Correctness of the Read/Conditional-Write and Query/Update protocols

Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson,
Michael K. Reiter, Jay J. Wylie

CMU-PDL-05-107

September, 2005

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Acknowledgements: This work is supported in part by Army Research Office grant number DAAD19-02-1-0389 and by Air Force Research Laboratory grant number FA8750-04-01-0238. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of Carnegie Mellon University or the U.S. Government or any of its agencies. We thank the members and companies of the PDL Consortium (including EMC, Engenio, Equallogic, Hewlett-Packard, HGST, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, and Sun) for their interest, insights, feedback, and support.

Keywords: Byzantine fault tolerance, asynchronous, read/conditional-write, query/update.

Abstract

The Read/Conditional-Write (R/CW) protocol provides linearizable reads and conditional-writes of individual objects. A client's conditional-write of an object succeeds only if the object has not been conditionally-written since it was last read by the client. In this sense, R/CW semantics are similar to those of a compare-and-swap register. If a conditional-write does not succeed, it aborts. The R/CW protocol supports multi-object reads and conditional-writes; such operations are strictly serializable. A variant of the R/CW protocol, the Query/Update (Q/U) protocol, provides an *operations-based* interface to clients: clients invoke query and update methods on objects rather than reading and writing objects in their entirety. The R/CW and Q/U protocols are correct in the asynchronous timing model and tolerate Byzantine failures of clients and servers.

1 Introduction

The Read/Conditional-Write (R/CW) protocol provides linearizable reads and conditional-writes of individual objects. A client's conditional-write of an object succeeds only if the object has not been conditionally-written since it was last read by the client. In this sense, R/CW semantics are similar to those of a compare-and-swap register. The R/CW protocol supports multi-object reads and conditional-writes; such operations are strictly serializable. A variant of the R/CW protocol, the Query/Update (Q/U) protocol, provides an *operations-based* interface to clients: clients invoke query and update methods on objects rather than reading and writing objects in their entirety. The R/CW and Q/U protocols are correct in the asynchronous timing model and tolerate Byzantine faults of clients and servers.

This technical report is a companion for the paper "Fault-scalable Byzantine fault-tolerant services" [1] and is made available to ensure timely dissemination of details elided from the paper. Motivation for these protocols, the intuition that underlies their structure, their relation to related work, and empirical results are found in the main paper. In this technical report, we present a proof of correctness and detailed pseudo-code for the R/CW protocol for individual objects (§2 and §3 respectively). We also discuss how to extend the R/CW protocol to accommodate multi-object conditional-writes (§4). Finally, we present extended pseudo-code for the Q/U protocol, a variant of the R/CW protocol, and an extended example execution of the Q/U protocol (§5).

2 Safety of the Read/Conditional-Write (R/CW) protocol

Using terminology developed throughout the proof, the outline of the proof of safety for the R/CW protocol for individual objects is as follows. First, we provide terminology and system definitions. Second, we define classification rules for candidates. Third, we define the types of conditional-writes, the pre-conditions for each type of conditional-write, and the post-conditions for each type of conditional-write. Fourth, we focus on the properties of a **write** that establishes a value candidate and a **copy** that establishes a value candidate; these two types of conditional-writes define *segments*. Such segments have start and end logical times that are shown to never overlap and that, in fact, the start of each segment corresponds to the end of another segment (except for the segment that starts at the well-defined initial value). As such, there is a single chain of values, called the *conditioned-on chain*, from the latest established value candidate back to the initial value candidate. The conditioned-on chain is defined by the conditioned-on timestamp of its members.

Given this view of segments we demonstrate that conditional-writes that complete can be totally ordered by logical timestamp and that this ordering is linearizable [5]. Moreover, given the segments and the linearized order of conditional-writes, we demonstrate that reads that return a value can be partially ordered by the logical timestamp of the value they return. This partial order of reads can be arbitrarily extended to a total order. This total order is shown to be linearizable, thus demonstrating that the set of all read and conditional-writes that return values is linearizable.

The R/CW protocol uses five types of conditional-writes. A **write** writes a new value candidate conditioned-on an established value candidate that is the latest candidate in the object history set. An **inline_write** writes a value candidate to additional servers—this is done if the latest timestamp in the object history set corresponds to a repairable value candidate. A **barrier** writes a barrier candidate—this is done if the latest timestamp in the object history set corresponds to an incomplete candidate (value or barrier). An **inline_barrier** writes a barrier candidate to additional servers—this is done if the latest timestamp in the object history set corresponds to a repairable barrier candidate. A **copy** copies a potential or established value candidate forward to a new timestamp—this is done if the latest timestamp in the object history set corresponds to a complete barrier candidate; the candidate written has the same data value as the latest potential or established candidate value that precedes the established barrier candidate.

2.1 Terminology

In this section terminology to describe the R/CW protocol is introduced. Some symbols and structures used in the pseudo-code are also used in the proof. Figure 1 on page 15 and Table 1 on page 15 may be helpful to the reader.

Definition 2.1 (*client, server, channels, shared keys, operation, request)***.** The R/CW protocol operates in a system comprised of *clients* and *servers*. Point-to-point authenticated *channels* exist among all servers and between all clients and servers. Channels are assumed to be unreliable, with the same properties as those used by Aguilera et al. in the crash-recovery model (i.e., channels do not create messages, channels may duplicate messages a finite number of times, and channels may drop messages a finite number of times) [2]. Such channels can be made reliable by repeated resends of requests. An infrastructure for deploying *shared keys* among pairs of servers is assumed to exist. Clients issue *reads* and *conditional-writes* to sets of servers. These are comprised of *requests* that a client sends directly to each server.

Definition 2.2 (*asynchronous timing model)***.** The R/CW protocol operates safely in an asynchronous system. No assumptions are made about the duration of message transmission delays or the execution rates of clients and servers except that they are non-zero.

Definition 2.3 (*hybrid failure model, Byzantine failures, crash-recovery failures, benign, malevolent, good, faulty)***.** Byzantine faulty components may exhibit arbitrary, potentially malicious, behavior [6]. Clients may be Byzantine faulty. The server model is a hybrid failure model [9, 10] of Byzantine and crash-recovery failures. We use the crash-recovery failure model of Aguilera et al. [2]. Servers have persistent storage that is durable through a crash and subsequent recovery. In the hybrid crash-recovery–Byzantine fault model, every server is either always-up, eventually-up, eventually-down, unstable, or malevolent. Since the Byzantine failure model is a strict generalization of the crash-recovery failure model, another term — malevolent — is used to categorize those servers that in fact exhibit out-of-specification, non-crash behavior. A server is *good* if it is either always-up or eventually-up (i.e., it may crash, but there is a time after which it is always-up). A server is *faulty* if it is unstable, eventually-down, or malevolent. As such, every server is either good or faulty. A server is *benign* if it obeys its specification except for crashes and recoveries. As such, every server is either benign or malevolent.

Definition 2.4 (*Computationally bounded adversary)***.** Clients and servers are assumed to be computationally bounded so that cryptographic primitives are effective.

Definition 2.5 (*universe, U, n, quorum, Q, quorum system, $\mathcal{Q}$)***.** The quorum system definition is based on that of Malkhi and Reiter [7]. We assume a universe U of servers such that $|U| = n$. A *quorum system* $\mathcal{Q} \subseteq 2^U$ is a non-empty set of subsets of U, every pair of which intersect. Each $Q \in \mathcal{Q}$ is called a *quorum*. The notation 2^{set} denotes the power set of *set*.

Definition 2.6 (*failure prone system, fault set, T, $\mathcal{T}$, malevolent fault set, B, $\mathcal{B}$)***.** We extend the definition of a failure prone system of Malkhi and Reiter [7] to accommodate the hybrid server failure model. We assume that in any execution, for $\mathcal{T} \subseteq 2^U$ any $T \in \mathcal{T}$ contains all faulty servers. We assume that in any execution, for $\mathcal{B} \subseteq 2^U$ any $B \in \mathcal{B}$ contains all malevolent servers. It follows from the definitions of faulty and malevolent that $\forall B \in \mathcal{B}, \exists T \in \mathcal{T} : B \subseteq T$.

Definition 2.7 (*candidate, accept, Data*)**.** A client conditional-write request generates an *candidate* at the server. Servers *accept* candidates if validation passes. The structure for a candidate, *Candidate* is given in Figure 1. Every candidate contains data denoted *Data*.

Definition 2.8 (*logical timestamp (timestamp), LT, LT_{CO}*)**.** *Logical timestamps* are used extensively in the R/CW protocol. Each candidate contains two logical timestamps: the logical timestamp of the candidate (LT) and the logical timestamp of the candidate it is conditioned-on (LT_{CO}). In the remainder of this paper, we refer just to *timestamps*, rather than logical timestamps.

Definition 2.9 (*comparing timestamps, =, <*)**.** Timestamps can be compared with the $=$ and $<$ operators. Equality ($=$) is defined naturally (all elements of the timestamp must be identical). Less than ($<$) is defined with the elements of the timestamp being compared in their order of definition (i.e., *Time*, then *BarrierFlag*, then *ClientID*, then *DataVerifier*, and finally *OHSVerifier*). To compare the *BarrierFlag* element, FALSE $<$ TRUE. To compare the *ClientID*, *DataVerifier*, and *OHSVerifier*, lexicographic comparisons are performed (e.g., `memcmp` could be performed).

Observation 2.10. We observe that although *LT.DataVerifier* is a cryptographic hash of the *Data* in a candidate, it is not guaranteed to be unique—many candidates in a replica history may have the same *Data*. We also observe that *LT.OHSVerifier* is unique for all candidates in a replica history—as will be seen, the object history set (and data) sent to a server by a client uniquely determine the candidate it accepts. A server can only accept a candidate once, since it is impossible for an object history set to be current (cf. Definition 2.34) at server s if s has already accepted the candidate corresponding to *ObjectHistorySet*. Finally, the *ClientID* is included in the timestamp to distinguish similar conditional-writes from different clients.

Definition 2.11 (*replica history, s.ReplicaHistory*)**.** If a server s accepts a candidate, it places the candidate in its replica history *s.ReplicaHistory*.

Definition 2.12 (*object history set, ObjectHistorySet*)**.** Clients read replica histories from servers. Clients store replica histories from servers in an array called the object history set (*ObjectHistorySet*). The object history set is indexed by server, e.g., *ObjectHistorySet*$[s]$ is equal to the last replica history returned from server s (i.e., *s.ReplicaHistory*).

Definition 2.13 (*initial candidate*)**.** There is a well known initial candidate: $\langle \mathbf{0}, \mathbf{0}, \perp \rangle$. All servers initialize their replica history to the initial value.

2.2 Candidates, constraints, and classification

In this section we present the constraints placed on quorum systems by the R/CW protocol. We define established and potential candidates. Based on the definitions of established and potential candidates, we develop Definition 2.21 and Definition 2.22 which define the quorum intersection properties necessary to provide read/conditional-write semantics.

Definition 2.14 (*established candidate*)**.** An established candidate is accepted at all of the benign servers in some quorum. Note a subset of servers in a quorum may be malevolent and we cannot specify what "accept" means at such servers.

Definition 2.15 (*repairable sets*)**.** We extend the quorum system definition to include *repairable sets*. Each quorum $Q \in \mathcal{Q}$ defines a set of repairable sets $\mathcal{R}(Q) \subseteq 2^{Q}$.

Definition 2.16 (*classifying a candidate complete)*. A candidate is classified *complete* if, given a set of server responses S, a quorum of servers share a common candidate:

$$\exists Q \in \mathcal{Q} : Q \subseteq S \Rightarrow \textit{complete}.$$

Definition 2.17 (*classifying a candidate repairable)*. A candidate is classified *repairable* if, given a set of server responses S, a repairable set share a common candidate and that candidate is not classifiable as complete:

$$(\forall Q \in \mathcal{Q} : Q \nsubseteq S) \wedge (\exists Q \in \mathcal{Q}, \exists R \in \mathcal{R}(Q) : R \subseteq S) \Rightarrow \textit{repairable}.$$

Definition 2.18 (*classifying a candidate incomplete)*. A candidate is classified *incomplete* if it is not classifiable as complete or repairable.

