Gryff: Unifying Consensus and Shared Registers

Matthew Burke
Cornell University

Audrey Cheng
Princeton University

Wyatt Lloyd
Princeton University

Abstract

Linearizability reduces the complexity of building correct applications. However, there is a tradeoff between using linearizability for geo-replicated storage and low tail latency. Traditional approaches use consensus to implement linearizable replicated state machines, but consensus is inefficient for workloads composed mostly of reads and writes.

We present the design, implementation, and evaluation of Gryff, a system that offers linearizability and low tail latency by unifying consensus with shared registers. Gryff introduces carstamps to correctly order reads and writes without incurring unnecessary constraints that are required when ordering stronger synchronization primitives. Our evaluation shows that Gryff's combination of an optimized shared register protocol with EPaxos allows it to provide lower service-level latency than EPaxos or MultiPaxos due to its much lower tail latency for reads.

1 Introduction

Large-scale web applications rely on replication to provide fault-tolerant storage. Increasingly, developers are turning to linearizable [32] storage systems because they reduce the complexity of implementing correct applications [2, 13, 17]. Recent systems from both academia [27, 35, 40, 52, 53, 57] and industry [6, 11, 14, 17, 23] demonstrate this trend.

Traditionally, linearizable storage systems for geo-replicated settings are built using state machine replication via consensus [33, 36, 37, 38, 45, 47, 50, 51]. These protocols are safe under the asynchronous network conditions that are common in wide-area networks. Furthermore, they provide the abstraction of a shared command log, which allows for the implementation of arbitrary deterministic state machines. Strong synchronization primitives, such as read-modify-write operations (rmws), can thus be used in applications built on top of these systems, further easing the programming burden on developers.

Linearizability for geo-replicated storage, however, comes with a tradeoff between strong guarantees and low latency. At least one communication delay between replicas is necessary to maintain a legal total order of operations [41], and in the wide-area, this communication incurs a considerable latency cost even in the best case. The tradeoff is starker for tail latency, where adverse conditions such as network delays, slow or failed replicas, and concurrent operations further delay responses to clients.

Tail latency is of particular importance for large-scale web applications, where end-user requests for high-level application objects fan-out into hundreds of sub-requests to storage services [18]. For example, when a user loads a page in a social networking service, an application server typically needs to invoke and wait for the completion of dozens of requests to replicas before returning the page to the client [2]. Only once the client receives the page can it begin loading additional assets and rendering the page. Thus, the median latency experienced by the end-user depends on the maximum of tens or hundreds of operations, which is dictated by the tail of the latency distribution.

Consensus protocols demonstrate the tradeoff between strong guarantees and low tail latency. Fundamentally, no protocol can solve consensus and guarantee termination in an asynchronous system with failures [24]. In practice, this impossibility result manifests as performance inefficiencies, such as serializing operations through a designated leader or delaying concurrent operations. In geo-replicated settings at scale, these inefficiencies impact tail latency.

In contrast, shared register protocols can implement linearizable shared registers, which support simple reads and writes, and guarantee termination in asynchronous systems with failures [5]. This translates to favorable tail latency for real protocols: shared register protocols are typically leaderless and often do not delay reads or writes, even if there are concurrent operations. The reads and writes provided by shared registers are the dominant types of operations in large-scale web applications [9]. Yet, shared registers are fundamentally too weak to directly implement strong synchronization primitives like rmws [31]. To resolve this tradeoff, the solution is to combine the strong synchronization provided by consensus with the favorable read/write tail latency of shared registers in a single protocol.

The idea of unifying consensus and shared registers is not new [8]. However, the only previous attempt of which we are aware is incorrect because it does not safely handle certain interleavings of operations. Our key insight is that protocol-level mechanisms for enforcing the interaction between rmws and reads/writes are difficult to reason about, which can lead to subtle safety violations. Instead, we argue the interaction be enforced at a deeper level, in the ordering mechanism itself, to simplify reasoning about correctness.

We introduce consensus-after-register timestamps, or

carstamps, a novel ordering mechanism for distributed storage to leverage this insight. Carstamps allow writes and rmws to concurrently modify the same state without serializing through a leader or incurring additional round trips. Reads use carstamps to determine consistent values without interposing on concurrent updates.

Gryff is our system that implements this ordering mechanism to achieve unification.[1] It is the first such system to be proven correct, implemented, and empirically evaluated. Gryff combines a multi-writer variant [43] of the ABD [5] protocol for reads and writes with EPaxos [47] for rmws. In addition to the challenges associated with unifying these protocols, we introduce an optimization to further rein in tail latency by reducing the frequency of reads taking multiple wide-area round trips.

We implemented Gryff in the same framework as EPaxos [47] and MultiPaxos [36] and evaluated its performance in a geo-replicated setting. Our evaluation shows that Gryff reduces the tradeoff between linearizability and low tail latency for workloads representative of large-scale web applications [10, 16, 17]. For moderate contention workloads, Gryff reduces p99 read latency to $\sim$56% of EPaxos, but has $\sim 2x$ higher write latency. This tradeoff allows Gryff to reduce service-level p50 latency to $\sim$60% of EPaxos for large-scale web applications whose requests fan-out into many storage-level requests.

In summary, the contributions of this paper include:

- A novel ordering mechanism, carstamps, that enables efficient unification of consensus with shared registers. (§3)
- The Gryff design that combines a shared register protocol with EPaxos to provide reads, writes, and rmws. (§4, §5)
- The implementation and evaluation of Gryff, which demonstrates its latency improvements. (§6)

2 Consensus vs. Shared Registers

This section covers preliminaries and then compares and contrasts consensus and shared register protocols. It looks at the interfaces they support, the ordering constraints they impose, and the ordering mechanisms they use.

Model and Preliminaries. We study systems comprised of a set P of m processes that communicate with each other over point-to-point message channels. Processes may fail according to the *crash failure model*: a failed process ceases executing instructions and its failure is not detectable by other processes. The system is *asynchronous* such that there is no upper bound on the time it takes for a message to be delivered and there is no bound on the relative speeds at which processes execute instructions.

Linearizability is a correctness condition for a concurrent object that requires (a) operations invoked by processes accessing the object appear to execute in some total order that is consistent with the semantics of the object (i.e., that is *legal*) and (b) the total order is consistent with the order that operations happened in real time [32]. Linearizability is a *local* property, meaning it holds for a collection of objects if and only if it holds for each individual object.

For the remainder of this text, we consider linearizable replication of a single object by omitting object identifiers; it is straightforward to compose instances of such a system to obtain a linearizable multi-object system.

2.1 State Machines and Consensus

State machine replication is the canonical approach to implementing fault-tolerant services [56]. It provides a fault-tolerant *state machine* that exposes the following interface:

- COMMAND$(c(\cdot))$: atomically applies a deterministic computation $c(\cdot)$ to the state machine and returns any outputs

Each command can include zero or more arguments, read local state, perform deterministic computation, and produce output. The state machine approach applies these commands one by one starting from the same initial state to move replicas through identical states. Thus, if some replicas fail, the remaining replicas still have the state and can continue to provide the service.

Applying commands in the same order on all replicas requires an ordering mechanism that is *stable*, i.e., a replica knows when a command's position is fixed and it will never receive an earlier command [56]. In asynchronous systems where processes can fail, consensus protocols [33, 36, 37, 38, 45, 47, 50, 51] are used to agree on this stable ordering.

Figure 1a shows the stable ordering provided by consensus protocols for state machine replication. Commands are assigned positions in a log and a command becomes stable once there are no empty slots preceding its own in the log.

2.2 Shared Registers and Their Protocols

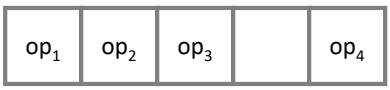
A *shared register* has the following interface:

- READ(): returns the value of the register
- WRITE(v): updates the value of the register to v

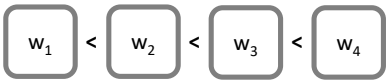
Shared registers provide a simple interface with read and write operations. They are less general than state machines as they provably cannot be used to implement consensus [31]. Shared register protocols replicate shared registers across multiple processes for fault tolerance [5, 22, 43].

Shared register protocols provide a linearizable ordering of operations. That ordering does not have to be stable, however, because each write operation fully defines the state of the object. Thus, a replica can safely apply a write w_4 even if it does not know about earlier writes. If an earlier write w_3 ever does arrive, the replica simply ignores that write because it already has the resulting state from applying w_3 and then w_4. Figure 1b shows shared register ordering where there is a total order of all writes (denoted by $<$) without stability.

[1] A gryffin is a mythological hybrid creature that combines the power of a lion with the speed of an eagle.



(a) Ordering in consensus protocols. Operations op_1, op_2, and op_3 are stable, but op_4 is not.



(b) Ordering in shared register protocols. No writes are stable.

Figure 1: Comparison of ordering in consensus and shared register protocols. Shared register protocols provide an unstable ordering where new writes can be inserted between writes that have already completed.

2.3 Shared Objects and Their Ordering

A *shared object* exposes the following interface:

- READ(): returns the value of the object
- WRITE(v): updates the value of the object to v
- RMW($f(\cdot)$): atomically reads the value v, updates the value to $f(v)$, and returns v

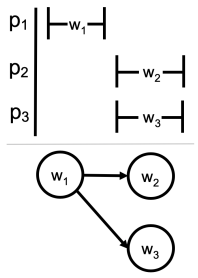
The abstraction of a shared object captures an intuitive programming model that is used in real-world systems [12, 15, 23, 44, 54, 55]. Most operations read or write data, but rmws support stronger primitives to synchronize concurrent accesses to data. For example, a conditional write can be implemented with a rmw by using a function $f(\cdot)$ that returns the new value to be written only if some condition is met.

Shared objects and state machines are equivalent in that an instance of one can be used to implement the other [31]. However, the difference is that shared objects expose a more restrictive interface for directly reading and writing state, as do shared registers. These simpler operations can be implemented more efficiently because their semantics impose fewer ordering constraints.

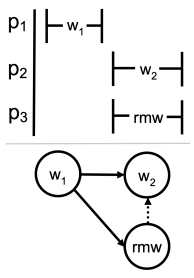
Yet, neither the stable ordering of state machine replication nor the unstable total ordering of shared register protocols is a good fit for shared objects. A stable order, on the one hand, over constrains how reads and writes are ordered and results in less efficient protocols. On the other hand, an unstable total order under constrains how rmws are ordered and results in an incorrect protocol.

Figure 2 demonstrates these different constraints. Consider the execution in Figure 2a where two processes, p_2 and p_3, write concurrently. Linearizability stipulates that w_2 and w_3 be ordered after w_1 because they are invoked after w_1 completes in real time. However, there is no stipulation for how w_2 and w_3 are ordered with respect to each other because the result of a write does not depend on preceding operations. Both $w_1 \rightarrow w_2 \rightarrow w_3$ and $w_1 \rightarrow w_3 \rightarrow w_2$ are valid.

Now consider the execution in Figure 2b involving a rmw. Process p_2 writes while p_3 concurrently executes a rmw. The *base update* of a rmw is the operation that writes the value that the rmw reads. Assume that w_1 is the base update of



(a) w_2 and w_3 may be arbitrarily ordered.



(b) if *rmw* reads w_1, it must be before w_2.

Figure 2: Solid arrows are real time ordering constraints. Dashed arrows are operation semantic constraints.

rmw. Then, not only does *rmw* need to be ordered after w_1, but no other write may be ordered between w_1 and *rmw*. This additional constraint ensures legality because the semantics of a rmw requires that it must appear to atomically read and update the object based on the value read. Thus, only $w_1 \rightarrow rmw \rightarrow w_2$ is a valid order.

3 Carstamps for Correct Ordering

Consensus-after-register timestamps, or *carstamps*, precisely capture the ordering constraints of shared objects. They provide the necessary stable order for rmws and the more efficient unstable order for reads and writes. This section describes the requirements of a precise ordering mechanism for shared objects and then describes carstamps.

3.1 Precise Ordering for Shared Objects

An ordering mechanism is an injective function $g : X \rightarrow Y$ from a set X of writes and rmws to a totally ordered set $(Y, <_Y)$. A mechanism g produces a total order $<_g$ on X: for all $x_1, x_2 \in X$, $x_1 <_g x_2$ if and only if $g(x_1) <_Y g(x_2)$.

Typically, replication protocols augment an ordering mechanism with protocol-level logic to enforce real time and legality constraints on the total order given by the ordering mechanism to provide linearizability. While the logic for enforcing real time constraints is often straightforward, legality constraints can be more complex.

Protocol-level Legality. For example, consider the Active Quorum Systems (AQS) protocol [7, 8]. AQS is the only prior protocol of which we are aware that attempts to combine consensus and shared registers and it does so with an unstable ordering mechanism. This allows for executions where a rmw *rmw* with base update u is ordered such that there exists a $y \in Y$ with $g(u) <_Y y <_Y g(rmw)$. This can result in an illegal total order when a write w is concurrent with *rmw* because w may be assigned $g(w) = y$. AQS contains no logic at the protocol-level to prevent this subtle scenario. We discuss such an execution in detail in Appendix C and describe how there does not exist a linearizable order of all operations.

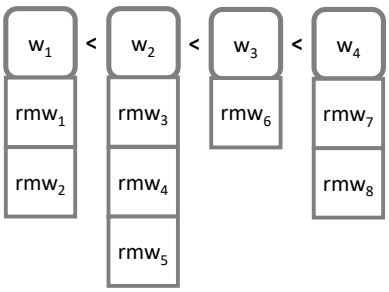


Figure 3: Unified ordering provided by carstamps for writes and rmws. Writes are unstably ordered while rmws are stably ordered with their base updates.

Ordering-level Legality. Our key insight is that the legality constraints of linearizability can be encoded in the ordering mechanism itself. An ordering mechanism that does this must ensure that for all $rmw \in X$ such that u is the base update of rmw, $g(u) <_Y g(rmw)$ and $g(u)$ is a *cover* of $g(rmw)$. This means that there is no $y \in Y$ such that $g(u) <_Y y <_Y g(rmw)$. With such an ordering mechanism, there is no need for protocol-level logic to prevent other writes in X from being assigned an illegal position in the total order between $g(u)$ and $g(rmw)$.

3.2 Carstamps

Our solution which leverages this insight is called *carstamp*s. A carstamp is a triple $cs = (ts, id, rmwc)$ with three fields: a logical timestamp *ts*, a process identifier *id*, and a rmw counter *rmwc*. The logical timestamp and process identifier can be used by a write protocol to form an unstable order of writes. A rmw *rmw* with base update u whose carstamp is cs_u is assigned a carstamp $cs_{rmw} = (cs_u.ts, cs_u.id, cs_u.rmwc + 1)$. The fields encode ordering constraints between operations via a lexicographical comparison such that $cs_1 < cs_2$ if and only if $cs_1.ts < cs_2.ts$ or $cs_1.ts = cs_2.ts$ and $cs_1.id < cs_2.id$ or $cs_1.ts = cs_2.ts$ and $cs_1.id = cs_2.id$ and $cs_1.rmwc < cs_2.rmwc$.

