

# Refined Quorum Systems

Rachid Guerraoui<sup>1</sup> and Marko Vukolić<sup>1,2</sup>

<sup>1</sup> School of Computer and Communication Sciences, EPFL, INR, Station 14, CH-1015 Lausanne, Switzerland

<sup>2</sup> IBM Zurich Research Laboratory, Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland  
rachid.guerraoui@epfl.ch, mvu@zurich.ibm.com

## Technical Report LPD-REPORT-2007-002

February 8, 2007

last revision April 23, 2009

**Abstract.** It is considered good distributed computing practice to devise object implementations that tolerate contention, periods of asynchrony and a large number of failures, but perform fast if few failures occur, the system is synchronous and there is no contention. This paper initiates the first study of quorum systems that help design such implementations by encompassing, at the same time, optimal resilience, as well as *optimal best-case complexity*.

We introduce the notion of a *refined* quorum system (RQS) of some set  $S$  as a set of three classes of subsets (quorums) of  $S$ : first class quorums are also second class quorums, themselves being also third class quorums. First class quorums have large intersections with all other quorums, second class quorums typically have smaller intersections with those of the third class, the latter simply correspond to traditional quorums. Intuitively, under uncontended and synchronous conditions, a distributed object implementation would expedite an operation if a quorum of the first class is accessed, then degrade gracefully depending on whether a quorum of the second or the third class is accessed. Our notion of refined quorum system is devised assuming a general adversary structure, and this basically allows algorithms relying on refined quorum systems to relax the assumption of independent process failures, often questioned in practice.

We illustrate the power of refined quorums by introducing two new optimal Byzantine-resilient distributed object implementations: an atomic storage and a consensus algorithm. Both match previously established resilience and best-case complexity lower bounds, closing open gaps, as well as new complexity bounds we establish here. Each of our algorithms is representative of a different class of architectures, highlighting the generality of the refined quorum abstraction.

**Keywords:** Quorums, Consensus, Complexity, Shared-memory emulations, Byzantine failures.

# 1 Introduction

## 1.1 Background

Quorum systems are powerful mathematical tools to reason about distributed implementations shared objects, in particular read/write storage (e.g., [4, 27, 40]) and consensus [8, 13, 34] abstractions. More specifically, quorum systems have been used (either explicitly or implicitly) to reason about distributed algorithms that tolerate process failures, as well as arbitrarily long periods of asynchrony, also called indulgent algorithms [19]. Originally, a quorum system was defined as a set of subsets that intersect [16], and this notion was key to reasoning about crash-resilient asynchronous algorithms. More sophisticated forms of quorum systems have been introduced to cope with Byzantine (malicious) failures [37]: these require larger intersections among subsets [40].

However, while being very useful to reason about the resilience dimension, traditional quorums (be they simple or Byzantine) are not adequate to capture the complexity dimension. This is particularly important given the appealing nature of *optimistic* distributed object implementations, e.g., [1, 7, 10, 12, 18, 20, 31, 36, 41, 46, 53]. In addition to being indulgent, these implementations are also geared to reduce best-case complexity, i.e., latency under situations of synchrony and no-contention, which are typically argued to be frequent in practice. More specifically, these implementations are tuned to expedite operations in uncontended and synchronous situations, provided “*enough*” servers are accessed. This very notion of “*enough servers to expedite an operation*” is crucial, but is not captured by traditional quorum systems. It is natural to seek for a mathematical abstraction that captures it in precise yet general terms. This was the motivation of this work.

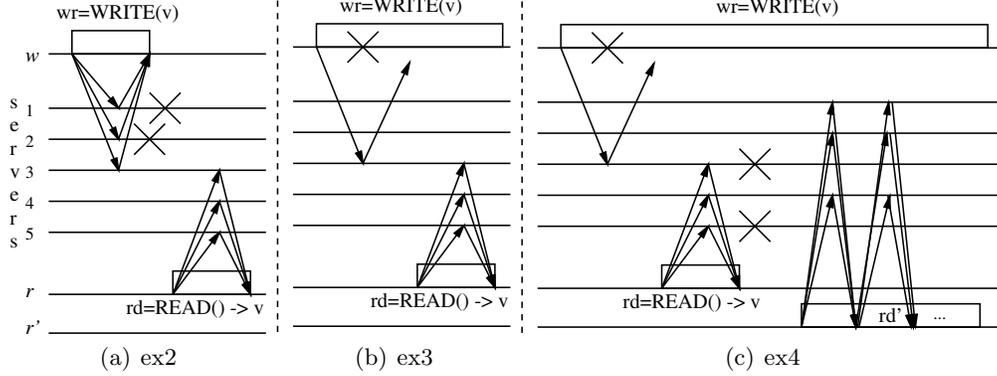
## 1.2 Example

To illustrate the motivation, consider the simple context of a crash-resilient asynchronous implementation of a wait-free atomic storage over a set of server processes [4]. It is known [11] that no optimally resilient atomic storage algorithm can have both **reads** and **writes** complete in a single communication round-trip (we simply say round), even if a single writer is involved (SWMR). For instance, the classical, optimally crash-resilient solution [4] (that assumes a majority of correct processes) requires two rounds for a **read**.

As we discussed earlier, it is practically appealing to look into best-case complexity and ask if it is possible to expedite *both reads* and *writes* within a single round in a synchronous and contention-free period. Clearly, if the reader (resp. the writer) access all servers in the first round, then it can immediately return a valid response. But do we need to access all servers for that? How many servers actually *need* to be accessed to achieve such a *fast termination* in best-case conditions?

Consider 5 servers implementing a crash-tolerant atomic wait-free storage assuming  $t = 2$  server failures (optimal resilience). We argue below that any algorithm that greedily expedites read/write operations in one round during synchronous and contention-free periods whenever  $S - t = 3$  servers are accessed, violates atomicity. This is depicted through several executions of such an algorithm (Figure 1):

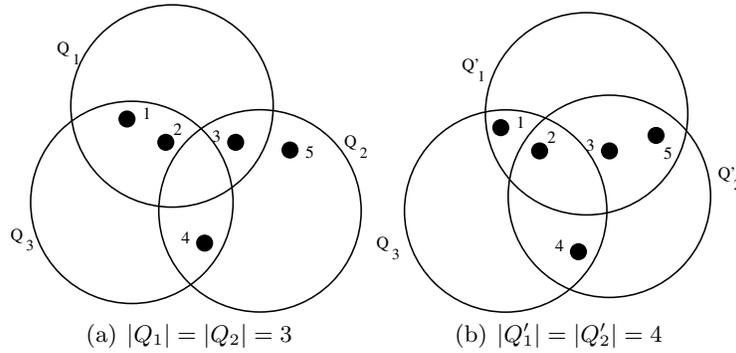
1. In the first execution (*ex1*), writer  $w$  invokes  $wr = \text{write}(v)$  and servers 4 and 5 are faulty. Then,  $wr$  writes value  $v$  into the subset of servers  $Q_1 = \{1, 2, 3\}$  and completes in a single round.
2. The second execution (*ex2*, Fig. 1(a)) is slightly different because servers 4 and 5 are actually correct. Yet  $wr$  also completes in a single round, after writing in  $Q_1$ . Then servers 1 and 2 crash



**Fig. 1.** Violation of atomicity in case the single-round operations access only 3 servers.

and a read  $rd$  (by the reader  $r$ ) is invoked. Assuming synchrony and no contention,  $rd$  accesses server set  $Q_2 = \{3, 4, 5\}$  and completes in a single round.

3. The third execution ( $ex3$ , Fig. 1(b)) is similar to  $ex2$  except that (1) the write is incomplete and writes only to server 3, (2) servers 1 and 2 (i.e., servers from the set  $Q_2 \setminus Q_1$ ) are correct, but the communication between the reader and servers from  $Q_2 \setminus Q_1$  is delayed. Read  $rd$  does not distinguish  $ex3$  from  $ex2$  and completes in a single round, returning  $v$ .
4. Finally, the fourth execution ( $ex4$ , Fig. 1(c)) extends  $ex3$  by: (1) the crash of servers 3 and 5 and (2) the invocation of read  $rd'$  by a different reader  $r'$ . This reader cannot return  $v$  using  $Q_3 = \{1, 2, 4\}$  regardless of how many rounds are used. Atomicity is violated.



**Fig. 2.** Quorum intersections

Essentially, atomicity is violated because  $Q_1 \cap Q_2 \cap Q_3 = \emptyset$  (Figure 2(a)). On the other hand, we can devise a storage algorithm that achieves fast termination whenever 4 servers are accessed.

For instance, consider the following algorithm, a variation of [4], in which all servers maintain 2 timestamp/value variables  $pw$  and  $w$ :

- On invoking  $wr = \text{write}(v)$ , as in [4], the writer increments its local timestamp  $ts$  and assigns it to value  $v$  and writes the pair to servers'  $pw$  variable. However, unlike in [4], the write completes in a single round only if it writes  $\langle ts, v \rangle$  to 4 servers, say  $Q'_1 = \{1, 2, 3, 5\}$ . Otherwise, if the writer

reaches only 3 servers in the first round (after waiting for some pre-specified time, complying with synchrony assumptions), the writer invokes the second round of `write` and writes  $\langle ts, v \rangle$  to servers'  $w$  variable. The `write` completes when the writer receives *acks* in the second round from any 3 servers.

- On the other hand, the first round of `read`, similarly as in [4], collects the servers' local copies  $pw$  and  $w$ . The reader selects a timestamp/value pair  $c_{max} = \langle ts_{max}, v \rangle$  with the highest timestamp. Unlike in [4], the `read` completes (and returns  $v$ ) at the end of the first round, if  $c_{max}$  is read in 3 different  $pw$  fields, or in some  $w$  field. It is very important to notice that this is always the case if there is no `read/write` contention and the reader receives a response from 4 (or more) servers in the first round, say  $Q'_2 = \{2, 3, 4, 5\}$ . Intuitively, the `read` may safely return after the first round if it makes sure that it leaves behind at least 3 servers with the knowledge of the latest value. This is the case both if the `read` accesses 3 servers from  $Q'_1$  (e.g.,  $Q'_1 \cap Q_2 = \{2, 3, 5\}$ ) that report the latest value in their  $pw$  variables, or if the reader selects the highest value from a server  $w$  variable, meaning that the writer already informed 3 servers about the latest value.
- Otherwise, if there is contention, or only 3 servers are available, the reader may not be able to return the value at the end of the first round. In this case, after reaching 3 servers in the first round, the reader proceeds to the second round, in which, as in [4], the reader writes back  $c_{max} = \langle ts_{max}, v \rangle$  to servers'  $pw$  field. The `read` completes when the reader receives *acks* from any 3 servers in the second round.

In the above example, the key to ensuring atomicity while allowing both `reads` and `writes` to terminate in a single round is to have  $Q'_1 \cap Q'_2 \cap Q_3 \neq \emptyset$ , where  $Q_3$  is any quorum in the system (in our case any subset of 3 or more servers). Namely (Figure 2(b)), in a system of 5 elements, any two subsets of 4 elements intersect with any subset of 3 elements. Basically, boosting complexity requires to access subsets of servers that have larger intersections than traditional quorums. The above example is (relatively) simple because we considered: a) crash failures only, b) a threshold adversary (at most  $t$  faulty processes) and c) no graceful degradation (i.e., achieving the next best possible latencies, when the best possible one (e.g., a single round) cannot be achieved).

The motivation underlying this paper is precisely to characterize the required intersection properties in a precise and general manner. We aim at a characterization that is necessary and sufficient for optimizing the best-case complexity of various distributed object implementations, in various failure models, under various adversary structures, and also considering graceful degradation.

### 1.3 Contributions

This paper introduces the notion of *refined quorum systems (RQS)*. In short, a refined quorum system of some set of elements  $S$  is a set of three classes of subsets (quorums) of  $S$ : first class quorums are also second class quorums, which are also third class quorums. Quorums of the first class have large intersections with quorums of other classes, those of the second class typically have smaller intersections with those of the third class, the latter simply correspond to traditional quorums. In the context of a distributed object implementation, the set  $S$  would typically contain the set of fault-prone server processes over which some object abstraction (e.g., storage or consensus) is implemented.

Intuitively, under uncontended and synchronous conditions, a distributed object implementation would expedite an operation if a quorum of the first class is available, then degrade gracefully depending on whether a quorum of the second or the third class is available. We argue that our

quorum notion is, in a sense, complete: there is no need for further refinement of quorums with the goal of optimizing best-case efficiency. Indeed, the properties provided by our third class quorums are anyway necessary for hindering the partitioning of the asynchronous system, which is key to any resilient distributed object implementation. Hence, there is no need to consider weaker intersection properties. Moreover, and as we show in this paper, optimally resilient and best-case efficient implementations of the seminal register and consensus abstractions have exactly *three* possible latencies under uncontended and synchronous conditions. This observation is of independent interest.

Our refined quorum systems are designed to handle a general adversary structure in which various subsets of processes can collude to defeat the protocol [26, 29, 40]. With such a general structure, we relax the often criticized assumption, of independent and identically distributed failures [1, 7, 10, 12, 18, 20, 36, 41, 46, 53].

We illustrate the power of our notion of refined quorum systems by introducing two new atomic object implementations. Each algorithm is interesting in its own right and is, in a precise sense, the first fully optimal protocol of its kind in terms of best case complexity.

- Our first object implementation is a new Byzantine-resilient asynchronous distributed storage algorithm implementing the atomic register abstraction. Such algorithms constitute an active area of research and are appealing alternatives to classical centralized storage systems based on specialized hardware [48]. The challenge when devising distributed storage algorithms is to ensure that *reads* and *writes* have low latency in most frequent situations, while (a) tolerating asynchrony and the failures of a large number of base servers (typically commodity disks) as well as any number of clients that access the storage (wait-freedom [24]) and (b) ensuring strong consistency (ideally atomicity [25, 33]). Using a refined quorum system, we present an atomic wait-free storage algorithm that combines optimal resilience with the lowest possible *read/write* latency in best-case conditions (no-contention and synchrony). Under such conditions, our algorithm expedites storage operations (*reads* and *writes*) in a single communication round-trip (or simply, round) if a first class quorum is accessed, in two rounds if a second class quorum is accessed and in three rounds otherwise. The latter case is when a third class quorum is available which is a necessary condition for resilience anyway. Our algorithm does not rely on any data authentication primitive, and matches the resilience and complexity lower bounds of [20, 42] (even when these bounds are extended to a general adversary structure), together with a new bound we establish in this paper. Our new bound captures the best-case complexity of gracefully degrading atomic storage implementations.
- Our second algorithm implements a Byzantine-resilient consensus abstraction in the general state machine replication (SMR) framework of [34], distinguishing different process roles: *proposers* that propose values to be learned by *learners* with the mediation of *acceptors*. Our algorithm is the first to tolerate (1) any number of Byzantine failures of proposers and learners, (2) the largest possible number of acceptor failures, and (3) arbitrarily long periods of asynchrony. On the other hand, under best-case conditions, our algorithm allows a value to be learned in only two message-delays in case a first class quorum is accessed, and in three (resp., four) message delays in case a second (resp., third) class quorum is accessed. Note here that (a) learning in a single message delay is obviously impossible with multiple or potentially Byzantine proposers, and (b) the availability of a third class quorum is anyway necessary for resilience. Our algorithm matches the resilience and complexity lower bounds of [35] (including when these bounds are extended to a general adversary structure), together with a new complementary bound we establish here on consensus algorithms that degrade gracefully in best-case

executions. These bounds state minimal conditions under which the state-machine replication approach can be made optimally resilient and best-case efficient. Until now, it was not clear whether the conditions of [35] were also sufficient. We show they are and we complement them.

We believe that it would have been very hard to devise such algorithms, especially in the context of a general adversary structure, without the notion of a refined quorum system, though we might be subjective here.

## 1.4 Roadmap

The rest of the paper is organized as follows. Section 2 first presents our quorum notion and illustrates how it generalizes previous ones through examples from the literature. Sections 3 and 4 introduce our two new distributed object implementations that exploit the full features of refined quorums. Proofs of correctness of our two algorithms are postponed to the appendices for better readability of the paper. We complete the overview of related work in Section 5. Then, we conclude the paper by pointing out some open research directions.