Definition 2.19 (*potential candidate)*. A potential candidate is accepted at all of the benign servers in some repairable set.

Definition 2.20 (*quorum size and asynchrony)*. To ensure that conditional-writes may complete in an asynchronous system,

$$\forall Q \in \mathcal{Q}, \forall T \in \mathcal{T} : Q \cup T \subseteq U.$$

Definition 2.21 (*Established candidate intersect potential candidate)*. We restrict the system such that an established candidate must intersect a potential candidate at at least one benign server:

$$\forall Q_i, Q_j \in \mathcal{Q}, \forall B \in \mathcal{B}, \forall R \in \mathcal{R}(Q_j) : Q_i \cap R \nsubseteq B.$$

Definition 2.22 (*established candidates classified as repairable)*. We restrict the system such that some repairable set of an established candidate fully intersects every other quorum; this ensures that an established candidate is classified as repairable or complete:

$$\forall Q_i, Q_j \in \mathcal{Q}, \forall B \in \mathcal{B}, \exists R \in \mathcal{R}(Q_i) : R \subseteq Q_i \cap Q_j \setminus B.$$

Definition 2.23 (*candidate, value candidate, barrier candidate)*. To clarify terminology used in the remainder of the proof, we may discuss established/potential candidates (which may be values or barriers), established/potential value candidates (which are not barriers), and established/potential barrier candidates (which are not values).

Observation 2.24. An established candidate may be classified as complete or repairable, but never incomplete. A potential candidate (that is not established) may be classified as repairable or incomplete, but never complete. A candidate that is classified as complete is established. A candidate that is classified as repairable may or may not be established. A candidate that is classified as incomplete is not established.

2.3 Classification tuple

In this section we define the *classification tuple* which is based on an object history set. The classification tuple is a summary of the object history set that indicates the latest value candidate, latest barrier candidate, and type of conditional-write that must be performed.

Definition 2.25 (*classification tuple, **rcw_classify*).** The function **rcw_classify** identifies the *classification tuple* of a given *ObjectHistorySet* (see Figure 3):

$$\langle \mathit{CWType}, \mathit{LatestCandidate}, \mathit{LatestBarrier}\rangle := \textbf{rcw\_classify}(\mathit{ObjectHistorySet})$$

Classification is performed by identifying the candidate with the latest timestamp, the value candidate with the latest timestamp that is classified either repairable or complete, and the barrier candidate with the latest timestamp that is classified either repairable or complete. This information determines which type of conditional-write must be performed. The terms in the classification tuple summarize the results of classification.
CWType is the type of conditional-write to perform.
LatestCandidate is the latest value candidate.
LatestBarrier is the latest barrier candidate.

Lemma 2.26. *CWType returned by **rcw_classify** is in the set* {WRITE, INLINE_WRITE, BARRIER, INLINE_BARRIER, COPY}.

Proof. This lemma is trivially true (see the pseudo-code).

Definition 2.27 (*conditioned on*). We refer to the object history set for a conditional-write as the *conditioned-on ObjectHistorySet*. We also refer to the *LatestCandidate* of the conditioned-on *ObjectHistorySet* (determined from classification) as the conditioned-on value candidate. Note that a candidate with logical timestamp *LT* is *conditioned on* the object history set with $\textbf{hash}(\mathit{ObjectHistorySet}) = \mathit{LT}.\mathit{OHSVerifier}$.

2.4 Conditional-write definitions

In this section, specific types of conditional-writes are defined. We develop the safety guarantees in terms of these types of conditional-writes. Servers can determine which type of conditional-write must be performed exclusively based on the *ObjectHistorySet*. The server constructs the candidate to accept (i.e., the tuple $\langle \mathit{LT}, \mathit{LT}_{\mathrm{CO}}, \mathit{Data}\rangle$) based on the classification of the conditioned-on object history set and the data value passed in (if it is a **write**).

Definition 2.28 (*write*). $\textbf{write}(\mathit{Data}, \mathit{ObjectHistorySet})$ conditionally-writes the value *Data*.

Definition 2.29 (*inline_write*). $\textbf{inline\_write}(\perp, \mathit{ObjectHistorySet})$ repairs *LatestCandidate*.

Definition 2.30 (*barrier*). $\textbf{barrier}(\perp, \mathit{ObjectHistorySet})$ conditionally-writes a barrier.

Definition 2.31 (*inline_barrier*). $\textbf{inline\_barrier}(\perp, \mathit{ObjectHistorySet})$ repairs *LatestBarrier*.

Definition 2.32 (*copy*). $\textbf{copy}(\perp, \mathit{ObjectHistorySet})$ copies forward *LatestCandidate*.

2.5 Conditional-write pre-conditions

In this section, we define the server validation that is performed before accepting a conditional-write candidate. For each type of conditional-write, the necessary pre-conditions are identified. Let $\mathit{CT} = \textbf{rcw\_classify}(\mathit{ObjectHistorySet})$.

```
    rcw_latest_time($\mathit{ObjectHistorySet}$)
100: return (max($\mathit{ObjectHistorySet}[U].\mathit{ReplicaHistory}.\mathit{LT}$))
```

Definition 2.33 (*current time, LT_{current}*)**.** The current time LT_{current} for a *ObjectHistorySet* is defined as:

$$LT_{\text{current}} = \begin{cases} CT.\mathit{LatestCandidate}.LT & \text{if } CT.\mathit{CWType} = \textsc{write}, \\ \mathbf{rcw\_latest\_time}(\mathit{ObjectHistorySet}) & \text{if } CT.\mathit{CWType} = \textsc{barrier}, \\ CT.\mathit{LatestCandidate}.LT & \text{if } CT.\mathit{CWType} = \textsc{inline\_write}, \\ CT.\mathit{LatestBarrier}.LT & \text{if } CT.\mathit{CWType} = \textsc{inline\_barrier}, \\ CT.\mathit{LatestBarrier}.LT & \text{if } CT.\mathit{CWType} = \textsc{copy}. \end{cases}$$

The current time identifies the logical timestamp prior to which the server's replica history does not affect classification. Note that **rcw_latest_time** is defined as $\mathbf{max}(\mathit{ObjectHistorySet}[U].\mathit{ReplicaHistory}.LT$ in the pseudo-code (See Figure 1100 on page 18).

Definition 2.34 (***current***)**.** For benign server s, if $\mathbf{max}(s.\mathit{ReplicaHistory}.LT) \leq LT_{\text{current}}$ then $\mathbf{current}(\mathit{ObjectHistorySet}) := \textsc{true}$.
Otherwise, $\mathbf{current}(\mathit{ObjectHistorySet}) := \textsc{false}$.

Observation 2.35. Validating authenticators is one aspect of server validation. If an HMAC in an authenticator does not validate for $\mathit{ObjectHistorySet}[s]$ then the server ignores $\mathit{ObjectHistorySet}[s]$ (i.e., the server sets $\mathit{ObjectHistorySet}[s] := \{\langle \mathbf{0}, \mathbf{0}, \perp \rangle\}$). As such, authenticator validation can only impact whether or not an *ObjectHistorySet* is current or not.

Lemma 2.36. *Benign server s accepts $\boldsymbol{write}(\mathit{Data}, \mathit{ObjectHistorySet})$ only if $CT.\mathit{CWType} = \textsc{write}$ and $\boldsymbol{current}(\mathit{ObjectHistorySet}) = \textsc{true}$.*

Proof. See pseudo-code for pre-condition.

Lemma 2.37. *Benign server s accepts $\boldsymbol{inline\_write}(\perp, \mathit{ObjectHistorySet})$ only if $CT.\mathit{CWType} = \textsc{inline\_write}$ and $\boldsymbol{current}(\mathit{ObjectHistorySet}) = \textsc{true}$.*

Proof. See pseudo-code for pre-condition.

Lemma 2.38. *Benign server s accepts $\boldsymbol{barrier}(\perp, \mathit{ObjectHistorySet})$ only if $CT.\mathit{CWType} = \textsc{barrier}$ and $\boldsymbol{current}(\mathit{ObjectHistorySet}) = \textsc{true}$.*

Proof. See pseudo-code for pre-condition.

Lemma 2.39. *Benign server s accepts $\boldsymbol{inline\_barrier}(\perp, \mathit{ObjectHistorySet})$ only if $CT.\mathit{CWType} = \textsc{inline\_barrier}$ and $\boldsymbol{current}(\mathit{ObjectHistorySet}) = \textsc{true}$.*

Proof. See pseudo-code for pre-condition.

Lemma 2.40. *Benign server s accepts $\boldsymbol{copy}(\perp, \mathit{ObjectHistorySet})$ only if $CT.\mathit{CWType} = \textsc{copy}$ and $\boldsymbol{current}(\mathit{ObjectHistorySet}) = \textsc{true}$.*

Proof. See pseudo-code for pre-condition.

2.6 Conditional-write post-conditions

In this section, we list the post-conditions for conditional-writes that establish candidates. The post-conditions are given in terms of an object history set comprised of the replica histories of the benign servers that accepted the conditional-write candidate. The post-condition takes effect as soon as the candidate is established (cf. Definition 2.14).

Definition 2.41 (*post-conditions, ObjectHistorySet′*)**.** We define $\mathit{ObjectHistorySet}'$ to be the object history set comprised of the replica histories returned from sufficient benign servers that accept conditional-write candidates to establish the candidate being conditionally-written. In the case of **inline_write** and **inline_barrier** conditional-writes, $\mathit{ObjectHistorySet}'$ includes the replica histories of the benign servers that accepted **write** candidates and **barrier** candidates respectively. Also within this section, we define $CT = \textbf{rcw\_classify}(\mathit{ObjectHistorySet})$ and $CT' = \textbf{rcw\_classify}(\mathit{ObjectHistorySet}')$.

Lemma 2.42. *If* $\textbf{\textit{write}}(\mathit{Data}, \mathit{ObjectHistorySet})$ *is established yielding* $\mathit{ObjectHistorySet}'$*, then letting* $X = CT'.\mathit{LatestCandidate}$*, we have that:*
$X.\mathit{Data} = \mathit{Data}$;
$X.LT.\mathit{Time} = \textbf{\textit{rcw\_latest\_time}}(\mathit{ObjectHistorySet}).\mathit{Time} + 1$;
$X.LT.\mathit{BarrierFlag} = \text{FALSE}$;
$X.LT.\mathit{ClientID} = \mathit{ClientID}$;
$X.LT.\mathit{DataVerifier} = \textbf{\textit{hash}}(\mathit{Data})$;
$X.LT.\mathit{OHSVerifier} = \textbf{\textit{hash}}(\mathit{ObjectHistorySet})$;
$X.LT_{\mathrm{CO}} = CT.\mathit{LatestCandidate}.LT$.

Proof. Look at the pseudo-code. Note that, because of the pre-conditions on a **write**, $X.LT.\mathit{Time} = CT.\mathit{LatestCandidate}.\mathit{Time} + 1$ is also true.

Lemma 2.43. *If* $\textbf{\textit{inline\_write}}(\perp, \mathit{ObjectHistorySet})$ *is established yielding* $\mathit{ObjectHistorySet}'$*, then* $CT'.\mathit{LatestCandidate} = CT.\mathit{LatestCandidate}$.

Proof. Look at the pseudo-code.

