are applied to the state. This means that no garbage collection is necessary. A disadvantage is that the response to a client operation must be delayed until all replicas in the designated majority have updated their application state. In other protocols, only one replica has to have updated its state and computed a response, because in case the replica fails another deterministic replica is guaranteed to compute the same response. Note that at this time each command that led to this state and response has been certified by a majority and therefore the state and response are recoverable even if this one replica crashes.

In Zab, the sequencer also speculatively applies client operations to compute state updates before they are decided. However, replicas only apply those state updates until after they have been decided. In Paxos, replicas only execute a command after it has been decided and there is no speculative execution.

B. Recovery

e) Coordinator/Sequencer Selection: All protocols have different approaches to coordinator and sequencer selection. The trade-offs between the coordinator selection protocols are unclear. However, VSR provides an advantage in selecting a sequencer that has the most up-to-date state (taking advantage of prefix ordering), and thus it does not have to recover this state from the other certifiers, simplifying and streamlining recovery.

f) Recovery Direction: In case of a recovery, Paxos allows the coordinator to recover the state of previous slots and, at the same time, the sequencer (the same process) can propose new commands for slots for which it already has retrieved sufficient state. No particular ordering is required here. However, all certifiers must send their certification state to the coordinator, and then the coordinator has to re-propose commands for slots. With Zab, the certifiers only send their BEV-stamps to the coordinator. If the coordinator is missing commands, it must obtain them from the certifier with the highest BEV-stamp, which may consist of an entire snapshot in certain cases. Since commands are state updates, the Zab coordinator recovers slots in prefix order.

Garbage collection of the certification state is important to ensure that the amount of state that has to be exchanged on recovery does not become too large.

There is a trade-off between how aggressively garbage is collected and recovery time. Often it is better for performance to bring a recovering replica up to date by replaying decided commands that it missed rather than by copying state. But if the log of decided commands becomes very large, performance suffers due to the overhead of transferring the log and replaying the commands.

With VSR, certifiers send a BEV-stamp to the coordinator. In contrast to Zab, the sequencer is chosen by the coordinator based on these BEV-stamps, and the coordinator does not need to obtain missing commands. The selected sequencer pushes its snapshot to the other certifiers. Because of prefix ordering, the snapshot has to be transferred and processed before new certification requests, possibly resulting in a performance hiccup.

g) Recovery Granularity: In VSR, state sent from the sequencer to the backups is the entire application state. For VSR, this state is transaction manager state and is small, but in general such an approach does not scale. However, in some cases that cost is unavoidable, even in the other protocols. For example, if a replica has a disk failure, replaying all commands from day 0 is not scalable either, and the recovering replica instead will have to seed its state from another one. In such a case, the replica will load a checkpoint and then replay missing commands to bring the checkpoint up-to-date—this technique is used in Zab (and in Harp as well). With passive replication protocols, replaying missing commands simply means applying state updates; with active replication protocols, replaying commands entails re-executing commands. Depending on the overheads of executing operations and the size of state update messages, one or the other approach may perform better.

h) Tolerating Memory Loss: Various protocols have described options for dealing with memory loss. An option suggested by VSR is to keep only a BEV-id on disk; the remaining of the state is in memory. This technique works only in restricted situations where at least one certifier has the most up-to-date state in memory, but can be used to improve availability. An option described for Paxos works by reconfiguring the set of certifiers and allows the entire state to be held in memory, but the technique cannot be applied within a single slot—availability will only be improved for the next slot. The techniques are orthogonal. The only approach that survives a total power loss is to keep all state in some form of stable storage.

VI. A Bit of History

Based on the discussion so far one may think that BEVs, sequencers, and so on were first introduced by protocols that implement BEV-Replication. However, we believe the first consensus protocol to use BEVs and sequencers is due to Dwork, Lynch, and Stockmeyer (DLS) [30]. While DLS can be used for state machine replication, an instance of the protocol can only make one decision, and one would have to instantiate DLS for each slot. BEV-Replication allows each BEV to decide multiple commands for better efficiency. Also, BEVs in DLS are countable and BEV $b+1$ cannot start until BEV b has run its course. Thus, DLS does not refine BEV-Replication.