## 2 Refined Quorum Systems

### 2.1 Definitions

A refined quorum system is expressed in the abstract context of a non-empty set  $S$  of elements, and an *adversary structure* (or, simply, *adversary*)  $\mathbf{B}$  defined as follows [26]:

**Definition 1.** *Let  $\mathbf{B}$  be any set of subsets of  $S$ .  $\mathbf{B}$  is an adversary (for  $S$ ) if:  $\forall B \in \mathbf{B}: B' \subseteq B \Rightarrow B' \in \mathbf{B}$ .*

Let  $\mathbf{RQS}$  be any set of subsets of  $S$ .

**Definition 2. Refined Quorum System.** *We say that  $\mathbf{RQS}$  is a refined quorum system for a set  $S$  and adversary  $\mathbf{B}$ , if  $\mathbf{RQS}$  has two subsets  $\mathbf{QC}_1 \subseteq \mathbf{QC}_2 \subseteq \mathbf{RQS}$  such that the following properties hold: (every  $\mathbf{QC}_i$  is called a quorum class, and elements of  $\mathbf{QC}_i$  are called class  $i$  elements)*

Property 1. *The intersection of any two elements of  $\mathbf{RQS}$  does not belong to  $\mathbf{B}$ , i.e.,*

$$- \forall Q, Q' \in \mathbf{RQS}: Q \cap Q' \notin \mathbf{B}.$$

Property 2. *The intersection of any two class 1 elements and any element of  $\mathbf{RQS}$  is not a subset of the union of any two elements of  $\mathbf{B}$ , i.e.,*

$$- \forall Q_1, Q'_1 \in \mathbf{QC}_1, \forall Q \in \mathbf{RQS}, \forall B_1, B_2 \in \mathbf{B}: Q_1 \cap Q'_1 \cap Q \not\subseteq B_1 \cup B_2.$$

Property 3. *The set difference between (i) an intersection of any class 2 element  $Q_2$  and any element  $Q$  of  $\mathbf{RQS}$ , and (ii) any element  $B$  of  $\mathbf{B}$ :*

(a) *does not belong to  $\mathbf{B}$  (we say  $P_{3a}(Q_2, Q, B)$  holds), or*

(b) *has a non-empty intersection with every class 1 element<sup>1</sup> (we say  $P_{3b}(Q_2, Q, B)$  holds), i.e.,*

<sup>1</sup> Assuming there is at least one class 1 element, i.e.,  $\mathbf{QC}_1 \neq \emptyset$ .



Quorum  $Q_1$  contains 5 elements and is a class 1 quorum, while quorum  $Q'$  contains 6 elements yet is only a class 3 quorum.

In the following, we give more illustrations of our quorum notions by explaining how it extends traditional ones. Later in the paper, we will introduce new optimal algorithms that make full use of our quorum notion. In the following, we consider that an adversary  $\mathbf{B}$  for a set of processes  $S$  contains all subsets of  $S$  that can simultaneously be Byzantine. In our description, a process that simply fails by crashing is not called Byzantine. We also denote by  $\mathbf{Q}_i$  the set of subsets of  $S$  that contains all subsets of  $S$  that contain all but at most  $i$  elements of  $S$ , i.e.,  $\mathbf{Q}_i = \{P | P \subseteq S \wedge |P| \geq |S| - i\}$ .

*Example 2.* Consider the case where: (a)  $\mathbf{B} = \{\emptyset\}$ , (b)  $\mathbf{QC}_1 = \mathbf{QC}_2 = \emptyset$  and (c)  $\mathbf{RQS} = \mathbf{Q}_{\lfloor(|S|-1)/2\rfloor}$ . In other words, every majority subset of  $S$  is a quorum. Property 1 is trivially satisfied. So are Properties 2 and 3, since  $\mathbf{QC}_1 = \mathbf{QC}_2 = \emptyset$ . This quorum system is typically used when devising algorithms that tolerate (a minority of) crash-failures, e.g., [4, 8, 16, 34, 44, 50].

*Example 3.* Consider the case of an adversary  $\mathbf{B}_{\lfloor(|S|-1)/3\rfloor}$ , where (a)  $\mathbf{QC}_1 = \mathbf{QC}_2 = \emptyset$  and (b)  $\mathbf{RQS} = \mathbf{Q}_{\lfloor(|S|-1)/3\rfloor}$ . In this case, each quorum contains more than two thirds of processes and Property 1 is satisfied. Properties 2 and 3 are also satisfied (since  $\mathbf{QC}_1 = \mathbf{QC}_2 = \emptyset$ ). Such a quorum system is typically used to tolerate (up to one third of) Byzantine failures, e.g., [7, 10, 42, 45].

*Example 4.* A refined quorum system for which  $\mathbf{QC}_1 = \mathbf{QC}_2 = \emptyset$  is a *disseminating quorum system* in the sense of [40]. In [40], disseminating quorum systems are used to build resilient distributed services that store authenticated (also called self-verifying) data. On the other hand, a refined quorum system in which  $\mathbf{QC}_1 = \emptyset$  and  $\mathbf{QC}_2 = \mathbf{RQS}$  is a *masking quorum system* in the sense of [40]. These systems have been used to build resilient distributed services that store unauthenticated data.

So far, in examples 2-4, we considered refined quorum systems in which  $\mathbf{QC}_1 = \emptyset$ . In the rest of the paper, we study the more general case where  $\mathbf{QC}_1 \neq \emptyset$ . This is the case where our refined quorum systems capture both the resilience and the best-case complexity dimensions of distributed algorithms.

*Example 5.* Consider the case of a refined quorum system where  $\emptyset \neq \mathbf{QC}_1 = \mathbf{QC}_2$ . Such a RQS corresponds to the quorum system used in [36] for the specific case  $\mathbf{B} = \{\emptyset\}$ , to devise a consensus algorithm that tolerates asynchronous periods and a threshold  $t$  of process (crash) failures, yet expedites decisions in best-case scenarios. In fact, although not used in the algorithm, the idea of a *fast* quorum (class 1 quorum in our terminology) was used to explain the algorithm. In the special case of an adversary  $\mathbf{B}_k$ , where (a)  $\mathbf{RQS} = \mathbf{Q}_t$ , and (b)  $\mathbf{QC}_1 = \mathbf{QC}_2 = \mathbf{Q}_q$  ( $q \leq t$ ), Property 2 is satisfied if  $|S| > 2q + t + 2k$  and Property 1 is satisfied if  $|S| > 2t + k$ . These inequalities correspond to Lamport's lower bounds for "asynchronous" consensus [35]. The special case of this RQS where  $k = q = t$  (i.e., where  $\mathbf{QC}_1 = \mathbf{RQS}$ ) corresponds to the quorum system used in [1, 41], also with the goal of boosting consensus latency.

*Example 6.* Even more interesting is the case where  $\emptyset \neq \mathbf{QC}_1 \neq \mathbf{QC}_2 \subseteq \mathbf{RQS}$  (e.g., Fig. 3), especially when  $\mathbf{RQS}$ ,  $\mathbf{QC}_1$  and the adversary are defined as in Example 5,  $\mathbf{QC}_2 = \mathbf{Q}_r$ , and  $0 \leq q < r \leq t$ . In other words, each quorum contains all but at most  $t$  processes, while class 1 (resp., class 2) quorums contain all but at most  $q$  (resp.,  $r$ ) processes.  $\mathbf{RQS}$  satisfies (i) Property 1

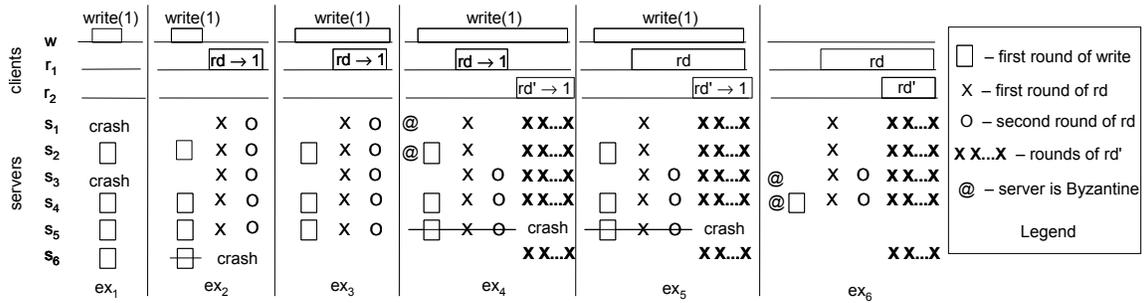
if  $|S| > 2t + k$ , (ii) Property 2 if  $|S| > t + 2k + 2q$ , and (iii) Property 3 if  $|S| > t + r + k + \min(k, q)$ , i.e., **RQS** is a refined quorum system if  $|S| > t + k + \max(t, k + 2q, r + \min(k, q))$ . This RQS corresponds to the quorum system used in [12, 20], and later in [53]. An important instantiation of this refined quorum system is the one with  $|S| = 3t + 1$  processes out of which  $t$  may be Byzantine ( $k = t$ ), and where all quorums are class 2 quorums ( $r = t$ ), whereas the set that contains all servers is a class 1 quorum ( $q = 0$ ).

*Example 7 (Intuition behind Property 3 of RQS).* As we just discussed, our RQS notion was implicitly used, in partial forms, in various distributed objects implementations. However, all examples we described assumed threshold adversary. This does not explain all the subtleness of RQS, notably of its Property 3, that becomes evident only when the general adversary structure is assumed.

Although the intuition behind Property 3 of RQS is not obvious, it should not be surprising that Property 3 is important to allow implementations that achieve both the best possible latency (e.g., 1 round in storage) and the next best possible latency (2 rounds in case of storage). Indeed, consider the following example in which there are 6 servers  $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$ , with adversary structure given by

$$B = \{\emptyset, \{s_1\}, \{s_2\}, \{s_3\}, \{s_4\}, \{s_1, s_2\}, \{s_3, s_4\}, \{s_2, s_4\}\}$$

with 3 quorums, **RQS** =  $\{Q_1, Q_2, Q'_2\}$ , where  $Q_1 = \{s_2, s_4, s_5, s_6\}$ ,  $Q_2 = \{s_1, s_2, s_3, s_4, s_5\}$  and  $Q'_2 = \{s_1, s_2, s_3, s_4, s_6\}$ . It is not very difficult to verify that  $Q_1$  is a class 1 quorum, where  $Q_2$  and  $Q'_2$  are class 2 quorums. Figure 4 depicts several executions of a possible best-case latency efficient atomic storage algorithm built over this RQS.



**Fig. 4.** Intuition behind Property 3 of RQS; executions of a possible atomic storage implementation.

In execution  $ex_1$  synchronous  $write(1)$  (we denote by  $wr$ ) must complete in a single round since it accesses a class 1 quorum  $Q_1$ . In  $ex_2$ ,  $wr$  completes as in  $ex_1$  (although  $s_1$  and  $s_3$  are correct). Therefore, in  $ex_2$ , synchronous and uncontended read  $rd$  must return value 1 after 2 rounds of communication with servers from  $Q_2$ . Moreover,  $r_1$  cannot distinguish  $ex_2$  from  $ex_3$  in which  $wr$  is slow, concurrent with  $rd$  and does not reach  $s_6$ . If we extend  $ex_3$  such that  $s_5$  crashes and servers from  $B_{12} = \{s_1, s_2\}$  are Byzantine and “forget” about round 2 of  $rd$ , we obtain  $ex_4$ . In  $ex_4$ ,  $rd'$  must return 1, but, at first, it is not clear that  $rd'$  should return the value in  $ex_4$  after communicating

only with servers from  $Q'_2$ . However,  $rd'$  in  $ex_4$  is indistinguishable from  $rd'$  in  $ex_5$  in which  $rd$  is simply slow and in which  $rd'$  must return a value (after some number of rounds) since all servers from  $Q'_2$  are correct (hence in both  $ex_4$  and  $ex_5$ ,  $rd'$  must return 1).

However, notice that, in  $ex_5$ , reader  $r_2$  has (just) enough information to return 1, only because Property 3 of RQS holds. For example since  $B_{34} = Q_2 \cap Q'_2 \setminus B_{12} = \{s_3, s_4\} \in \mathbf{B}$ ,  $P_{3a}(Q_2, Q'_2, B_{12})$  does not hold and consequently neither does  $P_{3a}(Q_2, Q'_2, B_{34})$ . Hence  $P_{3b}(Q_2, Q'_2, B_{34})$  must hold, i.e.,  $Q_1 \cap Q_2 \cap Q'_2 \setminus B_{34}$  is not empty. In our case server  $s_2$  in non-empty  $Q_1 \cap Q_2 \cap Q'_2 \setminus B_{34}$ , that was accessed in the first round of  $wr$ , is crucial for the ability of reader  $r_2$  to return 1 in  $rd'$ . Namely, if this intersection was empty (and hence if Property 3 was violated), the reader would not be able to deterministically distinguish  $rd'$  in  $ex_5$  from  $rd'$  in  $ex_6$  in which there is no  $wr$  and in which it must not return 1. Finally, notice that the server  $s_2$  in intersection of  $Q_2 \cap Q'_2 \setminus B_{34}$  and  $Q_1$  would not be important if  $P_{3a}(Q_2, Q'_2, B_{12})$  held, i.e., if  $B_{34}$  was not in  $\mathbf{B}$ . Then, in any execution, there would be at least one benign server in  $B_{34}$  and the reader would not have to worry about intersections with class 1 quorums.

In the following, we present two new algorithms that make full and explicit use of our notion of RQS.

### 3 Atomic storage

We show in this section how to use a refined quorum system to wait-free [24] implement the abstraction of a single-writer multi-reader (SWMR) atomic [33] storage with optimal resilience and complexity. Optimal resilience means here tolerating the maximal number of server failures while still ensuring wait-freedom in the face of contention and asynchrony (worst-case conditions). On the other hand, optimal complexity in our context means minimal operation latency in periods of synchrony and contention-freedom (best-case conditions). Our storage algorithm tolerates Byzantine servers yet does not rely on any data authentication primitive.<sup>2</sup>

In the following, and after few preliminaries on the model underlying our storage algorithm, we overview our algorithm and then state its optimality. This includes establishing a new tight bound on efficient atomic storage implementations. The full correctness proof of our atomic storage implementation is given in Appendix A.

#### 3.1 Model

**Processes and channels.** We model processes as a deterministic I/O automata [39]. Processes are interconnected with point-to-point communication channels. For ease of presentation, we assume a global clock that is not accessible to processes. The state of the communication channel between processes  $p$  and  $q$  is modeled as a set  $mset_{p,q} = mset_{q,p}$  containing messages that are sent but not yet received. We assume that every message  $m$  has two tags which identify the sender and the receiver of the message.

We model a distributed algorithm as a collection of automata  $A_p$ , each assigned to a process  $p$ . A computation of a process  $p$  proceeds in *steps* of  $A_p$ . A *correct* process  $p$  is one that executes an

<sup>2</sup> Although powerful, data authentication primitives do not provide deterministic guarantees, and might require (1) an infrastructure for key management (for solutions based on symmetric cryptography, e.g., [6]), or (2) non-negligible complexity of data encryption (e.g., [47]). These typically introduce overhead that one would like to avoid, especially in the best-case scenarios.

infinite number of steps of  $A_p$ . A process fails by *crashing* if it executes a finite number of steps. We say that a process is *benign* if it is correct or fails by crashing. A step of  $A$  is denoted by a pair of process id and message set  $\langle p, M \rangle$  ( $M$  might be  $\emptyset$ ). In step  $sp = \langle p, M \rangle$ , a benign process  $p$  atomically does the following (we say that  $p$  takes step  $sp$ ): (1) (*receive substep*)  $p$  removes the messages of  $M$  from  $mset_{p,*}$  (we also say:  $p$  receives the messages of  $M$ ), (2) (*computation substep*)  $p$  applies  $M$  and its current state  $st_p$  to  $A_p$ , which outputs a new state  $st'_p$  and a set of messages  $M'$  to be sent, and  $p$  adopts  $st'_p$  as its new state and (3) (*send substep*) puts the output messages ( $M'$ ) in  $mset_{p,*}$  (we also say:  $p$  sends the messages of  $M'$ ). We assume that local computation takes negligible time, i.e., the time necessary for a benign process to take a step is negligible.