By incrementing the lowest order field of the carstamp, each carstamp assigned to a base update of a rmw is guaranteed to cover its rmw. This stable ordering of rmws with their base updates is visualized in Figure 3. Writes are assigned to carstamps in the first row as part of an increasing unstable order. RMWs are assigned to carstamps in the column to which their base update belongs immediately below their base update.

Consider the example from Figure 2b and assume that w_1 is assigned carstamp $cs_{w_1} = (1,1,0)$ by p_1. Then, since *rmw* reads w_1, it will be assigned carstamp $cs_{rmw} = (1,1,1)$. Based on the lexicographical ordering of carstamps, there does not exist a carstamp cs such that $cs_{w_1} < cs < cs_{rmw}$, so w_2 cannot be arbitrarily re-ordered between w_1 and *rmw*.

4 Gryff Protocol

Gryff unifies shared registers with consensus using carstamps. It implements a linearizable shared object (§2) that tolerates the failure of up to f out of $n = 2f+1$ replicas. We divide its description into three components. First, we provide additional background including the shared register protocol and consensus protocol upon which its read, write, and rmw protocols are built (§4.1). Second, we describe how Gryff adapts these protocols with carstamps (§4.2,§4.3). Third, we describe an optimization to the base Gryff protocol that improves read latency in geo-replicated settings (§5).

In addition, in Appendix B we prove Gryff implements a shared object with linearizability. Appendix B also proves read/write wait-freedom—every read or write invoked by a correct process eventually completes—and rmw wait-freedom with partial synchrony—if there is a point in time after which the system is synchronous, every rmw invoked by a correct process eventually completes.

4.1 Background

Section 2 provides background on our model, linearizability, and state machines and shared registers in general. This subsection adds useful definitions and then describes the two specific protocols that Gryff adapts, a multi-writer variant [43] of ABD [5] and EPaxos [47].

Definitions. A subset of processes $R \subseteq P$ are *replicas* that store the value of the object. We assume reliable message delivery, which can be implemented on top of unreliable message channels via retransmission and deduplication.

Replicas are often deployed across a wide-area network such that inter-replica message delivery latency is on the order of tens of milliseconds. This is commonly done so that replica or network failures correlated by geographic region do not immediately cause the system to become unavailable. We say that a process p is *co-located* with a replica r if the message delivery latency between p and r is much less than the minimum inter-replica latency. Client processes running applications are typically co-located with a single replica, for example, within the same datacenter.

A *quorum system* $\mathcal{Q} \subseteq \mathcal{P}(R)$ over R is a set of subsets of R with the *quorum intersection property*: for all $Q_1, Q_2 \in \mathcal{Q}$, $Q_1 \cap Q_2 \neq \emptyset$. We use *quorum* both to mean a set of replicas in a particular quorum system and the size of such a set. Gryff can use any quorum system, but for liveness with up to f replica failures, we assume the use of the majority quorum system $\mathcal{Q}_{maj}$ such that $\forall Q \in \mathcal{Q}_{maj}. |Q| = f+1$.

A *coordinator* is a process that executes a read, write, or rmw protocol when it receives such an operation from an application. In shared register protocols, the coordinators are typically the client processes on which the application is running. In consensus protocols, the coordinators are typically one of the replicas to which client processes forward their requests. We assume all processes possess a unique *identifier*

that can be used when coordinating an operation to distinguish the coordinator from other processes.

Multi-Writer ABD. The multi-writer variant [43] of ABD [5] is a shared register protocol that requires two phases for both reads and writes. To provide a linearizable order of reads and writes, it associates a *tag* $t = (ts, id)$ with each write where *ts* is a logical timestamp and *id* is the identifier of the coordinator. Writes are ordered lexicographically by their tags. Each replica stores a value v and an associated tag t.

Reads and writes have two phases. A read begins with the coordinator reading the current tag and value from a quorum. Once it receives these, it determines the value that will be returned by the read by choosing the value associated with the maximum tag from the tags returned in the quorum. Then, the coordinator propagates this maximum tag and value to a quorum and waits for acknowledgments. We say that a replica *applies* a value v' and tag t' when it overwrites its v and t with v' and t' if $t' > t$. After a replica receives the propagated tag and value, it applies them and sends an acknowledgment to the coordinator.

A coordinator for a write follows a similar two-phase protocol, except instead of propagating the maximum tag t_{max} and associated value received in the first phase, it generates a new tag $t = (t_{max}.ts + 1, id)$ to associate with the value to be written where *id* is the identifier of the coordinator. In the second phase, the coordinator propagates this new tag and value to a quorum and waits for acknowledgments.

EPaxos EPaxos [47] is a consensus protocol that provides optimal commit latency in the wide-area. It has three phases in failure-free executions: PreAccept, Accept, and Commit. If a command commits on the *fast path*, the coordinator returns to the client after the PreAccept phase and skips the Accept phase. Otherwise, the command commits on the *slow path* after the Accept phase. Commands that do not read state complete at the beginning of the Commit phase; commands that do read state complete after a single replica, typically the coordinator, executes the command to obtain the returned state. The purpose of the PreAccept and Accept phases is to establish the *dependencies* for a command, or the set of commands that must be executed before the current command. The purpose of the Commit phase is for the coordinator to notify the other replicas of the agreed-upon dependencies.

PreAccept phase. The coordinator of a command constructs the preliminary dependency set consisting of all other commands of which the coordinator is aware that interfere (i.e., access the same state machine state) with it. It sends the command and its dependencies to a fast quorum of replicas. When replicas receive the proposed dependencies, they update them with any interfering commands of which they are aware that are not already in the set and respond to the coordinator with the possibly updated dependencies. If the leader receives a fast quorum of responses that all contain the same dependencies, it proceeds to the Commit phase.

v - value of shared object
cs - carstamp of shared object
prev - value and carstamp generated by the previously executed rmw
i - next unused instance number
cmds - two-dimensional array of instances indexed by replica id and instance number each containing:

- *cmd* - command to be executed
- *deps* - instances whose commands must be executed before this one
- *seq* - approximate sequence number of command used to break cycles in dependency graph
- *base* - possible base update for rmw
- *status* - status of instance

Figure 4: State at each replica.

Accept phase. Otherwise, the coordinator continues to the Accept phase where it builds the final dependencies for the command by taking the union of all the dependencies that it received in the PreAccept phase. It sends these to a quorum and waits for a quorum of acknowledgments before committing. Regardless of whether the command is committed after the first or second phase, once it is committed, a quorum store the same dependency set for the command.

Execution. Dependency sets for distinct commands define a dependency graph over all interfering commands. The EPaxos execution algorithm, separate from the commit protocol, executes all commands in the deterministic order specified by the graph. Cycles may exist in the graph, in which case a total order is determined by a secondary attribute called an approximate sequence number. We refer the reader to the EPaxos paper for more details [47].

4.2 Read & Write Protocols

The read and write protocols are based on multi-writer ABD. Figure 4 summarizes the state that is maintained at each replica. Algorithms 1 and 2 show the pseudocode for the coordinators and replicas. The key difference from multi-writer ABD is that replicas maintain a carstamp associated with the current value of the shared object instead of a tag so that rmws are properly ordered with respect to reads and writes.

Reads. We make the same observation as Georgiou et al. [26] that the second phase in the read protocol of multi-writer ABD is redundant when a quorum already store the value and associated carstamp chosen in the first phase. In such cases, the coordinator may immediately complete the read (Line 6 of Algorithm 1). Otherwise, it continues as normal to the second phase in order to propagate the observed value and carstamp to a quorum.

Writes. When generating a carstamp after the first phase of a write, the coordinator chooses the *ts* and *id* fields as

Algorithm 1: Read and write coordinator protocols.

```
1  procedure Coordinator::READ() at p ∈ P
2      send Read1 to all r ∈ R
3      wait to receive Read1Reply(v_r, cs_r) from all
         r ∈ Q ∈ 𝒬
4      cs_max ← max_{r∈Q} cs_r
5      v ← v_r : cs_r = cs_max
6      if ∀r ∈ Q : cs_r = cs_max then
7          return v
8      send Read2(v, cs_max) to all r ∈ R
9      wait to receive Read2Reply from all r ∈ Q' ∈ 𝒬
10     return v
11 procedure Coordinator::WRITE(v) at p ∈ P
12     send Write1 to all r ∈ R
13     wait to receive Write1Reply(cs_r) from all r ∈ Q ∈ 𝒬
14     cs_max ← max_{r∈Q} cs_r
15     cs ← (cs_max.ts + 1, id, 0)
16     send Write2(v, cs) to all r ∈ R
17     wait to receive Write2Reply from all r ∈ Q' ∈ 𝒬
```

Algorithm 2: Read and write replica protocols.

```
1  when replica r ∈ R receives a message m from p ∈ P do
2      case m = Read1 do
3          send Read1Reply(v, cs) to p
4      case m = Read2(v', cs') do
5          APPLY(v', cs')
6          send Read2Reply to p
7      case m = Write1 do
8          send Write1Reply(cs) to p
9      case m = Write2(v', cs') do
10         APPLY(v', cs')
11         send Write2Reply to p
12 procedure Replica::APPLY(v', cs')
13     if cs' > cs then
14         cs ← cs'
15         v ← v'
```

in multi-writer ABD. The *rmwc* field is reset to 0 (Line 15 of Algorithm 1). While not strictly necessary, this curbs the growth of the *rmwc* field in practical implementations.

4.3 Read-Modify-Write Protocol

Gryff's rmw protocol uses EPaxos to stably order rmws as commands in the dependency graph. Figure 4 summarizes the replica state. Algorithms 3 and 4 show the pseudocode for a rmw coordinator and replica message handling excluding the recovery procedure. Appendix B includes the pseudocode for the recovery procedure. The highlighted portions of the pseudocode show the changes from canonical EPaxos. We denote by I_{cmd} the set of commands of which the local replica is aware that interfere with *cmd*.

We make three high-level modifications to canonical EPaxos in order to unify its stable ordering with the unstable ordering of Gryff's read and write protocols.

1. A base update attribute, *base*, is decided by the replicas during the same process that establishes the dependencies and the approximate sequence number for a rmw.
2. A rmw completes after a quorum execute it.
3. When a rmw executes, it chooses its base update from between its *base* attribute and the result of the previously executed rmw *prev*. The result of the executed rmw is applied to the value and carstamp of the executing replica.

The first change adapts EPaxos to work with the unstable order of writes by fixing the write upon which it will operate. The second change adapts it to work with reads that bypass its execution protocol and directly read state. The third change ensures that concurrent rmws that choose the same initial base update are stably ordered using the ordering and execution protocols of EPaxos. We next discuss each of these changes in more detail.

Base Attribute. The *base* attribute associated with a rmw represents a possible base update on which the rmw will execute. Initially, the coordinator sets this to what it believes are the current value and carstamp of the shared object (Line 6 of Algorithm 3). When a replica receives a *PreAccept* message, it merges what it believes is the correct base update with the base update proposed by the coordinator (Line 5 of Algorithm 4). The fast path condition remains essentially unchanged: the coordinator commits the command if it receives *PreAcceptOK* responses from a fast quorum indicating that all replicas in the quorum agree on the attributes for the command. Otherwise, the coordinator merges all attributes it has received in the PreAccept phase and sends out the final attributes in the Accept phase.

Quorum Execute. In canonical EPaxos, a rmw completes after a single replica executes it because reads are executed through the same consensus protocol. Since Gryff's read protocol circumvents consensus and reads the state of the shared object directly from a quorum, a rmw must be executed at a quorum so that it is visible to reads that come after it in real time. This guarantees the rmw will be visible to future reads by the quorum intersection property.

Execution. The algorithm for determining the execution order of commands is unchanged from canonical EPaxos. The EXECUTE procedure in Algorithm 4 is called when a rmw *rmw* in the dependency graph committed at position (i, j) in the *cmds* array is ready to be executed.

In the procedure, the final base update for *rmw* is chosen to be the value and carstamp pair with the larger carstamp

Algorithm 3: RMW coordinator protocol.

1 **procedure** Coordinator::RMW($f(\cdot)$) at $c \in R$
  PreAccept Phase:
2 $i \leftarrow i+1$
3 $cmd \leftarrow f(\cdot)$
4 $seq \leftarrow 1+\max(\{cmds[j][k].seq | (j,k) \in I_{cmd}\} \cup \{0\})$
5 $deps \leftarrow I_{cmd}$
6 $base \leftarrow (v, cs)$
7 $cmds[id][i] \leftarrow (cmd, seq, deps, base,$ **pre-accepted**$)$
8 **send** $PreAccept(cmd, seq, deps, base, id, i)$ to all $r \in F \setminus \{c\}$ where $F \in \mathcal{F}$
9 **wait** to receive $PreAcceptOK(seq'_r, deps'_r, base'_r)$ from all $r \in F \setminus \{c\}$
10 **if** $\forall r_1, r_2 \in F \setminus \{c\} : seq'_{r_1} = seq'_{r_2} \wedge deps'_{r_1} = deps'_{r_2} \wedge base'_{r_1} = base'_{r_2}$ **then**
11 $deps, seq, base \leftarrow deps'_r, seq'_r, base'_r$: $r \in F \setminus \{c\}$
12 **goto Commit Phase**
  Accept Phase:
13 $deps \leftarrow \cup_{r \in F} deps_r$
14 $seq \leftarrow \max_{r \in F} seq_r$
15 $base \leftarrow base_r : \forall r' \in F. base_r.cs \geq base_{r'}.cs$
16 $cmds[id][i] \leftarrow (cmd, seq, deps, base,$ **accepted**$)$
17 **send** $Accept(cmd, seq, deps, base, id, i)$ to all $r \in Q \setminus \{c\}$ where $Q \in \mathcal{Q}$
18 **wait** to receive $AcceptOK$ from all $r \in Q \setminus \{c\}$
  Commit Phase:
19 $cmds[id][i] \leftarrow (cmd, seq, deps, base,$ **committed**$)$
20 **send** $Commit(cmd, seq, deps, base, id, i)$ to all $r \in R \setminus \{c\}$
21 **wait** to receive $Executed(v)$ from all $r \in Q' \in \mathcal{Q}$
22 **return** v

Algorithm 4: RMW replica protocol.

