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

Robbert van Renesse, Nicolas Schiper, Fred B. Schneider, *Fellow, IEEE*

Abstract—Paxos, Viewstamped Replication, and Zab are replication protocols for high-availability in asynchronous environments with crash failures. Claims have been made about their similarities and differences. But how does one determine whether two protocols are the same, and if not, how significant are the differences?
We address these questions using refinement mappings. Protocols are expressed as succinct specifications that are progressively refined to executable implementations. Doing so enables a principled understanding of the correctness of design decisions for implementing the protocols. Additionally, differences that have a significant impact on performance are surfaced by this exercise.

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

1 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]. If two protocols refine one another then we might argue that they are alike. But if they don't, how does one characterize the similarities and differences between two protocols?

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

- *Paxos* [3] is a state machine replication protocol [4], [5]. We consider a version of Paxos that uses the multi-decree Synod consensus algorithm described in [3], sometimes called Multi-Paxos. Implementations appear in Google's Chubby service [6], [7], Microsoft's Autopilot service [8] (used by Bing), and the Ceph distributed file system [9] (with interfaces now part of the standard Linux kernel).
- *Viewstamped Replication* (VSR) [10], [11] is a replication protocol originally targeted at replicating participants in a Two-Phase Commit (2PC) [12] protocol. VSR was used in the implementation of the Harp File System [13].
- Zab [14] (ZooKeeper Atomic Broadcast) is a replication protocol used for the *ZooKeeper* [15] configuration service. ZooKeeper was designed at Yahoo! and is now a popular open source product distributed by Apache.

These protocols seem to rely on many of the same principles. Citations [16], [17] assert that Paxos and Viewstamped Replication are "the same algorithm independently invented," "equivalent," or that "the view management protocols seem to be equivalent" [3]. The Zab paper [14] says that "Zab follows the abstract description of Paxos by Lampson [18]."

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

We also give informal levels of abstraction in Fig. 1, ranging from a highly abstract specification of a linearizable service [19] to concrete, executable specifications. Active and passive replication are common approaches for replicating a service and ensuring that behaviors are still linearizable. Multi-Consensus 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 in Fig. 1, the same specification is derived by following different paths of such design decisions. There is a qualitative difference between refinements that cross abstraction boundaries in Fig. 1 and those that do not. When crossing an 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*. Within the same abstraction, a refinement *restricts* behaviors. For example, one specification might decide commands out of order whereas a more restricted specification might decide them in order.

Using refinements, we identify and analyze commonalities and differences between the replication protocols. The refinements also suggest new variants of these protocols, characterizing conditions required for a linearizable service. Other publications have used refinements to specify replication protocols [18], [20], [21]. We employ

• *R. van Renesse, N. Schiper, and F. B. Schneider are with the Department of Computer Science, Cornell University, Ithaca, NY, 14853.*

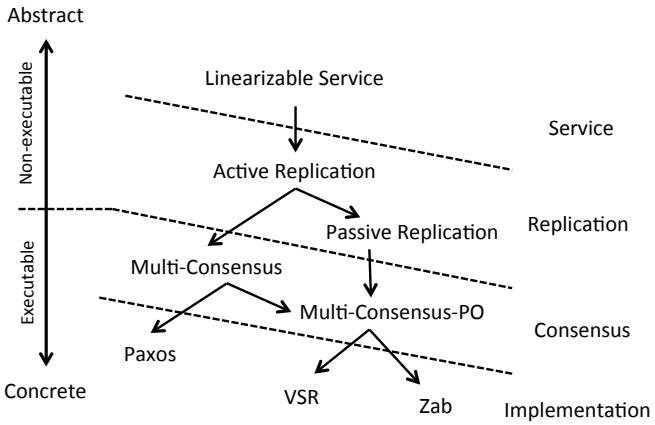


Fig. 1: An ordering of refinement mappings and informal levels of abstraction.

refinement for *comparing* Paxos, Viewstamped Replication, and Zab.

This paper is organized as follows. Section 2 introduces state transition specifications for linearizable replication as well as for active and passive replication. Section 3 presents Multi-Consensus, a canonical protocol that generalizes Paxos, VSR, and Zab: it forms a basis for comparison. *Progress summaries*, a new class of invariants, enable constructive reasoning about why and how these protocols work. We show how passive replication protocols refine Multi-Consensus by adding *prefix order* (*aka primary order*), creating Multi-Consensus-PO. Section 4 presents implementation details of Paxos, VSR, and Zab that constitute the final refinement steps to executable protocols. Section 5 discusses the implications of identified differences on performance. Section 6 gives a short overview of the history of concepts used in these replication protocols, and Section 7 is a conclusion. A proof that active replication refines the linearizable service is presented in Section 2.2, the details of the other refinements are available online at the Computer Society Digital Library [22].

2 Masking Failures

To improve the availability of a service, a common technique is replication. A *consistency criterion* defines expected responses to clients for concurrent operations. Ideally, the replication protocol ensures *linearizability* [19]—execution of concurrent client operations is equivalent to a sequential execution, where each operation is atomically performed at some point between its invocation and response.

2.1 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.