A *Byzantine* process  $p_B$  can perform arbitrary *actions*: (1)  $p_B$  can remove/put an arbitrary message  $m$  from/into  $mset_{p_B,*}$  at an arbitrary time  $t$ ,<sup>3</sup> and (2)  $p_B$  can change its state in an arbitrary manner. We say that a process is *faulty* if it is Byzantine or if it fails by crashing.

Given any algorithm  $A$ , an *execution* of  $A$  is an infinite sequence of steps of  $A$  taken by benign processes, and actions of Byzantine processes, such that the following properties hold for every benign process  $p$ : (1) initially, for each benign process  $q$ ,  $mset_{p,q} = \emptyset$ , (2) the current state in the first step of  $p$  is a special state *Init*, and (3) for each step  $\langle p, M \rangle$  of  $A$ , and for every message  $m \in M$ ,  $p$  is the receiver of  $m$  and  $\exists q, mset_{p,q}$  that contains  $m$  immediately before the step  $\langle p, M \rangle$  is taken. A *partial execution* is a finite prefix of some execution. A (partial) execution  $ex$  extends some partial execution  $ex'$  if  $ex'$  is a prefix of  $ex$ . At the end of a partial execution, all messages that are sent but not yet received are said to be *in transit*.

We assume that the system is asynchronous: there is no bound on communication delays. The system *may be* synchronous during certain periods of time. We say that the *system is synchronous* during time interval  $[t, t']$  if there is a constant  $\Delta$  ( $\Delta > 0$ ) is known to all correct processes, such that, for every two correct processes  $p_1$  and  $p_2$ , every message sent by  $p_1$  at time  $t_1$  such that  $[t_1, t_1 + \Delta] \subset [t, t']$ , is received by  $p_2$  at latest in the first receive substep taken by  $p_2$  after  $t_1 + \Delta$ .

Finally, we assume that communication channels are *reliable*, i.e., if there is a step  $sp_p$  in which some correct process  $p$  sends a message  $m$  to another correct process  $q$ , then (eventually)  $q$  takes a step  $sp_q$  in which  $q$  receives  $m$ .

**Distributed storage.** A distributed storage (or, simply, storage) can be viewed as a read/write abstraction implemented by a finite set of processes called *servers*, and a distinct, potentially unbounded, set of processes called *clients*. We assume that the set clients has two distinct subsets, a singleton *writer* and a set *readers*. We assume clients and servers are connected with point-to-point channels (defined as in Section 3.1). No channels are assumed among servers.

An atomic storage provides the illusion of sequential accesses by ensuring the atomicity of read/write operations [25, 33] (when there is no risk of ambiguity we say *operation* when we should be saying *operation execution*). We focus on *wait-free* [24] atomic storage algorithms in which every read/write operation invoked by a correct client eventually completes.

Clients access the storage through two operations: (1) *write*( $v$ ) (invoked by the writer), to write a value  $v$  in the storage, and (2) *read*() (invoked by readers), to read the value from the storage. We do not explicitly model the invocation and response steps of the storage operations. We simply say that an operation  $op$  invoked by client  $c$  is *complete* if  $c$  takes a response step for  $op$ . We assume that all benign servers are initialized with an initial value of the storage  $\perp$ , that is not in domain

<sup>3</sup> Informally, we assume that no (benign) process uses data authentication.

$\mathbb{D}$  of valid inputs of a write operation (we denote the set  $\mathbb{D} \cup \{\perp\}$  by  $\mathbb{D}_\perp$ ). No client  $c$  invokes a new operation before all operations previously invoked by  $c$  have completed.

Denote the time when operation  $op$  is invoked by  $t_{op_{inv}}$ . Denote the time when  $op$  completes by  $t_{op_{resp}} > t_{op_{inv}}$ . We say that a complete operation  $op'$  *precedes* operation  $op$  if  $t_{op'_{resp}} < t_{op_{inv}}$  (we also say  $op$  *follows*  $op'$ ). For two operations  $op$  and  $op'$ , if neither  $op$  precedes  $op'$ , nor  $op'$  precedes  $op$ , we say  $op$  and  $op'$  are *concurrent*. We say that an operation  $op$  is *uncontended* if  $op$  is not concurrent with any write operation. Moreover, we say that  $op$  is *synchronous* if the system is synchronous during the period between the invocation and completion of  $op$ .

Denoting the set of servers by  $S$ , and the adversary by  $\mathbf{B}$ , we construct a refined quorum system  $\mathbf{RQS}$  (obeying the properties defined in Section 2) known to all clients. In the above, we assume that, for any execution  $ex$ ,  $B_{ex} \in \mathbf{B}$ , where  $B_{ex}$  contains all servers Byzantine in  $ex$ . Moreover, any number of clients and servers may fail by crashing, as long as there is at least one quorum in  $\mathbf{RQS}$  that contains only correct servers.

**Round-based storage algorithms.** Our algorithm is *round-based*, i.e., each operation  $op$  (read or write) proceeds in series of *communication round-trips* (or, simply, *rounds*). In each round of  $op$ ,

1. Client  $c$  sends messages to a subset of servers (possibly all servers).
2. Servers on receiving such a message reply to client  $c$  before receiving any other messages. More precisely, any server  $s_i$  on receiving a message  $m$  in step  $sp1 = \langle s_i, M \rangle$  ( $m \in M$ ), where  $m$  is sent by the client  $c$ , replies to  $m$  either in step  $sp1$  itself, or in a subsequent step  $sp2$ , such that  $s_i$  does not receive any message in any step between  $sp1$  and  $sp2$  (including  $sp2$ ). Intuitively, this requirement forbids the server to wait for some other message before replying to  $m$ .
3. upon receiving a sufficient number  $k \geq 0$  of such replies, a client completes a round. The decision on whether and when a client should proceed to the next round is algorithm-specific.

In short, in round-based algorithms, servers can send messages to clients only in response to some particular messages received from a client. Since we assume that servers (resp., clients) are not interconnected with communication channels, no other messages are exchanged in a round based algorithm.

We express the latency (complexity) of an operation  $op$  of a round-based algorithm in terms of the number of rounds between the invocation and the completion of  $op$ . Let  $\mathbf{P}$  be any set of subsets of  $S$ .

**Definition 3.**  *$(m, \mathbf{P})$ -fast storage algorithm.* Consider any synchronous and uncontended operation  $op$  invoked by a correct client. We say that a storage algorithm  $A$  is  $(m, \mathbf{P})$ -fast if in every execution of  $A$  in which some set  $P \in \mathbf{P}$  contains only correct servers,  $op$  completes in at most  $m$  rounds, without using data authentication.

### 3.2 Atomic storage algorithm

Our storage algorithm is  $(m, \mathbf{QC}_m)$ -fast for all  $m \in \{1, 2, 3\}$ . Note that this implies that, in our algorithm, all synchronous and uncontended operations complete in at most 3 rounds.

The pseudocode of the algorithm is given in Figures 5, 6 and 7. The write code (Figure 5) is simple, thanks to the underlying RQS. The writer maintains a monotonically increasing local timestamp  $ts$  that is assigned to the written value  $v$  and sent to servers in each round (for simplicity, we sometimes say that the writer writes a pair  $\langle ts, v \rangle$ ).

---

```

Initialization:
0:  $ts := 0$ ;  $timeout := 2\Delta$ ;  $QC'_2 := \emptyset$ 

write( $v$ ) is {
1: inc( $ts$ )
2: round(1)
3: if acks received from some class 1 quorum then return(OK)
4: forall  $Q_2 \in QC_2$ 
5:   if acks received from  $Q_2$  then  $QC'_2 := QC'_2 \cup \{Q_2\}$ 
6: round(2)
7: if acks received from some quorum from  $QC'_2$  then return(OK)
8:  $QC'_2 := \emptyset$ ; round(3)
9: return(OK) }

round( $i$ ) is {
10: send  $wr(ts, v, QC'_2, i)$  to all servers
11: if  $i < 3$  then trigger( $timeout$ )
12: wait for (reception of  $wr\_ack(ts, i)$  from some quorum) and (expiration of  $timeout$ ) }

```

---

**Fig. 5.** Atomic storage algorithm: writer code

A write consists of at most three rounds. In every round  $rnd$ , the writer sends a  $wr\langle ts, v, QC'_2, rnd \rangle$  message containing value  $v$  to be written, along with a timestamp  $ts$  and a set of quorums (i.e., quorum ids)  $QC'_2$  to all servers (this set is empty in rounds 1 and 3 and only used in round 2, as explained below). In every round, the writer awaits acks from some quorum and, in the first two rounds, the expiration of the timer set to  $2\Delta$ .

If the writer receives acks from some class 1 quorum by the expiration of the timer, the write terminates. Otherwise, the writer proceeds to round 2. If, in round 1, the writer received acks from some class 2 quorums, the ids of these quorums are added to  $QC'_2$  (lines 4-5, Fig. 5). If the writer receives again acks from some quorum from  $QC'_2$  in round 2 (line 7, Fig. 5), the write terminates at the end of round 2. Finally, if this is not the case, the writer proceeds to round 3 and completes at the end of this round, upon reception of round 3 acks from any quorum.

The pseudocode of servers is given in Figure 6. Upon receipt of the message  $wr\langle ts, v, QC'_2, rnd \rangle$ , a server  $s_i$  stores the received data in its  $history_i$  matrix, by storing  $history_i[ts, j].pair = \langle ts, v \rangle$  for all  $j, 1 \leq j \leq rnd$  and by adding  $QC'_2$  to  $history_i[ts, rnd].sets$ .<sup>4</sup> Then, a server sends an ack to the client.

---

```

Initialization:
1:  $history_i[*, *].pair := \langle 0, \perp \rangle$ ;  $history_i[*, *].sets := \emptyset$ 

2: upon reception of a  $wr\langle ts, v, QC'_2, rnd \rangle$  message from some client  $c$  do
3:   for  $m = 1$  to  $rnd$  do
4:     if  $history_i[ts, m] = \langle \langle 0, \perp \rangle, \emptyset \rangle$  or  $history_i[ts, m] = \langle \langle ts, v \rangle, * \rangle$  then
5:        $history_i[ts, m].pair := \langle ts, v \rangle$ 
6:       if  $m = rnd$  then  $history_i[ts, m].sets := history_i[ts, m].sets \cup QC'_2$ 
7:     send  $wr\_ack(ts, rnd)$  to  $c$ 

8: upon reception of a  $rd\langle tsr, rnd \rangle$  message from the reader  $r_j$ 
9:   send  $rd\_ack\langle tsr, rnd, history_i \rangle$  to the reader  $r_j$ 

```

---

**Fig. 6.** Atomic storage algorithm: server  $s_i$  code

<sup>4</sup> Note here that, in order to simplify our algorithm, we assume that servers store the entire history of the shared variable they are implementing; we discuss this further in Section 5.

The read pseudocode, given in Figure 7, is more involved (in the following, unless stated otherwise line numbers refer to Fig. 7). As many other atomic storage algorithms (e.g., [4]) our algorithm consists of two parts: (1) the part that implements *regular* [33] storage (lines 20-35, along with the predicates defined in lines 3-9) and (2) the part that prevents read inversion and enforces atomicity, in which the readers write value back to a “sufficient” number of servers (lines 40-49, with predicates in lines 1-2).

In the first part of the read algorithm, the reader selects the timestamp/value pair (in line 35), that contains the value that the reader is going to return after a possible writeback. The first part of the algorithm consists in one or more rounds in which the reader sends  $\text{rd}\langle \text{read\_no}, \text{read\_rnd} \rangle$  (line 25), containing the unique id of a read  $\text{read\_no}$  (to distinguish messages sent by the same reader in different operations) and the round number  $\text{read\_rnd}$ . A server replies to a  $\text{rd}$  message by sending the entire history of the shared variable in a  $\text{rd\_ack}$  message in response (lines 8-9, Fig. 6). A round ends when the reader receives responses from all servers from some quorum  $Q$  (line 26). Specifically, in round 1, the reader: (a) also waits for the timer set to  $2\Delta$  to expire (lines 24 and 28), and (b) stores the ids of all class 2 quorums that responded to it in the set  $\mathbf{QC}'_2$  (lines 30-32). In general, we say quorum  $Q$  responds in a read if a reader receives at least one  $\text{rd\_ack}$  from every server in  $Q$  (lines 52-53). Remembering class 2 quorums that responded in round 1 will later reveal crucial for allowing 2-round best-case reads and single round best-case writes in the same implementation.

In the heart of the first part of the algorithm are predicates  $\text{valid}_j$  (for  $j \in \{1, 2, 3\}$ ), defined in lines 3-5. These predicates ensure that if some complete write (resp., read) operation  $op$  wrote (resp., selected) a pair  $c = \langle c.ts, c.val \rangle$ , in any read  $rd$  that follows  $op$ , for every quorum  $Q$  that responded in  $rd$  there is some  $j$  such that  $\text{valid}_j(c, Q)$  holds. Therefore, predicate  $\text{invalid}(c)$  (line 6) cannot hold in  $rd$  and, the reader cannot select a pair  $c_{sel}$  such that  $c_{sel}.ts < c.ts$  in line 35 of  $rd$ ; in other words,  $rd$  cannot return an older value than the one written/returned by  $op$ . The pair  $c_{sel}$  is selected in line 35, as the pair with the highest timestamp among all pairs  $c$  for which predicates  $\text{highCand}(c)$  (line 9) and  $\text{safe}(c)$  holds (line 8). Predicate  $\text{highCand}(c)$  implies that all pairs with a higher timestamp are invalid, i.e., that there are no possibly newer values that ought to be considered. On the other hand, predicate  $\text{safe}(c)$  guarantees that a selected value is not fabricated by Byzantine processes; roughly speaking, all servers from some set  $T \notin \mathbf{B}$  need to confirm  $c$  before a reader can select it. This prevents fabrication since, in every execution,  $T$  contains at least one benign server.

On the other hand, the second part of the algorithm that ensures atomicity, is orchestrated around the outcome of a *Best-Case Detector* BCD, defined by predicates in lines 1-2, and accessed RQS quorums. Roughly, BCD detects if a read operation is synchronous and uncontended. In the following, we explain the intuition behind the techniques used in our algorithm on the example of a one such uncontended and synchronous read  $rd$ .

Let  $wr$  be the last write that precedes  $rd$  and assume that  $wr$  wrote pair  $c = \langle ts, v \rangle$  in  $R$  rounds,  $R \in \{1, 2, 3\}$ . Note that, by atomicity,  $rd$  must return  $v$ .