1 **when** *replica* $r \in R$ *receives a message* m *from* $c \in R$ **do**
2 **case** $m = PreAccept(cmd, seq, deps, base, id_c, i)$ **do**
3 $seq' \leftarrow \max(\{seq\} \cup \{1 + cmds[j][k].seq | (j,k) \in I_{cmd}\})$
4 $deps' \leftarrow deps \cup I_{cmd}$
5 $base' \leftarrow$ **if** $cs > base.cs$ **then** (v, cs) **else** $base$
6 $cmds[id_c][i] \leftarrow (cmd, seq', deps', base',$ **pre-accepted**$)$
7 **send** $PreAcceptOK(seq', deps', base')$ to c
8 **case** $m = Accept(cmd, seq, deps, base, id_c, i)$ **do**
9 $cmds[id_c][i] \leftarrow (cmd, seq', deps', base',$ **accepted**$)$
10 **send** $AcceptOK$ to c
11 **case** $m = Commit(cmd, seq, deps, base, id_c, i)$ **do**
12 $cmds[id_c][i] \leftarrow (cmd, seq', deps', base',$ **committed**$)$
13 **procedure** Replica::EXECUTE(j,k)
14 $base \leftarrow cmds[j][k].base$
15 **if** $cmds[j][k].base.cs < prev.cs$ **then**
16 $base \leftarrow prev$
17 $v' \leftarrow cmds[j][k].cmd(base.v)$
18 $cs' \leftarrow (base.cs.ts, base.cs.id, base.cs.rmwc + 1)$
19 $prev \leftarrow (v', cs')$
20 APPLY(v', cs')
21 **send** $Executed(base.v)$ to replica j

between the result *prev* of the previously executed rmw and the *base* attribute of *rmw* (Line 15 of Algorithm 4). The *prev* variable is the most recent state of the shared object produced by the execution of a rmw whereas the *base* attribute is the most recent state of the shared object that the coordinator observed after *rmw* was invoked. In the absence of concurrent updates, these states are equivalent, so it is safe for the *rmw* to choose the state as the base update.

However, when rmws are concurrent, *prev* may be more recent than the *base* attribute of *rmw* because concurrent rmws were ordered and executed before *rmw*. In such cases, *rmw* must remain consistent with the stable order of rmws provided by EPaxos by executing on the most recent state.

The resulting value and carstamp of *rmw* are decided by executing the modify function $f(\cdot)$ on the value of the base update and incrementing the *rmwc* of the carstamp of the chosen base update. The replica finishes by applying the new value and carstamp and notifying the coordinator that the rmw has been executed.

5 Proxying Reads

The base Gryff read protocol, as described in the previous section, provides reads with single round-trip time latency from the coordinator to the nearest quorum including itself (1 RTT) when there are no concurrent updates. Otherwise, reads have at most 2 RTT latency. We discuss how read latency can be further improved in deployments across wide-area networks.

Because the round-trip time to the replica that is co-located with a client process is negligible relative to the inter-replica latency, replicas can coordinate reads for their co-located clients and utilize their local state in the read coordinator protocol to terminate after 1 RTT more often. When using this optimization, we say that the coordinating replica is a *proxy* for the client process's read.

Propagating Extra Data in Read Phase 1. The proxy includes in the *Read1* messages its current value v and carstamp cs. Upon receiving a *Read1* message with this additional information, a replica applies the value and carstamp before returning its current value and carstamp. This has the effect of ensuring every replica that receives the *Read1* messages will have a carstamp (and associated value) at least as large as the carstamp at the proxy when the read was invoked.

When this is the most recent carstamp for the shared object, the read is guaranteed to terminate after 1 RTT. This is because every *Read1Reply* that the coordinator receives will contain this most recent carstamp and associated value.

Updating the Proxy's Data. The proxy also applies the values and carstamps that it receives in *Read1Reply* messages as it receives them and before it makes the decision of whether or not to complete the read after the first phase. If every reply contains the same carstamp, then the read completes after 1 RTT even if the carstamp at the proxy when the read was invoked is smaller than the carstamp contained in every reply.

Given our assumption that each quorum contains $f+1$ replicas, these two modifications ensure that reads coordinated by a proxy r only take 2 RTT during normal operation when there is a concurrent update that arrives at the f nearest replicas to r in an order that interleaves with the *Read1* messages from r. Algorithm 7 in Appendix B describes the read proxy changes to base Gryff in pseudocode. Appendix B also contains a brief argument for why the read proxy optimization maintains the correctness of base Gryff.

Always Fast Reads When $n = 3$. This optimization increases the likelihood that a read completes in 1 RTT because the proxy replica is privy to more information—i.e., the number of replicas that contain the same value and carstamp—than a client process. Moreover, it allows Gryff to always provide 1 RTT reads when $n = 3$ since the proxy and any single other replica comprise a quorum. This optimization is, in some sense, the dual of the optimization that EPaxos [47] uses to always provide 1 RTT writes when $n = 3$. In both cases, the coordinator and the other replica in the quorum adopt each other's state so that the quorum always has the same state at the end of the first phase.

6 Evaluation

Gryff unifies consensus with shared registers to avoid the overhead of consensus for reads and writes. To quantify the benefits and drawbacks of this approach for storing data in geo-replicated, large-scale web applications, we ask:

- Do Gryff's shared register read and write protocols reduce read tail latency relative to the state-of-the-art? (§6.3)
- How do the read/write/rmw latency and throughput of Gryff compare to state-of-the-art protocols? (§6.4,§6.5)
- Does Gryff improve the median service-level latency for large scale web applications? (§6.6)

We find that, for workloads with moderate contention, Gryff reduces p99 read latency to ∼56% of EPaxos, but has ∼2x higher write latency. This tradeoff allows Gryff to reduce service-level p50 latency to ∼60% of EPaxos for large-scale web applications whose requests fan-out into many storage-level requests. Gryff and EPaxos each achieve a slightly higher maximum throughput than MultiPaxos due to their leaderless structure.

	CA	VA	IR	OR	JP
CA	0.2				
VA	72.0	0.2			
IR	151.0	88.0	0.2		
OR	59.0	93.0	145.0	0.2	
JP	113.0	162.0	220.0	121.0	0.2

Figure 5: Round trip latencies in ms between nodes in emulated geographic regions.

6.1 Baselines and Implementation

We evaluate Gryff against MultiPaxos and EPaxos. MultiPaxos [36], VR [50], Raft [51] and other protocols with leader-based architectures are used in commercial systems to provide linearizable replicated storage [14, 17, 23, 52]. While leader-based protocols have drawbacks in geo-replicated settings, their extensive use in real systems provides a practical measuring stick. EPaxos [47] is the state-of-the-art for geo-replicated storage.

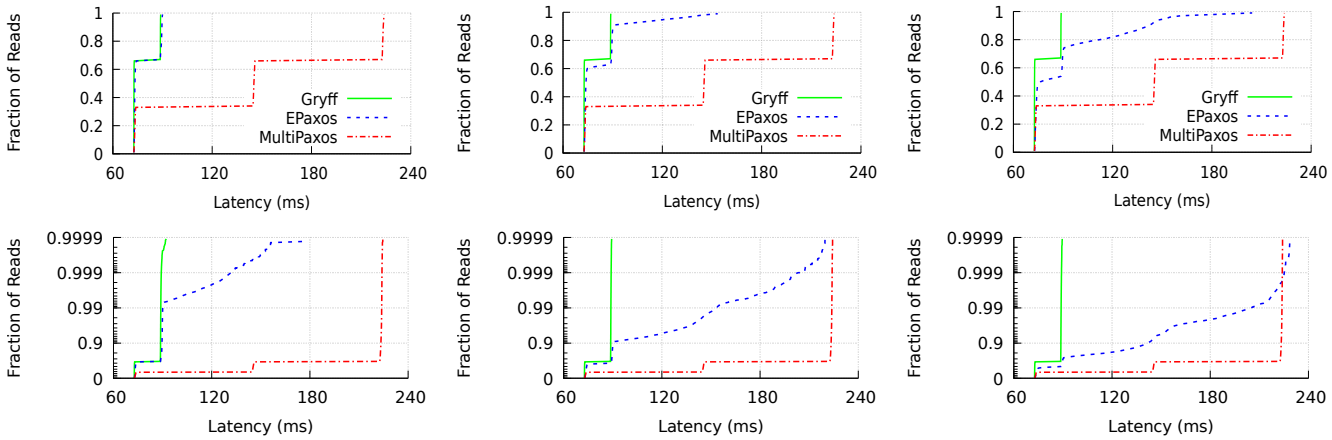
We implemented Gryff in Go using the framework of EPaxos to facilitate apples-to-apples comparisons between protocols. Our implementation is a multi-object storage system that uses the protocols as described in this paper with the addition of object identifiers to messages and state. Our code and experiment scripts are available online [29]. We use the existing implementation of MultiPaxos in the framework for our experiments. All of our experiments use the thrifty optimization for EPaxos, MultiPaxos, and Gryff. We use the read proxy optimization for Gryff.

6.2 Experimental Setup

Testbed. We run our experiments on the Emulab testbed [61] using pc3000 nodes. These node types have 1 Dual-Core 3 GHz CPU, 2 GB RAM, and 1 Gbps links to all other nodes. For three replica latency experiments, we emulate replicas in California (CA), Virginia (VA), and Ireland (IR). In five replica latency experiments, we add replicas in Oregon (OR) and Japan (JP). In all experiments, we place the MultiPaxos leader in CA.

We emulate wide-area network latencies using Linux's Traffic Control (tc) to add delays to outgoing packets on all nodes. Table 5 shows the configured round-trip times between nodes in different regions. We choose these numbers because they are the typical round-trip times between the corresponding Amazon EC2 availability regions.

Clients. For all experiments, we use 16 clients co-located with each replica. This number of clients provides enough load on the evaluated protocols to observe the effects of concurrent operations from many clients, but only moderately saturates the system. We avoid full saturation in order to isolate the protocol mechanisms that affect tail latency from hardware and software limitations at various levels in our stack. Clients perform operations in a closed loop.



(a) 2% conflicts. **(b) 10% conflicts.** **(c) 25% conflicts.**

Figure 6: Gryff's reads always complete in 1 RTT when $n = 3$. 99th percentile read latency is between 0 ms and 115 ms lower than EPaxos and 134 ms lower than MultiPaxos.

Measurement. Each experiment is run for 180 seconds and we exclude results from the first 15 seconds and last 15 seconds to avoid artifacts from start-up and cool-down. The latency for an individual operation is measured as the time between when a client invokes the operation and when it is notified of the operation's completion.

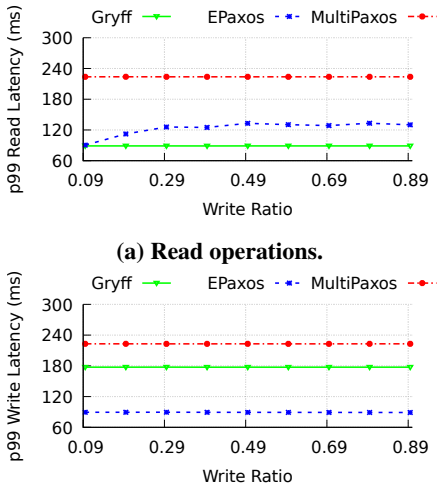
Conflicting Operations. When two operations target the same object in a storage system, we say the operations *conflict*. We use *conflict percentage* as a parameter in our workloads to control the percentage of operations from each client that target the same key. Workloads are highly skewed if and only if their conflict percentage is high.

6.3 Tail Latency

Gryff is designed to reduce the latency cost of linearizability for large scale web applications. Tail latency is of particular importance for these applications because end-user requests for high-level application objects typically fan-out into hundreds of sub-requests to storage services [2, 18]. The object can only be returned to the end-user once all of these sub-requests complete, so the median latency experienced by the end-user is dictated by the tail of the latency distribution for operations to these storage services.

6.3.1 Varying Conflict Percentage

To understand the read tail latency of Gryff and the baselines, we use a variant of the YCSB-B [16] workload that contains 94.5% reads, 4.5% writes, and 1.0% rmws. We examine a read-heavy distribution of operations because most large-scale web applications are read-heavy. For example, more than 99.7% of operations are reads in Google's advertising backend, F1 [17], 99.8% of operations in Facebook's TAO system are reads [10], and 3 out of 5 of YCSB's core workloads contain over 95% reads [16].



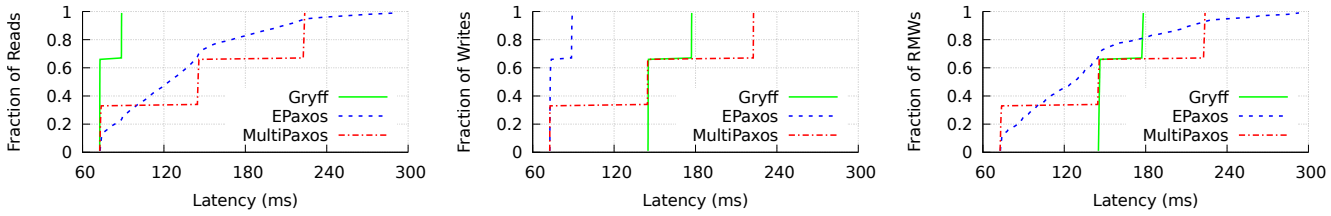
(a) Read operations.

(b) Write operations.

Figure 7: Gryff reduces p99 read latency between 1 ms and 44 ms relative to EPaxos and 134 ms relative to MultiPaxos for varying write percentages. EPaxos' p99 write latency is 89 ms lower than Gryff's p99 write latency regardless of write percentage and conflicts.

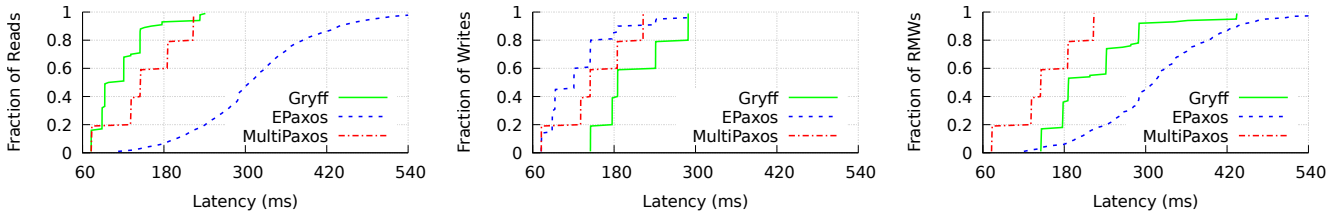
Figure 6a shows the results for three different conflict percentages with $n = 3$. In each sub-figure, a log-scale CDF up to p99.99 is shown below the normal-scale CDF.