Specification 1 Linearizable Service

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

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

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

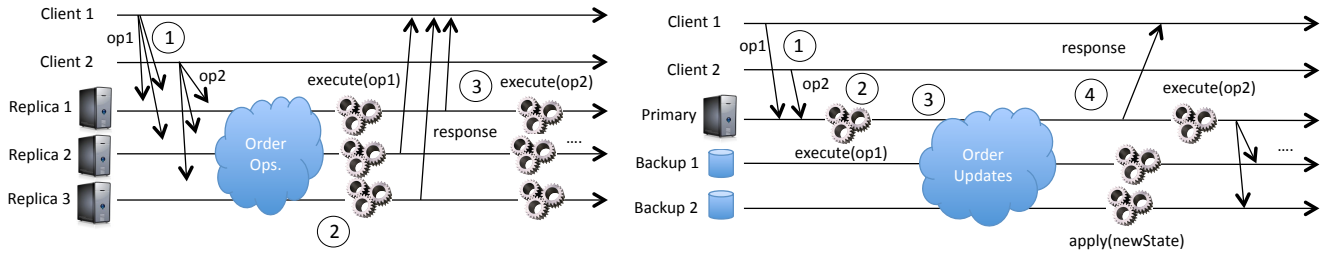
internal transition `execute`$(clt, op, result, newState)$:
precondition:
$(clt, op) \in inputs_\nu \wedge$
$(result, newState) = nextState(appState, (clt, op))$
action:
$appState := newState$
$outputs_\nu := outputs_\nu \cup \{((clt, op), result)\}$

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

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 is 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 variable is maintained by clt itself.
- $responded_{clt}$: the set of completed operations, also maintained by clt.

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 may not access internal variables. Internal transitions are performed by the service, and we demonstrate how this is done by implementing those transitions. The transitions of Specification Linearizable Service are:



(a) Active replication (b) Passive replication

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

- 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 (clt, op) to $inputs_\nu$ models clt sending a message containing op to the service. The client records 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 completed in $responded_{clt}$ to prevent this transition from being performed more than once per operation.

Specification Linearizable Service implies that a client cannot receive a response to an operation before invocation. However, the specification does allow each client operation to be executed an unbounded number of times. In an implementation, multiple executions could occur if the response to the client operation got lost by the network and the client retransmits its operation to the service. In practice, replicated services reduce or eliminate the probability that a client operation is executed more than once by keeping state about which operations have been executed. For example, a service could eliminate duplicate operations by attaching per-client sequence numbers to operations. In all of the specifications that follow, we omit logic for preventing operation duplication to simplify the presentation.

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 halts execution and later recovers and resumes execution is considered correct albeit, temporarily slow. A recovering process starts its execution exactly where it left it before halting, by relying on stable storage.
- **Failure Threshold**: Bound f defines the maximum number of replica processes that may fail; the number of client processes that may fail is unbounded.
- **Fairness**: Except for interface transitions at a crashed process, a transition that becomes continuously enabled is eventually executed.
- **Asynchrony**: There is no bound on the delay before a continuously enabled transition is executed.

We use refinement to show only that design decisions are safe under the assumption of Crash Failures. Intermediate specifications we derive do not necessarily guarantee liveness properties.[1] To show that the final refinements support liveness properties such as "an operation issued by a correct client is eventually executed", we require the existence of the Ω failure detector (see Section 4.2).

2.2 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* [4], [5], 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* [23], a primary replica runs a deterministic state machine, while backups only store copies of the primary replica's state. 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 a service implemented using active replication.

1) Clients submit operations to the service (`op1` and `op2` in Fig. 2a).
2) Replicas, starting out in the same state, execute 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

1. State transition specifications may include supplementary liveness conditions. If so, a specification Σ that refines a specification Σ' preserves both the *safety* and *liveness* properties of Σ'.

replica failures, message loss, and unpredictable delivery and processing delays. A fault-tolerant *consensus* protocol [24] is typically employed so that replicas will agree on the i^{th} operation for each i. 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 provided each instance of consensus eventually terminates.

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

1) Clients submit operations only to the primary.
2) The primary orders operations and computes new states and responses.
3) The primary forwards new states (so-called *state updates*) to each backup in the order generated.
4) The primary sends the response from an operation only after the corresponding state update has been successfully *decided* (this is made precise in Section 3).

Because two primaries may compete to have their state updates applied at the backups, replicas apply a state update u on the same state used by the primary to compute u. This is sometimes called the *prefix order* or *primary order* property [14].

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 corresponding to any sequential history of the two operations.

VSR and Zab employ passive replication; Paxos employs active replication. However, it is possible to implement one approach on the other. The Harp file system [13], 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. In doing so, the existing NFS servers do not have to be modified.

2.3 Refinement

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

2.3.1 Active Replication

We omit interface transitions `invoke`(clt, op) and `response`$(clt, op, result)$, which are the same as in Specification 1. Hereafter, a command cmd denotes a pair (clt, op).

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

Note that `propose`$(replica, slot, cmd)$ requires that $replica$ has not yet learned a decision for $slot$. While not necessary for safety, proposing a command for a slot that is known to be decided is wasted effort. It would make sense to require that replicas propose for the smallest value of $slot$ where both $proposals_{replica}[slot] = \varnothing$ and $learned_{replica}[slot] = \bot$. We do not require this only to simplify the refinement mapping between active and passive replication.