Lemma 2.44. *If* $\textbf{\textit{barrier}}(\perp, \mathit{ObjectHistorySet})$ *is established yielding* $\mathit{ObjectHistorySet}'$*, then,* $CT'.\mathit{LatestCandidate} = CT.\mathit{LatestCandidate}$*. Letting* $X = CT'.\mathit{LatestBarrier}$*, we have that:*
$X.\mathit{Data} = \perp$;
$X.LT.\mathit{Time} = \textbf{\textit{rcw\_latest\_time}}(\mathit{ObjectHistorySet}).\mathit{Time} + 1$;
$X.LT.\mathit{BarrierFlag} = \text{TRUE}$;
$X.LT.\mathit{ClientID} = \mathit{ClientID}$;
$X.LT.\mathit{DataVerifier} = \perp$;
$X.LT.\mathit{OHSVerifier} = \textbf{\textit{hash}}(\mathit{ObjectHistorySet})$;
$X.LT_{\mathrm{CO}} = CT.\mathit{LatestCandidate}.LT$.

Proof. Look at the pseudo-code.

Lemma 2.45. *If* $\textbf{\textit{inline\_barrier}}(\perp, \mathit{ObjectHistorySet})$ *is established yielding* $\mathit{ObjectHistorySet}'$*, then* $CT'.\mathit{LatestBarrier} = CT.\mathit{LatestBarrier}$ *and* $CT'.\mathit{LatestCandidate} = CT.\mathit{LatestCandidate}$.

Proof. Look at the pseudo-code.

Lemma 2.46. *If* ***copy***$(\perp, \mathit{ObjectHistorySet})$ *establishes a candidate yielding* $\mathit{ObjectHistorySet}'$*, then, letting* $X := CT'.\mathit{LatestCandidate}$*, we have that:*
$X.\mathit{Data} = CT.\mathit{LatestCandidate}.\mathit{Data}$*;*
$X.LT.\mathit{Time} = \textbf{\textit{rcw\_latest\_time}}(\mathit{ObjectHistorySet}).\mathit{Time} + 1$*;*
$X.LT.\mathit{BarrierFlag} = \text{FALSE}$*;*
$X.LT.\mathit{ClientID} = \mathit{ClientID}$*;*
$X.LT.\mathit{DataVerifier} = \textbf{\textit{hash}}(X.\mathit{Data})$*;*
$X.LT.\mathit{OHSVerifier} = \textbf{\textit{hash}}(\mathit{ObjectHistorySet})$*;*
$X.LT_{\mathrm{CO}} = CT.\mathit{LatestCandidate}.LT$*.*

Proof. Look at the pseudo-code. Note that, because of the pre-conditions on **copy**, $X.LT.\mathit{Time} = CT.\mathit{LatestBarrier}.\mathit{Time} + 1$ is also true.

2.7 Repair conditional-writes

In this section, we consider **inline_write** and **inline_barrier**. We show that for an execution that includes **inline_write** and **inline_barrier** conditional-writes, there exists an execution that does not include such operations that has identical server-side state transitions. Server-side state transitions refers to the state maintained by the server (i.e., its replica history and corresponding authenticator). As such, the remainder of the safety proof focuses on **write**, **barrier**, and **copy** conditional-writes.

In an obvious variation of the protocol, **inline_write** and **inline_barrier** conditional-writes are not employed: they are both replaced with a **barrier** followed by a **copy**. The **inline_write** and **inline_barrier** conditional-writes are included in the proof because they are included in our implementation of the R/CW protocol. In practice, such repairs reduce read/conditional-write contention, since a read initiating such repair does not contend with the conditional-write it repairs.

Lemma 2.47. *For every* ***inline_write*** *performed, there exists an execution in which the server-side state transition is identical, but in which the* ***inline_write*** *was not performed.*

Proof. Lemmas 2.42 and 2.43 show that the post-conditions (i.e., the server-side state transitions) for a **write** and **inline_write** are identical. From lemmas 2.36 and 2.37 we note that the pre-condition for an **inline_write** differs from that of a **write** only in the existence of the value candidate being repaired. As such, there exists another execution in which any server-side transition resulting from the acceptance of an **inline_write** is due to the acceptance of a **write**.

Lemma 2.48. *For every* ***inline_barrier*** *performed, there exists an execution in which the server-side state transition is identical, but in which the* ***inline_barrier*** *was not performed.*

Proof. Lemmas 2.44 and 2.44 show that the post-conditions (i.e., the server-side state transitions) for a **barrier** and **inline_barrier** are identical. From lemmas 2.38 and 2.39 we note that the pre-condition for an **inline_barrier** differs from that of a **barrier** only in the existence of the value candidate being repaired. As such, there exists another execution in which any server-side transition resulting from the acceptance of an **inline_barrier** is due to the acceptance of a **barrier**.

2.8 Write-CW segments

In this section, we define a write-cw segment. In the subsequent, we define a copy-CW segment. Then, we define a copy-CW segment-chain. From the properties of write-CW segments and copy-CW segment-chains, we demonstrate that all established value candidates are in the *condition-on chain.*

Definition 2.49 (*segment, LT_{begin}, LT_{end}*)**.** A *segment* is a logical timestamp interval. The logical timestamp that begins a segment is denoted LT_{begin}. The logical timestamp that ends a segment is denoted LT_{end}.

Definition 2.50 (*write-CW segment*)**.** Every **write** that establishes a candidate defines a write-CW segment. Consider established value candidate *Candidate* written by a **write**. The *write-CW segment* defined by *Candidate* has $LT_{\text{end}} = \mathit{Candidate.LT}$ $LT_{\text{begin}} = \mathit{Candidate.LT}_{\text{CO}}$.

Lemma 2.51. *For every write-CW segment in an execution, there are no potential barrier candidates with a timestamp in the range* $[LT_{\text{begin}}, LT_{\text{end}}]$.

Proof. Consider the pre-conditions for a **write** (cf. Lemma 2.36). Because of the pre-conditions, when a benign server accepts a write-CW candidate *Candidate*, the server has no history candidates between $\mathit{Candidate.LT}_{\text{CO}}$ (LT_{begin}) and $\mathit{Candidate.LT}$ (LT_{end}). Moreover, because benign servers only accept candidates conditioned-on a current object history set, such servers never accept a candidate in the range $[LT_{\text{begin}}, LT_{\text{end}}]$ at a later point in the execution. Because of the intersection property between established candidates and potential candidates (cf. Definition 2.21), there cannot exist a potential barrier with a timestamp in the range $[LT_{\text{begin}}, LT_{\text{end}}]$.

Lemma 2.52. *For each write-CW segment in in an execution there are no potential value candidates with a timestamp in the range* $(LT_{\text{begin}}, LT_{\text{end}})$.

Proof. This proof is similar to the proof for Lemma 2.51. However, there is a distinction between the range in which there are no potential barrier candidates and in which there are no potential values. By definition, there is an established (and therefore potential) value candidate at LT_{begin} and LT_{end}.

2.9 Copy-CW segments

The **write** is expected to be the "common case" in the R/CW protocol. The **copy** covers the "corner case" in which there is contention. Note that not all copy-CW segments are of interest; only those that are part of a copy-CW segment-chain are of interest. We define the copy-CW segment-chain in the subsequent.

Definition 2.53 (*copy-CW segment*)**.** Every **copy** that establishes a candidate defines a copy-CW segment. Consider established value candidate *Candidate* written by a **copy**. The *copy-CW segment* defined by *Candidate* has $LT_{\text{end}} = \mathit{Candidate.LT}$ $LT_{\text{begin}} = \mathit{Candidate.LT}_{\text{CO}}$.

Lemma 2.54. *For every copy-CW segment in an execution, there exists at least one established barrier candidate with timestamp in the range* $(LT_{\text{begin}}, LT_{\text{end}})$.

Proof. Consider the pre-conditions for a **copy** (cf. Lemma 2.40). The pre-conditions require there to be an established barrier candidate before a benign server will accept a **copy** candidate. Since the potential value candidate that defines the copy-CW segment is accepted at at least one benign server, the pre-conditions are true, and there exists an established barrier candidate with timestamp in the range $(LT_{\text{begin}}, LT_{\text{end}})$.

Lemma 2.55. *For each copy-CW segment in an execution, there are no established value candidates with a timestamp in the range* $(LT_{\text{begin}}, LT_{\text{end}})$.

Proof. We show that the potential value candidate that defines the copy-CW segment precludes an established value candidate in the range $(LT_{\text{begin}}, LT_{\text{end}})$. Consider the pre-conditions for a **copy** (cf. Lemma 2.40) and the intersection property between established and potential candidates (cf. Definition 2.21). If an established value candidate exists with a timestamp in the range $(LT_{\text{begin}}, LT_{\text{end}})$, classification would identify it as the conditioned-on value candidate since Lemma 2.22 ensures that an established value candidate is always classified as repairable. Since the conditioned-on value candidate has timestamp LT_{begin}, there does not exist an established value candidate with timestamp in the range $(LT_{\text{begin}}, LT_{\text{end}})$. The value candidate that is conditioned-on and the value candidate that ends the copy segment may both be established, which is why the range is $(LT_{\text{begin}}, LT_{\text{end}})$ and not $[LT_{\text{begin}}, LT_{\text{end}}]$.

2.10 Copy-CW segment-chains

In this section we define a copy-CW segment-chain. Such a chain consists of one or more copy-CW segments: the first segment in the chain begins with an established value candidate and the final segment in the chain ends with an established value candidate. All segments in between the established value candidates begin and end with a potential, but not established, value candidate. Not all copy-CW segments that exist in the timestamp interval between the two established value candidates that define the copy-CW segment-chain need be in the copy-CW segment-chain.

Definition 2.56 (*opening copy-CW segment*)**.** A copy-CW segment that begins with an established value candidate is called an *opening* copy-CW segment.

Definition 2.57 (*terminating copy-CW segment*)**.** A copy-CW segment that ends with an established value candidate is called a *terminating* copy-CW segment.

Definition 2.58 (*copy-CW segment-chain*)**.** A copy-CW segment-chain consists of an opening copy-CW segment, a terminating copy-CW segment, and a finite number (possibly zero) of copy-CW segments.

Observation 2.59. A copy-CW segment may be both an opening and terminating copy-CW segment and so a copy-CW segment-chain may consist of a single copy-CW segment.

Lemma 2.60. *For every non-opening copy-CW segment X in a copy-CW segment chain, there exists exactly one other copy-CW segment Y in the copy-CW segment chain, such that $X.LT_{\text{begin}} = Y.LT_{\text{end}}$.*

Proof. The pre-condition for a **copy** (see Lemma 2.40) and the post-conditions for a **copy** (see Lemma 2.46) ensure this.

Lemma 2.61. *For every non-terminating copy-CW segment X in a copy-CW segment-chain, there exists exactly one other copy-CW segment Y in the copy-CW segment-chain, such that $X.LT_{\text{end}} = Y.LT_{\text{begin}}$.*

Proof. The pre-condition for a **copy** (see Lemma 2.40) and the post-conditions for a **copy** (see Lemma 2.46) ensure this.

Definition 2.62 (*copy-CW segment-chain begin and end*)**.** A copy-CW segment chain begins at LT_{begin} of its opening copy-CW segment and ends at LT_{end} of its terminating copy-CW segment; as such, the copy-CW segment-chain has a LT_{begin} and LT_{end} as well.

Lemma 2.63. *For each copy-CW segment-chain in an execution, there are no established value candidates with a timestamp in the range $(LT_{\text{begin}}, LT_{\text{end}})$.*

Proof. Lemma 2.55 ensures that there are no established value candidates within the timestamp interval of each copy-CW segment in a copy-CW segment-chain. Lemmas 2.60 and 2.61 ensure that there are only copy-CW segments in the timestamp interval of the copy-CW segment-chain. Therefore, there are no established value candidates with a timestamp in the range $(LT_{\text{begin}}, LT_{\text{end}})$.

Observation 2.64. Once a barrier candidate is established, the only way a value candidate with a higher timestamp can be established is due to a **copy**. Until there is a terminating segment, the set of segments in the segment-chain may be unknown. Multiple potential value candidates can condition on the established value candidate with timestamp LT_{begin}. Therefore, there can exist multiple "cycles" of established barriers followed by multiple potential value candidates. In each of these "cycles" the set of potential value candidates, that result from **copy** conditional-writes, is a subset of the potential value candidates that precede the established barrier candidate and succeed LT_{begin}. Once there is a terminating segment, membership in the segment-chain is determined.