1 RTT Reads for Gryff. For $n = 3$ replicas, Gryff always completes reads in 1 RTT due to the read proxy optimization (§5). Figure 6 shows that clients in each region receive responses to their read requests after 1 RTT to the nearest quorum regardless of conflict percentage. Clients in CA are closest to the replicas in CA and VA and vice versa for clients in VA. This results in 66% of the reads completing in the round-trip time between CA and VA (72 ms). Clients in IR



(a) Reads. **(b) Writes.** **(c) RMWs.**

Figure 8: Gryff's writes take 2 RTT, which is always more than EPaxos when $n = 3$. MultiPaxos writes can be faster or slower than Gryff depending on client location and geographic setup.



(a) Reads. **(b) Writes.** **(c) RMWs.**

Figure 9: Gryff trades off worse write latency for better read and rmw latency relative to EPaxos when $n = 5$.

are closest to the replicas in IR and VA, so 33% of the reads complete in the round-trip time between IR and VA (88 ms).

Execution Dependencies Delay EPaxos. EPaxos always commits in 1 RTT for $n = 3$. However, a read cannot complete until a replica executes it and a replica can only execute it after receiving and executing its dependencies. This increases latency when a locally committed read has dependencies on operations that have not yet arrived at the local replica from other replicas. As shown in Figure 6a, these delays do not affect the p99 read latency of EPaxos when there are few conflicts. However, the log-scale CDF shows that a small number of reads are, in fact, delayed.

MultiPaxos has Client-dependent Stable Latency. The MultiPaxos leader can always commit and execute operations in 1 RTT to the nearest quorum. However, clients must also incur a 1 RTT delay to the leader. For clients co-located with the leader (in CA), this delay is negligible, so the latency experienced by these clients with MultiPaxos is less than or equal to the latency experienced with the other protocols. This is demonstrated in the 33rd percentile latencies in Figure 6. For clients not co-located with the leader, the latency is roughly 2 RTT.

Gryff improves 99th percentile read latency between 0 ms and 115 ms relative to EPaxos for low and high conflict percentages and 134 ms relative to MultiPaxos.

6.3.2 Varying Write Percentage

While Gryff's read tail latency is low for read-heavy workloads, we also quantify the tail latency under balanced and write-heavy workloads. To do so, we fix the conflict percentage at 2% and measure the 99th percentile latency of read and write operations for workloads containing 1% rmws and varying ratios of reads and writes. We vary the write percentage from 9.5% to 89.5% and the read percentage from 89.5% to 9.5%. Figure 7 shows the results for $n = 3$ replicas.

Gryff and MultiPaxos Unaffected. The write percentage does not affect Gryff's write latency because its write protocol arbitrarily orders concurrent writes. Similarly, MultiPaxos commits writes through the same path regardless of conflicting operations.

EPaxos Reads Slowdown. With increasing write percentage, the chance that a read obtains a dependency increases even with a fixed conflict percentage (Figure 7a). Unlike reads, writes do not need to be executed before they complete, so they still complete as soon as they are committed. This only takes 1 RTT in EPaxos when $n = 3$. EPaxos dominates Gryff and MultiPaxos for p99 write latency.

Five Replica Varying Write Ratio. We run the same workload with $n = 5$ and show the results in Figure 12 in Appendix A. Gryff can no longer always complete reads in 1 RTT, but due to the low conflict percentage it still achieves a p99 read latency of 1 RTT regardless of write percentage. EPaxos can no longer always commit in 1 RTT. This especially impacts EPaxos' p99 write latency, which becomes approximately the same as Gryff (290 ms).

6.4 Read/Write/RMW Latency

We also quantify the latency distributions of write and rmws in Gryff relative to that of the baselines. For these experiments, we use a variant of the YCSB-A workload with 49.5%

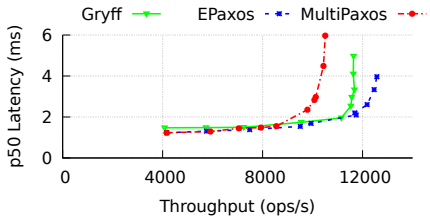


Figure 10: Gryff's throughput at saturation is within 7.5% of EPaxos and is higher than MultiPaxos.

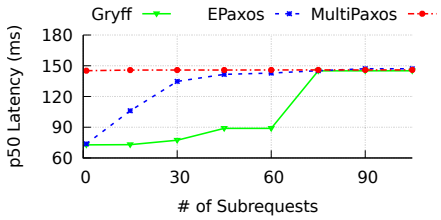


Figure 11: Gryff improves service-level p50 latency when the expected tail-at-scale request contains many reads.

reads, 49.5% writes, and 1.0% rmws with 25% conflicts. The balance between reads and writes allows us to observe the effects that interleavings of operations with different semantics have on the performance of the evaluated protocols. Similarly, the high conflict percentage reveals performance when concurrent operations to the same object interleave.

Figure 8 shows the cumulative distribution functions of the latencies for each operation type for $n = 3$ replicas. Figure 9 shows the same for $n = 5$.

1 RTT Reads for Gryff. For $n > 3$, Gryff often completes reads in 1 RTT, but sometimes takes 2 RTT. Figure 9a demonstrates this behavior as the tail surpasses the 1 RTT latency for any region.

EPaxos Writes are Fast, Reads are Slower. EPaxos dominates Gryff and MultiPaxos for write latency because it always commits in a single round trip for $n = 3$ (Figure 8b) and often commits in a single round trip for $n = 5$ (Figure 9b). As discussed in Section 6.3.1, reads cannot complete until they are executed, so when there are more replicas and more concurrent writes, EPaxos' read latency increases due to the increased likelihood that reads acquire dependencies on updates from other regions.

2 RTT Writes for Gryff. Writes in Gryff takes 2 RTT to complete. Figure 8b demonstrates the gap between EPaxos and Gryff for $n = 3$. When $n > 3$ replicas (Figure 9b), EPaxos still typically completes writes faster than Gryff because it only takes 2 RTT when conflicting concurrent operations arrive at replicas in the intersections of their fast quorums in different orders.

Less Blocking for RMWs in Gryff. Gryff achieves 2 RTT rmws when there are no conflicts and 3 RTT when there are. While Gryff must still block the execution of rmws until all dependencies have been received and executed, Gryff experiences significantly less blocking than EPaxos. This is because EPaxos needs to have dependencies on writes, but Gryff's rmw protocol does not.

EPaxos dominates Gryff for write latency. For $n = 3$, the p50 write latency of Gryff is 72 ms higher and the p99 write latency is 89 ms higher than EPaxos.

6.5 Throughput

We measure median latency at varying levels of load in a local-area cluster. Again, we use the variant of YCSB-A with 49.5% reads, 49.5% writes, and 1.0% rmws with 25% conflicts. Figure 10 shows the results for $n = 3$. We find that Gryff's throughput at saturation is about 11,600 ops/s, within 7.5% of EPaxos. This is also about 1,200 ops/s higher than the maximum throughput of MultiPaxos. Like EPaxos, Gryff does not require a single replica to be involved in the execution of every operation, so it achieves better scalability and load-balancing than leader-based protocols.

Gryff Scales Better. We run the same workload with $n = 5$ and show the results in Figure 13 in Appendix A. Gryff's maximum throughput is higher than EPaxos because EPaxos can no longer always commit on the fast path. Each operation that commits on the slow path on EPaxos requires an additional quorum of messages and replies, which causes the system to more quickly saturate.

6.6 Tail at Scale

Our primary experiments show that Gryff improves read latency relative to our baselines. However, p50 write and p50 rmw latency are lower in EPaxos for $n = 3$. For other parts of the distributions and for MultiPaxos, the latency tradeoff is not comparable. To understand how these tradeoffs with EPaxos and MultiPaxos affect the performance of large-scale web applications whose structure resembles the common structure discussed in Section 6.3, we ran experiments that emulate end-user requests.

We emulate the request pattern of an application preparing a high-level object for an end-user. The object is composed of m sub-requests to the storage system that are drawn from a fixed distribution of reads, writes, and rmws. For example, in order to display a profile page in a social network, dozens of requests to the storage systems that store profile information must be initiated simultaneously [10]. The latency of one of these *tail at scale requests* is the maximum latency of all of its sub-requests. Thus, the median latency of tail at scale requests depends on the tail latency of the sub-requests.

The large-scale web applications whose workloads we emulate are typically read-heavy (§6.3). Moreover, they are often highly skewed. Facebook engineers report that a small

set of objects account for a large fraction of total read and write operations in the social graph [2]. This experiment uses a 99%/0.9%/0.1% read/write/rmw workload with 25% conflicts. We vary the number of sub-requests m from 1 to 105 in increments of 15. Figure 11 summarizes the results.

Fast Reads Improve Median End-to-end Latency. Gryff's median latency is lower than that of EPaxos and MultiPaxos when fewer than half of the tail at scale requests are expected to contain a write or rmw operation. Compared to EPaxos' p50 latency, Gryff's is up to 57 ms lower for $n = 3$.

Five Replica Tail-at-scale. We run the same workload with $n = 5$ and show the results in Figure 14 in Appendix A. All protocols follow trends similar to the $n = 3$ case. However, Gryff cannot always complete reads in 1 RTT, so the longer tail of the read latency distribution causes the median latency of these tail at scale requests to increase at a smaller number of sub-requests. Similarly, EPaxos can no longer always commit in 1 RTT, so its tail latency is 2 RTTs plus the delay from blocking for dependencies.

7 Related Work

We review related work in geo-replicated storage systems and combining consensus with shared registers.

EPaxos. EPaxos [47] is the state-of-the-art for linearizable replication in geo-replicated settings. Our evaluation shows that EPaxos dominates Gryff for blind write latency. On the other hand, Gryff dominates EPaxos for read latency and its rmw latency ranges from higher to lower as the contention in the workload increases. This tradeoff is possible because Gryff only uses consensus for operations that require it.

Read Leases. Read leases allow clients to read replicated state from leaseholders by requiring updates to the replicated state be acknowledged by the leaseholders before completing [28, 49]. While this enables reads that need only communicate with a single replica, it sacrifices write availability when a leaseholder fails until the lease expires. Furthermore, to implement read leases safely, clocks at each process must have bounded skew, which is not satisfied by current commodity clocks [25]. Given these difficult availability and safety tradeoffs, we do not consider read leases in the context of Gryff or the baseline systems, but we believe they can be adapted to Gryff's write and rmw protocols.

Other Linearizable Protocols. Paxos [36], VR [50], Fast Paxos [38], Generalized Paxos [37], Mencius [45], Raft [51], Flexible Paxos [33], CAESAR [4], and SD Paxos [62] are consensus protocols that are used to implement linearizable replicated storage systems by ensuring the Agreement property for state machine replication [56]. Other systems, such as Sinfonia [1] and Zookeeper [34], use similarly expensive coordination protocols (2PC and atomic broadcast respectively) to provide strong consistency. CURP [53], Chain Replication [59], and other primary-backup protocols [3] achieve good performance when failures are detectable. Gryff guarantees linearizability in systems with undetectable failures for reads, writes, and rmws and only incurs expensive coordination overhead when needed.

ABD [5] provides linearizable reads and writes with guaranteed termination in asynchronous settings. Subsequent work has established the conditions under which linearizable shared register protocols can provide fast—i.e., complete in 1 RTT—reads [20] or writes [22]. Gryff maintains the performance benefits of these protocols for reads and writes and incorporates rmws for when application developers need stronger synchronization primitives.

Weaker Semantics for Lower Latency. Other geo-replicated systems eschew strong consistency for weaker consistency models that support lower latency operations. PNUTS [15] provides per-timeline sequential consistency, OCCULT [46], COPS [42], and GentleRain [19] provide causal consistency. ABD-Reg [60] provides regularity. Moreover, some systems provide hybrid consistency: Pileus [58], Gemini [39], and ICG [30] allow some operations to be strongly consistent and other operations to be weakly consistent. Gryff provides linearizability to free developers from reasoning about complex consistency models.

Consensus and Shared Registers. Active Quorum Systems (AQS) [7, 8], to our knowledge, was the first attempt to combine consensus with shared registers. We found that AQS allows for non-linearizable executions because its ordering mechanism is unstable for rmws (Appendix C). In contrast, Gryff uses carstamps to stably order rmws with their base updates while allowing for efficient reads and writes with an unstable order. In addition, Gryff is implemented and empirically evaluated.

Cassandra [44] provides reads and writes with tunable consistency and implements a compare-and-swap for applications that occasionally need stronger synchronization. Unlike Gryff, Cassandra's reads and writes are not linearizable by default and its compare-and-swap is not consistent when operating on data also accessed via reads and writes.

8 Conclusion

Gryff unifies consensus and shared registers with carstamps. This reduces latency by avoiding the cost of consensus for the common case of reads and writes. Our evaluation shows that the reduction in latency for individual operations reduces the median service-level latency to $\sim$60% of EPaxos for large-scale web applications.

Acknowledgments We thank our shepherd, Patrick P. C. Lee, and the anonymous reviewers for their helpful comments. We are grateful to Christopher Hodsdon, Theano Stavrinos, Natacha Crooks, Soumya Basu, Haonan Lu, and Khiem Ngo for their extensive feedback. This work was supported by the National Science Foundation under grants number CNS-1824130 and CNS-1932829.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *ACM Symposium on Operating System Principles (SOSP)*, 2007.

[2] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. Challenges to Adopting Stronger Consistency at Scale. In *ACM SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[3] P. A. Alsberg and J. D. Day. A Principle for Resilient Sharing of Distributed Resources. In *International Conference on Software Engineering*, 1976.

[4] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran. Speeding up Consensus by Chasing Fast Decisions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[5] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[6] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Conference on Innovative Data Systems Research (CIDR)*, 2011.

[7] A. Bessani. Active Quorum Systems: Specification and Correctness Proof. Technical report, Technical Report DIFCULTR201002, University of Lisbon, 2010.

[8] A. Bessani, P. Sousa, and M. Correia. Active Quorum Systems. In *Workshop on Hot Topics in System Dependability (HotDep)*, 2010.

[9] K. Birman, G. Chockler, and R. van Renesse. Toward a Cloud Computing Research Agenda. *ACM SIGACT News*, 40(2):68–80, 2009.

[10] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference (ATC)*, 2013.

[11] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[13] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure Coding Objects across Global Data Centers. In *USENIX Annual Technical Conference (ATC)*, 2017.

[14] CockroachDB. `https://www.cockroachlabs.com/`, 2020.

[15] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *Proceedings of the VLDB Endowment (PVLDB)*, 2008.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*, 2010.

[17] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Googles Globally-Distributed Database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[18] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[19] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *ACM Symposium on Cloud Computing (SoCC)*, 2014.

[20] P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty. How Fast can a Distributed Atomic Read be? In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2004.

[21] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[22] B. Englert, C. Georgiou, P. M. Musial, N. Nicolaou, and A. A. Shvartsman. On the Efficiency of Atomic Multi-reader, Multi-writer Distributed Memory. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2009.