To show that active replication refines Specification 1, we first show how the internal state of Specification 1 is derived from the state of Specification 2. The internal state in Specification 1 is $appState$. For our refinement mapping, its value is the copy of the application state that is being maintained by the replica (or one of the replicas) with the highest version number.

To complete the refinement mapping, we also must show how transitions of active replication map onto enabled transitions of Specification 1, or onto *stutter steps* (no-ops with respect to Specification 1). The `propose`, `decide`, and `learn` transitions are always stutter steps, because they do not update $appState_{replica}$ of any replica. An `update`$(replica, cmd, res, newState)$ transition corresponds to `execute`$(clt, op, res, newState)$ in Specification 1, where $cmd = (clt, op)$ and $replica$ is the first replica to apply op. Transition `update` is a stutter step if the executing replica is not the first to apply the update.

2.3.2 Passive Replication

Passive replication (Specification 3) also uses slots, and proposals are tuples $(oldState, (cmd, res, newState))$ consisting of the state prior to executing a command, a command, the output of executing the command, and a new state that results from applying the command. In an actual implementation, the old state and new state would each be represented by an identifier along with a state update, rather than by the entire value of the state.

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

Transition `propose`$(replica, slot, cmd, res, newState)$ is performed when a primary $replica$ proposes applying cmd to $shadowState_{replica}$ for $slot$, resulting in output res.

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:
$cmd \in inputs_{\nu} \wedge learned_{replica}[slot] = \bot$
action:
$proposals_{replica}[slot] := proposals_{replica}[slot] \cup \{cmd\}$

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`$(replica, slot, cmd,$
$res, newState)$:
precondition:
$cmd \in inputs_{\nu} \wedge slot = shadowVersion_{replica} \wedge$
$learned_{replica}[slot] = \bot \wedge$
$(res, newState) = nextState(shadowState_{replica}, cmd)$
action:
$proposals_{replica}[slot] := proposals_{replica}[slot] \cup$
$\{(shadowState_{replica}, (cmd, res, newState))\}$
$shadowState_{replica} := newState$
$shadowVersion_{replica} := slot + 1$

internal transition `decide`$(slot, cmd, res, newState)$:
precondition:
$decisions[slot] = \bot \wedge$
$\exists r, s : (s, (cmd, res, newState)) \in proposals_{r}[slot] \wedge$
$(slot > 1 \Rightarrow decisions[slot - 1] = (-, -, s))$
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:
$outputs_{\nu} := outputs_{\nu} \cup \{(cmd, res)\}$
$appState_{replica} := newState$
$version_{replica} := version_{replica} + 1$

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

State $shadowState_{replica}$ is what the primary calculated for the previous slot (even though that state is not necessarily decided yet, and it may never be decided). Proposals for a slot are stored in a set, since a primary may propose to apply different commands for the same slot if there are repeated changes of primaries.

Transition `decide`$(slot, cmd, res, newState)$ specifies that only one of the proposed new states can be decided. Because cmd was performed speculatively, the `decide` transition checks that the state decided in the prior slot, if any, matches state s to which replica r applied cmd; prefix ordering is ensured.

Similar to active replication, transition `learn` models a replica learning the decision associated with 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; in passive replication, only the primary performs client operations and backups simply obtain the resulting states.

Replicas that are experiencing unexpected delay (e.g., due the crash suspicion of the primary by a failure detector) can start acting as primary by performing transition `resetShadow` to update their speculative state and version, respectively stored in variables $shadowState_{replica}$ and $shadowVersion_{replica}$. The new shadow state may itself be speculative; it must be a version at least as recent as the latest learned state.

3 A Generic Protocol

Specifications 2 and 3 contain internal variables and transitions that need to be refined for an executable implementation. We start by refining active replication. Multi-Consensus (Specification 4) refines active replication and contains no internal variables or transitions. As before, `invoke` and `response` transitions (and corresponding

Our term	Paxos [3]	VSR [10]	Zab [14]	meaning
replica	learner	cohort	server/observer	stores copy of application state
certifier	acceptor	cohort	server/participant	maintains consensus state
sequencer	leader	primary	leader	certifier that proposes orderings
round	ballot	view	epoch	round of certification
round-id	ballot number	view-id	epoch number	uniquely identifies a round
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 round
command	proposal	event record	transaction	a pair of a client id and an operation to be performed
round-stamp	N/A	viewstamp	zxid	uniquely identifies a sequence of proposals

TABLE 1: Translation between terms used in this paper and in the various replication protocols under consideration.

variables) have been omitted—they are the same as in Specification 1. Transitions `propose` and `update` of Specification 2 are omitted for the same reason. In this section we explain how and why Multi-Consensus works. Details of the refinement mappings are available online at the Computer Society Digital Library [22].

3.1 Certifiers and Rounds

Multi-Consensus 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 faulty processes, we require that $n \geq 2f + 1$ must hold.
- An unbounded number of *round*s.[2]

In each round, Multi-Consensus assigns to at most one certifier the role of *sequencer*. The sequencer of a round certifies at most one command for each slot. Other certifiers can imitate the sequencer, certifying the same command for the same slot and round; if two certifiers certify a command in the same slot and the same round, then it must be the same command. Moreover, a certifier cannot retract a certification. Once a majority of certifiers certify the command within a round, the command is *decided* (and because certifications cannot be retracted the command will remain decided thereafter). In Section 3.4 we show why two rounds cannot decide different commands for the same slot.