First, it is crucial to see that, in a synchronous and uncontended read like  $rd$ , the first part of the algorithm takes *only* a single round. Since  $rd$  is uncontended, during  $rd$  benign servers store values with timestamps only as high as  $ts$ . As we already intuited above, in  $rd$ , for every quorum  $Q$  that responds in round 1, there is some  $j$  such that  $\text{valid}_j(c, Q)$  holds. To see this, notice that  $wr$  completed either in: (a) single round ( $R = 1$ ) by accessing class 1 quorum  $Q_1$ , or (b) in more than one round ( $R \in \{2, 3\}$ ). Then, it is not difficult to see that for every quorum  $Q$ , in case: (a)  $\text{valid}_1(c, Q)$  holds (by Property 2 of RQS), whereas in (b)  $\text{valid}_2(c, Q)$  holds (by Property 1

---

Definitions:

- 1:  $BCD(c, 1, R) ::= \exists Q_1 \in \mathbf{QC}_1, \exists Q_R \in \mathbf{QC}_R, \exists \mathbf{Set} \subseteq \mathbf{QC}_2 \cup \{\emptyset\}:$   
 $(Q_1 \cap Q_R \subseteq \{s_i \in S \mid \mathit{history}[i, c.ts, R] = \langle c, \mathbf{Set} \rangle\}) \wedge ((R \neq 2) \vee (Q_R \in \mathbf{Set}))$
- 2:  $BCD(c, 2, R) ::= \{Q_2 \in \mathbf{QC}'_2 \mid \exists Q_R \in \mathbf{QC}_R: Q_R \cap Q_2 \subseteq \{s_i \in S \mid \mathit{history}[i, c.ts, R].\mathit{pair} = c\}\}$
- 3:  $\mathit{valid}_1(c, Q) ::= \exists T \subseteq Q, \forall s_i \in T : (T \not\subseteq \mathbf{B}) \wedge (\mathit{history}[i, c.ts, 1].\mathit{pair} = c)$
- 4:  $\mathit{valid}_2(c, Q) ::= \exists s_i \in Q : \mathit{history}[i, c.ts, 2].\mathit{pair} = c$
- 5:  $\mathit{valid}_3(c, Q) ::= \exists Q_2 \in \mathbf{QC}_2, \exists B \in \mathbf{B}, \forall s_i \in Q_2 \cap Q \setminus B, \exists \mathbf{Set}_i \subseteq \mathbf{QC}_2:$   
 $(P_{3b}(Q_2, Q, B)) \wedge (\mathit{history}[i, c.ts, 1] = \langle c, \mathbf{Set}_i \rangle) \wedge (Q_2 \in \mathbf{Set}_i)$
- 6:  $\mathit{invalid}(c) ::= \exists Q \in \mathbf{Responded}: \neg(\mathit{valid}_1(c, Q) \vee \mathit{valid}_2(c, Q) \vee \mathit{valid}_3(c, Q)) \vee (c.ts > \mathit{highest\_ts})$
- 7:  $\mathit{read}(c, i) ::= \exists \mathit{rnd} \in \{1, 2\} : \mathit{history}[i, c.ts, \mathit{rnd}].\mathit{pair} = c$
- 8:  $\mathit{safe}(c) ::= \{s_i \in S \mid \mathit{read}(c, i)\} \notin \mathbf{B}$
- 9:  $\mathit{highCand}(c) ::= \forall c' \in \mathbb{N}_0 \times \mathbb{D}_\perp, \forall s_i \in S : \mathit{read}(c', i) \wedge (c'.ts > c.ts) \Rightarrow \mathit{invalid}(c')$

Initialization:

- 10:  $\mathit{timeout} := 2\Delta; \mathit{history}[*], [*], [*] := \langle \langle 0, \perp \rangle, \emptyset \rangle; \mathit{highest\_ts} := 0; \mathit{read\_no} := 0$

$\mathit{read}()$  is {

- 20:  $\mathit{read\_rnd} := 0; \mathbf{QC}'_2 := \emptyset; \mathbf{Responded} := \emptyset$
- 21: **inc**( $\mathit{read\_no}$ )
- 22: **repeat**
- 23:   **inc**( $\mathit{read\_rnd}$ )
- 24:   **if**  $\mathit{read\_rnd} = 1$  **then** **trigger**( $\mathit{timeout}$ )
- 25:   send  $\mathit{rd}\langle \mathit{read\_no}, \mathit{read\_rnd} \rangle$  to all servers
- 26:   **wait for** receive  $\mathit{rd\_ack}\langle \mathit{read\_no}, \mathit{read\_rnd}, * \rangle$  from some quorum
- 27:   **if**  $\mathit{read\_rnd} = 1$  **then**
- 28:     **wait for** expiration of  $\mathit{timeout}$
- 29:      $\mathit{highest\_ts} :=$  highest timestamp  $\mathit{hts} \in \mathbb{N}_0$  such that  $\exists c \in \mathbb{N}_0 \times \mathbb{D}_\perp, \exists s_i \in S : \mathit{read}(c, i) \wedge c.ts = \mathit{hts}$
- 30:     **forall**  $Q_2 \in \mathbf{QC}_2$
- 31:       **if** acks received from  $Q_2$  **then**  $\mathbf{QC}'_2 := \mathbf{QC}'_2 \cup \{Q_2\}$
- 32:     **endif**
- 33:      $C := \{c \in \mathbb{N}_0 \times \mathbb{D}_\perp \mid (\mathit{safe}(c) \wedge \mathit{highCand}(c))\}$
- 34:     **until**  $C \neq \emptyset$
- 35:      $c_{sel} := c \in C : (\forall c' \in C : c.ts \geq c'.ts)$
- 40:   **if**  $(\exists i \in \{1, 2, 3\} : BCD(c_{sel}, 1, i))$  **and**  $(\mathit{read\_rnd} = 1)$  **then** **return**( $c_{sel}.val$ )
- 41:   **if**  $(\exists i \in \{1, 2, 3\} : BCD(c_{sel}, 2, i) \neq \emptyset)$  **and**  $(\mathit{read\_rnd} = 1)$  **then**
- 42:     **if**  $(\exists i \in \{2, 3\} : BCD(c_{sel}, 2, i) \neq \emptyset)$  **then** **writeback**( $2, c_{sel}, \emptyset$ )
- 43:     **trigger**( $\mathit{timeout}$ )
- 44:     **writeback**( $1, c_{sel}, BCD(c_{sel}, 2, 1)$ )
- 45:     **wait for** expiration of  $\mathit{timeout}$
- 46:     **if** acks received from some quorum from  $BCD(c_{sel}, 2, 1)$  **then** **return**( $c_{sel}.val$ )
- 47:     **writeback**( $2, c_{sel}, \emptyset$ )
- 48:   **endif**
- 49:   **writeback**( $1, c_{sel}, \emptyset$ ); **writeback**( $2, c_{sel}, \emptyset$ )
- 50: **upon** reception of  $\mathit{rd\_ack}\langle \mathit{read\_no}, \mathit{read\_rnd}, \mathit{history}_i \rangle$  from server  $s_i$  **do**
- 51:    $\mathit{history}[i] := \mathit{history}_i$
- 52: **upon** received at least one  $\mathit{rd\_ack}$  message from every server  $s_i$  in some quorum  $Q \in \mathbf{RQS}$  **do**
- 53:    $\mathbf{Responded} := \mathbf{Responded} \cup Q$

$\mathit{writeback}(\mathit{round}, c, \mathbf{Set})$  is {

- 60: send  $\mathit{wr}\langle c.ts, c.val, \mathbf{Set}, \mathit{round} \rangle$  message to all servers
  - 61: **wait for** reception of  $\mathit{wr\_ack}\langle c.ts, \mathit{round} \rangle$  message from some quorum
  - 62: **if**  $\mathit{round} = 2$  **then** **return**( $c_{sel}.val$ ) }
- 

**Fig. 7.** Atomic storage algorithm: reader code

of RQS). Hence,  $invalid(c)$  cannot hold at the end of round 1. Moreover, since  $rd$  is synchronous, it gets a response from at least one quorum  $Q_c$  containing only correct servers; hence,  $safe(c)$  also holds. Considering possible pairs  $c'$  with higher timestamp than  $ts$  (which could have been reported by Byzantine servers only), it is not difficult to see that, for such  $c'$ , none of the predicates  $valid_j(c', Q_c)$  for  $j \in \{1, 2, 3\}$  can hold. Hence, in the case of synchronous and uncontended read  $rd$  the reader selects a pair  $c_{sel} = c$  written by the last preceding write in line 35, and the first part of the algorithm (lines 20-35) takes only a single round.

Then, the reader proceeds to the second part of the **read** algorithm (that guarantees atomicity, lines 40-49); basically, this is a sophisticated *writeback* procedure, based on the outcome of a BCD. The reader queries BCD with  $c_{sel} = \langle ts, v \rangle$  as a parameter (line 40) and the outcome governs the remainder of the writeback procedure. Namely:

1. If the reader received acks from a class 1 quorum (containing only correct servers) in round 1,  $BCD(c_{sel}, 1, R)$  holds (line 1) and  $rd$  completes at the end of round 1, *without* any writeback whatsoever (line 40). Recall here that  $R$  denotes the number of rounds in which  $wr$  completed and hence suggests the class of the quorum that was available to the writer. Notice that, by line 1,  $BCD(c_{sel}, 1, R)$  holds only if there is a class 1 quorum  $Q_1$  and a class  $R$  quorum  $Q_R$  such that *all* servers from  $Q_1 \cap Q_R$  had received a round  $R$   $wr$  message containing  $ts$  and  $v$  (either from the writer or some reader writing-back the pair  $c_{sel}$ ) and responded to the reader. Since read  $rd$  is synchronous and uncontended,  $BCD(c_{sel}, 1, R)$  is guaranteed to hold in case  $rd$  accesses a class 1 quorum of correct processes in the first round.
2. Else, if the reader received acks from some class 2 quorum(s)  $Q_2$  (containing only correct servers) in round 1, then set  $\mathbf{X} = BCD(c_{sel}, 2, R)$  (line 2) is non-empty set of quorums (since  $Q_2 \in \mathbf{X}$ ). Indeed, notice that  $\mathbf{X} = BCD(c_{sel}, 2, R)$  contains a set of all class 2 quorums  $Q_2$  such that: (a) the reader received replies from  $Q_2$  in round 1 of  $rd$  (i.e.,  $Q_2 \in QC'_2$ ), and (b) there is a class  $R$  quorum  $Q_R$  such that *all* servers from  $Q_2 \cap Q_R$  received the round  $R$   $wr$  message containing  $ts$  and  $v$ . Since read  $rd$  is both synchronous and uncontended,  $\mathbf{X}$  is guaranteed to contain all class 2 quorums (containing only correct processes) that replied to the reader in round 1.

Since  $\mathbf{X}$  is non-empty,  $rd$  proceeds to round 2 (or, in other words, the first round of the writeback procedure, line 41). If  $R \in \{2, 3\}$ , then the reader sends  $wr\langle ts, v, \emptyset, rnd \rangle$ , with  $rnd = 2$  to all servers, waits for acks from some quorum and returns (lines 42 and 60-62). In this case, reader writesback with  $rnd = 2$  since it knows that the writer already completed wrote the value to some quorum (writing the value to servers using  $wr\langle ts, v, *, rnd \rangle$  message with  $rnd = 2$  conveys that the client knows that all servers from some quorum have already stored pair  $\langle ts, v \rangle$ ).

Else, if  $R = 1$ , the first round of the writeback procedure (round 2 of  $rd$ ) is more sophisticated, since the reader cannot be sure that all servers from some quorum already stored  $\langle ts, v \rangle$  (recall here the executions depicted in Fig. 4, in Example 7, of Section 2.2). Namely, in this case, the reader: (a) triggers a timer (line 43), (b) sends  $wr\langle ts, v, \mathbf{X}, 1 \rangle$  to all servers (line 44), and (c) waits for acks until some quorum responds and the timer expires (lines 45 and 60-62). The uncontended and synchronous **read** completes at the end of round 2 (the first round of the writeback) only if the reader receives acks from some quorum from  $\mathbf{X} = BCD(c_{sel}, 2, 1)$  (line 46).

Writing class 2 quorum ids contained in the  $\mathbf{X}$ , is crucial for allowing 2 round best-case reads to be combined with single round best-case writes. For example, if  $ex_5$  of Figure 4 is applied

to our algorithm, the reader in  $r_1$  would be writing back the value in the second round of  $rd$  precisely as described above.

3. Otherwise, if no quorum from  $X$  replies, the second round of the writeback procedure (i.e., the third round of  $rd$ ) is invoked (line 47). Note that the `read` takes at most 2 rounds in the second part of the algorithm (i.e., in lines 40-49). Hence, when the `read` is synchronous and uncontended, it completes in at most 3 rounds.

The correctness proof of our atomic storage algorithm is given in Appendix A. The critical part of the proof consists of several short lemmas. The main theorems (that make use of the above mentioned lemmas), despite being long, are easy to follow, since these are mainly case-by-case analysis, where the intersection properties of RQS allow the critical lemmas to be easily applied.

### 3.3 Optimality

Consider the space of round-based storage algorithms. Let  $Q, Q^{(i)}$  (for  $i \in \{1, 2, 3\}$ ) be any sets of subsets of (the set of servers)  $S$ . We say that an algorithm  $A$  is  $(Q, B)$ -atomic, if  $A$  wait-free implements an atomic SWMR storage despite the adversary  $B$  provided that in every execution of  $A$ , there is a set  $Q \in Q$  that contains only correct servers. Moreover, denote by  $P3(Q^{(1)}, Q^{(2)}, Q^{(3)})$  (resp.,  $P1(Q^{(3)})$ ;  $P2(Q^{(1)}, Q^{(3)})$ ) the property obtained from Property 3 (resp., Property 1; Property 2) of Definition 2 by replacing  $QC_i$  with  $Q^{(i)}$ , for  $i = 1 \dots 3$ . The minimality of our RQS is captured via the following three theorems.

**Theorem 1.** *If an algorithm  $A$  is  $(Q^{(3)}, B)$ -atomic, then  $P1(Q^{(3)})$  holds.*

**Theorem 2.** *If a  $(Q^{(3)}, B)$ -atomic algorithm  $A$  is  $(1, Q^{(1)})$ -fast, then  $P2(Q^{(1)}, Q^{(3)})$  holds.*

**Theorem 3.** *If a  $(Q^{(3)}, B)$ -atomic algorithm  $A$  is both  $(1, Q^{(1)})$ -fast (for some  $Q^{(1)} \neq \emptyset$ ) and  $(2, Q^{(2)})$ -fast, then  $P3(Q^{(1)}, Q^{(2)}, Q^{(3)})$  holds.*

As a corollary of Theorems 1–3, our atomic storage implementation of Figure 7 is optimally resilient and has optimal (best-case) complexity.

Theorem 1 has been established for the special case of threshold-based quorums and with an implicit notion of quorums in [42]. Moreover, a restricted form of Theorem 2 was proved in [20], which considered atomic storage implementations in which synchronous and uncontended read/write operations can complete in a single round, in the context of optimally resilient atomic storage implementations in the threshold-based hybrid failure model [49]. It is not very difficult to extend these bounds to the general adversary structure and the RQS setting. Theorem 3 is entirely novel, and particularly interesting, due to the unusual *or* condition that appears in Property 3 of RQS. In the following we prove Theorem 3.