[23] etcd. `https://etcd.io/`, 2020.

[24] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[25] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat. Exploiting a Natural

Network Effect for Scalable, Fine-grained Clock Synchronization. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[26] C. Georgiou, N. C. Nicolaou, and A. A. Shvartsman. On the Robustness of (Semi) Fast Quorum-Based Implementations of Atomic Shared Memory. In *International Symposium on Distributed Computing (DISC)*, 2008.

[27] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable Consistency in Scatter. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[28] C. G. Gray and D. R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *ACM Symposium on Operating System Principles (SOSP)*, 1989.

[29] Gryff Implementation. `https://www.github.com/matthelb/gryff/`, 2020.

[30] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. Incremental Consistency Guarantees for Replicated Objects. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[31] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[32] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[33] H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696*, 2016.

[34] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX Annual Technical Conference (ATC)*, 2010.

[35] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[36] L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[37] L. Lamport. Generalized Consensus and Paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[38] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2): 79–103, 2006.

[39] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[40] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[41] R. J. Lipton and J. Sandberg. PRAM: A Scalable Shared Memory. Technical report, Technical Report TR-180-88, Princeton University, 1988.

[42] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[43] N. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *International Symposium on Fault Tolerant Computing*, 1997.

[44] P. Malik and A. Lakshman. Cassandra - A Decentralized Structured Storage System. In *ACM SIGOPS Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2009.

[45] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[46] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[47] I. Moraru, D. G. Andersen, and M. Kaminsky. There Is More Consensus in Egalitarian Parliaments. In *ACM Symposium on Operating System Principles (SOSP)*, 2013.

[48] I. Moraru, D. G. Andersen, and M. Kaminsky. A Proof of Correctness for Egalitarian Paxos. Technical report, Technical Report CMU-PDL-13-111, Carnegie Mellon University, 2013.

[49] I. Moraru, D. G. Andersen, and M. Kaminsky. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. In *ACM Symposium on Cloud Computing (SoCC)*, 2014.

[50] B. M. Oki and B. H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1988.

[51] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference (ATC)*, 2014.

[52] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAM-

Cloud. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[53] S. J. Park and J. Ousterhout. Exploiting Commutativity For Practical Fast Replication. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[54] Redis. `https://redis.io/`, 2020.

[55] Riak. `https://riak.com/products/riak-kv/`, 2020.

[56] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[57] K. Taranov, G. Alonso, and T. Hoefler. Fast and strongly-consistent per-item resilience in key-value stores. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2018.

[58] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-Based Service Level Agreements for Cloud Storage. In *ACM Symposium on Operating System Principles (SOSP)*, 2013.

[59] R. van Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[60] M. Vukolić. Quorum Systems: with Applications to Storage and Consensus. *Synthesis Lectures on Distributed Computing Theory*, 3(1):1–146, 2012.

[61] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

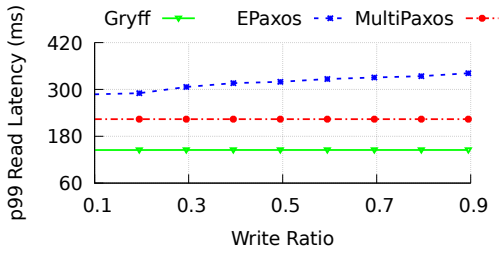
[62] H. Zhao, Q. Zhang, Z. Yang, M. Wu, and Y. Dai. SDPaxos: Building Efficient Semi-Decentralized Geo-Replicated State Machines. In *ACM Symposium on Cloud Computing (SoCC)*, 2018.

A Additional Experiments

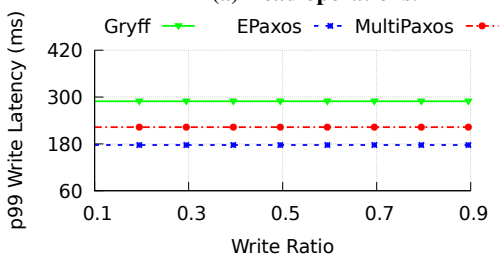
This appendix contains figures for experiments run with $n = 5$ replicas. Discussions of these results are in the main body of the paper in Section 6.

B Proof of Correctness

The proof of correctness for Gryff is presented in five parts. First, we define our model and introduce definitions (§B.1). Second, we describe the remainder of the rmw protocol (§B.2). Third, we prove safety for the base protocol (§B.3).



(a) Read operations.



(b) Write operations.

Figure 12: Gryff has better p99 read latency for $n = 5$ because, even though reads sometimes complete in 2 RTT, enough still complete in 1 RTT that the p99 latency is determined by 2 RTT in a region (CA) where the nearest quorum are relatively close (72 ms per RTT). EPaxos cannot always commit reads or writes in 1 RTT, so its latency increases relative to $n = 3$.

Fourth, we prove liveness for the base protocol (§B.4). Fifth, we argue that the read proxy optimization maintains the desired correctness properties (§B.5).

B.1 Preliminaries

We introduce the system model (§B.1.1) and define a shared object (§B.1.2).

B.1.1 Model

The system is comprised of a set P of *processes* $\{p_1, ..., p_m\}$. A subset $R \subseteq P$ of processes are *replicas* $\{r_1, ..., r_n\}$. Processes communicate with each other over point-to-point message channels. We assume *reliable message delivery*. This abstraction can be implemented on top of unreliable message channels that guarantee eventual delivery via retransmission and deduplication.

Processes may fail according to the *crash failure model*: a failed process ceases executing instructions and its failure is not detectable by other processes. The system is *asynchronous* such that there is no upper bound on the time it takes for a message to be delivered and there is no bound on relative speeds at which processes execute instructions.

Processes are state machines that deterministically transition between states when an *event* occurs. A process interacts with its environment via a set of objects O. The process may

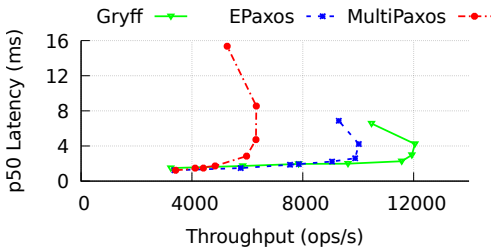


Figure 13: Gryff's throughput at saturation is higher than both EPaxos and MultiPaxos when $n = 5$.

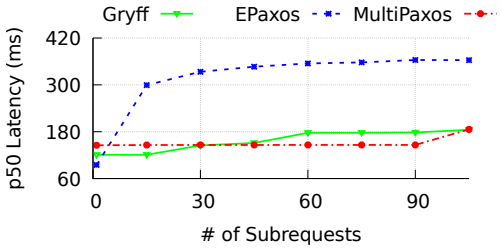


Figure 14: For $n = 5$, the difference in service-level p50 latency is larger because reads in EPaxos suffer from more blocking with more replicas and clients executing operations.

receive an operation op for an object via an invocation event $inv(op)$. The process indicates the result of the operation by generating a response event $resp(op)$. Internal events are the modification of local state at a process, the sending or receipt of a message, and the failure of process. We denote the process associated with an event e by $process(e)$.

An *execution* is an infinite sequence of events generated when the processes run a distributed algorithm. A *partial execution* is a finite prefix of some execution. A process is *correct* in an execution if there are infinite number of events associated with it. Otherwise, the process is *faulty*. Given a set of processes P and an execution e, we denote the set of correct processes in P by $alive(e,P)$, and the set of faulty processes in P by $faulty(e,P)$.

We borrow histories and related definitions from Herlihy and Wing [32]. A *history* h of an execution e is an infinite sequence of operation invocation and response events in the same order as they appear in e. A history may also be defined with respect to a partial execution e'; such a history is a finite sequence. A *subhistory* of a history h is a subsequence of the events of h.

We denote by $ops(h)$ the set of all operations whose invocations appear in h. An invocation is *pending* in a history if no matching response follows the invocation. If h is a history, $complete(h)$ is the maximal subsequence of h consisting only of invocations and matching responses. A history h is *complete* if it contains no pending invocations.

A history h is *sequential* if (1) the first event of h is an invocation and (2) each invocation, except possibly the last, is immediately followed by a matching response and each response is immediately followed by an invocation.

A *process subhistory*, $h|i$, of a history h is the subsequence of all events in h which occurred at p_i. An object subhistory h/o is similarly defined for an object $o \in O$. Two histories h and h' are *equivalent* if $\forall 1 \leq i \leq m.h|i = h'|i$. A history h is *well-formed* if $\forall 1 \leq i \leq m.h|i$ is sequential. We assume all histories are well-formed.

A set S of histories is *prefix-closed* if, whenever h is in S, every prefix of h is also in S. A *single-object* history is one in which all events are associated with the same object. A *sequential specification* for an object $o \in O$ is a prefix-closed set of single-object sequential histories for o. A sequential history h is *legal* if $\forall o \in O.h/o$ belongs to the sequential specification for o.

A history induces an irreflexive partial order on $ops(h)$, denoted $<_h$, as $op_1 <_h op_2$ if and only if $resp(op_1) < inv(op_2)$ in h.

A *quorum system* $\mathcal{Q} \subseteq \mathcal{P}(R)$ over R is a set of subsets of R with the *quorum intersection property*: for all $Q_1, Q_2 \in \mathcal{Q}$, $Q_1 \cap Q_2 \neq \emptyset$. We use *quorum* both to mean a set of replicas in a particular quorum system and the size of such a set.

B.1.2 Shared Objects

A shared object is a data type that supports the following operations:

- READ(): returns the value of the object
- WRITE(v): updates the value of the object to v
- RMW($f(\cdot)$): atomically reads the value v of the object, updates the value to $f(v)$, and returns v

We use $reads(h)$, $writes(h)$, and $rmws(h)$ to denote the set of all operations that are reads, writes, and rmws in $ops(h)$ respectively. We use $updates(h) = writes(h) \cup rmws(h)$ to denote the set of operations which update the state of a shared object in $ops(h)$. We use $observes(h) = reads(h) \cup rmws(h)$ to denote the set of operations which observe the state of a shared object in $ops(h)$.

Definition B.1. *(Shared Object Specification) A sequential object subhistory h/o belongs to the* sequential specification of a shared object *if for each $op \in observes(h/o)$ such that $resp(op) \in h/o$, $resp(op)$ contains the value of the latest preceding operation $u \in updates(h/o)$ or if there is no preceding update, then $resp(op)$ contains the initial value of o.*

B.2 Recovery for RMW Protocol

Algorithms 5 and 6 show the modifications to the basic EPaxos recovery protocol. In addition to the replica state in Figure 4, each replica also maintains *epoch*, the current epoch used in generating ballot numbers, and b, the highest ballot number seen in the current epoch. Each instance in the

Algorithm 5: Recovery coordinator protocol for rmws.

1 **when** *replica* $r \in R$ *suspects replica* $c \in R$ *failed while committing instance* j **do**
2 $\quad ballot \leftarrow (epoch, (b+1), id_r)$
3 $\quad$ **send** $Prepare(ballot, id_c, j)$ to all $r \in R$
4 $\quad$ **wait** to receive $PrepareOK(cmd_r, seq_r, deps_r, base_r, status_r, ballot_r)$ from all $r \in Q \in \mathscr{Q}$
5 $\quad \mathscr{R} \leftarrow \{(cmd_r, seq_r, deps_r, base_r, status_r) \mid \forall r' \in Q : ballot_r \geq ballot_{r'}\}$
6 $\quad$ **if** $(cmd, seq, deps, base, \textbf{\textit{committed}}) \in \mathscr{R}$ **then**
7 $\quad\quad$ **run Commit Phase** for $(cmd, seq, deps, base)$ at (id_c, j)
8 $\quad$ **else if** $(cmd, seq, deps, base, \textbf{\textit{accepted}}) \in \mathscr{R}$ **then**
9 $\quad\quad$ **run Accept Phase** for $(cmd, seq, deps, base)$ at (id_c, j)
10 $\quad$ **else if** $\exists S \subseteq \mathscr{R} :$ $(cmd_c, seq_c, deps_c, base_c, status_c) \notin S) \wedge$ $(|S| \geq \lfloor \frac{n}{2} \rfloor) \wedge$ $(\forall reply_1, reply_2 \in S. reply_1 = reply_2 \wedge reply_1.status = \textbf{\textit{pre-accepted}})$ **then**
11 $\quad\quad$ **run Accept Phase** for $(cmd_r, seq_r, deps_r, base_r) \in S$ at (id_c, j)
12 $\quad$ **else if** $(cmd, seq, deps, base, \textbf{\textit{pre-accepted}}) \in \mathscr{R}$ **then**
13 $\quad\quad$ **run PreAccept Phase** for *cmd* at (id_c, j), avoid fast path
14 $\quad$ **else**
15 $\quad\quad$ **run PreAccept Phase** for *no-op* at (id_c, j), avoid fast path

cmds array also contains a *ballot* number that is only used during recovery.

Note that the only change Gryff makes is that the *base* attribute is recovered along with the *deps* and *seq* attributes. To support optimized EPaxos, similar changes must be made to the optimized recovery protocol. We refer the reader to the optimized recovery protocol description in the EPaxos technical report [48] and our implementation of Gryff [29] for more details.

B.3 Proof of Linearizability

More Definitions. A *consistency condition* is specified by a particular set of schedules. Linearizability [32] is a strong consistency condition that reduces the complexity of building correct applications.

Definition B.2 (Linearizability)**.** *A complete history h satisfies* linearizability *if there exists a legal total order τ of $ops(h)$ such that $\forall op_1, op_2 \in ops(h). op_1 <_h op_2 \implies op_1 <_\tau op_2$.*

Algorithm 6: Recovery replica protocol for rmws.

1 **when** *replica* $r \in R$ *receives a message* m *from* $x \in R$ **do**
2 $\quad$ **case** $m = Prepare(ballot, j, k)$ **do**
3 $\quad\quad$ **if** $ballot > cmds[j][k].ballot$ **then**
4 $\quad\quad\quad cmds[j][k].ballot = ballot$
5 $\quad\quad\quad$ **send** $PrepareOK(cmds[j][k])$ to x
6 $\quad\quad$ **else**
7 $\quad\quad\quad$ **send** *NACK* to x

Given a particular consistency condition, we are interested in whether a system enforces the condition for all possible partial executions.

Definition B.3. *The system* provides consistency condition C *if, for every partial execution e of the system, the history h of e can be extended to some history h' such that $complete(h')$ is in C.*

Unless otherwise noted, the rest of this section considers a complete history h produced by the distributed algorithm specified in Algorithms 1, 2, 3, and 4 in the main body of the paper and Algorithms 5 and 6 in this appendix.