Each round has a *round-id* that uniquely identifies the round. Rounds are totally ordered by round-ids. A round is in one of three modes: *pending*, *operational*, or *wedged*. One round is the first round (it has the smallest round-id), and initially only that round is operational. Other rounds initially are pending. Two possible transitions on the mode of a round are:

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

This implies that at any time at most one round is operational and that wedged rounds can never become unwedged.

3.2 Tracking Progress

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

2. Table 1 translates between terms used in this paper and those found in the papers describing the protocols under consideration.

Progress Summaries: A *progress summary* is a pair $\langle rid, cmd \rangle$ where rid is the identifier of a round and cmd is a proposed command or $\perp$, satisfying:

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

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

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

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

3.3 Normal Case Processing

Each certifier $cert$ *supports* exactly one round-id rid_{cert}, initially 0. The *normal case* holds when a majority of certifiers support the same round-id, and one of these certifiers is sequencer (signified by $isSeq_{cert}$ being `true`).

Transition $\texttt{certifySeq}(cert, slot, \langle rid, cmd \rangle)$ is performed when sequencer $cert$ certifies command cmd for the given slot and round. The condition $progrsum_{cert}[slot] = \langle rid, \perp \rangle$ holds only if no command can be decided in this slot by a round with an id lower than rid_{cert}. The transition requires that $slot$ is the lowest empty slot of the sequencer. If the transition is performed, then $cert$ updates $progrsum_{cert}[slot]$ to reflect that a command decided in this round must be cmd. Sequencer $cert$ also notifies all other certifiers by adding $(cert, slot, \langle rid, cmd \rangle)$ to set $certifics_\nu$ (modeling a broadcast to the certifiers).

A certifier that receives such a message checks whether the message contains the same round-id that it is currently supporting and whether the progress summary in the message exceeds its own progress summary for the same slot. If so, then the certifier updates its own progress summary and certifies the proposed command (transition $\texttt{certify}(cert, slot, \langle rid, cmd \rangle)$).

Transition $\texttt{observeDecision}(replica, slot, cmd)$ at $replica$ is enabled if a majority of certifiers in the same

Specification 4 Multi-Consensus

var $rid_{cert}, isSeq_{cert}, progrsum_{cert}[1...], certifics_{\nu}, snapshots_{\nu}$

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

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

interface transition `certify`$(cert, slot, \langle rid, cmd \rangle)$:
precondition:
$\exists cert' : (cert', slot, \langle rid, cmd \rangle) \in certifics_{\nu} \wedge$
$rid_{cert} = rid \wedge \langle rid, cmd \rangle \succ progrsum_{cert}[slot]$
action:
$progrsum_{cert}[slot] := \langle rid, cmd \rangle$
$certifics_{\nu} := certifics_{\nu} \cup \{(cert, slot, \langle rid, cmd \rangle)\}$

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

interface transition `supportRound`$(cert, rid, proseq)$:
precondition:
$rid > rid_{cert}$
action:
$rid_{cert} := rid; isSeq_{cert} := \texttt{false}$
$snapshots_{\nu} :=$
$snapshots_{\nu} \cup \{(cert, rid, proseq, progrsum_{cert})\}$

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

round have certified cmd in $slot$. If so, then the command is decided and, as explained in the next section, all replicas that undergo the `observeDecision` transition for this slot will decide on the same command. Indeed, internal variable $decisions[slot]$ of Specification 2 is defined to be cmd if each certifier $cert$ in a majority have certified $(cert, slot, \langle rid, cmd \rangle)$ for some rid (and $\perp$ if no such rid exists). The details of the refinement mapping are available at the Computer Society Digital Library [22].

3.4 Recovery

When an operational round is no longer certifying proposals, perhaps because its sequencer has crashed or is slow, the round can be wedged and a round with a higher round-id can become operational. A certifier $cert$ transitions to supporting a new round-id rid and prospective sequencer $proseq$ (transition `supportRound`$(cert, rid, proseq)$). This transition only increases rid_{cert}. The transition sends the certifier's *snapshot* by adding it to the set $snapshots_{\nu}$. A snapshot is a four-tuple $(cert, rid, proseq, progrsum_{cert})$ containing the certifier's identifier, its current round-id, the identifier of $proseq$, and the certifier's list of progress summaries. Note that a certifier can send at most one snapshot for each round.

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

Transition `recover`$(cert, rid, \mathcal{S})$ is enabled at $cert$ if $\mathcal{S}$ contains snapshots for rid and sequencer $cert$ from a majority of certifiers. The sequencer helps ensure that the round does not decide commands inconsistent with prior rounds. For each slot, sequencer $cert$ determines the maximum progress summary $\langle r, cmd \rangle$ for the slot in the snapshots contained in $\mathcal{S}$. It then sets its own progress summary for the slot to $\langle rid, cmd \rangle$. It is easy to see that $rid \geq r$.

We argue that $\langle rid, cmd \rangle$ satisfies the definition of progress summary in Section 3.2. All certifiers in $\mathcal{S}$ support rid and form a majority. Thus, it is not possible for any round between r and rid to decide a command, because none of these certifiers can certify a command in those rounds. There are two cases:

- If $cmd = \perp$, then no command can be decided before r, so no command can be decided before rid. Hence, $\langle rid, \perp \rangle$ is a correct progress summary.
- If $cmd \neq \perp$, then a command decided by r or a round prior to r must be cmd. Since no command can be decided by rounds between r and rid, progress summary $\langle rid, cmd \rangle$ is correct.