2.11 The condition-on chain

In this section we show that from the latest established value candidate in an execution back to logical time $\mathbf{0}$, there exists a continuous chain of write-CW segments and copy-CW segment-chains. We refer to this chain as the condition-on chain. In this section we use the term *segment* to refer to either a write-CW segment or a copy-CW segment-chain.

Definition 2.65 (*condition-on chain*)**.** The condition-on chain for an execution is comprised of the latest established value candidate in the execution as well as each candidate found by repeatedly traversing condition on timestamps.

Lemma 2.66. *Every established value candidate is in the condition-on chain.*

Proof. Consider an execution in which there is a single established value candidate. By definition, this single established value candidate is $\langle \mathbf{0}, \mathbf{0}, \perp \rangle$. Now, consider an execution in which there are two established value candidates. Clearly, one is $\langle \mathbf{0}, \mathbf{0}, \perp \rangle$ and the other is the latest established value candidate. Following the condition on timestamp of the latest established value candidate leads to $\langle \mathbf{0}, \mathbf{0}, \perp \rangle$. Lemma 2.52 proves that there are no potential or established value candidates within the timestamp interval of a write-CW segment, but that there is an established value candidate at the beginning and at the end. Lemma 2.63 proves that there are no established value candidates within the timestamp interval of a copy-CW segment-chain, but that there is an established value candidate at the beginning and at the end. Since the latest established value candidate is either part of a write-CW segment chain or a copy-CW segment-chain, and since there are only two established value candidates, then both established value candidates are in the condition-on chain. Now, consider an execution in which there are x established value candidates. The latest established value candidate (i.e., the x^{th} established value candidate) is part of either a write-CW segment or a copy-CW segment-chain that begins with the next latest established value candidate (i.e., the $(x-1)^{st}$). By induction then, all established value candidates are in the condition-on chain (and the condition-on chain ends at $\langle \mathbf{0}, \mathbf{0}, \perp \rangle$).

2.12 Linearizable reads and conditional-writes

Intuitively, linearizability [5] requires that each read return a value consistent with some execution in which each read and write is performed at a distinct point in time between when the client invokes the operation and when the operation returns. For the purposes of the R/CW protocol, we consider the linearizability of reads and conditional-writes (rather than reads and writes).

The condition-on chain that results from write-CW segments and copy-CW segment-chains induces a total order on all established value candidates. This total order is sufficient to demonstrate the linearizability of all conditional-writes that establish a candidate. Reads are shown to be partially ordered by the established value candidate they return and to obey their real-time ordering relation. Since multiple distinct reads may return the same established value candidate, the total order on established value candidates provides only a partial ordering. Copy-CW segment-chains that literally copy an established value candidate may induce ordering on a subset of reads that return the same "data" (albeit different established value candidates). The partial ordering on reads can be arbitrarily extended to a total ordering, and thus the set of all reads that return a candidate and conditional-writes that established is linearizable.

Lemma 2.67. *All established value candidates are totally ordered by their timestamps.*

Proof. Lemma 2.66 proves that all established value candidates are in the condition-on chain and that the condition-on chain totally orders all established value candidates by timestamp.

Definition 2.68 (*conditional-write begins***).** A conditional-write *begins* once a benign server accepts a conditional-write candidate that corresponds to the conditional-write.

Definition 2.69 (*conditional-write ends***).** A conditional-write *ends* once the candidate corresponding to it is an established value candidate. As such, we just refer to a conditional-write that is established.

Definition 2.70 (*read begins***).** A read *begins* once a read response from a benign server is received.

Definition 2.71 (*read ends***).** A read *ends* once it returns a candidate.

Lemma 2.72. *A read that returns a candidate, returns an established value candidate.*

Proof. Look at the pseudo-code; repair is attempted until the pre-conditions for a **write** are met (i.e., until an established value candidate is latest).

Lemma 2.73. *The set of reads by benign clients in an execution are partially ordered by the timestamp of the established value candidate returned.*

Proof. As Lemma 2.67 proves, all established value candidates in the execution are totally ordered. The total order on established value candidates induces a partial order on all reads (which return only established value candidates, due to Lemma 2.72). For a read to return a candidate, the candidate must be established before the read ends (due to the classification rules Definition 2.17 and Definition 2.16). It does not matter if the candidate returned by the read was established prior to the read beginning, or if it is established at some point during the read. In either case the candidate returned by the read is consistent with the a partial order.

Observation 2.74. Lemma 2.73 excludes reads by malevolent clients because malevolent clients can return arbitrary values from reads. Indeed, such clients need not even issue a read request before returning a, potentially forged, candidate.

Lemma 2.75. *In an execution, conditional-writes that establish value candidates and the values returned by reads are linearizable.*

Proof. Lemma 2.67 proves that all established value candidates are totally ordered by timestamp. This ordering is consistent with the real-time ordering of the conditional-write operations that yielded the established value candidates. Lemma 2.73 proves that reads are partially ordered by the timestamp of the candidate returned. The partial ordering can be totally ordered so as to be consistent with the real-time ordering of the read operations. As such, conditional-writes that establish value candidates and the values returned by reads are linearizable.

2.13 Liveness

The R/CW protocol as described provides a very weak liveness guarantee, namely that it is possible to make progress.

Lemma 2.76. *It is possible to complete a conditional-write.*

Proof. Given any object history set, there is a sequence of client operations, that if each completes, allows a client to complete a conditional-write. Notice that it is possible that a client performs the following chain of operations: **barrier**, **inline_barrier**, **copy**, **inline_write**, and, finally, **write**. All possible operations are in this chain and it ends with a conditional-write. Since every *ObjectHistorySet* requires one of **barrier**, **inline_barrier**, **copy**, **inline_write**, or **write** (see Lemma 2.26), then it is possible from any system state to perform a conditional-write.

Unfortunately, Lemma 2.76 does not guarantee that eventually a new value is conditionally-written to the system. Malevolent components can prevent correct clients from making progress.

Corollary 2.77. *In a benign execution, the R/CW protocol is obstruction-free [4].*

Proof. In the absence of contention from other clients and of malevolent components, Lemma 2.76 ensures that a client will complete a conditional-write in a finite number of steps.

3 Read/conditional-write pseudo-code

In this section pseudo-code for the read/conditional-write (R/CW) protocol is given. The pseudo-code is tailored for a threshold quorum system construction described in the following section.

3.1 Threshold quorum constraints

In this section we develop constraints on threshold quorum systems that meet the definitions from Section 2.2. This construction of threshold quorum system can directly be applied to the RT-system quorum construction in [8].

Consider a threshold quorum system in which all $Q \in \mathcal{Q}$ are of size q, all $\mathcal{R}(Q)$ are of size r, all $T \in \mathcal{T}$ are of size t, all $B \in \mathcal{B}$ are of size b, and the universe is of size n.

From Definition 2.20:

$$q + t \leq n. \tag{1}$$

From Definition 2.21:

$$n < q + r - b. \tag{2}$$

From Definition 2.22

$$n + r + b \leq 2q. \tag{3}$$

Combining (1) and (2):

$$\begin{aligned} q + t \leq n &< q + r - b, \\ t + b &< r. \end{aligned} \tag{4}$$

Combining (1) and (3):

$$\begin{aligned} q + t \leq n &\leq 2q - r - b, \\ r + t + b &\leq q. \end{aligned} \tag{5}$$

Combining (2) and (3):

$$\begin{aligned} n - q + b < r &\leq 2q - n - b, \\ 2n &< 3q - 2b, \\ n &< \frac{3q - 2b}{2}. \end{aligned} \tag{6}$$

These constraints can be summarized as:

$$\begin{aligned} t + b &< r; \\ r + t + b &\leq q; \\ q + t &\leq n; \\ n &< \mathbf{min}\left[q + r - b, \frac{3q - 2b}{2}\right]. \end{aligned}$$

To construct a threshold quorum parameterized by $\Delta \geq 0$:

$$\begin{aligned} r &= t + b + 2\Delta + 1; \\ q &= 2t + 2b + 2\Delta + 1 \quad (= r + t + b); \\ n &= 3t + 2b + 3\Delta + 1 \quad (= q + t + \Delta). \end{aligned}$$

With this construction, the upper bound on n is obeyed. First, $q + r - b = 3t + 2b + 4\Delta + 2 > n$. Second, $\frac{3q-2b}{2} = \frac{6t+4b+6\Delta+3}{2} = 3t + 2b + 3\Delta + 1.5 > n$. Since n is bound from above by $\frac{3q}{2}$, the greatest throughput-scalability that can be achieved via threshold quorums is 1.5×.

3.2 Symbols and data structures

Symbols used in the pseudo-code are listed in Table 1. Enumerations, structures, and types used in the pseudo-code are given in Figure 1.

Symbol	Description
s	A specific server.
U	The universe of all servers.
$\mathit{ObjectHistorySet}$	An object history set.
$\mathit{ReplicaHistory}$	A replica history.
$s.\mathit{ReplicaHistory}$	The replica history of server s.
α	An authenticator (array of HMACs).
LT	Logical timestamp.
LT_{CO}	Conditioned-on logical timestamp.
$\mathbf{0}$	Well known initial logical timestamp.
$\perp$	A null value; sometimes used to indicate an unused argument/answer.
Q	A quorum.
$\mathcal{Q}$	The quorum system.
n	The size of the universe of servers
q	The size of each quorum in a threshold quorum system; threshold for classifying a candidate complete.
r	The size of each repairable set for a quorum; threshold for classifying a candidate repairable.
t	The threshold number of faulty servers.
b	The threshold number of Byzantine faulty servers.

Table 1. Symbols used in the pseudo-code.

```
/* Enumerations. */
/* Types of operations. */
$CWType \in \{$WRITE, INLINE_WRITE, BARRIER, INLINE_BARRIER, COPY$\}$
/* Structures. */
/* Logical timestamps. */
$LT \equiv \{$
    $Time$                                      /* Major component of logical time. */
    $BarrierFlag$                               /* Boolean flag indicating barrier or value. */
    $ClientID$                                  /* Client ID. */
    $DataVerifier$                              /* Hash of data value. */
    $OHSVerifier$                               /* Hash of conditioned-on $ObjectHistorySet$. */
$\}$
/* Candidate (initialized to $\langle \mathbf{0}, \mathbf{0}, \perp \rangle$). */
$Candidate \equiv \{$
    $LT$                                        /* Logical timestamp of candidate. */
    $LT_{\text{CO}}$                            /* Timestamp of conditioned-on candidate. */
    $Data$                                      /* Candidate's data. */
$\}$
/* Types. */
/* Replica history is an ordered set of candidates. */
$ReplicaHistory \equiv \{Candidate\}$
/* Authenticator is an array of HMACs indexed by server ($U$ is the universe of servers). */
$\alpha \equiv \text{HMAC}[U]$
/* Object history set is an array of replica histories indexed by server. */
$ObjectHistorySet \equiv \langle ReplicaHistory, \alpha \rangle [U]$
```

Figure 1. Enumerations, structures, and types for the R/CW pseudo-code.