The *coordinator* of a read or write is the invoking process. For rmws, the coordinator is the replica that notifies the invoking process its rmws has been executed. We assume that each $u \in updates(h)$ writes a unique value.

Definition B.4. *A complete operation $op \in observes(h)$* observes an update $u \in updates(h)$ *if the value returned in $resp(op)$ was written by u.*

Definition B.5. *The* carstamp cs_{op} assigned to a complete operation $op \in ops(h)$ *is:*

- *If $op \in writes(h)$, cs_{op} is the carstamp determined on Line 15 of Algorithm 1.*
- *If $op \in rmws(h)$, cs_{op} is the carstamp determined by Property B.4.*
- *If $op \in reads(h)$, cs_{op} is the carstamp cs_u assigned to the update u that op observes.*

Structure. We abstract the implementation details of the rmw protocol into four sufficient properties. The proofs of the subsequent lemmas and theorem assume that the rmw protocol provides these properties. At the end of this subsection, we prove that Gryff's rmw protocol does exactly this.

Property B.1. *(Freshness) Every complete $rmw \in rmws(h)$ is assigned a carstamp such that $\forall Q \in \mathscr{Q}. cs_{rmw} > \min_{r \in Q} cs_r$ where cs_r is the carstamp at r when rmw is invoked.*

Property B.2. *(Propagation) For every complete $rmw \in rmws(h)$ there exists a $Q \in \mathscr{Q}$ such that $\forall r \in Q. cs_r \geq cs_{rmw}$ where cs_r is the carstamp at r when rmw completes.*

Property B.3. *(Uniqueness) For all complete* $rmw_1, rmw_2 \in rmws(h)$, $cs_{rmw_1} \neq cs_{rmw_2}$.

Property B.4. *(Assignment) Every complete* $rmw \in rmws(h)$ *is assigned the carstamp* $cs_{rmw} = (cs_u.ts, cs_u.id, cs_u.rmwc + 1)$ *where u is the update that rmw observes.*

The linearizability proof follows a linear structure. We first prove that the carstamps assigned to each operation respect the real time order of h in Lemmas B.1-B.5. These proofs leverage the quorum intersection property. Then, we prove that a partial order on operations induced by their carstamps respects both the real time order of h and the legality condition for shared objects in Lemmas B.6-B.10. Finally, we connect these lemmas in Theorem B.1 to show that a total order of this partial order satisfies linearizability.

Lemma B.1. *After a replica* $r \in R$ *executes the* APPLY *function with tuple* (v, cs) *and before it executes any other instruction,* $cs_r \geq cs$ *where* cs_r *is the carstamp at r.*

Proof. By the condition on Line 13 of Algorithm 2. □

Lemma B.2. $\forall r \in R, cs_r$ *monotonically increases where* cs_r *is the carstamp at r.*

Proof.

1. cs_r is only modified via the APPLY function.

   PROOF: By the fact that, out of all of the replica pseudocode in Algorithms 2, 3, 4, 5, and 6, the APPLY function in Algorithm 2 is the only place that cs_r is assigned a value.

2. Q.E.D.

   PROOF: By Lemma B.1 and 1. □

Lemma B.3. *If an operation* $op \in ops(h)$ *is complete, then after* $resp(op)$ *there exists a* $Q \in \mathcal{Q}$ *such that* $\forall r \in Q.cs_r \geq cs_{op}$ *where* cs_r *is the carstamp at r.*

Proof.

1. Let op be an operation in $ops(h)$
2. CASE: $op \in writes(h)$

   2.1. Let $Q \in \mathcal{Q}$ be the quorum from which the coordinator of op receives *Write2Reply* messages.

   PROOF: By the hypothesis that op is complete and the requirement that the coordinator of op waits to receive *Write2Reply* messages from a quorum before completing op (Line 17 of Algorithm 1).

   2.2. Each $r \in Q$ received a *Write2* message for op containing (v, cs_{op}) where v is the value written by op.

   PROOF: By 2.1 and that a replica sends a *Write2Reply* message for op to the coordinator of op only if it receives a *Write2* message for op containing (v, cs_{op}).

   2.3. Each $r \in Q$ applied (v, cs_{op}) before sending a *Write2Reply* message for op.

   PROOF: By 2.1, 2.2, and the requirement that a replica sends a *Write2Reply* message after it applies the tuple it received in a *Write2* message (Line 10 of Algorithm 2).

   2.4. Q.E.D.

   By Lemma B.1, Lemma B.2, and 2.3.

3. CASE: $op \in reads(h)$

   3.1. CASE: op completed after Read Phase 1 (Line 7 of Algorithm 1).

   3.1.1. Let $Q \in \mathcal{Q}$ be the quorum from which the coordinator of op receives *Read1Reply* messages.

   PROOF: By the hypothesis that op is complete and the requirement that the coordinator of op waits to receive *Read1Reply* messages from a quorum before completing op (Line 3 of Algorithm 1).

   3.1.2. When each $r \in Q$ sent their *Read1Reply* message, $cs_r = cs_{op}$ where cs_r is the carstamp at r.

   PROOF: By 3.1.1, Definition B.5, the case 3.1 assumption, and the fast read condition (Line 6 of Algorithm 1).

   3.1.3. Q.E.D.

   PROOF: By Lemma B.2 and 3.1.2.

   3.2. CASE: op completed after Read Phase 2 (Line 10 of Algorithm 1).

   3.2.1. Let $Q \in \mathcal{Q}$ be the quorum from which the coordinator of op receives *Read2Reply* messages.

   PROOF: By the hypothesis that op is complete, the case 3.2 assumption, and the requirement that the coordinator of op waits to receive *Read2Reply* messages from a quorum before completing op in Read Phase 2 (Line 9 of Algorithm 1).

   3.2.2. Each $r \in Q$ received a *Read2* message for op containing (v, cs_{op}) where v is the value written by op.

   PROOF: By 3.2.1 and that a replica sends a *Read2Reply* message for op to the coordinator of op only if it receives a *Read2* message for op containing (v, cs_{op}).

   3.2.3. Each $r \in Q$ applied (v, cs_{op}) before sending a *Read2Reply* message.

   PROOF: By 3.2.1, 3.2.2, and the requirement that a replica sends a *Read2Reply* message after it applies

the tuple it received in a *Read2* message (Line 5 of Algorithm 2).

3.2.4. Q.E.D.

By Lemma B.1, Lemma B.2, and 3.2.3.

4. CASE: $op \in rmws(h)$

PROOF: By Property B.2.

5. Q.E.D.

PROOF: By 1, 2, 3, and 4. □

Lemma B.4. *For all operations $op \in ops(h)$ and updates $u \in updates(h)$, $op <_h u \implies cs_{op} < cs_u$.*

Proof.

1. Let $Q_{op} \in \mathcal{Q}$ be a quorum such that $\forall r \in Q_{op}.cs_r \geq cs_{op}$ where cs_r is the carstamp at r when u is invoked.

   PROOF: By the hypothesis that *op* completed before u was invoked and Lemma B.3.

2. Let u be an update in $updates(h)$.
3. CASE: $u \in writes(h)$

   3.1. Let $Q_u \in \mathcal{Q}$ be the quorum from which the coordinator of u receives *Write1Reply* messages and cs_{max} be the largest carstamp contained in these messages.

   PROOF: By the hypothesis that u is complete and the requirement that the coordinator of u waits to receive *Write1Reply* messages from a quorum before completing u (Line 13 of Algorithm 1).

   3.2. Let $r \in Q_{op} \cap Q_u$ be a replica.

   PROOF: By 1, 3.1, and the Quorum Intersection property.

   3.3. *op* completed before r received a *Write1* message for u.

   PROOF: By the hypothesis that *op* completed before u was invoked and 3.2.

   3.4. The *Write1Reply* message that r sent for u contains a carstamp $cs_r \geq cs_{op}$.

   PROOF: By 1 and 3.3.

   3.5. The coordinator for u assigns u the carstamp $cs_u = (cs_{max}.ts + 1, id, 0)$ where $cs_{max} \geq cs_r$ and id is the id of the coordinator for u.

   PROOF: By 3.1 and the assignment of a carstamp to u (Lines 14 and 15 of Algorithm 1).

   3.6. Q.E.D.

   PROOF: By 3.4, and 3.5.

4. CASE: $u \in rmws(h)$

   PROOF: By 1 and Property B.1.

5. Q.E.D.

   PROOF: By 2, 3, and 4. □

Lemma B.5. *For all operations $op \in ops(h)$ and reads $\rho \in reads(h)$, $op <_h \rho \implies cs_{op} \leq cs_\rho$.*

Proof.

1. Let u be the update that ρ observes.

   PROOF: By the hypothesis that ρ is complete and Definition B.4.

2. CASE: $u = op$

   2.1. $cs_\rho = cs_u = cs_{op}$

   PROOF: By the assumption of case 2, 1, and Definition B.5.

   2.2. Q.E.D.

   PROOF: By 2.1.

3. CASE: $u \neq op$

   3.1. Let $Q_{op} \in \mathcal{Q}$ be a quorum such that $\forall r \in Q_{op}.cs_r \geq cs_{op}$ where cs_r is the carstamp at r when ρ is invoked.

   PROOF: By the hypothesis that *op* completed before ρ was invoked and Lemma B.3.

   3.2. Let $Q_\rho \in \mathcal{Q}$ be the quorum from which the coordinator of ρ receives *Read1Reply* messages and cs_{max} be the largest carstamp contained in these messages.

   PROOF: By the hypothesis that ρ is complete and the requirement that the coordinator of ρ waits to receive *Read1Reply* messages from a quorum before completing ρ (Line 3 of Algorithm 1).

   3.3. Let $r \in Q_{op} \cap Q_\rho$ be a replica.

   PROOF: By 3.1, 3.2, and the Quorum Intersection property.

   3.4. *op* completed before r received a *Read1* message for ρ.

   PROOF: By the hypothesis that *op* completed before u was invoked and 3.3.

   3.5. The *Read1Reply* message that r sent for ρ contains a carstamp $cs_r \geq cs_{op}$.

   PROOF: By 3.1 and 3.4.

   3.6. The coordinator for ρ chooses u to be the update corresponding to cs_{max}.

PROOF: By 3.2 and the selection of an update to observe for ρ (Lines 4 and 5 of Algorithm 1).

3.7. Q.E.D.

PROOF: By 3.5 and 3.6.

4. Q.E.D.

PROOF: By 1, 2, and 3. □

We define the relation $<_\psi$ on $ops(h)$ as follows:

- $\forall op_1, op_2 \in ops(h).\, cs_{op_1} < cs_{op_2} \implies op_1 <_\psi op_2$.
- $\forall \rho \in reads(h)$ such that ρ observes an update $u \in updates(h)$, $u <_\psi r$. $\forall u' \in updates(h)$ such that $u <_\psi u'$, $r <_\psi u'$.
- $\forall \rho_1, \rho_2 \in reads(h)$ such that ρ_1 and ρ_2 observe the same update u, $inv(\rho_1) < inv(\rho_2) \implies \rho_1 <_\psi \rho_2$.
- $\forall op_1, op_2, op_3 \in ops(h).\, op_1 <_\psi op_2 \wedge op_2 <_\psi op_3 \implies op_1 <_\psi op_3$.

Less formally, $<_\psi$ orders operations by their carstamps and inserts reads in between the updates that the reads observe and subsequent updates.

Lemma B.6. *For all $u_1, u_2 \in updates(h)$, $u_1 <_h u_2 \implies u_1 <_\psi u_2$.*

Proof.

1. $cs_{u_1} < cs_{u_2}$.

   PROOF: By the hypothesis that $u_1 <_h u_2$ and Lemma B.4.

2. Q.E.D.

   PROOF: By 1 and the definition of $<_\psi$. □

Lemma B.7. *For all $u \in updates(h)$ and $\rho \in reads(h)$, $u <_h \rho \implies u <_\psi \rho$.*

Proof.

1. $cs_u \leq cs_\rho$.

   PROOF: By the hypothesis that $u <_h \rho$ and Lemma B.5.

2. CASE: $cs_u < cs_\rho$.

   PROOF: By the definition of $<_\psi$.

3. CASE: $cs_u = cs_\rho$.

   3.1. ρ observes u

   PROOF: By the assumption of case 3 and Definition B.4.

   3.2. Q.E.D.

   PROOF: By 3.1 and the definition of $<_\psi$.

4. Q.E.D.

   PROOF: By 1, 2, and 3. □

Lemma B.8. *For all $\rho \in reads(h)$ and $u \in updates(h)$, $\rho <_h u \implies \rho <_\psi u$.*

Proof.

1. $cs_\rho < cs_u$.

   PROOF: By the hypothesis that $\rho <_h u$ and Lemma B.4.

2. Q.E.D.

   PROOF: By 1 and the definition of $<_\psi$. □

Lemma B.9. *For all $\rho_1, \rho_2 \in reads(h)$, $\rho_1 <_h \rho_2 \implies \rho_1 <_\psi \rho_2$.*

Proof.

1. $cs_{\rho_1} \leq cs_{\rho_2}$.

   PROOF: By the hypothesis that $\rho_1 <_h \rho_2$ and Lemma B.5.

2. CASE: $cs_{\rho_1} < cs_{\rho_2}$

   PROOF: By the definition of $<_\psi$.

3. CASE: $cs_{\rho_1} = cs_{\rho_2}$

   3.1. $resp(\rho_1) < inv(\rho_2)$

   PROOF: By the hypothesis that $\rho_1 <_h \rho_2$.

   3.2. $inv(\rho_1) < inv(\rho_2)$

   PROOF: By 3.1.

   3.3. Q.E.D.

   PROOF: By 3.2 and the definition of $<_\psi$.

4. Q.E.D.

   PROOF: By 1, 2, and 3. □

Lemma B.10. *If τ is a topological sort of $<_\psi$, τ is a legal total order of $ops(h)$.*

Proof.

1. Let $op \in observes(h)$ be an operation that observes an update $u \in updates(h)$.

   PROOF: By the hypothesis that op is completed.

2. CASE: $op \in reads(h)$.

   2.1. There is no u' such that $u <_\psi u' <_\psi op$.

   PROOF: By the assumption of case 2 and the definition of $<_\psi$.

   2.2. There is no u' such that $u <_\tau u' <_\tau op$.

PROOF: By the hypothesis that τ is a topological sort of $<_\psi$ and 2.1.

2.3. Q.E.D.

PROOF: By 2.2, the definition of legal, and Definition B.1.

3. CASE: $op \in rmws(h)$.

3.1. $cs_{op} = (cs_u.ts, cs_u.id, cs_u.rmwc + 1)$.

PROOF: By Property B.4.

3.2. SUFFICES ASSUME: $\exists u' \in updates(h)$ with carstamp $cs_{u'}$ such that $u <_\psi u' <_\psi op$.
PROVE: False.