Chandra and Toueg's work on consensus [31] formalized the conditions under which consensus protocols terminate by encapsulating synchrony assumptions in the form of failure detectors. Their consensus protocol resembles the Paxos single-decree Synod protocol and refines BEV-Replication.

To the best of our knowledge, the idea of using majority intersection to avoid potential inconsistencies first appears in Thomas [38]. Quorum replication [38], [39] supports only storage objects with `read` and `write` operations (or, equivalently, `get` and `put` operations in the case of a Key-Value Store). This is less general than state machine-based services that can be replicated with active or passive replication. But quorum replication do not require consensus [40] and is thus not subject to the FLP impossibility result [32].

VII. Conclusion

Paxos, VSR, and Zab are well-known replication protocols for an asynchronous environment that admits a bounded number of crash failures. The paper describes a specification for BEV-Replication, a generic specification that contains important design features that the protocols share. These features include an unbounded number of totally ordered BEVs, a static set of certifiers, and at most one sequencer per BEV.

The protocols then differ in how they refine BEV-Replication. We were able to disentangle fundamentally different design decisions in the three protocols and consider their impact on performance. Most importantly, compute-intensive services are better off with a passive replication strategy such as used in VSR and Zab (provided that state updates are of a reasonable size). To achieve predictable low-delay performance for short operations during both normal case execution and recovery, an active replication strategy without designated majorities, such as used in Paxos, is the best option.

References

[1] L. Lamport, "Specifying concurrent program modules," *Trans. on Programming Languages and Systems*, vol. 5, no. 2, pp. 190–222, Apr. 1983.

[2] R. Milner, "An algebraic definition of simulation between programs," in *Proc. of the 2nd Int. Joint Conference on Artificial Intelligence*, D. Cooper, Ed. London, UK: William Kaufmann, British Computer Society, Sep. 1971, pp. 481–489.

[3] N. A. Lynch and F. W. Vaandrager, "Forward and backward simulations, ii: Timing-based systems," *Inf. Comput.*, vol. 128, no. 1, pp. 1–25, 1996.

[4] L. Lamport, "The part-time parliament," *Trans. on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[5] ——, "Time, clocks, and the ordering of events in a distributed system," *CACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[6] F. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[7] M. Burrows, "The Chubby Lock Service for loosely-coupled distributed systems," in *7th Symposium on Operating System Design and Implementation*, Seattle, WA, Nov. 2006.

[8] T. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proc. of the 26th ACM Symp. on Principles of Distributed Computing*. Portland, OR: ACM, May 2007, pp. 398–407.

[9] M. Isard, "Autopilot: Automatic data center management," *Operating Systems Review*, vol. 41, no. 2, pp. 60–67, Apr. 2007.

[10] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI06)*, Nov. 2006.

[11] B. Oki and B. Liskov, "Viewstamped Replication: A general primary-copy method to support highly-available distributed systems," in *Proc. of the 7th ACM Symp. on Principles of Distributed Computing*. Toronto, Ontario: ACM SIGOPS-SIGACT, Aug. 1988, pp. 8–17.

[12] B. Liskov and J. Cowling, "Viewstamped Replication revisited," MIT, Tech. Rep. MIT-CSAIL-TR-2012-021, Jul. 2012.

[13] B. Lampson and H. Sturgis, "Crash recovery in a distributed data storage system," Xerox PARC, Palo Alto, CA, Tech. Rep., 1976.

[14] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams, "Replication in the Harp file system," in *Proc. of the Thirteenth ACM Symp. on Operating Systems Principles*, Pacific Grove, CA, Oct. 1991.

[15] F. Junqueira and B. Reed, "A simple totally ordered broadcast protocol," in *2nd Workshop on Large-Scale Distributed Systems and Middleware*, vol. 341. New York, NY: Association for Computing Machinery, 2008.

[16] F. Junqueira, B. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Int'l Conf. on Dependable Systems and Networks (DSN-DCCS'11)*. IEEE, 2011.

[17] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," in *USENIX Annual Technology Conference*, 2010.

[18] B. Lampson, "How to build a highly available system using consensus," in *Distributed systems (2nd Ed.)*, S. Mullender, Ed. New York, NY: ACM Press/Addison-Wesley Publishing Co., 1993, pp. 1–17.