*Proof.* Theorem 3 states that there is no  $(Q^{(3)}, B)$ -atomic storage algorithm that is both  $(1, Q^{(1)})$ -fast (for some  $Q^{(1)} \neq \emptyset$ ) and  $(2, Q^{(2)})$ -fast, if Property 3 of RQS is violated. Assume by contradiction that such a storage algorithm  $A$  exists even if Property 3 of RQS is violated. Consider a simple SWMR storage algorithm with a single writer  $w$  and two distinct readers  $w \neq r_1 \neq r_2 \neq w$ . In the following, we denote by  $\overline{X}$  the set  $S \setminus X$ , where  $X$  is any subset of the set  $S$  (recall that  $S$  denotes the set of all servers). Negating  $P3(Q^{(1)}, Q^{(2)}, Q^{(3)})$  (Property 3 of RQS) yields (having in mind  $Q^{(1)} \neq \emptyset$ ):

$$\exists Q_1 \in \mathbf{Q}^{(1)}, \exists Q_2 \in \mathbf{Q}^{(2)}, \exists Q \in \mathbf{Q}^{(3)}, \exists B'_1, B_2 \in \mathbf{B}: (Q_2 \cap Q \setminus B'_1 = B_2) \wedge (Q_1 \cap Q_2 \cap Q \setminus B'_1 = \emptyset).$$

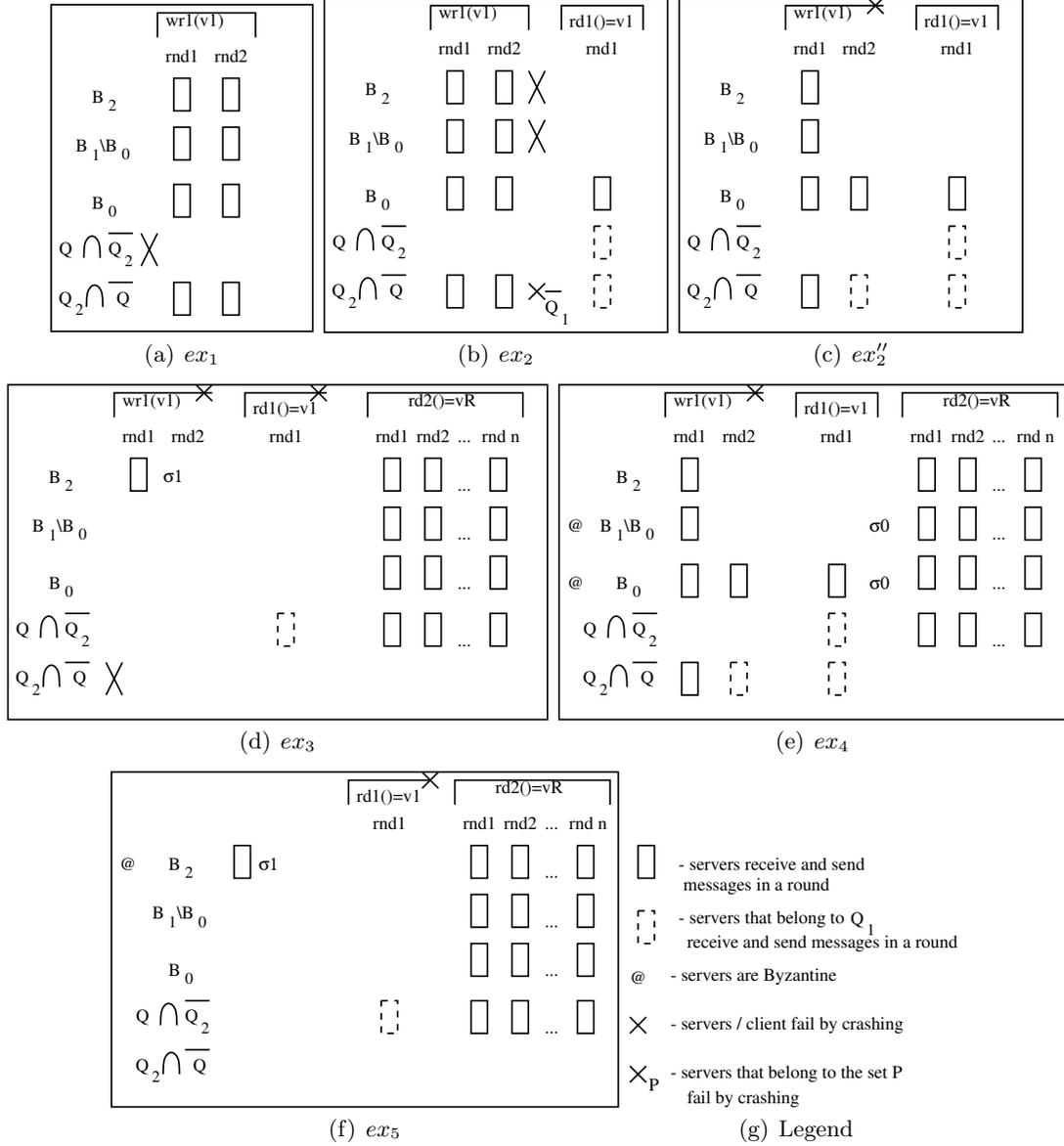
In the following, we denote the set  $Q_1 \cap Q_2 \cap Q$  by  $B_0$  and  $Q_2 \cap Q \cap B'_1$  by  $B_1$ . Having in mind that  $\mathbf{B}$  is an adversary for  $S$ , it is straightforward to verify the following:

- $B_0, B_1 \subseteq B'_1$ ,
- $B_0, B_1 \in \mathbf{B}$ , and
- $Q_2 \cap Q = B_1 \cup B_2$ .

Moreover, since  $B_0 \subseteq B'_1$  and  $B_0 \subseteq Q_2 \cap Q$ , we have  $B_0 \subseteq B_1$ . Hence,  $Q_2 \cap Q \cap \overline{Q_1} = B_2 \cup (B_1 \setminus B_0)$ .

To exhibit a contradiction, we construct several partial executions (sketched in Figure 8) of the algorithm  $A$  including one in which atomicity is violated. More specifically, in this particular partial execution, a read operation returns a value that was never written.

- Let  $ex_1$  be the execution in which all servers from  $Q_2$  are correct, while all others (i.e., those from  $\overline{Q_2}$ ) fail by crashing at the beginning of the execution. Furthermore, let  $wr_1$  be the write operation invoked at time  $t_1$  by the correct writer  $w$  in  $ex_1$  to write a value  $v_1 \neq \perp$  in the storage. Moreover, assume that the system is synchronous in  $ex_1$ . Hence,  $wr_1$  is synchronous and uncontended. Since  $A$  is  $(2, \mathbf{Q}^{(2)})$ -fast,  $wr_1$  completes in  $ex_1$ , say at time  $t'_1$ , in at most two communication rounds, after the writer receives the replies in round 2 from servers from  $Q_2$ .
- Let  $ex'_1$  be the partial execution that ends at  $t'_1$ , such that  $ex'_1$  is identical to  $ex_1$  up to time  $t'_1$ , except that in  $ex'_1$  servers from  $\overline{Q_2}$  do not crash, but, due to asynchrony, all messages sent by the writer to  $\overline{Q_2}$  during  $wr_1$  remain in transit. Since the writer cannot distinguish  $ex'_1$  from  $ex_1$ ,  $wr_1$  completes in  $ex'_1$ , in two communication rounds, at time  $t'_1$ .
- Let the partial execution  $ex_2$  extend  $ex'_1$  such that: (1) servers from  $\overline{Q_1}$  crash at  $t'_1$ , (2)  $rd_1$  is a synchronous read operation invoked by the correct reader  $r_1$  after  $t'_1$ , and (3) no other operation is invoked (hence,  $rd_1$  is uncontended). Since  $A$  is  $(1, \mathbf{Q}^{(1)})$ -fast,  $rd_1$  completes in a single round (since a set  $Q_1$  of servers is correct) at time  $t_2$  and returns  $v_1$ . Moreover, let  $ex_2$  end at  $t_2$ . All messages that were in transit in  $ex'_1$  remain in transit in  $ex_2$ .
- Let  $ex'_2$  be the partial execution identical to  $ex_2$  except that in  $ex'_2$  servers from  $\overline{Q_1}$  do not crash, but, due to asynchrony, the message sent from  $r_1$  to servers in  $\overline{Q_1}$  during  $rd_1$  remains in transit in  $ex'_2$ . Since  $r_1$  and all servers, except those from  $\overline{Q_1}$ , cannot distinguish  $ex'_2$  from  $ex_2$ ,  $rd_1$  completes in  $ex'_2$  in a single round, at time  $t_2$ , and returns  $v_1$ .
- Let  $ex''_2$  be the partial execution identical to  $ex'_2$  except that, in  $ex''_2$ : (1) the writer crashes during  $wr_1$  and its round 2 messages are not received by any servers from  $\overline{Q_1} \cup \overline{Q_2}$  (i.e., only servers from  $Q_1 \cap Q_2$  receive the round 2 message from the writer). Note that all servers from the set  $B_2 \cup (B_1 \setminus B_0)$  belong to  $\overline{Q_1}$  and, hence, do not receive a round 2 message from the writer. Since  $r_1$  and all servers, except those from  $\overline{Q_1} \cap Q_2$ , cannot distinguish  $ex''_2$  from  $ex'_2$ ,  $rd_1$  completes in  $ex''_2$  at time  $t_2$  and returns  $v_1$ .
- Consider now a partial execution  $ex_3$  slightly different from  $ex''_2$  in which the writer (resp., the reader  $r_1$ ) crashes during the round 1 of  $wr_1$  (resp.,  $rd_1$ ) such that the round 1 messages sent by the writer (resp.,  $r_1$ ) in  $wr_1$  (resp.,  $rd_1$ ) are received only by servers from the set  $B_2$  (resp.,  $Q \cap \overline{Q_2} \cap Q_1$ ). We refer to the state of servers that belong to the set  $B_2$  after sending the reply to the round 1 message of  $wr_1$  as to  $\sigma_1$ . In  $ex_3$ , all servers are correct except those from the set  $\overline{Q}$  that fail by crashing at the beginning of partial execution  $ex_3$ . Assume that the writer crashes at time  $t_{fail_w}$  and that  $r_1$  crashes at time  $t_{fail_r} > t_{fail_w}$ . Let  $rd_2$  be a read operation invoked by the correct reader  $r_2 \neq r_1$  at time  $t'_3 > \max(t_{fail_r}, t_2)$ . Since all servers from the set



**Fig. 8.** Illustration of the partial executions used in the proof of Theorem 3. Only servers that belong to the set  $Q_2 \cup Q$  are depicted.

- $Q$  are correct in  $ex_3$  and  $A$  is a  $(\mathcal{Q}^{(3)}, \mathcal{B})$ -atomic storage algorithm,  $rd_2$  eventually completes, at some point in time  $t_3$ , after  $n$  communication rounds and returns value  $v_R$ .
- Let  $ex_4$  be a partial execution identical to  $ex_2''$  except that in  $ex_4$ : (1) a read operation  $rd_2$  is invoked by the correct reader  $r_2$  at  $t_3'$  (as in  $ex_3$ ), (2) due to asynchrony all messages sent by servers from  $\overline{Q}$  to  $r_2$  are delayed until after  $t_3$  (i.e., until after  $n^{\text{th}}$  round of  $rd_2$ ) and (3) in  $ex_4$ , all servers from  $B_1$  (and  $B_0$ , since  $B_0 \subseteq B_1$ ) are Byzantine: these servers forge their state at time  $t_2$  to  $\sigma_0$  (the initial state of servers); otherwise, servers from  $B_1$  obey the protocol (including with respect to the writer and the reader  $r_1$ ). Note that  $r_2$  and servers from  $Q \setminus B_1 = B_2 \cup (Q \cap \overline{Q_2})$  cannot distinguish  $ex_4$  from  $ex_3$  and, hence,  $rd_2$  completes in  $ex_4$  at time  $t_3$  (as in  $ex_3$ ) and returns  $v_R$ . On the other hand,  $r_1$  cannot distinguish  $ex_4$  from  $ex_2''$ . Hence,  $rd_1$  completes in a single round and returns  $v_1$ . By atomicity, since  $rd_1$  precedes  $rd_2$ ,  $v_R$  must equal  $v_1$ .
  - Consider now the partial execution  $ex_5$ , identical to  $ex_3$ , except that in  $ex_5$ : (1)  $wr_1$  is never invoked, (2) servers from  $B_2$  are Byzantine in  $ex_5$  and forge their state to  $\sigma_1$  (see  $ex_3$ ); otherwise, servers from  $B_2$  send the same messages as in  $ex_3$ , and (3) servers from  $\overline{Q}$  do not crash in  $ex_5$ , but, due to asynchrony, all messages sent from servers from  $\overline{Q}$  to  $r_2$  are delayed until after  $t_3$  (i.e.,  $n^{\text{th}}$  round of  $rd_2$ ). The reader  $r_2$  and servers from  $Q \setminus B_2 = B_1 \cup (Q \cap \overline{Q_2})$  cannot distinguish  $ex_5$  from  $ex_3$ , so  $rd_2$  completes at time  $t_3$  and returns  $v_R$ , i.e.,  $v_1$  (see  $ex_4$ ). However, by atomicity, in  $ex_5$ ,  $rd_2$  must return  $\perp$ , the initial value of the atomic storage. Since  $v_1 \neq \perp$ ,  $ex_5$  violates atomicity.  $\square$

## 4 Consensus

In this section, we give the second example of using RQS to obtain a novel implementation of an important abstraction, optimal in terms of resilience and best-case complexity. The example we consider here is that of implementing a *consensus* abstraction, which is at the heart of the state machine replication technique [32], widely considered for building general reliable services (beyond the storage abstraction).

The consensus algorithm we present tolerates Byzantine failures of processes and unbounded periods of asynchrony. In fact, it is the first consensus algorithm that tolerates an unbounded number of Byzantine proposers and learners (in practical state machine replication algorithms, unbounded number of proposers and learners would typically be translated into the unbounded number of clients). The algorithm is optimal in terms of resilience as well as complexity, matching the lower bounds of [35] and closing, we believe, a very important gap. The notion of complexity considered here is again best-case complexity for this is considered practically appealing on the one hand and, on the other hand, the worst-case complexity of a consensus algorithm that tolerates arbitrarily long periods of asynchrony is anyway unbounded. Our algorithm expedites the consensus decision under best-case conditions (synchrony and no contention) without using data authentication primitives; however, when best-case conditions are not met, data authentication primitives are indeed used.

In the remainder of this section, after presenting the model, we describe our consensus algorithm and then state its optimality. The correctness proof of our consensus algorithm is given in Appendix B.

### 4.1 Model

We model processes and channels in the same way as in Section 3.1, with the following differences:

- In contrast to Section 3.1: (1) channels are not assumed to be reliable (i.e., messages can be lost), (2) processes that access RQS *can be* Byzantine, and (3) processes that form RQS may directly with each other (this also illustrates application of RQS to different models).
- We assume that the system is eventually synchronous [13] (this is crucial to circumvent the impossibility of an asynchronous consensus [14]). Eventual synchrony means that there is a point in time  $GST$  (Global Stabilization Time), not known to processes, such that, after  $GST$ , the system is synchronous. In addition, we assume that all messages sent before  $GST$ , are either received by  $GST$  or lost.
- We allow messages to be authenticated with digital signatures [47]; however, we disallow the use of authenticated messages in best-case executions (to avoid, in best-case executions, the inherent practical latency overhead introduced by signatures). We denote by  $\langle m \rangle$  an unauthenticated message, and by  $\langle m \rangle_{\sigma_x}$  an authenticated message, i.e., a message signed by process  $x$ . We assume that no Byzantine process  $p_B$  cannot forge a digital signature of some benign process  $p$ , i.e., if  $p_B$  sends  $\langle m \rangle_{\sigma_p}$ , in execution  $ex$  then  $p$  already sent  $\langle m \rangle_{\sigma_p}$  in  $ex$ .

Our consensus framework is composed of three sets of processes: *proposers*, *acceptors* and *learners* [34]. Roughly, proposers propose values (from domain  $\mathbb{D}$ ) that are to be agreed upon by learners, where the role of acceptors is to help learners agree. In this paper, as in [52], we assume that the set *acceptors* does not intersect with the set *proposers*  $\cup$  *learners*, i.e., no proposer or learner can be an acceptor (note that we allow a proposer to be also a learner). We assume that every proposer  $p$  is initialized with a single proposal value and all processes are interconnected with point-to-point communication channels.

An algorithm solves consensus if it satisfies the following properties.

- (Validity:) If a benign learner learns a value  $v$  and all proposers are benign, then some proposer proposes  $v$ ;<sup>5</sup>
- (Agreement:) No two benign learners learn different values;
- (Termination:) If a correct proposer proposes a value, then eventually, every correct learner learns a value.

We construct a refined quorum system  $RQS$  around the set *acceptors* for an adversary  $\mathbf{B}$ , such that  $RQS$  is known to all processes. Besides Byzantine acceptors that may belong to adversary, any number of proposers and learners can be Byzantine. Consensus *safety* (i.e., Validity and Agreement) is guaranteed as long as the set of Byzantine acceptors in any execution belongs to  $\mathbf{B}$ , while consensus *liveness* (i.e., Termination) is ensured if there is a correct quorum of acceptors  $Q_c \in RQS$ .

We say that an execution  $ex$  is a *best-case* execution if, in  $ex$ : (1) there is no contention, i.e., (a) all proposers are benign and (b) exactly one proposer  $p$  proposes, say some value  $v$  at time  $t$  (and  $p$  is correct) and (2) the system is synchronous (during  $[t, t + 4\Delta]$ ). Let  $\mathbf{P}$  be any set of subsets of *acceptors*.

**Definition 4.** ( $(m, \mathbf{P})$ –fast consensus algorithm. We say that a consensus algorithm is  $(m, \mathbf{P})$ –fast if in every best-case execution  $ex$  in which some set  $P \in \mathbf{P}$  contains only correct acceptors, all correct learners learn  $v$  in  $m + 1$  message delays<sup>6</sup>, without using authenticated messages.

<sup>5</sup> The Validity property, as stated in [35], “Only a value proposed by a proposer can be learned“, is clearly impossible to ensure in the presence of Byzantine proposers.

<sup>6</sup> In our round-by-round eventually synchronous model [15, 30], a single *message delay* corresponds to a single *round*. In the following, when explaining our algorithm, instead of the term *round*, we use the term *message delay*

In the following, we present a novel consensus algorithm, based on RQS, that is  $(m, \mathbf{QC}_m)$ -fast for all  $m \in \{1, 2, 3\}$ .