3.2.1. $cs_u < cs_{u'} < cs_{op}$.

PROOF: By assumption 3.2 and the definition of $<_\psi$.

3.2.2. CASE: $cs_u.ts < cs_{u'}.ts$

3.2.2.1. $cs_{op}.ts < cs_{u'}.ts$.

PROOF: By the assumption of case 3.2.2 and 3.1.

3.2.2.2. Q.E.D.

PROOF: By 3.2.2.1 and 3.2.1.

3.2.3. CASE: $cs_u.ts = cs_{u'}.ts$ and $cs_u.id < cs_{u'}.id$.

3.2.3.1. $cs_{op}.ts = cs_{u'}.ts$ and $cs_{op}.id < cs_{u'}.id$.

PROOF: By the assumption of case 3.2.3 and 3.1.

3.2.3.2. Q.E.D.

PROOF: By 3.2.3.1 and 3.2.1.

3.2.4. CASE: $cs_u.ts = cs_{u'}.ts$, $cs_u.id = cs_{u'}.id$, and $cs_u.rmwc < cs_{u'}.rmwc$.

3.2.4.1. $cs_{op}.ts = cs_{u'}.ts$ and $cs_{op}.id = cs_{u'}.id$.

PROOF: By the assumption of case 3.2.4 and 3.1.

3.2.4.2. $cs_{op}.rmwc = cs_u.rmwc + 1 \leq cs_{u'}.rmwc$.

PROOF: By the assumption of case 3.2.4 and 3.1 and that the *rmwc* component of a carstamp is a natural number.

3.2.4.3. CASE: $u' \in writes(h)$

3.2.4.3.1. $cs_{u'}.rmwc = 0$

PROOF: By the assignment of a carstamp to u' (Lines 14 and 15 of Algorithm 1).

3.2.4.3.2. Q.E.D.

PROOF: By 3.2.4.3.1 and 3.2.4.2.

3.2.4.4. CASE: $u' \in rmws(h)$

3.2.4.4.1. $cs_{op}.rmwc \neq cs_{u'}.rmwc$.

PROOF: By 3.2.4.1 and Property B.3.

3.2.4.4.2. $cs_{op}.rmwc < cs_{u'}.rmwc$.

PROOF: By 3.2.4.4.1 and 3.2.4.2.

3.2.4.4.3. Q.E.D.

PROOF: By 3.2.4.1, 3.2.4.4.2, and 3.2.1.

3.2.4.5. Q.E.D.

PROOF: By 3.2.4.3 and 3.2.4.4.

3.3. Q.E.D.

PROOF: By 3.2, the definition of legal, and Definition B.1.

4. Q.E.D.

PROOF: By 1, 2, and 3. □

Theorem B.1. *The system implements a shared object with linearizability.*

Proof. Consider a partial execution e with history h. Let h' be h with a response for each pending operation in $updates(h)$ appended to h. Let $h'' = complete(h')$.

1. Let op_1 and op_2 be operations in $ops(h'')$. We prove that $op_1 <_h op_2 \implies op_1 <_\psi op_2$.
2. CASE: $op_1, op_2 \in updates(h'')$.

PROOF: By Lemma B.6.

3. CASE: $op_1 \in updates(h'')$ and $op_2 \in reads(h'')$.

PROOF: By Lemma B.7.

4. CASE: $op_1 \in reads(h'')$ and $op_2 \in updates(h'')$.

PROOF: By Lemma B.8.

5. CASE: $op_1, op_2 \in reads(h'')$.

PROOF: By Lemma B.9.

6. Let τ be a topological sort of $<_\psi$ on $ops(h'')$.
7. τ is a legal total order on $ops(complete(h'))$.

PROOF: By 6 and Lemma B.10.

8. Q.E.D.

PROOF: By 1, 2, 3, 4, 5, and 7. □

RMW Properties. In order to prove that Gryff's rmw protocol provides the aforementioned properties, we rely on the correctness of EPaxos [48]. Because replicas act as coordinators for a rmw invoked by other processes, the failure of a replica during a rmw before the invoking process learns of the result may cause the invoking process to submit its rmw to another replica. Replicas must be able to recognize duplicates, only execute the rmw once, and store the result until the invoking process generates a response event.

This issue affects all protocols that rely on a subset of processes to coordinate the execution of operations on behalf of other processes. In Gryff, if a process learns that a pending rmw has been executed by at least one replica, it must ensure that a quorum have executed the rmw before completing it. A replica can ensure this by sending *Commit* messages with the appropriate attributes to all replicas. Replicas that receive *Commit* messages for a rmw they have already executed can immediately reply with an *Executed* message. For brevity, we omit the duplicate execution check for a replica receiving a rmw in Algorithm 3 and assume that if a replica has already executed a rmw, it will skip to Line 20 of Algorithm 3.

We assume the use of the majority quorum system $\mathcal{Q}_{maj}$ such that $\forall Q \in \mathcal{Q}_{maj}. |Q| = \lfloor \frac{n}{2} \rfloor + 1$. This assumption implies each quorum is a subset of a fast quorum and equivalent to a slow quorum in canonical EPaxos.

Definition B.6. *A command γ is* committed *at a replica $r \in R$ if the cmds array at r contains an instance with γ as the command and* ***committed*** *as the status.*

Lemma B.11. *The system provides Property B.1.*

Proof. Let *rmw* be an operation in $rmws(h)$ and $Q \in \mathcal{Q}$ be a quorum.

1. *rmw* committed with attributes that are the union of the attributes computed by each $r \in S$ where $S \supseteq Q'$ for some $Q' \in \mathcal{Q}$.

   1.1. *rmw* commits with basic EPaxos or with optimized EPaxos.

   1.2. CASE: *rmw* commits with basic EPaxos.

   PROOF: By Step 1.1 of the proof of Theorem 4 in the EPaxos technical report, which states that *rmw* is committed with the union of attributes from $\lfloor \frac{n}{2} \rfloor + 1$ replicas, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

   1.3. CASE: *rmw* commits with optimized EPaxos.

   There are two sub-cases:

   1.3.1. CASE: *rmw* commits without running the recovery procedure.

   PROOF: By 1.2 and that a fast quorum in optimized EPaxos is larger than a majority quorum because this case reduces to 1.2 with the fast quorum size reduced from $n-1$.

   1.3.2. CASE: *rmw* commits through the optimized recovery procedure.

   1.3.2.1. CASE: *rmw* commits before step 7 of the optimized recovery procedure, or after exiting one of the Else branches in step 7.

   PROOF: By Step 2.1 of Theorem 7 of the EPaxos technical report, which states that *rmw* must have been pre-accepted by a majority of replicas, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

   1.3.2.2. CASE: *rmw* committed after exiting the optimized recovery procedure on the If branch in step 7.

   PROOF: By Step 2.2.2 of Theorem 7 of the EPaxos technical report, which states that *rmw* must have been pre-accepted by a majority of replicas, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

   1.3.2.3. Q.E.D.

   PROOF: By 1.3.2.1 and 1.3.2.2.

   1.3.3. Q.E.D.

   PROOF: By 1.3.1 and 1.3.2.

   1.4. Q.E.D.

   PROOF: By 1.1, 1.2, and 1.3.

2. The *base* attribute of *rmw* is chosen such that $base.cs \geq \max_{r \in S} cs_r \geq \max_{r \in Q'} cs_r$ where cs_r is the carstamp at r when *rmw* is invoked.

   2.1. *rmw* committed after the PreAccept Phase or the Accept Phase. Note that the basic recovery procedure and optimized recovery procedure always exit by running the PreAccept, Accept, or Commit phase. Each of these is reducible to committing after the PreAccept phase or Accept phase.

   2.2. CASE: *rmw* committed after the PreAccept Phase (Line 12 of Algorithm 3).

   2.2.1. When each $r \in S$ sent their *PreAcceptOK* message, $cs_r = base.cs$ where cs_r is the carstamp at r.

   PROOF: By 1, the case 2.2 assumption, and the fast path condition (Line 10 of Algorithm 3).

   2.2.2. Q.E.D.

   PROOF: By Lemma B.2 and 2.2.1.

   2.3. CASE: *rmw* committed after the Accept Phase.

   2.3.1. CASE: The Accept phase is run during normal processing.

   PROOF: By Lemma B.2 and the selection of *base* in the Accept Phase (Line 15 of Algorithm 3).

   2.3.2. CASE: The Accept phase is run during recovery (either basic or optimized).

   PROOF: By the fact that the recovery procedures exit directly to the Accept phase only if *rmw* has previously been pre-accepted by a majority.

   2.4. Q.E.D.

PROOF: By 2.1, 2.2, and 2.3.

3. $base.cs \geq \min_{r \in Q} cs_r$.

   PROOF: By 2 and the Quorum Intersection property ($\max_{r \in Q'} cs \geq \min_{r \in Q \cap Q'} cs_r \geq \min_{s \in Q'} cs_r$).

4. $cs_{rmw} > base.cs$.

   PROOF: By the generation of the carstamp of *rmw* (Line 18 of Algorithm 4).

5. Q.E.D.

   PROOF: By 3 and 4. □

Lemma B.12. *The system provides Property B.2.*

Proof. Let *rmw* be an operation in $rmws(h)$.

1. After *rmw* completes, $\exists Q \in \mathcal{Q}$ such that each $r \in Q$ has executed *rmw*.

   PROOF: By the hypothesis that *rmw* is complete and the requirement that the coordinator only completes *rmw* when it has received *Executed* messages from a quorum (Line 21 of Algorithm 3).

2. Each $r \in Q$ applied cs_{rmw}.

   PROOF: By 1 and that a replica only sends an *Executed* message for *rmw* if it has applied the carstamp and value of *rmw* (Line 20 of Algorithm 3).

3. Q.E.D.

   PROOF: By 2, Lemma B.1, and Lemma B.2. □

Lemma B.13. *The system provides Property B.3.*

Proof. Let rmw_a and rmw_b be operations in $rmws(h)$.

1. Either rmw_a is executed before rmw_b or vice versa.

   PROOF: By Theorem 4 and Theorem 7 from the EPaxos technical report, that the logic for determining the *deps* and *seq* attributes of a command remains unchanged from EPaxos, and that the logic for determining the execution order of commands remains unchanged from EPaxos.

2. CASE: rmw_a is executed before rmw_b.

   2.1. $cs_{rmw_a} < cs_{rmw_b}$.

   2.1.1. For any two interfering commands rmw_a and rmw_b, there is a sequence of zero of more interfering commands that are executed between rmw_a and rmw_b. Let this sequence be $rmw_a = rmw_1, ..., rmw_k = rmw_b$.

   PROOF: By Theorem 4 and Theorem 7 from the EPaxos technical report.

   2.1.2. Proof by induction on the sequence $rmw_1, ..., rmw_k$.

   2.1.2.1. Base case: $k = 2$ (rmw_2 immediately follows rmw_1).

   2.1.2.1.1. $prev.cs = cs_{rmw_1}$.

   PROOF: By the assumption of the base case 2.1.2.1 and that *prev* is only modified when a rmw is executed (Line 19 of Algorithm 4).

   2.1.2.1.2. $cs_{rmw_2} > prev.cs$.

   PROOF: By the generation of cs_{rmw_2} to be larger than *prev* at the time that rmw_2 is executed (Lines 15, 16, and 18 of Algorithm 4).

   2.1.2.1.3. Q.E.D.

   PROOF: By 2.1.2.1.1 and 2.1.2.1.2.

   2.1.2.2. ASSUME: $cs_{rmw_1} < cs_{rmw_i}$.
   PROVE: $cs_{rmw_1} < cs_{rmw_{i+1}}$.

   2.1.2.2.1. $prev.cs = cs_{rmw_i}$.

   PROOF: By the assumption that rmw_i was the last rmw to be executed and that *prev* is only modified when a rmw is executed (Line 19 of Algorithm 4).

   2.1.2.2.2. $cs_{rmw_{i+1}} > prev.cs$

   PROOF: By the generation of $cs_{rmw_{i+1}}$ to be larger than *prev* at the time that rmw_{i+1} is executed (Lines 15, 16, and 18 of Algorithm 4).

   2.1.2.2.3. Q.E.D.

   PROOF: By 2.1.2.2.1 and 2.1.2.2.2.

   2.1.3. Q.E.D.

   PROOF: By 2.1.1 and 2.1.2.

   2.2. Q.E.D.

   PROOF: By 2.1.

3. CASE: rmw_2 is executed before rmw_1.

   PROOF: By symmetry with case 2.

4. Q.E.D.

   PROOF: By 1, 2, and 3. □

Lemma B.14. *The system provides Property B.4.*

Proof. Let *rmw* be an operation in $rmws(h)$.

1. Let $u \in updates(h)$ be the update that *rmw* observes.

   PROOF: By the assumption that *rmw* is complete.

2. Let cs_u be the carstamp chosen on Lines 14 and 16 of Algorithm 4.

PROOF: By 1 and Definition B.4.

3. Q.E.D.

PROOF: By 2, Definition B.5, and the generation of cs_{rmw} (Line 18 of Algorithm 4). □

Lemmas B.11, B.12, B.13, and B.14 imply that Gryff's rmw protocol satisfies the assumptions needed to prove Theorem B.1.

B.4 Proof of Wait-Freedom

More Definitions. Wait-freedom is a strong liveness property that guarantees a correct process can always make progress regardless of concurrent operations invoked by other processes.

Definition B.7. *(Wait-Freedom) A subset $S \subseteq ops(h)$ of operations are* wait-free *in a history h with execution e if $\forall op \in S.process(inv(op)) \in alive(e,P) \implies resp(op) \in h$.*

Unless otherwise noted, the rest of this section considers an execution e with history h produced by the distributed algorithm specified in Algorithms 1, 2, 3, and 4 in the main body of the paper and Algorithms 5 and 6 in this appendix.

We assume that there are $n = 2f+1$ replicas and that up to f replicas may fail and any number of other processes may fail in e. Thus, we assume the use of the majority quorum system $\mathscr{Q}_{maj}$ such that $\forall Q \in \mathscr{Q}_{maj}. |Q| = f+1$.

Structure. We first prove that Gryff's reads and writes are wait-free in Theorems B.2 and B.3. To prove wait-freedom for rmws, we discuss why the synchrony assumption must be strengthened from asynchrony to partial synchrony. With this stronger assumption, we restate the liveness property of EPaxos and use this to prove that Gryff's rmws are wait-free in Theorem B.5.

Theorem B.2. *The system provides read wait-freedom.*

Proof. 1. Let op be an operation in $reads(h)$.
2. The coordinator of op is correct.

PROOF: By the definition of a coordinator of a read and by the hypothesis that $process(inv(op)) \in alive(e,P)$.