[19] P. Alvaro, T. Condie, N. Conway, J. Hellerstein, and R. Sears, "I do declare: Consensus in a logic language (slides)," in *Proc. of NetDB'09*, ser. see slides at http://netdb09.cis.upenn.edu/slides/idodeclare.pdf, Big Sky, MT, Oct. 2009.

[20] C. Cachin, "Yet another visit to Paxos," IBM Research, Zurich, Switzerland, Tech. Rep. RZ3754, 2009.

[21] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *Trans. on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[22] C. Sprenger and D. Basin, "Developing security protocols by refinement," in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS '10. ACM, 2010, pp. 361–374.

[23] A. Shankar and S. Lam, "Construction of network protocols by stepwise refinement," in *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, ser. Lecture Notes in Computer Science, J. Bakker, W.-P. Roever, and G. Rozenberg, Eds. Springer Berlin Heidelberg, 1990, vol. 430, pp. 669–695.

[24] R. Van Renesse, N. Schiper, and F. B. Schneider, "Vive la Différence: Paxos vs. Viewstamped Replication vs. Zab," arXiv, Tech. Rep. 1309.5671, Sep. 2013.

[25] P. Alsberg and J. Day, "A principle for resilient sharing of distributed resources," in *Proc. of the 2nd Int. Conf. on Software Engineering (ICSE'76)*. San Francisco, CA: IEEE, Oct. 1976, pp. 627–644.

[26] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg, "The primary-backup approach," in *Distributed systems (2nd Ed.)*, S. Mullender, Ed. New York, NY: ACM Press/Addison-Wesley Publishing Co., 1993.

[27] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, Apr. 1980.

[28] C. Gray and D. Cheriton, "Leases: an efficient fault-tolerant mechanism for distributed file cache consistency," in *Proc. of the Twelfth ACM Symp. on Operating Systems Principles*, Litchfield Park, AZ, Nov. 1989.

[29] N. Schiper and S. Toueg, "A robust and lightweight stable leader election service for dynamic systems," in *Int'l Conf. on Dependable Systems and Networks (DSN'08)*. ieee, 2008, pp. 207–216.

[30] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," in *Proc. of the 3rd ACM Symp. on Principles of Distributed Computing*. Vancouver, BC: ACM SIGOPS-SIGACT, Aug. 1984, pp. 103–118.

[31] T. Chandra and S. Toueg, "Unreliable failure detectors for asynchronous systems," in *Proc. of the 11th ACM Symp. on Principles of Distributed Computing*. Montreal, Quebec: ACM SIGOPS-SIGACT, Aug. 1991, pp. 325–340.

[32] M. Fischer, N. Lynch, and M. Patterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[33] J. Kirsch and Y. Amir, "Paxos for system builders: an overview," in *Proc. of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS'08)*, 2008, pp. 1–6.

[34] J. Lorch, A. Adya, W. Bolosky, R. Chaiken, J. Douceur, and J. Howell, "The SMART way to migrate replicated stateful services," in *Proc. of the 1st Eurosys Conference*. Leuven, Belgium: ACM, Apr. 2006, pp. 103–115.

[35] K. Birman, D. Malkhi, and R. van Renesse, "Virtually synchronous methodology for dynamic service replication," Microsoft Research, Tech. Rep. MSR-TR-2010-151, 2010.

[36] P. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring Paxos: A high-throughput atomic broadcast protocol," in *Proc. of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'10)*, Jun. 2010, pp. 527–536.

[37] W. Bolosky, D. Bradshaw, R. Haagens, N. Kusters, and P. Li, "Paxos replicated state machines as the basis of a high-performance data store," in *Proc. of the 8th Symp. on Networked Systems Design and Implementation*, Boston, MA, 2011.

[38] R. Thomas, "A solution to the concurrency control problem for multiple copy databases," in *Proc. of COMPCON 78 Spring*. Washington, D.C.: IEEE Computer Society, Feb. 1978, pp. 88–93.

[39] D. Gifford, "Weighted voting for replicated data," in *Proc. of the 7th ACM Symp. on Operating Systems Principles*. Pacific Grove, CA: ACM SIGOPS, Dec. 1979, pp. 150–162.

[40] H. Attiya, A. B. Noy, and D. Dolev, "Sharing memory robustly in message passing systems," *J. ACM*, vol. 42, no. 1, pp. 121–132, 1995.