The sequencer sets $isSeq_{cert}$ flag upon recovery. As a result, the sequencer can propose new commands. The normal case for the round begins and holds as long as a majority of certifiers support the corresponding round-id.

3.5 Passive Replication

Specification 4 does not satisfy prefix ordering (Section 2.2) because any proposal can be decided in a slot, including a slot that does not correspond to a state decided in the prior slot. Thus Multi-Consensus does not refine Passive Replication. One way of implementing prefix ordering would be for the primary to delay

proposing a command for a slot until it knows decisions for all prior slots. But that would be slow.

A better solution is to refine Multi-Consensus and obtain a specification that also refines Passive Replication as well as satisfying prefix ordering. We call this specification Multi-Consensus-PO. Multi-Consensus-PO guarantees that each decision is the result of an operation applied to the state decided in the prior slot (except for the first slot). We complete the refinement by adding two preconditions:[3]

(i) In Multi-Consensus-PO, slots have to be decided in sequence. To guarantee this, commands are certified in order by adding the following precondition to transition `certify`: $slot > 1 \Rightarrow \exists c \neq \bot : progrsum_{cert}[slot - 1] = \langle rid, c \rangle$. Thus, if in some round there exists a majority of certifiers that have certified a command in $slot$, then there also exists a majority of certifiers that have certified a command in the prior slot.

(ii) To guarantee that a decision in $slot$ is based on the state decided in the prior slot, we add the following precondition to transition `certifySeq`:
$slot > 1 \Rightarrow \exists s :$
$cmd = (s, -) \wedge$
$progrsum_{cert}[slot - 1] = \langle rid, (-, (-, -, s)) \rangle$.
This works due to properties of progress summaries: if a command has been or will be decided in round rid or a prior round for $slot - 1$, then it is the command in $progrsum_{cert}[slot - 1]$. Therefore, if the sequencer's proposal for $slot$ is decided in round rid, the proposal will be based on the state decided in $slot - 1$. If the primary and the sequencer are co-located, as they usually are, this transition precondition is satisfied automatically, because the primary computes states in order.

Multi-Consensus-PO inherits transitions `invoke` and `response` from Specification 1 as well as transitions `propose`, `update`, and `resetShadow` from Specification 3. The variables contained in these transitions are inherited as well.

Passive replication protocols 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 message and overwrite their own snapshot with its contents before they certify new commands in this round. As a result, these $progrsum_{cert}$ slots have the same round identifier at each certifier and can thus be maintained as a separate variable.

3. These two preconditions, combined with the condition that $isSeq_{cert}$ must hold in transition `certifySeq`, serve as the primary order and integrity properties defined in [25] to implement primary-backup replication. The condition $isSeq_{cert}$ is equivalent to the barrier function τ_{Paxos} used in the same paper to implement primary-backup on top of a *white-box* consensus service.

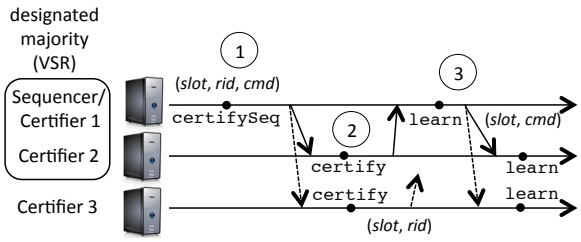


Fig. 3: Normal case processing at three certifiers. Dots indicate transitions, and arrows between certifiers are messages.

4 Implementation

Specifications Multi-Consensus and Multi-Consensus-PO do not contain internal variables or transitions. However, they only specify which transitions may be performed, not when transitions should be performed. We discuss final refinements of these specifications that produce Paxos, VSR, and Zab.

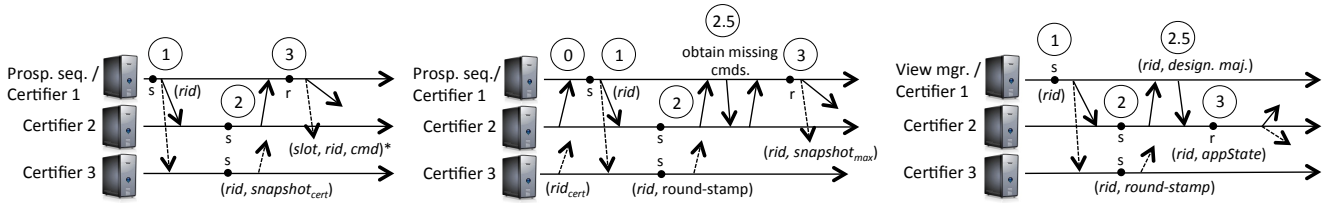
4.1 Normal Case

We first turn to implementing state and transitions of Multi-Consensus and Multi-Consensus-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, this state is 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. This state can be implemented as ordinary program variables. In Zab, $progrsum_{cert}$ is implemented by a queue of commands. In VSR, $progrsum_{cert}$ is replaced by the application state and a counter that records the number of updates made to the state in this round. In Paxos, a progress summary is simply a pair consisting of a round identifier and a command.