3.3 Client-side

Client-side functions are listed in Figure 2. To perform a read a null object history set (i.e., $\perp$), is sent in the conditional-write request (cf. line400). In the pseudo-code shown, reads retry repair until a candidate is classified complete; they could abort instead. Conditional-writes are shown to abort if they do not establish a candidate; they could retry repair instead. The quorum probing policy shown in **c_rcw_quorum_rpc** is inefficient. A more efficient approach to probing for a quorum of responses is implemented in the prototype. Functions for determining the classification tuple of an object history set are listed in Figure 3.

```
c_rcw_initialize() :
300: /* The version history returned from each server in the read operation is kept in ObjectHistorySet. */
301: for each ($s \in U$) do
302:    $ObjectHistorySet[s].ReplicaHistory := \{\langle \mathbf{0}, \mathbf{0}, \perp \rangle\}$
303:    $ObjectHistorySet[s].\alpha := \perp$
304: end for

c_rcw_read() :
400: $\langle \perp, ObjectHistorySet \rangle :=$ c_rcw_quorum_rpc($\perp, \perp$)          /* Passing in ($\perp, \perp$) indicates a read request. */
401: $\langle CWType, Candidate, \perp \rangle :=$ rcw_classify($ObjectHistorySet$)
402: if ($CWType \neq$ WRITE) then
403:    /* Perform repair. */
404:    $ObjectHistorySet :=$ c_rcw_repair($ObjectHistorySet$)
405:    $\langle CWType, Candidate, \perp \rangle :=$ rcw_classify($ObjectHistorySet$)
406:    /* Since repair returned, $CWType =$ WRITE. */
407: end if
408: return ($\langle$SUCCESS, $Candidate.Data \rangle$)

c_rcw_write($Data$, $ObjectHistorySet$):                                        /* WRITE. */
500: $\langle Order, ObjectHistorySet \rangle :=$ c_rcw_quorum_rpc($Data$, $ObjectHistorySet$)
501: if ($Order \geq q$) then
502:    return ($\langle$SUCCESS, $\perp \rangle$)
503: end if
504: /* Otherwise, perform repair and then retry. */
505: $ObjectHistorySet :=$ c_rcw_repair($ObjectHistorySet$)
506: return (c_rcw_write($Data$, $ObjectHistorySet$))

c_rcw_repair($ObjectHistorySet$):                  /* INLINE_WRITE, BARRIER, INLINE_BARRIER, and COPY. */
600: repeat
601:    backoff()                                                  /* Backoff to avoid livelock. */
602:    /* Perform a barrier or copy (depends on ObjectHistorySet). */
603:    $\langle \perp, ObjectHistorySet \rangle :=$ c_rcw_quorum_rpc($\perp$, $ObjectHistorySet$)
604:    $\langle CWType, Candidate, \perp \rangle :=$ rcw_classify($ObjectHistorySet$)
605: until ($CWType =$ WRITE)
606: return ($ObjectHistorySet$)

c_rcw_quorum_rpc($Data$, $ObjectHistorySet$) :
700: $ResponseSet := \emptyset$
701: $Count := 0$
702: repeat
703:    /* Eliding probing policy. For simplicity broadcast to all servers. */
704:    for each ($s \in U \setminus ResponseSet.s$) do
705:       send($s$, $Data$, $ObjectHistorySet$)
706:    end for
707:    if (poll() = TRUE) then
708:       $\langle s, Status, \langle ReplicaHistory, \alpha \rangle \rangle :=$ receive()
709:       if ($s \notin ResponseSet.s$) then
710:          $ObjectHistorySet[s] := \langle ReplicaHistory, \alpha \rangle$
711:          $ResponseSet := ResponseSet \cup \langle s \rangle$
712:       end if
713:       if ($Status =$ SUCCESS) then
714:          $Count := Count + 1$
715:       end if
716:    end if
717: until ($\exists Q \subseteq ResponseSet : Q \in \mathcal{Q}$)
718: return ($\langle Count, ObjectHistorySet \rangle$)
```

Figure 2. Client-side R/CW pseudo-code.

```
rcw_classify($ObjectHistorySet$):
800: /* Get latest object version, barrier version, and timestamp. */
801: $LatestCandidate :=$ rcw_latest_candidate($ObjectHistorySet$, FALSE)
802: $LatestBarrier :=$ rcw_latest_candidate($ObjectHistorySet$, TRUE)
803: $LT_{\text{latest}} :=$ rcw_latest_time($ObjectHistorySet$)
804: /* Determine which type of operation to perform. */
805: if $(LT_{\text{latest}} = LatestCandidate.LT) \wedge$ (rcw_order($LatestCandidate, ObjectHistorySet$) $\geq q$) then
806:     $CWType :=$ WRITE
807: else if $(LT_{\text{latest}} = LatestCandidate.LT) \wedge$ (rcw_order($LatestCandidate, ObjectHistorySet$) $\geq r$) then
808:     $CWType :=$ INLINE_WRITE
809: else if $(LT_{\text{latest}} = LatestBarrier.LT) \wedge$ (rcw_order($LatestBarrier, ObjectHistorySet$) $\geq q$) then
810:     $CWType :=$ COPY
811: else if $(LT_{\text{latest}} = LatestBarrier.LT) \wedge$ (rcw_order($LatestBarrier, ObjectHistorySet$) $\geq r$) then
812:     $CWType :=$ INLINE_BARRIER
813: else
814:     $CWType :=$ BARRIER
815: end if
816: return $(\langle CWType, LatestCandidate, LatestBarrier \rangle)$

rcw_latest_candidate($ObjectHistorySet, BarrierFlag$)
900: $CandidateSet := \{Candidate :$ (rcw_order($Candidate, ObjectHistorySet$) $\geq r) \wedge$
901:       $(Candidate.LT.BarrierFlag = BarrierFlag)\}$
902: $Candidate := (Candidate : (Candidate \in CandidateSet) \wedge (Candidate.LT = \mathbf{max}(CandidateSet.LT)))$
903: return $(Candidate)$

rcw_order($Candidate, ObjectHistorySet$) :
1000: return $(|\{s \in U : Candidate \in ObjectHistorySet[s].ReplicaHistory\}|)$

rcw_latest_time($ObjectHistorySet$)
1100: return $(\mathbf{max}(ObjectHistorySet[U].ReplicaHistory.LT))$
```

Figure 3. R/CW classification of an object history set.

3.4 R/CW server pseudo-code

Server-side functions are listed in Figure 4. As presented, replica histories include data for every candidate: this is inefficient. In practice, the data in a candidate is treated specially. Only the data for the latest value candidate is included in the replica history returned to clients. Even then, it is not included in responses to a **write** or **inline_write** since the client knows the data value it sent the server. If the client needs a previous candidate's data, it can request it via its timestamp.

To be safe in the crash recovery failure model, servers must update their state in an atomic step after accepting a candidate. Also due to the crash-recovery failure model, clients re-send requests until they receive a response. Thus, a server may receive repeated requests. To handle repeated requests, a server checks its replica history to determine if it has already accepted the requested candidate (line 1315). To reclaim storage space, servers can prune their replica histories (lines 1324–1328). Unfortunately, if storage space is reclaimed, it is not always possible for servers to determine if they have accepted a candidate. As such, there is (in theory) a tension between bounded storage capacity and client's being able to determine if they established a candidate or not.

Pseudo-code for the server-side function **s_rcw_setup** is listed in Figure 5. This function sets up the candidate for a server to accept based on the conditioned-on object history set and data sent by the client.

```
**s_rcw_initialize**() :
1200: $s.\mathit{ReplicaHistory} := \{\langle \mathbf{0}, \mathbf{0}, \bot \rangle\}$
1201: $\forall s' \in U, s.\alpha[s'] := \mathbf{hmac}(s, s', s.\mathit{ReplicaHistory})$

**s_rcw_request**(*Data*, *ObjectHistorySet*) :
1300: /* Reply to read requests. */
1301: if ($\mathit{ObjectHistorySet} = \bot$) then
1302:    $\mathbf{reply}(s, \text{SUCCESS}, s.\langle \mathit{ReplicaHistory}, \alpha \rangle)$
1303: end if
1304: /* Validate authenticators. */
1305: for each ($s' \in U$) do
1306:    if ($\mathbf{hmac}(s, s', \mathit{ObjectHistorySet}[s'].\mathit{ReplicaHistory}) \neq \mathit{ObjectHistorySet}[s'].\alpha[s]$) then
1307:       $\mathit{ObjectHistorySet}[s].\mathit{ReplicaHistory} := \{\langle \mathbf{0}, \mathbf{0}, \bot \rangle\}$
1308:    end if
1309: end for
1310: /* Setup candidate and determine operation type. */
1311: $\langle \mathit{CWType}, \mathit{Candidate}, \mathit{LT}_{\text{current}} \rangle := \mathbf{s\_rcw\_setup}(\mathit{Data}, \mathit{ObjectHistorySet})$
1312: /* Determine if this is a repeated request. */
1313: if ($\mathit{Candidate} \in s.\mathit{ReplicaHistory}$) then
1314:    /* Reply with success, but send current history. */
1315:    $\mathbf{reply}(s, \text{SUCCESS}, s.\langle \mathit{ReplicaHistory}, \alpha \rangle)$
1316: end if
1317: /* Validate that conditioned-on object history set is current. */
1318: if ($\mathbf{max}(s.\mathit{ReplicaHistory}.\mathit{LT}) > \mathit{LT}_{\text{current}}$) then
1319:    $\mathbf{reply}(s, \text{FAIL}, s.\langle \mathit{ReplicaHistory}, \alpha \rangle)$
1320: end if
1321: atomic
1322:    $s.\mathit{ReplicaHistory} := s.\mathit{ReplicaHistory} \cup \mathit{Candidate}$
1323:    $\forall s' \in U, s.\alpha[s'] := \mathbf{hmac}(s, s', s.\mathit{ReplicaHistory})$
1324:    if ($\mathit{CWType} \in \{\text{WRITE}, \text{INLINE\_WRITE}\}$) then
1325:       /* By definition, the conditioned-on candidate is established. */
1326:       $\mathit{PrunedHistory} := \{\mathit{Candidate}' : (\mathit{Candidate}' \in s.\mathit{ReplicaHistory}) \wedge (\mathit{Candidate}'.\mathit{LT} < \mathit{Candidate}.\mathit{LT}_{\text{CO}})\}$
1327:       $s.\mathit{ReplicaHistory} := s.\mathit{ReplicaHistory} \setminus \mathit{PrunedHistory}$
1328:    end if
1329: end atomic
1330: $\mathbf{reply}(s, \text{SUCCESS}, s.\langle \mathit{ReplicaHistory}, \alpha \rangle)$
```

Figure 4. Server side R/CW pseudo-code for server s.

```
**s_rcw_setup**($Data$, $ObjectHistorySet$) :
1400: $\langle CWType, LatestCandidate, LatestBarrier\rangle :=$ **rcw_classify**($ObjectHistorySet$)
1401: $LT_{\mathrm{CO}} := LatestCandidate.LT$
1402: **if** ($CWType =$ WRITE) **then**
1403:     $LT.Time :=$ **rcw_latest_time**($ObjectHistorySet$)$.Time + 1$   $(= LatestCandidate.LT.Time + 1)$
1404:     $LT.BarrierFlag :=$ FALSE
1405:     $LT.ClientID := ClientID$          /* Client ID is known from authenticated channel. */
1406:     $LT.DataVerifier :=$ **hash**($Data$)
1407:     $LT.OHSVerifier :=$ **hash**($ObjectHistorySet$)
1408:     $LT_{\mathrm{current}} := LatestCandidate.LT$   $(= LT_{\mathrm{CO}})$
1409: **else if** ($CWType =$ BARRIER) **then**
1410:     $Data := \perp$
1411:     $LT.Time :=$ **rcw_latest_time**($ObjectHistorySet$)$.Time + 1$
1412:     $LT.BarrierFlag :=$ TRUE
1413:     $LT.ClientID := ClientID$
1414:     $LT.DataVerifier := \perp$
1415:     $LT.OHSVerifier :=$ **hash**($ObjectHistorySet$)
1416:     $LT_{\mathrm{current}} := LT$
1417: **else if** ($CWType =$ COPY) **then**
1418:     $Data := LatestCandidate.Data$
1419:     $LT.Time :=$ **rcw_latest_time**($ObjectHistorySet$)$.Time + 1$   $(= LatestBarrier.LT.Time + 1)$
1420:     $LT.BarrierFlag :=$ FALSE
1421:     $LT.ClientID := ClientID$
1422:     $LT.DataVerifier :=$ **hash**($Data$)
1423:     $LT.OHSVerifier :=$ **hash**($ObjectHistorySet$)
1424:     $LT_{\mathrm{current}} := LatestBarrier.LT$
1425: **else if** ($CWType =$ INLINE_WRITE) **then**
1426:     $\langle LT, LT_{\mathrm{CO}}, Data\rangle := LatestCandidate$
1427:     $LT_{\mathrm{current}} := LT$
1428: **else**
1429:     /* $CWType =$ INLINE_BARRIER */
1430:     $\langle LT, LT_{\mathrm{CO}}, Data\rangle := LatestBarrier$
1431:     $LT_{\mathrm{current}} := LT$
1432: **end if**
1433: **return** $(\langle CWType, \langle LT, LT_{\mathrm{CO}}, Data\rangle, LT_{\mathrm{current}}\rangle)$
```

Figure 5. Server-side R/CW logic for setting up the candidate to accept.