## 4.2 Consensus Algorithm

The algorithm consists of two modules: (1) a *Locking* module that ensures safety, and (2) an *Election* module used to help ensure liveness. The algorithm proceeds in a sequence of *views*. In every view, a single proposer is the *leader*, except in the initial view ( $initView = 0$ ) where all proposers can be seen as leaders. The *Locking* module consists of a *consult* (Fig. 12) and an *update* phase (Fig. 10). In  $initView$ , the proposer, on proposing a value, skips the consult phase and executes immediately the update phase (see Fig. 9). In the following, we first explain the update phase.

---

```

Initialization:
view, initView := 0

propose(v) is {
  if (view ≠ initView) then
    consult phase
  endif
  update phase }

upon pj is elected
  propose(v)

```

---

**Fig. 9.** The *Locking* module: High level pseudocode of a proposer  $p_j$

**Update phase.** Communication pattern of the *update* phase is illustrated in Figure 11; it takes 4 communication steps and allows correct learners to learn a value in  $m + 1$  communication steps in best-case executions in which there is a quorum of class  $m$  which contains only correct acceptors (for  $m \in \{1, 2, 3\}$ ). This phase proceeds in the following communication steps (line numbers refer to Fig. 10):

1. Proposer  $p$  sends a message  $m = \text{prepare}$  to all acceptors (line 9) containing: (a) its proposal value  $v$ , (b) the view number  $view$ , and (c) the array of authenticated messages  $vProof$  that originates from some quorum  $Q$  of acceptors. Roughly,  $vProof$  serves as a certificate for the proposed value  $v$ . We detail how  $vProof$  is constructed later when explaining the consult phase. It is important to notice that, in the  $initView$ ,  $vProof$  equals  $nil$  (i.e., contains no messages).
2. Benign acceptor  $a_j$ , upon receiving  $m = \text{prepare}\langle v, view, vProof, Q \rangle$  from  $p$ , such that  $view = view_{a_j}$  (i.e., if  $a_j$  is in  $view$ ), checks if (line 31): (a) (unless  $view = initView$ ) whether  $p$  is the leader of  $view$  and whether  $vProof$  matches value  $v$  (this is done using the  $choose()$  function that is explained later in details), and (b)  $a_j$  did not already receive a  $\text{prepare}$  message in  $view$ . If these checks succeed,  $a_j$  stores  $v$  into a local variable  $Prep$  and the view number in the set variable  $Prep_{view}$  (we simply say,  $a_j$  prepares  $v$  in  $view$ ). If  $Prep$  was already equal to  $v$ , then  $a_j$  simply adds  $view$  to the set  $Prep_{view}$  (line 32). Then  $a_j$  echoes  $v$  by sending an  $\text{update}_1\langle v, view, \emptyset \rangle$  message to all acceptors and learners (line 33).

---

to prevent confusion with the notion of a round in a sense of a *communication round-trip* used in our storage algorithm (that, in a sense, corresponds to two message delays).

---

**at every proposer  $p_j$ :**  
Initialization:  
 $view, initView := 0; vProof := nil; Q := \emptyset$   
9: send **prepare** $\langle v, view, vProof, Q \rangle$  to *acceptors*

---

**at every acceptor  $a_j$ :**  
Initialization:  
 $view_{a_j} := initView; Prep_{view}, old, Update_{proof}[*], Update_{view}[*], Update_Q[*], * := \emptyset; Prep, Update[*] := nil$

**upon received  $m = \text{prepare}\langle v, view_{a_j}, vProof, Q \rangle$  from  $p_i$**   
31: **if**  $(w \in Prep_{view} \Rightarrow w < view_{a_j})$  **and**  $(view_{a_j} = initView$  **or**  $(p_i$  is leader **and**  $v$  matches **choose** $\langle v, vProof, Q \rangle))$  **then**  
32:     **if**  $Prep = v$  **then**  $Prep_{view} := Prep_{view} \cup \{view_{a_j}\}$  **else**  $Prep := v; Prep_{view} := \{view_{a_j}\}$   
33:     send  $m_1 = \text{update}_1\langle v, view_{a_j}, \emptyset \rangle$  to  $acceptors \cup learners; old := old \cup m_1$

**upon received  $m = \text{update}_{step}\langle v, view_{a_j}, * \rangle$  from some quorum  $Q$  and  $v = Prep$  and  $view_{a_j} \in Prep_{view}$  (for  $step \in \{1, 2\}$ )**  
34: **if**  $Update[step] = v$  **then**  $Update_{view}[step] := Update_{view}[step] \cup \{view_{a_j}\}$   
35:     **else**  $Update[step] := v; Update_{view}[step] := \{view_{a_j}\}; Update_Q[step, *] := \emptyset; Update_{proof}[step, *] := \emptyset$   
36: **if**  $(Q \notin Update_Q[step, view_{a_j}]$  **and**  $step = 1)$  **or**  $(Update_Q[step, view_{a_j]} = \emptyset$  **and**  $step = 2)$  **then**  
37:      $Update_Q[step, view_{a_j}] := Update_Q[step, view_{a_j}] \cup Q$   
38:     send  $m_{step+1} = \text{update}_{step+1}\langle v, view_{a_j}, Q \rangle$  to  $acceptors \cup learners; old := old \cup m_{step+1}$

---

**at every acceptor and learner  $x$ :**  
**upon received the same  $\text{update}_1\langle v, view, * \rangle$  from  $Q_1 \in QC_1$**   
51: **if**  $x$  has not yet decided **then decide** $\langle v \rangle$

**at every learner  $l_j$ :**  
**upon  $l_j$  decides  $v$**   
60: **learn** $\langle v \rangle$

**upon received the same  $\text{update}_2\langle v, view, Q_2 \rangle$  from  $Q_2 \in QC_2$**   
52: **if**  $x$  has not yet decided **then decide** $\langle v \rangle$

**upon received the same  $\text{update}_3\langle v, view, * \rangle$  from  $Q_3 \in RQS$**   
53: **if**  $x$  has not yet decided **then decide** $\langle v \rangle$

---

Fig. 10. The *Locking* module: *update* phase

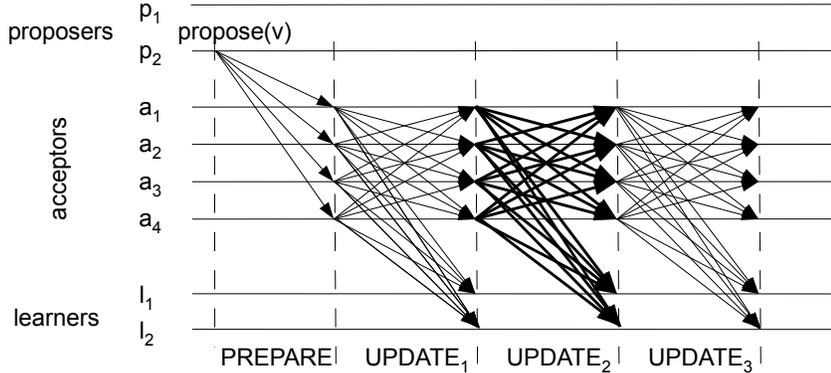


Fig. 11. Communication pattern of the *update* phase. An acceptor may send multiple  $\text{update}_2$  messages (bolded).

3. Benign acceptor  $a_j$ , upon receiving  $\text{update}_1$  messages from some quorum  $Q$  with the same value  $v$  and view number  $view$ , checks if its local view equals  $view$  and if it already prepared a message with a value  $v$  in  $view$ . If this check succeeds,  $a_j$  performs the following local computations (we say  $a_j$  1-updates  $v$  in  $view$  with quorum  $Q$ ). In case the local variable  $Update[1]$  does not equal  $v$  (i.e., if a new value is 1-updated — line 35),  $a_j$ : (i) stores  $v$  into  $Update[1]$  and  $view$  into  $Update_{view}[1]$ , and (ii) empties the sets  $Update_Q[1, *]$  and  $Update_{proof}[1, *]$ . In case a value  $v$  was already 1-updated (in some previous view — line 34),  $a_j$  simply adds  $view$  into  $Update_{view}[1]$ . Then,  $a_j$  adds the identifier of the quorum  $Q$  into the set  $Update_Q[1, view]$  and sends an  $\text{update}_2\langle v, view, Q \rangle$  message to all acceptors and learners (here, an  $\text{update}_2$  message is sent once per every different quorum  $Q$  — line 38).
4. A benign acceptor  $a_j$ , upon receiving  $\text{update}_2$  message from some quorum  $Q$ , performs the similar steps as when receiving a quorum of  $\text{update}_1$  messages (we say  $a_j$  2-updates  $v$  in  $view$ ), including sending an  $\text{update}_3$  message containing  $v$  and  $view$  to all acceptors and learners (lines 34-38). The differences with respect to the step (3) are captured in lines 36-38; namely in step (4) an acceptor: (i) adds only one quorum  $Q$  (the first one) to  $Update_Q[2, *]$  per view (lines 36 and 37), and (ii) sends only one  $\text{update}_3\langle v, view, * \rangle$  message per view to other acceptors and learners.

Moreover, all acceptors and learners decide on  $v$  upon receiving  $\text{update}_1$  messages with the same value  $v$  and view number  $view$  from a class 1 quorum (line 51). Similarly, acceptors and learners decide on  $v$  upon receiving the same  $\text{update}_2\langle v, view, Q_2 \rangle$  messages from some class 2 quorum  $Q_2$  (note here that, besides value and the view number, the quorum identifier within  $\text{update}_2$  messages must be the same — line 52). Finally, acceptors and learners decide on  $v$  upon receiving  $\text{update}_3$  messages with the  $v$  and  $view$  from any (class 3) quorum of acceptors (line 53). Besides, a benign learner  $l_j$  learns  $v$  as soon as  $l_j$  decides  $v$  (line 60).

The above scheme guarantees that, in the best case execution, in which only a single proposer proposes in the *initView* and the system is synchronous, all correct learners learn  $v$  in two (resp., three; four) message delays in case a class 1 (resp., class 2; class 3) quorum of correct servers is available.<sup>7</sup> Note that, in the above sequence, all messages are unauthenticated.

**Consult phase.** In the best-case execution, the *Election* module, responsible for view changes, does not change the view before all correct acceptors (and learners) decide  $v$ . However, if more than one proposer proposes in *initView*, or some proposer is Byzantine, or if the system is asynchronous, the *Election* module might designate a different proposer  $p_i$  to be the leader for the new view  $w$  (see Fig. 9). In our algorithm, the idea behind the structure of view changes is similar to the one in the algorithm of Castro and Liskov [7]. The core difference is in the way our algorithm chooses the proposal value in the new view (*choose()* function, Fig. 13, explained later in details).

Proposer  $p_i$  starts the consult phase of a new view  $w$  by sending the *new\_view* message to acceptors (line 2, Fig. 12). The *new\_view* message contains a view number and a set of messages, *viewProof*, which are provided to the proposer by the *Election* module. The set *viewProof* consists of signed (authenticated) messages from a quorum of acceptors — this vouches for the authenticity of the *new\_view* message. After sending the *new\_view* message,  $p_i$  waits for a quorum  $Q$  of *valid* signed acks (line 4) containing the last prepared, 1-updated and 2-updated values, along with the

<sup>7</sup> Since an availability of a class 3 quorum is anyway assumed, our algorithm guarantees that a value will be learned by all correct learners in at most four message delays in any best-case execution.

---

**at every proposer  $p_j$ :**  
Initialization:  
 $view, initView := 0; viewProof, vProof := nil; \mathbf{faulty} := \emptyset$   
2: send  $\mathbf{new\_view}\langle view, viewProof \rangle$  to *acceptors*  
3: **repeat**  
4:   **wait for** valid acks from some quorum  $Q \in RQS \setminus \mathbf{faulty}$   
5:    $vProof :=$  array of received acks from  $Q$   
6:    $(v, abort) := \mathbf{choose}(v, vProof, Q)$   
7:   **if**  $abort$  **then**  $\mathbf{faulty} := \mathbf{faulty} \cup \{Q\}$   
8: **until**  $\neg(abort)$

---

**at every acceptor  $a_j$ :**  
Initialization:  
 $view_{a_j} := initView; Prep_{view}, old, Update_{proof}[*], Update_{view}[*], Update_Q[*] := \emptyset; Prep, Update[*] := nil$

**upon** received  $\mathbf{new\_view}\langle view, viewProof \rangle$  from  $p_i$   
21: **if**  $(view > view_{a_j})$  **and** ( $p_i$  is the leader of  $view$ ) **and** ( $viewProof$  matches  $view$ ) **then**  
22:    $view_{a_j} := view$   
23:    $\forall step \in \{1, 2\}, \forall w : w \in Update_{view}[step] \wedge Update_{proof}[step, w] = \emptyset$  **do**  
24:     send  $\mathbf{sign\_req}\langle Update[step], w, step \rangle$  to some quorum in  $Update_Q[step, w]$   
25:   **for** every sent  $\mathbf{sign\_req}\langle Update[step], w, step \rangle$  message  
26:     **wait for** acks with a valid signature from some subset of *acceptors*  $T_{step, w}, T_{step, w} \notin \mathbf{B}$   
27:      $Update_{proof}[step, w] :=$  received acks from  $T_{step, w}$   
28:     send  $\mathbf{new\_view\_ack}\langle view_{a_j}, Prep, Prep_{view}, Update[1..2], Update_{view}[1..2], Update_{proof}[1..2, *], Update_Q[1..2, *] \rangle_{\sigma_{a_j}}$  to  $p_i$

**upon** received  $\mathbf{sign\_req}\langle v, w, step \rangle$  from  $a_i$   
29: **if**  $m = \mathbf{update}_{step}\langle v, w, * \rangle \in old$  **then** send  $\mathbf{sign\_ack}\langle m \rangle_{\sigma_{a_j}}$  to  $a_i$

---

**Fig. 12.** The *Locking* module: *consult* phase

corresponding view numbers. An acceptor  $a_j$  acks a  $\mathbf{new\_view}$  message only if (line 21, Fig. 12): (a) the view number  $w$  is higher than the acceptor's local view number  $view_{a_j}$ , (b)  $p_i$  is the leader of the view  $w$  (i.e., if  $i = w \bmod |proposers|$ ), and (c) the set  $viewProof$  matches  $w$ , i.e., if  $viewProof$  proof contains a quorum of authenticated messages vouching that  $p_i$  may issue a  $\mathbf{new\_view}$  message for a view  $w$ .

An ack (i.e., a  $\mathbf{new\_view\_ack}$  message — line 28, Fig. 12) for a view  $w$  is considered *valid* in line 4, Fig. 12 if every value  $v_{step}$  in  $Update[step]$  and every view number  $w'$  in  $Update_{view}[step]$  ( $step \in \{1, 2\}$ ) is accompanied by a set of signatures,  $Update_{proof}[step, w']$ . Here, every  $Update_{proof}[step, w']$  must be a set of signed  $\mathbf{update}_{step}\langle v_{step}, w', * \rangle$  messages sent from all acceptors from some subset of acceptors that is not an element of an adversary (to guarantee that a message is signed by at least one benign acceptor). An acceptor  $a_j$  must obtain all the necessary sets of signatures before replying to the  $\mathbf{new\_view}$  message — this is done in lines 23-27, Fig. 12, unless  $a_j$  already possesses the required proofs.

Then, the leader of view  $w$ ,  $p_i$ , evaluates acks from  $Q$  using the  $\mathbf{choose}()$  function (line 6, Fig. 12 and Fig. 13).

**Choose function.** The  $\mathbf{choose}()$  function is the heart of our algorithm and relies on RQS properties to guarantee consensus safety. This function ensures the following crucial property: *if any value  $v$  is decided in a view  $w$ , then benign acceptors in a view higher than  $w$  accept only  $v$ .* We sketch the arguments (based on RQS properties) behind this property, for the view  $w + 1$  (which gives the base step of the induction-based proof). In the following, we refer to Figure 13.