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

1) Sequencer $cert$ proposes a command for the next open slot and sends a message to the other certifiers (maps to `certifySeq`$(cert, slot, \langle rid, cmd \rangle)$). In VSR and Zab, the command is a state update that results from executing the client operation; in Paxos, the command is the operation itself.
2) Upon receipt by a certifier $cert$, if $cert$ supports the round-id rid in the message, then $cert$ updates its slot and replies to the sequencer (transition `certify`$(cert, slot, \langle rid, cmd \rangle)$). If not, the message can be ignored. With VSR and Zab, prefix-ordering must be ensured, and $cert$ only replies to the sequencer if its progress summary for $slot - 1$ contains a non-empty command for rid.
3) If the sequencer receives successful responses from a majority of certifiers (transition `observeDecision`),



(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 `supportRound` and a `recover` transition. Messages of the form (x, y)* contain multiple (x,y) tuples.

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).

The various protocols reflect other different design decisions:

- In VSR, a specific majority of certifiers is determined a priori and fixed for each round. 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 could be updated before some proposed command is decided, so if another command is decided then 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.
- Some versions of Paxos use leases [26], [3] for read-only operations. Leases have the advantage that read-only operations can be served at a single replica while still guaranteeing linearizability. This requires synchronized clocks (or clocks with bounded drift), and the sequencer obtains 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.
- ZooKeeper, which is built upon Zab, offers the option to use leasing or to have any replica handle read-only operations individually, circumventing Zab. The latter is efficient, but a replica might not have learned the latest decided proposals so its clients can receive results based on stale state (such reads satisfy sequential consistency [27]).

For replicas to learn about decisions, two options exist:

- Certifiers can respond 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.
- 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.

4.2 Recovery

Fig. 4 illustrates the recovery steps in the protocols.

With Paxos, a certifier $proseq$ that notices a lack of progress (typically the certifier designated by a weak leader election protocol Ω [28]) may start the following process to try to become sequencer itself (see Fig. 4a):

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

Prospective sequencers in Zab are designated by Ω as well. When Ω determines that a sequencer has become unresponsive, it initiates a protocol (see Fig. 4b) in which a new sequencer is elected:

- Step 0: Ω proposes a prospective sequencer $proseq$ and notifies the certifiers. Upon receipt, a certifier sends a message containing the round-id it supports to $proseq$.
- Step 1: Upon receiving such messages from a majority of certifiers, prospective sequencer $proseq$ selects a round-id rid that is one larger than the maximum it received, transitions to supporting it (transition $\texttt{supportRound}(proseq, rid, proseq)$), and broadcasts this to the other certifiers for approval.
- Step 2: Upon receipt, if certifier $cert$ can support rid and has not agreed to a certifier other than $proseq$ becoming sequencer of the round, then it performs transition $\texttt{supportRound}(cert, rid, proseq)$. Zab exploits prefix ordering to optimize the recovery protocol. Instead of sending its entire snapshot to the prospective sequencer, a certifier that transitions to supporting round rid sends a *round-stamp*. A round-stamp is a

lexicographically ordered pair consisting of the round-id in the snapshot (the same for all slots) and the number of slots in the round for which it has certified a command.

- Step 2.5: Once $proseq$ receives responses from a majority, it computes the maximum round-stamp and determines if commands are missing and retrieves them from the certifier $cert_{max}$ with the highest received round-stamp. If $proseq$ is missing too many commands (e.g. if $proseq$ did not participate in the last round $cert_{max}$ participated in), $cert_{max}$ sends its entire snapshot to $proseq$.
- Step 3: After receiving the missing commands, $proseq$ broadcasts its snapshot. In practice the snapshot is a checkpoint of its state with a sequence of state updates, to the certifiers (transition $\texttt{recover}(proseq, rid, \mathcal{S})$). Certifiers acknowledge receipt of this snapshot and, upon receiving acknowledgments from a majority, $proseq$ learns that it is now the sequencer of rid and broadcasts a *commit* message before resuming the normal case protocol (not shown in the picture).

In VSR, each round-id has a pre-assigned *view manager* v that is not necessarily the sequencer. A round-id is a lexicographically ordered pair comprising a number and the process identifier of the view manager.

The view manager v of round-id starts the following recovery procedure if the protocol seems stalled (see Fig. 4c):

- Step 1: v starts supporting round rid (transition $\texttt{supportRound}(v, rid, v)$), and it queries at least a majority of certifiers.
- Step 2: Upon receipt of such a query, a certifier $cert$ starts supporting rid (transition $\texttt{supportRound}(cert, rid, v)$). Similar to Zab, $cert$ sends its round-stamp to v.
- Step 2.5: Upon receiving round-stamps from a majority of certifiers, view manager v uses the set of certifiers that responded as the designated majority for the round and assigns the sequencer role to the certifier p that reported the highest round-stamp. The view manager then notifies certifier p, requesting it to become sequencer.
- Step 3: Sequencer p, having the latest state, broadcasts its snapshot (transition $\texttt{recover}(p, rid, \mathcal{S})$). In the case of VSR, the state that the new sequencer sends is its application state rather than a snapshot.

4.3 Garbage Collection

Multi-Consensus has each certifier accumulating state about all slots, which does not scale. For Paxos, the issue has received attention in [7], [29]. In VSR, no garbage collection is required. Certifiers and replicas are co-located, and they only store the most recent round-id they adopted; application state is updated upon certification of a command. During recovery, the sequencer simply sends application state to the replicas and, consequently, any decided commands need not be replayed.