3. $|alive(e,R)| \geq f+1$

PROOF: By the assumption that at most f out of $2f+1$ replicas can fail in any execution.

4. The coordinator sends a *Read1* message for op to every replica $r \in R$.

PROOF: By 2 and Line 2 of Algorithm 1.

5. Each $r \in alive(e,R)$ delivers a *Read1* message for op.

PROOF: By 4, the assumption that $r \in alive(e,R)$, and the assumption that the network guarantees eventual reliable message delivery.

6. Each $r \in alive(e,R)$ sends a *Read1Reply* message for op to the coordinator.

PROOF: By 5, the assumption that $r \in alive(e,R)$, and that the message handler for a *Read1* message contains no blocking instructions or conditional branches (Algorithm 2).

7. The coordinator delivers *Read1Reply* messages from a quorum $Q \in \mathscr{Q}$.

PROOF: By 2, 3, 6, the assumption that the network guarantees eventual reliable message delivery, and the assumption that the majority quorum system $\mathscr{Q}_{maj}$ is used.

8. CASE: $\forall r \in Q. cs_r = cs_{max}$

PROOF: By 7, the assumption of the case and that the coordinator generates $resp(op)$ when this assumption holds (Lines 6 and 7 of Algorithm 1).

9. CASE: $\exists r \in Q : cs_r \neq cs_{max}$

9.1. The coordinator sends a *Read2* message for op to every replica $r \in R$.

PROOF: By 2, the assumption of the case, and Line 8 of Algorithm 2.

9.2. Each $r \in alive(e,R)$ delivers a *Read2* message for op.

PROOF: By 9.1, the assumption that $r \in alive(e,R)$, and the assumption that the network guarantees eventual reliable message delivery.

9.3. Each $r \in alive(e,R)$ sends a *Read2Reply* message for op to the coordinator.

PROOF: By 9.2, the assumption that $r \in alive(e,R)$, and that the message handler for a *Read2* message contains no blocking instructions or conditional branches on sending a reply (Algorithm 2).

9.4. The coordinator delivers *Read2Reply* messages from a quorum $Q \in \mathscr{Q}$.

PROOF: By 2, 3, 9.3, the assumption that the network guarantees eventual reliable message delivery, and the assumption that the majority quorum system $\mathscr{Q}_{maj}$ is used.

9.5. Q.E.D.

PROOF: By 7, 9.4, and the fact that the coordinator generates a $resp(op)$ after receiving a quorum of *Read2Reply* messages (Line 10 of Algorithm 1).

10. Q.E.D.

PROOF: By 7, 8, and 9. □

Theorem B.3. *The system provides write wait-freedom.*

Proof. 1. Let *op* be an operation in $writes(h)$.

2. The coordinator of *op* is correct.

   PROOF: By the definition of a coordinator of a write and by the hypothesis that $process(inv(op)) \in alive(e,P)$.

3. $|alive(e,R)| \geq f+1$

   PROOF: By the assumption that at most f out of $2f+1$ replicas can fail in any execution.

4. The coordinator sends a *Write1* message for *op* to every replica $r \in R$.

   PROOF: By 2 and Line 12 of Algorithm 2.

5. Each $r \in alive(e,R)$ delivers a *Write1* message for *op*.

   PROOF: By 4, the assumption that $r \in alive(e,R)$, and the assumption that the network guarantees eventual reliable message delivery.

6. Each $r \in alive(e,R)$ sends a *Write1Reply* message for *op* to the coordinator.

   PROOF: By 5, the assumption that $r \in alive(e,R)$, and that the message handler for a *Write1* message contains no blocking instructions or conditional branches (Algorithm 2).

7. The coordinator delivers *Write1Reply* messages from a quorum $Q \in \mathcal{Q}$.

   PROOF: By 2, 3, 6, the assumption that the network guarantees eventual reliable message delivery, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

8. The coordinator sends a *Write2* message for *op* to every replica $r \in R$.

   PROOF: By 2, 7, and Line 16 of Algorithm 2.

9. Each $r \in alive(e,R)$ delivers a *Write2* message for *op*.

   PROOF: By 8, the assumption that $r \in alive(e,R)$, and the assumption that the network guarantees eventual reliable message delivery.

10. Each $r \in alive(e,R)$ sends a *Write2Reply* message for *op* to the coordinator.

    PROOF: By 9, the assumption that $r \in alive(e,R)$, and that the message handler for a *Write2* message contains no blocking instructions or conditional branches on sending a reply (Algorithm 2).

11. The coordinator delivers *Write2Reply* messages from a quorum $Q \in \mathcal{Q}$.

    PROOF: By 2, 3, 10, the assumption that the network guarantees eventual reliable message delivery, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

12. Q.E.D.

    PROOF: By 11 and the fact that the coordinator generates a $resp(op)$ after receiving a quorum of *Write2Reply* messages (Algorithm 1). □

Note that Theorems B.2 and B.3 rely on our weak network assumption that messages are eventually delivered and do not require any stronger assumptions about the synchrony of the system. Eventual message delivery only precludes infinitely long partitions in the network, which is unlikely to occur in any practical system.

RMW Wait-Freedom. The FLP impossibility result implies that no consensus protocol can provide both safety and liveness in asynchronous systems where processes can fail [24]. Because rmw can solve consensus [31], this also implies that no rmw protocol can provide both.

The rest of this section shows that Gryff's rmw protocol provides wait-freedom if we relax the system model from asynchrony to partial synchrony [21]. In the partial synchrony model, there are two bounds Δ and Φ such that after some unknown point in time during an execution of the system, all messages are delivered within Δ time of when they are sent and all correct processes take at most Φ time between the execution of instructions.

As in the proof of linearizability, we rely on the correctness of EPaxos in the partial synchrony model [48].

Theorem B.4. *EPaxos guarantees with high probability that every proposed command will eventually be committed by every $r \in alive(e,R)$ as long as messages eventually reach their destination before their recipient times out.*

Lemma B.15. *With high probability, every $r \in alive(e,R)$ executes every rmw that commits.*

Proof. Let r be a correct replica, *rmw* be an operation in $rmws(h)$, and D be the transitive closure of the set of dependencies for *rmw* determined by the commit protocol.

1. With high probability, every $rmw' \in D$ eventually commits at r.

   PROOF: By Theorem B.4.

2. With high probability, every $rmw' \in D$ is executed at r. Proof by generalized induction on D.

   2.1. Base case: $rmw_0 \in D$ is the first rmw committed in e.

   PROOF: By the assumption that r is correct, the assumption of the base case 2.1, and that the EPaxos execution

algorithm contains no blocking instructions for commands with no dependencies.

2.2. ASSUME: For any $rmw'' \in D$ such that rmw'' is before rmw' in the EPaxos execution order, rmw'' is executed at r.
PROVE: rmw' is executed at r

PROOF: By the assumption that r is correct, the induction hypothesis 2.2, and that the EPaxos execution algorithm only blocks the execution of a command until all of its dependencies have executed.

2.3. Q.E.D.

By 1, 2.1, and 2.2.

3. With high probability, after all $rmw' \in D$ have executed, rmw will be executed.

3.1. CASE: rmw is in its own strongly connected component in the dependency graph.

PROOF: By the execution order specified by the EPaxos execution algorithm, which requires every dependency of a command to be executed before the command is executed.

3.2. CASE: rmw is in a cycle in the dependency graph.

PROOF: By the execution order specified by the EPaxos execution algorithm, which requires that cycles be broken in order of *seq*, and the fact that rmw may be executed before some of its dependencies within the same cycle.

3.3. Q.E.D.

PROOF: By 3.1 and 3.2.

4. Q.E.D.

PROOF: By 2 and 3. □

Theorem B.5. *If there is a point in time after which the system is synchronous with bounds Δ and Φ, the system provides rmw wait-freedom with high probability.*

Proof. Let op be an operation in $rmws(h)$.
1. $|alive(e,R)| \geq f+1$

PROOF: By the assumption that at most f out of $2f+1$ replicas can fail in any execution.

2. With high probability, every $r \in alive(e,R)$ commits an instance containing op.

PROOF: By the hypothesis that there is a finite time after which all messages are delivered within Δ time of when they are sent and Theorem B.4.

3. With high probability, every $r \in alive(e,R)$ executes op.

PROOF: By 2 and Lemma B.15.

4. With high probability, every $r \in alive(e,R)$ sends an *Executed* message for op to the coordinator.

By 3 and that there are no blocking instructions or conditional branches on sending an *Executed* message in the EXECUTE function.

5. With high probability, the coordinator delivers an *Executed* message for op from a quorum $Q \in \mathcal{Q}$.

PROOF: By 1, 4, the assumption that the network guarantees eventual reliable message delivery, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

6. Q.E.D.

PROOF: By 5 and the fact that the coordinator generates a $resp(op)$ after receiving a quorum of *Executed* messages. □

B.5 Read Proxy Correctness

Algorithm 7: The modified read coordinator protocol and *Read1* message handler for using the read proxy optimization.

```
1  procedure Coordinator::READ(v, cs) at p ∈ P
2      send Read1(v, cs) to all r ∈ R
3      wait to receive Read1Reply(v_r, cs_r) from all
         r ∈ Q ∈ 𝒬
4      for r ∈ Q do
5          APPLY(v_r, cs_r)
6      cs_max ← max_{r∈Q} cs_r
7      v ← v_r : cs_r = cs_max
8      if ∀r ∈ Q : cs_r = cs_max then
9          return v
10     send Read2(v, cs_max) to all r ∈ R
11     wait to receive Read2Reply from all r ∈ Q' ∈ 𝒬
12     return v
13 when replica r ∈ R receives a message m from p ∈ P do
14     case m = Read1(v', cs') do
15         APPLY(v', cs')
16         send Read1Reply(v, cs) to p
```

The pseudocode for the read proxy optimization described in Section 5 is in Algorithm 7. We briefly argue that the optimization does not change the correctness proofs.

The optimization changes the definition of the coordinator of a read from the invoking process to the replica that notifies the invoking process of the result of the read. Neither the definition change nor the added logic for the optimization affect the proof of linearizability because the value that a read observes is still chosen to be the one associated with

the maximum carstamp on a quorum. Reads can be executed multiple times without affecting the state of the shared object, so it is safe for a client to timeout after a finite time t and forward its read to another replica if it suspects the initial coordinator failed.

The proof of wait-freedom for reads remains the same, but needs a small clarification in the proof of Step 2. Since at most f replicas can fail, a client will eventually forward its read to a correct replica that will complete the read coordinator protocol. This will happen after at most $f \cdot t$ time, which is finite.

C Non-Linearizable AQS Execution

AQS [8] attempts to exploit the same observation that Gryff does about the relationship between shared register and consensus protocols to improve performance under the Byzantine failure model. In Figure 15, we demonstrate an explicit execution of AQS that exhibits non-linearizable behavior.

Here, process p_1 first issues and completes w_1 with $ts = (1,1)$ that is seen by all replicas (Figure 15.1). After this write has completed, process p_2 begins w_2 and sees w_1 with $ts = (1,1)$, so it chooses $ts = (2,2)$ for w_2 (Figure 15.2). This write then pauses, and process p_3 issues rmw_3 to primary s_4. The primary gathers state from all replicas and picks $base\_state = \langle w_1, ts = (1,1)\rangle$ (Figure 15.3). The primary then generates an updated state v_1 based on w_1 and sends PRE-PREPARE messages to all replicas. These messages are accepted by all replicas because w_1 is the most recent state they have observed (Figure 15.4). All replicas then broadcast PREPARE messages to all other replicas, and the messages are received and accepted. All replicas then broadcast COMMIT messages (Figure 15.5) and rmw_3 pauses. Process p_2 now finishes w_2 by sending out a second round of messages with $ts = (2,2)$, and all replicas accept and apply this write (Figure 15.6). Shortly after, replicas receive COMMIT messages from all other replicas for rmw_3, forming a commit certificate. All replicas generate $ts_1 = succ(ts = (1,1), s_4) = (2,4)$ and apply rmw_3 (Figure 15.7). Process p_4 now issues a read ρ_4, and the read completes in one round, returning $ts = (2,4)$ from rmw_3 (Figure 15.8).

There is no legal total order for this execution because rmw_3 must follow w_1 with no writes in between because rmw_3 picks $base\_state = \langle w_1, ts = (1,1)\rangle$. Thus, rmw_3 must be ordered before w_2. We also must have ρ_4 ordered after both rmw_3 and w_2 because it begins in real time after both operations have finished. The read ρ_4 sees rmw_3, so rmw_3 must be ordered after w_2. Thus, there is no legal total order of operations and linearizability is not satisfied.

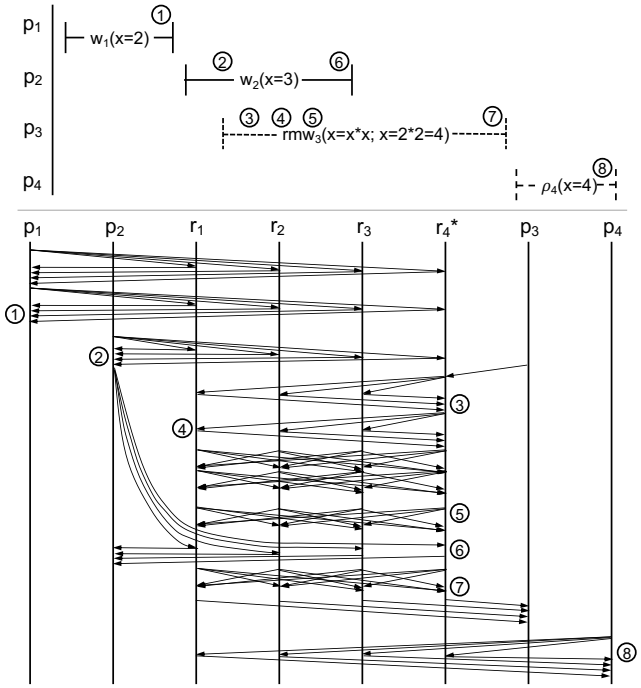


Figure 15: Labeled numbers represent the following events: 1. p_1 issues and completes w_1 with $ts = (1,1)$. 2. p_2 issues w_2 and gets back $ts = (1,1)$; the process then picks $ts = (2,2)$ for w_2. 3. The primary s_4 picks $base\_state = \langle w_1, ts = (1,1)\rangle$. 4. All replicas accept PRE-PREPARE messages because w_1 is the most recent state observed. 5. All replicas broadcast COMMIT messages to all other replicas. 6. All replicas apply w_2 because $ts = (2,2) > ts = (1,1)$. 7. All replicas apply rmw_3 because $ts = (2,4) > ts = (2,2)$. 8. p_4 issues and completes ρ_4 in 1 round, returning rmw_3 with $ts = (2,4)$.