4 Multi-object operations

In this section we discuss how to extend the R/CW protocol so that conditional-write operations may condition on multiple objects. Multi-object writes provide a different safety guarantee than single-object writes: strict serializability [3] instead of linearizability [5]. Linearizability only applies to protocols that operate on individual objects, whereas strict serializability guarantees that all writes across all objects, even multi-object writes, are partially ordered in a reasonable manner.

A multi-object conditional-write atomically writes a set of objects. To perform such a conditional-write, a client includes a conditioned-on OHS for each object being written. The set of objects and each object's corresponding conditioned-on OHS, are referred to as the multi-object history set (*multi-OHS*). Each server locks its local version of each object in the multi-OHS and validates that each conditioned-on OHS is current. The local locks are acquired in object ID order to avoid the possibility of local deadlocks. Assuming validation passes for all of the objects in the multi-OHS, the server accepts the conditional-write for all the objects in the multi-OHS simultaneously.

So long as a candidate is classified as complete or incomplete, no additional logic is required. However, to repair multi-object conditional-writes, clients must determine which objects are in the multi-OHS. As such, the multi-OHS is included in the timestamp. Note that all objects written by a multi-object conditional-write have the same multi-OHS in their timestamp.

To illustrate multi-object repair, consider a multi-object conditional-write that writes two objects, o_a and o_b. The multi-object conditional-write results in a candidate for each object, c_a and c_b respectively. Now, consider a client that queries o_a and classifies c_a as repairable. To repair c_a, the client must fetch a current object history for o_b because o_b is in the multi-OHS of c_a. If c_a is in fact established, then there could exist a subsequent established candidate at o_b that conditions on c_b. If c_a is not established, then subsequent operations at o_b may preclude c_b from ever being established (e.g., a barrier and a copy that establishes another candidate at o_b with a higher timestamp than c_b). The former requires that c_a be reclassified as complete. The latter requires that c_a be reclassified as incomplete.

Such reclassification of a repairable candidate, based on the objects in its multi-OHS, is called *classification by deduction*. If the repairable candidate lists other objects in its multi-OHS, then classification by deduction must be performed. If classification by deduction does not result in reclassification, then repair is performed. Repair, like in the case of individual objects, consists of barrier and copy operations. The multi-OHS for multi-object barriers and multi-object copies have the same set of objects in them as the multi-OHS of the repairable candidate. Because multi-object operations are atomic, classification by deduction cannot yield conflicting reclassifications: either all of the objects in the multi-OHS are classified as repairable, some are classified complete (implying that all are complete), or some are classified incomplete (implying that all are incomplete).

5 Query/Update (Q/U) protocol

The Query/Update (Q/U) protocol is a variant of the R/CW protocol; it is described, in detail, in the main paper [1]. The correctness of the Q/U protocol is based on the correctness of the R/CW protocol.

This section presents extended pseudo-code for the query/update (Q/U) protocol. Given the Q/U protocol pseudo-code and the R/CW protocol pseudo-code, it is clear the the one is a variant of the other. The Q/U protocol pseudo-code is somewhat longer and more detailed than the R/CW protocol pseudo-code because of the operations-based interface it provides, the need for object syncing, and the inclusion of some performance optimizations in the pseudo-code.

```
structures, types, & enumerations:
1500: /* Enumerations. */
1501: $Class \in \{\text{QUERY}, \text{UPDATE}\}$
1502: $Type \in \{\text{METHOD}, \text{INLINE\_METHOD}, \text{COPY}, \text{BARRIER}, \text{INLINE\_BARRIER}\}$
1503: /* Structures. */
1504: $Operation \equiv \{$
1505:     $Method$                         /* Method to invoke on the object. */
1506:     $Class$                          /* Class of operation. */
1507:     $Argument$                       /* Argument(s) passed into method. */
1508: $\}$
1509: $LT \equiv \{$                       /* Logical timestamp. */
1510:     $Time$                           /* Major component of logical time. */
1511:     $BarrierFlag$                    /* TRUE for barriers. */
1512:     $ClientID$                       /* Client ID. */
1513:     $Operation$                      /* Operation to be performed on conditioned-on object version. */
1514:     $ObjectHistorySet$               /* Conditioned-on ObjectHistorySet. */
1515: $\}$     /* Operation and ObjectHistorySet replaced with single hash in the implementation. */
1516: $Candidate \equiv \{$                /* Candidates are initialized to $\langle \mathbf{0}, \mathbf{0} \rangle$. */
1517:     $LT$                             /* Timestamp of corresponding object version. */
1518:     $LT_{\text{CO}}$                 /* Timestamp of conditioned-on object version. */
1519: $\}$
1520: /* Types. */
1521: $ReplicaHistory \equiv \{Candidate\}$    /* An ordered set of candidates. */
1522: $\alpha \equiv \text{HMAC}[U]$  /* An authenticator is an array of HMACs indexed by server ($U$ is the universe of servers). */
1523: $ObjectHistorySet \equiv \langle ReplicaHistory, \alpha \rangle [U]$    /* An array of replica histories indexed by server. */
```

Figure 6. Enumerations, structures, and data types used in pseudo-code.

In this section we also present a longer example execution than in the main paper. Moreover, the example execution is for a queue object that requires the semantics provided by the Q/U protocol.

5.1 Data structures

Symbols used in the pseudo-code are listed in Table 1 in §3. Enumerations, structures, and types used in the pseudo-code are given in Figure 6.

5.2 Client-side

Pseudo-code for client-side functions is give in Figure 7. The pseudo-code for the query **c_qu_fetch** includes the optimization to handle servers executing queries optimistically if the clients object history set is not current. It also includes the optimization (although not complete pseudo-code) for returning answers from the latest complete object version, even if there is a later incomplete candidate. This optimization avoids queries contending with a single client performing updates.

Pseudo-code for the function **c_qu_quorum_rpc** is given in Figure 8. Like in the R/CW pseudo-code, a rudimentary quorum probing policy is shown. Note **c_qu_quorum_rpc** differs from **c_rcw_quorum_rpc** in some substantive ways. The voting and synthesis of responses performed on lines 2017 to 2020 is to ensure that the answer returned matches that of a benign server. (A Byzantine faulty server could respond with success but supply the incorrect answer.)

Figure 9 lists pseudo-code for the function **qu_classify**. The construction of candidates for inline repair of value candidates and barrier candidates is included in this extended pseudo-code.

```
c_qu_initialize():                                                  /* Client initialization. */
1600: for each ($s \in U$) do
1601:     $ObjectHistorySet[s].ReplicaHistory := \{\langle \mathbf{0}, \mathbf{0} \rangle\}$
1602:     $ObjectHistorySet[s].\alpha := \perp$
1603: end for

c_qu_increment(Argument):                                           /* Example update operation. */
1700: $Operation := \langle \mathbf{inc}, \textsc{update}, Argument \rangle$
1701: $\langle Answer, Order, ObjectHistorySet \rangle :=$ c_qu_quorum_rpc($Operation, ObjectHistorySet$)
1702: while ($Order < q$) do
1703:     $ObjectHistorySet :=$ c_qu_repair($ObjectHistorySet$)
1704:     $\langle Answer, Order, ObjectHistorySet \rangle :=$ c_qu_quorum_rpc($Operation, ObjectHistorySet$)
1705: end while
1706: return ($\langle Answer \rangle$)

c_qu_fetch():                                                       /* Example query operation. */
1800: $Operation := \langle \mathbf{fetch}, \textsc{query}, \perp \rangle$
1801: $\langle Answer, Order, ObjectHistorySet \rangle :=$ c_qu_quorum_rpc($Operation, ObjectHistorySet$)
1802: while ($Order < q$) do
1803:     /* See if query was executed optimistically. */
1804:     $\langle Type, \perp, \perp \rangle :=$ qu_classify($ObjectHistorySet$)
1805:     if ($Type = \textsc{method}$) then
1806:         return ($\langle Answer \rangle$)
1807:     end if
1808:     /* Try and avoid update-query contention */
1809:     if ($Order < r$) then
1810:         /* Eliding details of (i) Determining if the latest two entries in ObjectHistorySet are the incomplete */
1811:         /* candidate and the established candidate it is conditioned on, and (ii) determing the answer */
1812:         /* $Answer'$ returned by the conditioned-on candidate. */
1813:         return ($\langle Answer' \rangle$)
1814:     end if
1815:     $ObjectHistorySet :=$ c_qu_repair($ObjectHistorySet$)
1816:     $\langle Answer, Order, ObjectHistorySet \rangle :=$ c_qu_quorum_rpc($Operation, ObjectHistorySet$)
1817: end while
1818: return ($\langle Answer \rangle$)

c_qu_repair(ObjectHistorySet):                                      /* Deal with failures and contention. */
1900: $\langle Type, \perp, \perp \rangle :=$ qu_classify($ObjectHistorySet$)
1901: while ($Type \neq \textsc{method}$) do
1902:     backoff()                                                   /* Backoff to avoid livelock. */
1903:     /* Perform a barrier or copy (depends on ObjectHistorySet). */
1904:     $\langle \perp, \perp, ObjectHistorySet \rangle :=$ c_qu_quorum_rpc($\perp, ObjectHistorySet$)
1905:     $\langle Type, \perp, \perp \rangle :=$ qu_classify($ObjectHistorySet$)
1906: end while
1907: return ($ObjectHistorySet$)
```

Figure 7. Client side pseudo-code.

```
c_qu_quorum_rpc(Operation, ObjectHistorySet) :          /* Quorum RPC from client to servers. */
2000: $ResponseSet := SuccessSet := \emptyset$
2001: repeat
2002:    /* Eliding probing policy. For simplicity broadcast to all servers. */
2003:    for each $(s \in U \setminus ResponseSet.s)$ do
2004:       send($s$, $Operation$, $ObjectHistorySet$)
2005:    end for
2006:    if (poll() = TRUE) then
2007:       $\langle s, Status, Answer, \langle ReplicaHistory, \alpha \rangle\rangle$ := receive()
2008:       if $(s \notin ResponseSet.s)$ then
2009:          $ObjectHistorySet[s] := \langle ReplicaHistory, \alpha \rangle$
2010:          $ResponseSet := ResponseSet \cup \langle s \rangle$
2011:       end if
2012:       if $(Status = \text{SUCCESS})$ then
2013:          $SuccessSet := SuccessSet \cup \langle Answer, \langle ReplicaHistory, \alpha \rangle\rangle$
2014:       end if
2015:    end if
2016: until $(\exists Q \subseteq ResponseSet : Q \in \mathcal{Q})$
2017: /* Use voting to identify response from benign server. */
2018: $VoteCount$ := max($\forall Response \in ResponseSet$ : qu_count($Response$, $SuccessSet$))
2019: $Response$ := ($Response$ : qu_count($Response$, $SuccessSet$) = $VoteCount$)
2020: return ($\langle Response.Answer$, qu_count($Response$, $SuccessSet$), $ObjectHistorySet\rangle$)
```