4.4 Liveness

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

To guarantee progress, some round with a correct sequencer must eventually not get preempted by a higher round [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 rounds [29].

In this section, we look at how the various protocols optimize progress.

4.4.1 Partial Memory Loss

If certifiers keep 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, then as long as a majority eventually recovers their state from before the crash, the service can continue operating.

4.4.2 Total Memory Loss

In [7, §5.1], 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. This optimization is flawed and it breaks the invariant that a certifier's round-id can only increase over time (confirmed by the authors of [7]). 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 [3] shows how the set of certifiers can be reconfigured to tolerate total memory loss, and this has been worked out in greater detail in Microsoft's SMART project [33] and later for Viewstamped Replication as well [11]. Zab also supports reconfiguration [34].

5 Discussion

Table 2 summarizes some differences between Paxos, VSR, and Zab. These differences demonstrate that the protocols do not refine one another; they also have pragmatic consequences, as discussed below. The comparisons are based on published algorithms; actual implementations may vary. We organize the discussion around normal case processing and recovery overheads.

What	Section	Paxos	VSR	Zab
replication style	2.2	active	passive	passive
read-only operations	4.1	leasing	certification	read any replica/leasing
designated majority	4.1, 4.2	no	yes	no
time of execution	4.1	upon decision	upon certification	depends on role
sequencer selection	4.2	majority vote or deterministic	view manager assigned	majority vote
recovery direction	4.2	two-way	from sequencer	two-way/from sequencer
recovery granularity	4.2	slot-at-a-time	application state	command prefix
tolerates memory loss	4.4.1, 4.4.2	reconfigure	partial	reconfigure

TABLE 2: Overview of important differences between the various protocols.

5.1 Normal Case

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

5.1.0.2 Read-only Optimizations: Paxos and Zookeeper support leasing for read-only operations, but there is no reason why leasing could not be added to VSR. Indeed, Harp (built on VSR) uses leasing. A lease improves latency of read-only operations in the normal case, but it delays recovery in case of a failure. ZooKeeper offers the option whether to use leases. Without leases, ZooKeeper clients read any replica at any time. Doing so may cause replicas to read stale state.

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

5.1.0.4 Time of Command Execution: In VSR, replicas apply state updates speculatively when they are certified, possibly before they are decided. Commands are forgotten as soon as they are applied to the state. So no garbage collection is necessary. A disadvantage is that 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 must update its state and compute a response, because if the replica fails then another deterministic replica is guaranteed to compute the same response. At the time of failing each command that led to this state and response has been certified by a majority and, therefore, the state and response are recoverable.

In Zab, the primary also speculatively applies client operations to compute state updates before these updates are decided. Replicas apply state updates after those updates have been decided. In Paxos, there is no speculative execution.

5.2 Recovery

5.2.0.1 Sequencer Selection: VSR selects a sequencer that has the most up-to-date state (taking advantage of prefix ordering), so VSR does not have to recover state from the other certifiers, which simplifies and streamlines recovery.

5.2.0.2 Recovery Direction: Paxos allows the prospective sequencer to recover state of previous slots and, at the same time, propose new commands for slots where it has already retrieved sufficient state. However, all certifiers must send their certification state to the prospective sequencer before it can re-propose commands for slots. With VSR, the sequencer is the certifier with the highest round-stamp, and it does not need to recover state from the other certifiers. A similar optimization is sketched in the description of the Zab protocol (and implemented in the Zookeeper service).

With Paxos and Zab, garbage collection of the certification state is important for ensuring that the amount of state exchanged on recovery does not become too large. A recovering replica can be brought up-to-date faster by replaying decided commands that were missed rather than by copying state.

With VSR, the selected sequencer pushes its snapshot to the other certifiers. The snapshot has to be transferred and processed prior to processing new certification requests, possibly resulting in a performance interruption.

5.2.0.3 Recovery Granularity: In VSR, the entire application state is sent from the sequencer to the backups. For VSR, this state is transaction manager state and is small, but this approach does not scale. However, in some cases that cost is unavoidable. For example, if a replica has a disk failure, then replaying all commands from time 0 is not scalable either, and the recovering replica instead will have to seed its state from another one. In this case, the replica will load a checkpoint and then replay missing commands to bring the checkpoint up-to-date—a technique used in Zab (and 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 overhead of executing operations and the size of state update messages, one or the other approach may perform better.

5.2.0.4 Tolerating Memory Loss: An option suggested by VSR is to retain on disk only a round-id; the remaining state is kept in memory. This technique works only in restricted situations, where at least one certifier has the most up-to-date state in memory.

6 A Bit of History

We believe the first consensus protocol to use rounds and sequencers is due to Dwork, Lynch, and Stockmeyer (DLS) [30]. Rounds in DLS are countable and round $b + 1$ cannot start until round b has run its course. Thus, DLS does not refine Multi-Consensus(even though Multi-Consensus does use rounds and sequencers).

Chandra and Toueg's work on consensus [31] formalized conditions under which consensus protocols terminate. Their consensus protocol resembles the Paxos single-decree Synod protocol, and it refines Multi-Consensus.