---

Definitions:

- 1:  $Cand_2(v, w) ::= \exists Q_1 \in \mathbf{QC}_1, \exists B \in \mathbf{B}, \forall a_j \in (Q_1 \cap Q) \setminus B : (w \in vProof[a_j].Prep_{view}) \wedge (vProof[a_j].Prep = v)$
- 2:  $C_3(v, w, char, Q_2, B) ::= \forall a_j \in (Q_2 \cap Q) \setminus B : P_{3char}(Q_2, Q, B) \wedge (vProof[a_j].Update[1] = v) \wedge (w \in vProof[a_j].Update_{view}[1]) \wedge (Q_2 \in vProof[a_j].Update_Q[1, w])$
- 3:  $Cand_3(v, w, char) ::= \exists Q_2 \in \mathbf{QC}_2, \exists B \in \mathbf{B} : C_3(v, w, char, Q_2, B)$
- 4:  $Valid_3(v, w, char) ::= \forall Q_2 \in \mathbf{QC}_2, \forall B \in \mathbf{B}, \forall a_j \in Q_2 \cap Q, \forall w' \in \mathbb{N} : C_3(v, w, char, Q_2, B) \Rightarrow ((vProof[a_j].Prep = v) \wedge (w \in vProof[a_j].Prep_{view})) \vee (w' \in vProof[a_j].Prep_{view} \Rightarrow w' > w)$
- 5:  $Cand_4(v, w) ::= \exists a_j \in Q : (vProof[a_j].Update[2] = v) \wedge (w \in vProof[a_j].Update_{view}[2])$

**choose**( $v', vProof, Q$ ) **returns**( $v_{ret}, abort$ ) **is** {

- 10:  $v_{ret} := v'; abort := false$
  - 11: **if**  $\exists v \in \mathbb{D}, \exists w \in \mathbb{N}_0, \exists char \in \{ 'a', 'b' \} : Cand_2(v, w) \vee Cand_3(v, w, char) \vee Cand_4(v, w)$  **then**
  - 12:      $view_{max} := \max\{w \in \mathbb{N}_0 \mid \exists v \in \mathbb{D}, \exists char \in \{ 'a', 'b' \} : Cand_2(v, w) \vee Cand_3(v, w, char) \vee Cand_4(v, w)\}$
  - 13:     **if**  $\exists v \in \mathbb{D} : Cand_3(v, view_{max}, 'a') \vee Cand_4(v, view_{max})$  **then**
  - 14:          $v_{ret} := v$ ; **return**
  - 15:     **if**  $\exists v, v' \in \mathbb{D} : (v \neq v') \wedge Cand_3(v, view_{max}, 'b') \wedge Cand_3(v', view_{max}, 'b')$  **then**
  - 16:          $abort := true$ ; **return**
  - 17:     **if**  $\exists v \in \mathbb{D} : Cand_3(v, view_{max}, 'b')$  **then**
  - 18:         **if**  $Valid_3(v, view_{max}, 'b')$  **then**  $v_{ret} := v$  **else**  $abort := true$
  - 19:         **return**
  - 20:      $v_{ret} := v \in \mathbb{D} : Cand_2(v, view_{max})$ ; **return**
  - 21: **else return**
- 

**Fig. 13.** *choose()* function

Let  $v$  be the value decided by some benign process (acceptor or learner) in view  $w$  upon receiving  $update_k$  messages from some class  $k$  quorum  $Q_k$ . Then, for any  $Q$ , substituting for  $Q_1$  (resp.,  $Q_2$ ;  $Q_3$ ),  $Cand_2(v, w)$  holds in line 1 (resp.,  $Cand_3(v, w, char)$  holds for some  $char \in \{ 'a', 'b' \}$  in line 3;  $Cand_4(v, w)$  holds in line 5). In this case, we say that  $v$  is a *candidate* with view number  $w$ . It is not difficult to see that there can be no candidate  $v$  with view  $w' > w$  (since no benign acceptor prepares or updates any value in a view higher than  $w$ ), i.e.,  $w = view_{max}$  in line 12. Hence, *choose()* may return only the candidate with view number  $w$ . However, *choose()* faces the challenging tasks of prioritizing different candidate values with the same view number  $w$ . In the following, we illustrate how *choose()* relies on RQS properties to safely prioritize candidate values, by focusing a particular example in which  $v$  was decided in view  $w$  after some benign acceptor or learner receives  $update_1$  messages from some class 1 quorum  $Q_1$ . In this case, since  $v$  was decided, *choose()* must not return a value  $v' \neq v$ . We show that, for a valid  $vProof$  that consists of `new_view_ack` messages sent by *any* quorum  $Q$ , *choose*( $*$ ,  $vProof$ ,  $Q$ ) either returns  $v$  or sets the *abort* flag. In the latter case, we will show that  $Q$  actually contains some Byzantine acceptor; in this case, the proposer will simply wait for additional `new_view_ack` messages from other acceptors and reinvoke *choose()* until it encounters a quorum that does not contain Byzantine acceptors, when *choose()* will not abort (see lines 3-8, Fig. 12).

As we suggested above, in this case,  $Cand_2(v, w)$  holds. To see this, denote the set of Byzantine acceptors in any execution  $ex$  by  $B_{ex} \in \mathbf{B}$ . Then  $vProof$  contains `new_view_ack` messages from all acceptors from the set  $X = Q_1 \cap Q \setminus B_{ex}$ . Since all acceptors from  $X$  are benign they all correctly inform the proposer that they prepared  $v$  in view  $w$  and, hence,  $Cand_2(v, w)$  holds. The following arguments show that the low priority that candidate values for which  $Cand_2(v, w)$  holds have in *choose()* (such candidate values can be returned only in line 20), does not compromise safety:

1. If, for value,  $v' \neq v$ : (i)  $Cand_3(v', w, 'a')$  or (ii)  $Cand_4(v, w)$  hold,  $v'$  will be returned in line 14. However, this is not possible. In case (i), there are class 2 quorum  $Q_2$  and a set  $B \in \mathbf{B}$  such that

all acceptors from  $Y = Q_2 \cap Q \setminus B$  reported they 1-updated  $v'$  in  $w$ . Since,  $P_{3a}(Q_2, Q, B)$  holds, this includes at least one benign acceptor. Similarly, in case (ii) some acceptor  $a_j \in Q$  reported that it 2-updated  $v'$  in  $w$ . This means that in case (ii), just like in case (i), there is at least one benign acceptor that 1-updated  $v'$  in  $w$  (in a valid  $vProof$ ,  $vProof[a_j].Update_{proof}[2]$  contains a set of signatures from all acceptors from the set  $T \notin \mathbf{B}$  confirming that they 1-updated  $v'$  in  $w$ ). Since benign acceptors 1-update  $v$  in  $w$  only if all acceptors from  $Q' \setminus B_{ex}$  (for some quorum  $Q'$ ) prepared  $v$  in  $w$  — by Property 1 of RQS, since  $Q_1 \cap Q' \setminus B_{ex}$  is non-empty, this would imply that at least one benign acceptor prepared both  $v$  and  $v'$  in  $w$ . A contradiction.

2. It is less obvious that  $choose()$  cannot return  $v' \neq v$  in line 18, if both  $Cand_3(v', w, 'b')$  and  $Valid_3(v', w, 'b')$  hold. However, this is also impossible. Namely, in this case, there is a class 2 quorum  $Q_2$  and  $B \in \mathbf{B}$ , such that all acceptors from  $Y = Q_2 \cap Q \setminus B$  claim they 1-updated  $v'$  in  $w$ . If  $Y \not\subseteq B_{ex}$  there is one benign acceptor that indeed 1-updated  $v'$  in  $w$  — a contradiction follows the argument in the previous paragraph. Otherwise, if  $Y \subseteq B_{ex}$ , since  $Valid_3(v', w, 'b')$  holds, all (benign) acceptors from  $Z = Q_2 \cap Q \setminus B_{ex}$  prepared  $v'$  in  $w$ .<sup>8</sup> If  $P_{3b}(Q'_2, Q, B_{ex})$  holds, this implies that  $Z \cap Q_1$  is non-empty, i.e., at least one benign acceptor prepared both  $v$  and  $v'$  in  $w$  — a contradiction. The last possibility, that  $P_{3a}(Q_2, Q, B_{ex})$  holds, implies that  $P_{3a}(Q_2, Q, Y)$  also holds (since  $Y \subseteq B_{ex}$ ). However, this implies  $Q_2 \cap Q \setminus Y \notin \mathbf{B}$ , which contradicts the definition of  $Y$ .
3. Finally, it should not be difficult to see, by applying Property 2 of RQS, that  $Cand_2(v', w)$  cannot hold at the same time as  $Cand_2(v, w)$ , for some  $v' \neq v$ . Hence, choosing a candidate value in line 20 is not ambiguous.

Finally, in the following we explain why  $choose(*, vProof, Q)$  never aborts in case quorum  $Q$  contains only benign acceptor. Assume, by contradiction, that  $choose(*, vProof, Q)$  aborts, yet  $Q$  contains only benign acceptors.

1. If  $choose()$  aborts in line 16, there are two values  $v$  and  $v' \neq v$  such that both  $Cand_3(v, w, 'b')$  and  $Cand_3(v', w, 'b')$  hold. In this case there are (supposedly benign) acceptors  $a_i, a_j \in Q$  such that  $a_i$  (resp.,  $a_j$ ) claims it 1-updated  $v$  (resp.,  $v'$ ) in  $w$ , i.e., that it received  $update_1\langle v, w, \emptyset \rangle$  ( $update_1\langle v', w, \emptyset \rangle$ ) messages from some quorum  $Q'$  (resp.,  $Q''$ ) of acceptors. By Property 1 of RQS,  $Q \cap Q' \setminus B_{ex} \neq \emptyset$ , i.e., there is at least one benign acceptor  $a_k$  that sent both  $update_1\langle v, w, \emptyset \rangle$  and  $update_1\langle v', w, \emptyset \rangle$ , i.e.,  $a_k$  prepared both  $v$  and  $v'$  in view  $w$ . A contradiction.
2. If  $choose()$  aborts in line 18 then, there is a value  $v$  such that  $Cand_3(v, w, 'b')$  holds, yet  $Valid_3(v, w, 'b')$  does not hold. In this case, there is a class 2 quorum  $Q_2$  and a (benign) acceptor  $a_i \in Q$  that claims it received  $update_1\langle v, w, \emptyset \rangle$  messages from all (supposedly benign) acceptors from  $Q_2 \cap Q$ , i.e., that all acceptors from  $Q_2 \cap Q$  prepared  $v$  in  $w$ . However, since  $Valid_3(v, w, 'b')$  does not hold, there is a benign acceptor  $a_j \in Q_2 \cap Q$  for which the predicate  $P = P_1 \vee P_2$ , such that:

$$P_1 ::= (vProof[a_j].Prep = v) \wedge (w \in vProof[a_j].Prep_{view}), \text{ and} \\ P_2 ::= w' \in vProof[a_j].Prep_{view} \Rightarrow w' > w.$$

does not hold. Since  $a_j$  prepared  $v$  in  $w$ , there are two possibilities.

<sup>8</sup> Acceptors from  $Z$  could not have prepared a value in a view higher than  $w$  to satisfy  $Valid_3(v', w, 'b')$  since we consider only  $vProof$  for  $w+1$  in this example. Benign acceptors must be in the view lower than the one for which they send a `new_view_ack` message.

- (a)  $a_j$  never prepared a value different than  $v$  in views higher than  $w$ ; in this case  $P_1$  must hold, since  $a_j$  never removes  $w$  from its variable  $Prep_{view}$  (line 32, Fig. 10) — a contradiction.
- (b)  $a_j$  prepared a value different from  $v$  in a view higher than  $w$ , say in view  $w'$ . Then (line 32, Fig. 10),  $Prep_{view}$  at  $a_j$  contains only view numbers higher than  $w$ , i.e.,  $P_2$  must hold — a contradiction.

**The *Election* module.** The Election module given in Figure 14 is very simple and guarantees progress in case the system is eventually synchronous. It is based on an exponential increase of the timeouts (maintained by acceptors) between views. This scheme can be seen as inefficient, and impact the worst-case performance of our algorithm. Different optimizations of this simple scheme are possible, but these are out of the scope of this paper.

The pseudocode of the *Locking* module, combined from pseudocodes of Figures 9, 10 and 12 is given in Figure 15. The additional part of the *Locking* module consists of lines 40 and 101-104, Fig. 15, that serve solely to halt a consensus instance, i.e., to permanently stop the view-change mechanism and to halt proposers. Otherwise, lines 40 and 101-104 of Fig. 15 can be omitted.

The correctness proof of our consensus algorithm is given in Appendix B. In the following we state the optimality of our algorithm.

---

```

at every acceptor  $a_j$ :
 $suspectTimeout, initTimeout := 5\Delta; nextView_{a_j} := initView$                                 % Initialization

upon reception of  $prepare(*, initView, *, *)$  or  $sync$  message for the first time
0: trigger( $suspectTimeout$ )

upon expiration of ( $suspectTimeout$ )
1:  $suspectTimeout := suspectTimeout * 2$ 
2: inc( $nextView_{a_j}$ )
3:  $nextLeader := nextView_{a_j} \bmod |proposers|$ 
4: send  $view\_change(nextView_{a_j})_{\sigma_{a_j}}$  to  $p_{nextLeader}$ 
5: trigger( $suspectTimeout$ )

upon  $decide(v)$ 
7: send  $decision(v)$  to  $acceptors$ 

upon reception of a valid  $decision(v)$  from some quorum  $Q \in RQS$ 
8: stop( $suspectTimeout$ )

```

---

```

at every proposer  $p_j$ :

upon reception of  $view\_change(nextView)_{\sigma_{a_i}}$  with the same  $nextView$  from all  $a_i$  from some  $Q \in RQS$ 
10: if  $nextView > view$  then
11:    $viewProof := \cup$  received signed  $view\_change(nextView)$  messages
12:    $view := nextView$ 
13:   elect( $self$ )

upon  $p_j$  proposed a value for the first time
101: wait some preset time
102: send  $sync$  to  $acceptors$ 
103: send  $(decision\_pull)$  to  $acceptors$ 

upon received  $decision(v)$  (with the same  $v$ ) from some quorum  $Q \in RQS$ 
104: halt

```

---

**Fig. 14.** The *Election* module

### 4.3 Optimality

We say that an algorithm  $A$  implements  $(\mathbf{Q}, \mathbf{B})$ -consensus if  $A$  ensures consensus Validity and Agreement, as long as, for any execution  $ex$  of  $A$ , the set of acceptors Byzantine in  $ex$  belongs to  $\mathbf{B}$ , as well as Termination in case the system is eventually synchronous and there is a set  $Q \in \mathbf{Q}$  that contains only correct acceptors. In addition, analogously to Section 3.3, we define the properties  $P1(\mathbf{Q}^{(3)})$ ,  $P2(\mathbf{Q}^{(1)}, \mathbf{Q}^{(3)})$  and  $P3(\mathbf{Q}^{(1)}, \mathbf{Q}^{(2)}, \mathbf{Q}^{(3)})$ , where  $\mathbf{Q}^{(i)}$  is some set of subsets of *acceptors*. The following theorems capture the minimality of our RQS, assuming  $|\text{proposers}| \geq 2$  and  $|\text{learners}| \geq 3$ .<sup>9</sup>

**Theorem 4.** *If an algorithm  $A$  implements  $(\mathbf{Q}^{(3)}, \mathbf{B})$ -consensus, then  $P1(\mathbf{Q}^{(3)})$  holds.*

**Theorem 5.** *If a  $(\mathbf{Q}^{(3)}, \mathbf{B})$ -consensus algorithm  $A$  is  $(1, \mathbf{Q}^{(1)})$ -fast, then  $P2(\mathbf{Q}^{(1)}, \mathbf{Q}^{(3)})$  holds.*

**Theorem 6.** *If a  $(\mathbf{Q}^{(3)}, \mathbf{B})$ -consensus algorithm  $A$  is both  $(1, \mathbf{Q}^{(1)})$ -fast (for some  $\mathbf{Q}^{(1)} \neq \emptyset$ ) and  $(2, \mathbf{Q}^{(2)})$ -fast, then  $P3(\mathbf{Q}^{(1)}, \mathbf{Q}^{(2)}, \mathbf{Q}^{(3)})$  holds.*

Theorem 4 is not new; it follows directly from [29]. Moreover, in the special threshold case, where (a)  $\mathbf{B} = \mathbf{B}_k$ , (b) all elements of  $\mathbf{Q}^{(1)}$  (resp.,  $\mathbf{Q}^{(3)}$ ) contain all but at most  $q$  (resp.,  $t$ ) acceptors, and (c)  $q = t - 2k$ , Theorems 4–5 correspond to the lower bounds identified in [35].