Figure 8. Quorum RPC pseudo-code.

```
qu_classify(ObjectHistorySet):
2100: /* Get latest object version, barrier version, and timestamp. */
2101: $LatestObjectVersion$ := qu_latest_candidate($ObjectHistorySet$, FALSE)
2102: $LatestBarrierVersion$ := qu_latest_candidate($ObjectHistorySet$, TRUE)
2103: $LT_{\text{latest}}$ := qu_latest_time($ObjectHistorySet$)
2104: /* Determine which type of operation to perform. */
2105: if $(LT_{\text{latest}} = LatestObjectVersion.LT) \wedge$ (qu_order($LatestObjectVersion$, $ObjectHistorySet$) $\geq q$) then
2106:    $Type$ := METHOD
2107: else if $(LT_{\text{latest}} = LatestObjectVersion.LT) \wedge$ (qu_order($LatestObjectVersion$, $ObjectHistorySet$) $\geq r$) then
2108:    $Type$ := INLINE_METHOD
2109: else if $(LT_{\text{latest}} = LatestBarrierVersion.LT) \wedge$ (qu_order($LatestBarrierVersion$, $ObjectHistorySet$) $\geq q$) then
2110:    $Type$ := COPY
2111: else if $(LT_{\text{latest}} = LatestBarrierVersion.LT) \wedge$ (qu_order($LatestBarrierVersion$, $ObjectHistorySet$) $\geq r$) then
2112:    $Type$ := INLINE_BARRIER
2113: else
2114:    $Type$ := BARRIER
2115: end if
2116: return ($\langle Type, LatestObjectVersion, LatestBarrierVersion \rangle$)

qu_latest_candidate(ObjectHistorySet, BarrierFlag)
2200: $CandidateSet := \{Candidate$ : (qu_order($Candidate$, $ObjectHistorySet$) $\geq r) \wedge$
2201:       $(Candidate.LT.BarrierFlag = BarrierFlag)\}$
2202: $Candidate := (Candidate : (Candidate \in CandidateSet) \wedge (Candidate.LT =$ max$(CandidateSet.LT)))$
2203: return ($Candidate$)

qu_order(Candidate, ObjectHistorySet) :
2300: return $(|\{s \in U : Candidate \in ObjectHistorySet[s].ReplicaHistory\}|)$

qu_latest_time(ObjectHistorySet)
2400: return (max($ObjectHistorySet[U].ReplicaHistory.LT$))
```

Figure 9. Classification of an object history set.

5.3 Server-side

Figure 10 lists pseudo-code for server-side functions. The logic for inline repair of value candidates and barrier candidates is included. Optimistic execution of queries is shown. The logic necessary to handle repeated requests is given (i.e., checking for the candidate in the replica history, and storing/retrieving answers in addition to object versions). Pruning replica histories (and garbage collecting object versions and answers) is shown.

Figure 11 lists pseudo-code for the server-side function **s_qu_setup**. The logic for constructing candidates for inline repairs of value candidates and barrier candidates is included.

Figure 12 lists pseudo-code for object sync. Like **c_qu_quorum_rpc**, **s_qu_object_sync** broadcasts its requests which is inefficient. In the implementation, servers are probed based on the information in the conditioned-on object history set.

```
s_qu_initialize() :                                                        /* Initialize server s. */
2500: $s.ReplicaHistory := \{\langle \mathbf{0}, \mathbf{0} \rangle\}$
2501: $\forall s' \in U, s.\alpha[s'] :=$ hmac$(s, s', s.ReplicaHistory)$

s_qu_request($Operation$, $ObjectHistorySet$) :                            /* Handle request at server s. */
2600: $Answer := \bot$                          /* BARRIER and INLINE_BARRIER return null answers. */
2601: /* Validate authenticators. */
2602: for each $(s' \in U)$ do
2603:    if (hmac$(s, s', ObjectHistorySet[s'].ReplicaHistory) \neq ObjectHistorySet[s'].\alpha[s]$) then
2604:       $ObjectHistorySet[s'].ReplicaHistory := \{\langle \mathbf{0}, \mathbf{0} \rangle\}$   /* Cull invalid ReplicaHistorys. */
2605:    end if
2606: end for
2607: /* Setup candidate and determine operation type. */
2608: $\langle Type, \langle LT, LT_{CO} \rangle, LT_{current} \rangle :=$ s_qu_setup$(Operation, ObjectHistorySet)$
2609: /* Determine if this is a repeated request. */
2610: if $(\langle LT, LT_{CO} \rangle \in s.ReplicaHistory)$ then
2611:    $\langle \bot, Answer \rangle :=$ retrieve$(LT)$
2612:    reply$(s$, SUCCESS, $Answer, s.\langle ReplicaHistory, \alpha \rangle)$   /* Reply with success, but send current history. */
2613: end if
2614: /* Validate that conditioned-on object history set is current. */
2615: if (qu_latest_time$(s.ReplicaHistory) > LT_{current}$) then
2616:    /* Optimistically execute query operations on latest object version. */
2617:    if $(Operation.Class =$ QUERY$)$ then
2618:       $Object :=$ retrieve(qu_latest_time$(s.ReplicaHistory)$)   /* Retrieve latest local object version. */
2619:       $\langle \bot, Answer \rangle := Operation.Method(Object, Operation.Argument)$
2620:    end if
2621:    /* $Answer = \bot$ except in the case of optimistic query execution. */
2622:    reply$(s$, FAIL, $Answer, s.\langle ReplicaHistory, \alpha \rangle)$
2623: end if
2624: if $(Type \in \{$METHOD, INLINE_METHOD, COPY$\})$ then
2625:    /* Retrieve conditioned-on object so that method can be invoked. */
2626:    $\langle Object, Answer \rangle :=$ retrieve$(LT_{CO})$                /* Attempt to retrieve it locally. */
2627:    if $((Object = \bot) \wedge (LT_{CO} > \mathbf{0}))$ then
2628:       $\langle Object, Answer \rangle :=$ s_qu_object_sync$(LT_{CO})$   /* Retrieve from other servers via object sync. */
2629:    end if
2630: end if
2631: if $(Type \in \{$METHOD, INLINE_METHOD$\})$ then
2632:    $\langle Object, Answer \rangle := Operation.Method(Object, Operation.Argument)$
2633:    if $(Operation.Class =$ QUERY$)$ then
2634:       reply$(s$, SUCCESS, $Answer, s.\langle ReplicaHistory, \alpha \rangle)$
2635:    end if
2636: end if
2637: atomic
2638:    $s.ReplicaHistory := s.ReplicaHistory \cup \{\langle LT, LT_{CO} \rangle\}$   /* Update replica history. */
2639:    $\forall s' \in U, s.\alpha[s'] :=$ hmac$(s, s', s.ReplicaHistory)$      /* Update authenticator. */
2640:    if $(Type \in \{$METHOD, INLINE_METHOD, COPY$\})$ then
2641:       store$(LT, \langle Object, Answer \rangle)$   /* Store object version locally indexed by logical timestamp. */
2642:    end if
2643:    if $(Type \in \{$METHOD, INLINE_METHOD$\})$ then
2644:       /* By definition, the conditioned-on candidate is established. */
2645:       $PrunedHistory := \{Candidate : (Candidate \in s.ReplicaHistory) \wedge (Candidate.LT < LT_{CO})\}$
2646:       $s.ReplicaHistory := s.ReplicaHistory \setminus PrunedHistory$
2647:       remove_garbage$(PrunedHistory.LT)$   /* Remove local object versions of pruned candidates. */
2648:    end if
2649: end atomic
2650: reply$(s$, SUCCESS, $Answer, s.\langle ReplicaHistory, \alpha \rangle)$
```

Figure 10. Server side pseudo-code.

```
$\mathbf{s\_qu\_setup}(\mathit{Operation}, \mathit{ObjectHistorySet})$ :
2700: $\langle \mathit{Type}, \mathit{LatestObjectVersion}, \mathit{LatestBarrierVersion} \rangle := \mathbf{qu\_classify}(\mathit{ObjectHistorySet})$
2701: $LT_{\mathrm{CO}} := \mathit{LatestObjectVersion}.LT$
2702: if $(\mathit{Type} = \textsc{method})$ then
2703:     $LT.\mathit{Time} := \mathbf{qu\_latest\_time}(\mathit{ObjectHistorySet}).\mathit{Time} + 1$
2704:     $LT.\mathit{BarrierFlag} := \textsc{false}$
2705:     $LT.\mathit{ClientID} := \mathit{ClientID}$                    /* Client ID is known from authenticated channel. */
2706:     $LT.\mathit{Operation} := \mathit{Operation}$
2707:     $LT.\mathit{ObjectHistorySet} := \mathit{ObjectHistorySet}$
2708:     $LT_{\mathrm{current}} := \mathit{LatestObjectVersion}.LT \quad (= LT_{\mathrm{CO}})$
2709: else if $(\mathit{Type} = \textsc{barrier})$ then
2710:     $LT.\mathit{Time} := \mathbf{qu\_latest\_time}(\mathit{ObjectHistorySet}).\mathit{Time} + 1$
2711:     $LT.\mathit{BarrierFlag} := \textsc{true}$
2712:     $LT.\mathit{ClientID} := \mathit{ClientID}$
2713:     $LT.\mathit{Operation} := \perp$
2714:     $LT.\mathit{ObjectHistorySet} := \mathit{ObjectHistorySet}$
2715:     $LT_{\mathrm{current}} := LT$
2716: else if $(\mathit{Type} = \textsc{copy})$ then
2717:     $LT.\mathit{Time} := \mathbf{qu\_latest\_time}(\mathit{ObjectHistorySet}).\mathit{Time} + 1$
2718:     $LT.\mathit{BarrierFlag} := \textsc{false}$
2719:     $LT.\mathit{ClientID} := \mathit{ClientID}$
2720:     $LT.\mathit{Operation} := LT_{\mathrm{CO}}.\mathit{Operation}$
2721:     $LT.\mathit{ObjectHistorySet} := \mathit{ObjectHistorySet}$
2722:     $LT_{\mathrm{current}} := \mathit{LatestBarrierVersion}.LT$
2723: else if $(\mathit{Type} = \textsc{inline\_method})$ then
2724:     $LT := \mathit{LatestObjectVersion}.LT$
2725:     $LT_{\mathrm{CO}} := \mathit{LatestObjectVersion}.LT_{\mathrm{CO}}$
2726:     $LT_{\mathrm{current}} := LT$
2727: else
2728:     /* $\mathit{Type} = \textsc{inline\_barrier}$ */
2729:     $LT := \mathit{LatestBarrierVersion}.LT$
2730:     $LT_{\mathrm{CO}} := \mathit{LatestBarrierVersion}.LT_{\mathrm{CO}}$
2731:     $LT_{\mathrm{current}} := LT$
2732: end if
2733: return $(\langle \mathit{Type}, \langle LT, LT_{\mathrm{CO}} \rangle, LT_{\mathrm{current}} \rangle)$
```

Figure 11. Server side pseudo-code to set up candidate and identify type of operation.

```
s_qu_object_sync($LT$)
2800: /* Pseudo-code does not deal with servers pruning replica histories or garbage collecting object versions. */
2801: $ResponseSet := \emptyset$
2802: repeat
2803:    /* Eliding details of probing based on object history set. */
2804:    for each ($s \in U$) do
2805:       send($s, LT$)
2806:    end for
2807:    if (poll() = TRUE) then
2808:       $\langle Object, Answer \rangle :=$ receive()
2809:       $ResponseSet := ResponseSet \cup \{\langle Object, Answer \rangle\}$
2810:       if (qu_count($\langle Object, Answer \rangle, ResponseSet$) $> b + 1$) then
2811:          return ($\langle Object, Answer \rangle$)
2812:       end if
2813:    end if
2814: until (FALSE)

s_qu_object_source($LT$)
2900: $\langle Object, Answer \rangle :=$ c_qu_fetch($LT$)
2901: if ($Object \neq \perp$) then
2902:    reply($s, \langle Object, Answer \rangle$)
2903: end if

qu_count($Element, Set$) :
3000: return ($|\{Element : Element \in Set\}|$)
```

Figure 12. Pseudo-code for object syncing at server s.