To the best of our knowledge, majority intersection to avoid potential inconsistencies first appears in Thomas [35]. Quorum replication [35] supports only storage objects with `read` and `write` operations (or, equivalently, `get` and `put` operations in the case of a Key-Value Store).

7 Conclusion

Paxos, VSR, and Zab are three well-known replication protocols for asynchronous environments that admit bounded numbers of crash failures. This paper describes a specification for Multi-Consensus, a generic specification that contains important design features that the three protocols share. The features include an unbounded number of totally ordered rounds, a static set of certifiers, and at most one sequencer per round.

The three protocols differ in how they refine Multi-Consensus. We disentangled fundamentally different design decisions in the three protocols and considered impact on performance. 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.

Acknowledgments

We are grateful for the anonymous reviews and discussions with Flavio Junqueira and Marco Serafini. The authors are supported in part by AFOSR grants FA2386-12-1-3008, F9550-06-0019, by the AFOSR MURI "Science of Cyber Security: Modeling, Composition, and Measurement" as AFOSR grant FA9550-11-1-0137, by NSF grants 0430161, 0964409, 1040689, 1047540, and CCF-0424422 (TRUST), by ONR grants N00014-01-1-0968 and N00014-09-1-0652, by DARPA grants FA8750-10-2-0238 and FA8750-11-2-0256, by DOE ARPA-e grant DE-AR0000230, by MDCN/iAd grant 54083, and by grants from Microsoft Corporation, Facebook Inc., and Amazon.com.

References

[1] L. Lamport, "Specifying concurrent program modules," *Trans. on Programming Languages and Systems*, vol. 5, no. 2, pp. 190–222, Apr. 1983.
[2] 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.
[3] L. Lamport, "The part-time parliament," *Trans. on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.
[4] ——, "Time, clocks, and the ordering of events in a distributed system," *CACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.
[5] 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.
[6] M. Burrows, "The Chubby Lock Service for loosely-coupled distributed systems," in *7th Symposium on Operating System Design and Implementation*, Seattle, WA, Nov. 2006.
[7] 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.
[8] M. Isard, "Autopilot: Automatic data center management," *Operating Systems Review*, vol. 41, no. 2, pp. 60–67, Apr. 2007.
[9] 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.
[10] 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.
[11] B. Liskov and J. Cowling, "Viewstamped Replication revisited," MIT, Tech. Rep. MIT-CSAIL-TR-2012-021, Jul. 2012.
[12] B. Lampson and H. Sturgis, "Crash recovery in a distributed data storage system," Xerox PARC, Palo Alto, CA, Tech. Rep., 1976.
[13] 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.
[14] 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.
[15] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," in *USENIX Annual Technology Conference*, 2010.
[16] 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.
[17] C. Cachin, "Yet another visit to Paxos," IBM Research, Zurich, Switzerland, Tech. Rep. RZ3754, 2009.
[18] B. Lampson, "The ABCDs of Paxos," in *Proc. of the 20th ACM Symp. on Principles of Distributed Computing*. Newport, RI: ACM Press, 2001.
[19] 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.
[20] J. Aizikowitz, "Designing distributed services using refinement mappings," Ph.D. dissertation, Cornell University, 1990.
[21] L. Lamport, "Byzantizing Paxos by refinement," in *Proceedings of the 25th International Conference on Distributed Computing*, ser. DISC'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 211–224.
[22] "http://www.computer.org/csdl/trans/tq/index.html."

[23] 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.
[24] 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.
[25] F. P. Junqueira and M. Serafini, "On barriers and the gap between active and passive replication," in *Distributed Computing*. Springer, 2013, vol. 8205, pp. 299–313.
[26] 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.
[27] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979.
[28] 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.
[29] 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.
[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. 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.
[34] A. Shraer, B. Reed, D. Malkhi, and F. Junqueira, "Dynamic reconfiguration of primary/backup clusters," in *Annual Technical Conference (ATC'12)*. USENIX, Jun. 2012.
[35] 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.

Robbert van Renesse is a Principal Research Scientist in the Department of Computer Science at Cornell University. He received a Ph.D. from the Vrije Universiteit in Amsterdam in 1989. After working at AT&T Bell Labs in Murray Hill he joined Cornell in 1991. He was associate editor of IEEE Transactions on Parallel and Distributed Systems from 1997 to 1999, and he is currently associate editor for ACM Computing Surveys. His research interests include the fault tolerance and scalability of distributed systems. Van Renesse is an ACM Fellow.

Nicolas Schiper is a postdoctoral associate at the Computer Science Department at Cornell University, USA. He obtained his Ph.D. degree in 2009 from the University of Lugano, Switzerland. His research lies at the intersection of the theory and practice of distributed systems. Nicolas Schiper is particularly interested in scalability and energy efficiency aspects of fault-tolerance.

Fred B. Schneider is Samuel B. Eckert Professor of Computer Science at Cornell University. Schneider is a fellow of AAAS (1992), ACM (1995), IEEE (2008), a member of NAE (2011) and a foreign member of its Norwegean equivalent NKTV (2012). He was named Professor-at-Large at the University of Tromso (Norway) in 1996 and was awarded a Doctor of Science honoris causa by the University of NewCastle-upon-Tyne in 2003 for his work in computer dependability and security. He received the 2012 IEEE Emanuel R. Piore Award.