5.4 Example Q/U protocol execution

An example execution of the Q/U protocol is given in Table 2 for a queue object. The caption explains the structure of, and notation used in, the table. The example is for an object that exports three methods: **enq** (an update that enqueues a value), **deq** (an update that dequeues a value), and **front** (a query that returns the value at the front of the queue). The server configuration is based on the smallest quorum system for $b = 1$. Clients perform optimistic queries for the **front** method; the conditioned-on OHS sent with the **enq** and **deq** methods is not shown in the table. The sequence of client operations is divided into six sequences of interest by horizontal double lines: the initial queue state, four illustrative sequences of operations, and the final queue state. For illustrative purposes, clients X and Z interact with the object's preferred quorum (the first five servers) and Y a non-preferred (the last five servers).

In the first sequence, the queue state is initialized to the well known null value at the well known logical time zero. The second sequence demonstrates failure- and concurrency-free execution: client X performs a **front** and **enq** that each complete in a single phase. Client Y performs a **front** in the second sequence that requires repair. Since there is no contention, Y performs inline repair. Server s_5 performs object syncing to process the inline repair request. It also optimistically performs the **front** after object syncing and returns ♣ to the client.

In the fourth sequence, concurrent updates by X and Y both abort because of contention; however, X establishes a candidate. It is possible for an update to abort, but establish a candidate because the established threshold is b less than quorum size (4 in this case). At server s_4, the **enq** from Y arrives before the **enq** from X. As such, it returns its replica history $\{\langle 3, 1\rangle, \langle 1, \mathbf{0}\rangle\}$ with FAIL. The candidate $\langle 3, 1\rangle$ in this replica history dictates that the timestamp of the barrier be $4\mathbf{b}$. For illustrative purposes, Y backs off. Client X subsequently completes barrier and copy operations.

Notice though that the replica histories returned by the servers in response to Y's **enq** requests are pruned. Servers prune their replica histories whenever they accept an update operation (or an inline repair), since such operations are conditioned on established candidates. Servers do not prune their replica histories when they accept copy and barrier operations, since such operations may condition on potential candidates. For example, server s_1 returns $\{\langle 2, 1\rangle, \langle 1, \mathbf{0}\rangle\}$, rather than $\{\langle 2, 1\rangle, \langle 1, \mathbf{0}\rangle, \langle \mathbf{0}, \mathbf{0}\rangle\}$, because the object history set sent by X with **enq** operation $\langle 2, 1\rangle$ proved that $\langle 1, \mathbf{0}\rangle$ was established.

In the fifth sequence, X crashes during an **enq** operation and yields a potential candidate. The remaining clients Y and Z perform concurrent **front** operations at different quorums; this illustrates how a potential candidate can be classified as either repairable or incomplete. Y and Z concurrently attempt a barrier and inline repair respectively. In this example, Y establishes a barrier before Z's inline repair requests arrive at servers s_3 and s_4. As such, client Z aborts its inline repair operation and backs off. Subsequently, Y completes a copy operation and establishes a candidate $\langle 8, 5\rangle$ that copies forward the established candidate $\langle 5, 2\rangle$. Then, Y dequeues a value from the queue. After backing off, Z attempts a **front** operation that requires an inline repair to complete. Notice that queries such as **front** can be piggy-backed on inline repairs. Finally, in the sixth sequence, the final server state is listed.

References

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine fault-tolerant services. *Symposium on Operating Systems Principles*, 2005.

Operation	s_0	s_1	s_2	s_3	s_4	s_5	**Result**
Initial queue state	$\{\mathbf{0}\},\perp$	$\{\mathbf{0}\},\perp$	$\{\mathbf{0}\},\perp$	$\{\mathbf{0}\},\perp$	$\{\mathbf{0}\},\perp$	$\{\mathbf{0}\},\perp$	
X completes **front**()	$\{\mathbf{0}\},\perp$	$\{\mathbf{0}\},\perp$	$\{\mathbf{0}\},\perp$	$\{\mathbf{0}\},\perp$	$\{\mathbf{0}\},\perp$		**0** complete, return $\perp$
X completes **enq**($\clubsuit$)	$\langle 1,\mathbf{0}\rangle,\perp,[\clubsuit]$	$\langle 1,\mathbf{0}\rangle,\perp,[\clubsuit]$	$\langle 1,\mathbf{0}\rangle,\perp,[\clubsuit]$	$\langle 1,\mathbf{0}\rangle,\perp,[\clubsuit]$	$\langle 1,\mathbf{0}\rangle,\perp,[\clubsuit]$		1 established
Y begins **front**()...		$\{1,\mathbf{0}\},\clubsuit$	$\{1,\mathbf{0}\},\clubsuit$	$\{1,\mathbf{0}\},\clubsuit$	$\{1,\mathbf{0}\},\clubsuit$	$\{\mathbf{0}\},\perp$	1 repairable
...Y performs **inline**()						$\langle 1,\mathbf{0}\rangle,\clubsuit,[\clubsuit]$	1 complete, return $\clubsuit$
X attempts **enq**($\diamondsuit$)...	$\langle 2,1\rangle,\perp,[\clubsuit\diamondsuit]$	$\langle 2,1\rangle,\perp,[\clubsuit\diamondsuit]$	$\langle 2,1\rangle,\perp,[\clubsuit\diamondsuit]$	$\langle 2,1\rangle,\perp,[\clubsuit\diamondsuit]$	$\{3,1\},\text{FAIL}$		2 potential
Y attempts **enq**($\heartsuit$)		$\{2,1\},\text{FAIL}$	$\{2,1\},\text{FAIL}$	$\{2,1\},\text{FAIL}$	$\langle 3,1\rangle,\perp,[\clubsuit\heartsuit]$	$\langle 3,1\rangle,\perp,[\clubsuit\heartsuit]$	Y backs off
...X completes **barrier**()	$\langle 4\mathbf{b},2\rangle,\perp$	$\langle 4\mathbf{b},2\rangle,\perp$	$\langle 4\mathbf{b},2\rangle,\perp$	$\langle 4\mathbf{b},2\rangle,\perp$	$\langle 4\mathbf{b},2\rangle,\perp$		4**b** established
...X completes **copy**()	$\langle 5,2\rangle,\perp,[\clubsuit\diamondsuit]$	$\langle 5,2\rangle,\perp,[\clubsuit\diamondsuit]$	$\langle 5,2\rangle,\perp,[\clubsuit\diamondsuit]$	$\langle 5,2\rangle,\perp,[\clubsuit\diamondsuit]$	$\langle 5,2\rangle,\perp,[\clubsuit\diamondsuit]$		5 established
X crashes in **enq**($\spadesuit$)	$\langle 6,5\rangle,\perp,[\clubsuit\diamondsuit\spadesuit]$	$\langle 6,5\rangle,\perp,[\clubsuit\diamondsuit\spadesuit]$	$\langle 6,5\rangle,\perp,[\clubsuit\diamondsuit\spadesuit]$				6 potential
Y begins **front**()...		$\{6,5\},\clubsuit$	$\{6,5\},\clubsuit$	$\{5,4\mathbf{b},2,1\},\clubsuit$	$\{5,4\mathbf{b},2,1\},\clubsuit$	$\{3,1\},\clubsuit$	6 incom., 5 repair.
Z begins **front**()...	$\{6,5\},\clubsuit$	$\{6,5\},\clubsuit$	$\{6,5\},\clubsuit$	$\{5,4\mathbf{b},2,1\},\clubsuit$	$\{5,4\mathbf{b},2,1\},\clubsuit$		6 repairable
...Y completes **barrier**()		$\langle 7\mathbf{b},5\rangle,\perp$	$\langle 7\mathbf{b},5\rangle,\perp$	$\langle 7\mathbf{b},5\rangle,\perp$	$\langle 7\mathbf{b},5\rangle,\perp$	$\langle 7\mathbf{b},5\rangle,\perp$	7**b** established
...Z attempts **inline**()				$\{7\mathbf{b},...,1\},\text{FAIL}$	$\{7\mathbf{b},...,1\},\text{FAIL}$		Z backs off
...Y completes **copy**()		$\langle 8,5\rangle,\clubsuit,[\clubsuit\diamondsuit]$	$\langle 8,5\rangle,\clubsuit,[\clubsuit\diamondsuit]$	$\langle 8,5\rangle,\clubsuit,[\clubsuit\diamondsuit]$	$\langle 8,5\rangle,\clubsuit,[\clubsuit\diamondsuit]$	$\langle 8,5\rangle,\clubsuit,[\clubsuit\diamondsuit]$	8 established, ret. $\clubsuit$
Y completes **deq**()		$\langle 9,8\rangle,\clubsuit,[\diamondsuit]$	$\langle 9,8\rangle,\clubsuit,[\diamondsuit]$	$\langle 9,8\rangle,\clubsuit,[\diamondsuit]$	$\langle 9,8\rangle,\clubsuit,[\diamondsuit]$	$\langle 9,8\rangle,\clubsuit,[\diamondsuit]$	9 complete, return $\clubsuit$
Z begins **front**()...	$\{6,5\},\clubsuit$	$\{9,8\},\diamondsuit$	$\{9,8\},\diamondsuit$	$\{9,8\},\diamondsuit$	$\{9,8\},\diamondsuit$		9 repairable
...Z completes **inline**()	$\langle 9,8\rangle,\diamondsuit,[\diamondsuit]$						9 complete, return $\diamondsuit$
Final queue state	$\{9,8\},[\diamondsuit]$	$\{9,8\},[\diamondsuit]$	$\{9,8\},[\diamondsuit]$	$\{9,8\},[\diamondsuit]$	$\{9,8\},[\diamondsuit]$	$\{9,8\},[\diamondsuit]$	

Table 2. Example Q/U protocol execution for a queue object that supports **enq** (update), **deq** (update), and **front** (query) operations. Operations performed by three clients (X, Y, and Z) are listed in the left column. The middle columns lists candidates stored by and replies (replica histories or status codes) returned by six benign servers ($s_0, ..., s_5$). For **enq**, **deq**, **copy**, and **inline** operations, the triple listed is the timestamp/conditioned-on timestamp pair, answer from the server, and queue state. If an operation is rejected, because the conditioned-on OHS is not current, then the server reply is listed: the server's replica history and FAIL. Excepting queue state, the listing for **barrier** operations is the same as the other update operations. For **front** operations, a query operation, the replica history and answer returned by the server are listed. The results listed in the right column that identify candidates as established or potential are based on the servers being benign. Time "flows" from the top row to the bottom row. Candidates are denoted $\langle LT, LT_{\mathrm{CO}}\rangle$. Only $LT.Time$ with "**b**" appended for barriers is shown for timestamps (i.e., client ID, operation, and object history set are not shown). Replica historires are denoted $\{LT_3, LT_2, ...\}$. Only the candidate's LT, not the LT_{CO}, is listed in the replica history. Queue state is listed, in order, delimited by square brackets "[" and "]".

[2] Marcos K. Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, **13**(2):99–125. Springer-Verlag, 2000.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[4] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: double-ended queues as an example. *International Conference on Distributed Computing Systems* (Providence, RI, 19–22 May 2003), pages 522–529. IEEE, 2003.

[5] Maurice P. Herlihy and Jeanette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, **12**(3):463–492. ACM, July 1990.

[6] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, **4**(3):382–401. ACM, July 1982.

[7] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, **11**(4):203–213. Springer-Verlag, 1998.

[8] Dahlia Malkhi, Michael Reiter, and Avishai Wool. The load and availability of Byzantine quorum systems. *SIAM Journal of Computing*, **29**(6):1889–1906. Society for Industrial and Applied Mathematics, April 2000.

[9] Fred J. Meyer and Dhiraj K. Pradhan. Consensus with dual failure modes. *IEEE Transactions on Parallel and Distributed Systems*, **2**(2):214–222. IEEE, April 1991.

[10] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. *Symposium on Reliable Distributed Systems* (10–12 October 1988), pages 93–100. IEEE, 1988.