In the following, we prove Theorem 6. To strengthen the optimality result established by Theorem 6 we assume that proposers and learners may not be Byzantine, yet that any number of proposers and learners may fail by crashing.

*Proof. Preliminaries.* To precisely prove Theorem 6, we assume full information protocols in the *round-by-round* eventually synchronous model [15, 30]. The assumption of a full information protocol is indeed without loss of generality, since if, in some particular algorithm  $A$ , a process  $p$  does not send a message to the process  $q$  in round  $rnd$ , we model this by having process  $p$  send to  $q$  a default message  $msg_{nil}$  in  $rnd$  and  $q$  does not change its state upon reception of a message  $msg_{nil}$ . Moreover, denote by  $m_{rnd}^j[i]$  the message sent by process  $j$  to process  $i$  in round  $rnd$  of some execution. For presentation simplicity we assume that, in each round, every process combines all the messages  $m_{rnd}^j[i]$  it is about to send in  $rnd$  and sends the same message  $m_{rnd}^j$  to all processes, such that every process  $i$  (including Byzantine ones) ignores all the portions of the message except  $m_{rnd}^j[i]$  (it is not difficult to see that this is indeed without loss of generality).

Assume, by contradiction, that there is a  $(\mathbf{Q}^{(3)}, \mathbf{B})$ -consensus that is both  $(1, \mathbf{Q}^{(1)})$ -fast (for some  $\mathbf{Q}^{(1)} \neq \emptyset$ ) and  $(2, \mathbf{Q}^{(2)})$ -fast such that  $P3(\mathbf{Q}^{(1)}, \mathbf{Q}^{(2)}, \mathbf{Q}^{(3)})$  is violated, i.e.:

$$\exists Q_1 \in \mathbf{Q}^{(1)}, \exists Q_2 \in \mathbf{Q}^{(2)}, \exists Q \in \mathbf{Q}^{(3)}, \exists B'_1, B_2 \in \mathbf{B}: (Q_2 \cap Q \setminus B'_1 = B_2) \wedge (Q_1 \cap Q_2 \cap Q \setminus B'_1 = \emptyset).$$

In the following, we denote the set  $Q_1 \cap Q_2 \cap Q$  by  $B_0$  and  $Q_2 \cap Q \cap B'_1$  by  $B_1$ . Having in mind that  $\mathbf{B}$  is an adversary for  $S$ , it is straightforward to see that (i)  $B_0, B_1 \subseteq B'_1$ , (ii)  $B_0, B_1 \in \mathbf{B}$ , and (iii)  $Q_2 \cap Q = B_1 \cup B_2$ . Moreover, since  $B_0 \subseteq B'_1$  and  $B_0 \subseteq Q_2 \cap Q$ , we have  $B_0 \subseteq B_1$ . Furthermore, denote by  $\overline{X}$  the set *acceptors*  $\setminus X$ , where  $X$  is any subset of *acceptors*. Hence,  $Q_2 \cap Q \cap \overline{Q_1} = B_2 \cup (B_1 \setminus B_0)$ .

Denote by  $p_0$  and  $p_1$  two distinct proposers ( $p_0 \neq p_1$ ) (such proposers exist since  $|\text{proposers}| \geq 2$ ). Since there are at least three learners, there exists a learner  $l_1 \in \text{learners} \setminus \{p_0, p_1\}$ . Denote by

<sup>9</sup> We exclude here the special cases where  $|\text{proposers}| = 1$ ,  $|\text{learners}| \leq 2$  or  $\text{acceptors} \cap (\text{proposers} \cup \text{learners}) \neq \emptyset$ . These have to be addressed separately.

---

**at every proposer  $p_j$ :**  
Initialization:  
 $view, initView := 0$ ;  $viewProof, vProof := nil$ ;  $Q, \mathbf{faulty} := \emptyset$   
**propose**( $v$ ) is {  
1: **if** ( $view \neq initView$ ) **then** % consult phase  
2:   send **new\_view**( $view, viewProof$ ) to *acceptors*  
3:   **repeat**  
4:     **wait for** valid acks from some quorum  $Q \in RQS \setminus \mathbf{faulty}$   
5:      $vProof :=$  array of received acks from  $Q$   
6:     ( $v, abort$ ) := **choose**( $v, vProof, Q$ )  
7:     **if**  $abort$  **then**  $\mathbf{faulty} := \mathbf{faulty} \cup \{Q\}$   
8:     **until**  $\neg(abort)$   
9:   send **prepare**( $v, view, vProof, Q$ ) to *acceptors* % update phase  
  
**upon**  $p_j$  is elected  
10: **propose**( $v$ )

---

**at every acceptor  $a_j$ :**  
Initialization:  
 $view_{a_j} := initView$ ;  $Prep_{view}, old, Update_{proof}[*], Update_{view}[*], Update_Q[*], * := \emptyset$ ;  $Prep, Update[*] := nil$   
  
**upon** received **new\_view**( $view, viewProof$ ) from  $p_i$  % consult phase (lines 21-29)  
21: **if** ( $view > view_{a_j}$ ) **and** ( $p_i$  is the leader of  $view$ ) **and** ( $viewProof$  matches  $view$ ) **then**  
22:    $view_{a_j} := view$   
23:    $\forall step \in \{1, 2\}, \forall w : w \in Update_{view}[step] \wedge Update_{proof}[step, w] = \emptyset$  **do**  
24:     send **sign\_req**( $Update[step], w, step$ ) to some quorum in  $Update_Q[step, w]$   
25:   **for** every sent **sign\_req**( $Update[step], w, step$ ) message  
26:     **wait for** acks with a valid signature from some subset of *acceptors*  $T_{step, w}, T_{step, w} \notin \mathbf{B}$   
27:      $Update_{proof}[step, w] :=$  received acks from  $T_{step, w}$   
28:   send **new\_view\_ack**( $view_{a_j}, Prep, Prep_{view}, Update[1..2], Update_{view}[1..2], Update_{proof}[1..2, *], Update_Q[1..2, *]$ ) $_{\sigma_{a_j}}$  to  $p_i$   
  
**upon** received **sign\_req**( $v, w, step$ ) from  $a_i$   
29: **if**  $m = update_{step}(v, w, *) \in old$  **then** send **sign\_ack**( $m$ ) $_{\sigma_{a_j}}$  to  $a_i$   
  
**upon** received  $m = prepare(v, view_{a_j}, vProof, Q)$  from  $p_i$  % update phase (lines 31-38 and 51-60)  
31: **if** ( $w \in Prep_{view} \Rightarrow w < view_{a_j}$ ) **and** ( $view_{a_j} = initView$  **or** ( $p_i$  is leader **and**  $v$  matches **choose**( $v, vProof, Q$ ))) **then**  
32:   **if**  $Prep = v$  **then**  $Prep_{view} := Prep_{view} \cup \{view_{a_j}\}$  **else**  $Prep := v$ ;  $Prep_{view} := \{view_{a_j}\}$   
33:   send  $m_1 = update_1(v, view_{a_j}, \emptyset)$  to *acceptors*  $\cup$  *learners*;  $old := old \cup m_1$   
  
**upon** received  $m = update_{step}(v, view_{a_j}, *)$  from some quorum  $Q$  **and**  $v = Prep$  **and**  $view_{a_j} \in Prep_{view}$  (for  $step \in \{1, 2\}$ )  
34: **if**  $Update[step] = v$  **then**  $Update_{view}[step] := Update_{view}[step] \cup \{view_{a_j}\}$   
35:   **else**  $Update[step] := v$ ;  $Update_{view}[step] := \{view_{a_j}\}$ ;  $Update_Q[step, *] := \emptyset$ ;  $Update_{proof}[step, *] := \emptyset$   
36: **if** ( $Q \notin Update_Q[step, view_{a_j}]$  **and**  $step = 1$ ) **or** ( $Update_Q[step, view_{a_j}] = \emptyset$  **and**  $step = 2$ ) **then**  
37:    $Update_Q[step, view_{a_j}] := Update_Q[step, view_{a_j}] \cup Q$   
38:   send  $m_{step+1} = update_{step+1}(v, view_{a_j}, Q)$  to *acceptors*  $\cup$  *learners*;  $old := old \cup m_{step+1}$   
  
**upon** reception of (decision\_pull) from a process  $q$   
40: **if** decided  $v$  **then** send **decision**( $v$ ) to *acceptors*  $\cup \{q\}$

---

<p><b>at every acceptor and learner <math>x</math>:</b>  <b>upon</b> received the same <math>update_1(v, view, *)</math> from <math>Q_1 \in QC_1</math>  51: <b>if</b> <math>x</math> has not yet decided <b>then</b> <b>decide</b>(<math>v</math>)</p> <p><b>upon</b> received the same <math>update_2(v, view, Q_2)</math> from <math>Q_2 \in QC_2</math>  52: <b>if</b> <math>x</math> has not yet decided <b>then</b> <b>decide</b>(<math>v</math>)</p> <p><b>upon</b> received the same <math>update_3(v, view, *)</math> from <math>Q_3 \in RQS</math>  53: <b>if</b> <math>x</math> has not yet decided <b>then</b> <b>decide</b>(<math>v</math>)</p>	<p><b>at every learner <math>l_j</math>:</b>  <b>upon</b> <math>l_j</math> decides <math>v</math>  60: <b>learn</b>(<math>v</math>)</p>
--	---

---

**at every learner  $l_j$ :**  
**upon**  $l_j$  received **decision**( $v$ ) from some subset of *acceptors*  $T, T \notin \mathbf{B}$   
101: **if**  $l_j$  has not yet learned a value **then** **learn**( $v$ )  
  
**upon** value not learned  
102: **wait** some preset time  
103: **if** value not learned **then** send (decision\_pull) to *acceptors*

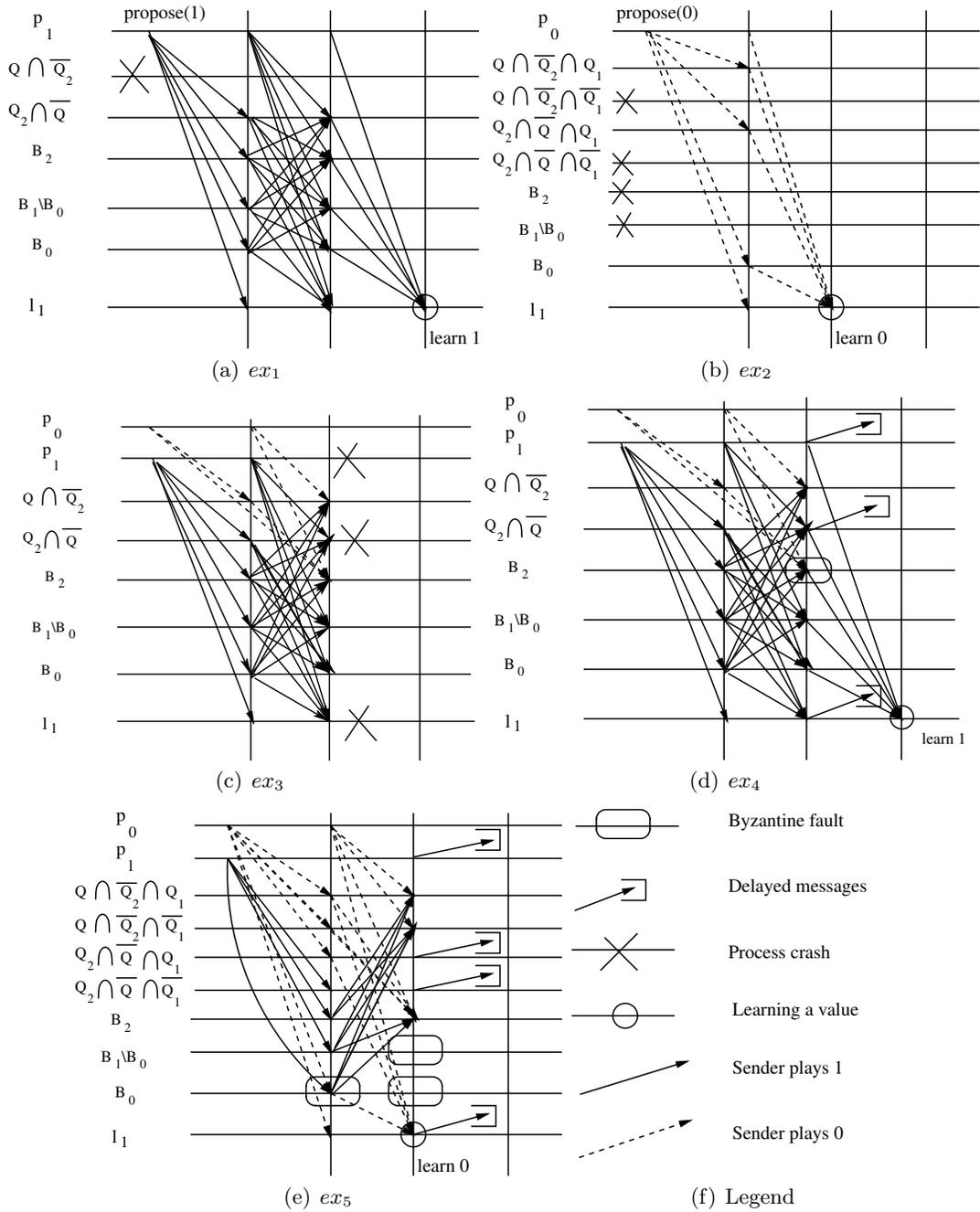
---

$l_2$  some learner different from  $l_1$ . Without loss of generality, we assume that if  $l_2 \in \{p_0, p_1\}$  then  $p_0 = l_2$ . Recall that  $(proposers \cup learners) \cap acceptors = \emptyset$ .

We only consider the cases where  $p_0$  proposes 0 and  $p_1$  proposes 1 (as this is sufficient to prove the theorem). Let  $m0 = m_0^{p_0}$  (resp.,  $m1 = m_1^{p_1}$ ) be the message sent by  $p_0$  (resp.,  $p_1$ ) in round 1 of some best case execution  $ex$ , when  $p_0$  (resp.,  $p_1$ ) is correct and proposes 0 (resp., 1). We say that a process  $a_i$  plays 0 (resp. 1) to some process  $a_j$  in round 2 of  $ex$  if  $a_j$  cannot distinguish, at round 2, execution  $ex$  from some execution  $ex'$  in which (1)  $a_i$  has received  $m0$  (resp.  $m1$ ) from  $p_0$  (resp.,  $p_1$ ) in the first round, and (2)  $a_i$  is correct.

To exhibit a contradiction, we construct several (partial) executions (sketched in Figure 16) of the algorithm  $A$ , including the one in which *agreement* is violated. In these executions, we consider only processes from the set  $acceptors \cup \{p_0, p_1, l_1, l_2\}$ . Other processes can be assumed w.l.o.g. to fail by crashing at the beginning of each of the following executions.

- Let  $ex_1$  be the best-case execution in which all processes fail by crashing at the beginning of  $ex_1$ , except  $p_1, l_1$  and the acceptors from  $Q_2$  that are correct in  $ex_1$  (such an execution is possible since  $p_1$  and  $l_1$  are not acceptors, and, hence,  $p_1$  and  $l_1$  are not in  $\overline{Q_2}$ ). Moreover, a correct proposer  $p_1$  proposes 1 in round 1 at the beginning of  $ex_1$  (we denote this time by  $t_0$ ). Since the system is synchronous in the first three rounds of a best-case execution  $ex_1$  (i.e., during  $[t_0, t_0 + 3\Delta]$ ), all round 1-3 messages exchanged among all correct processes are delivered in  $ex_1$ . By our assumption that  $A$  is  $(2, Q_2)$ -fast,  $l_1$  learns 1 by the end of round 3 (i.e., in three message-delays).
- Let  $ex_2$  be the best case execution in which all processes fail by crashing at the beginning of  $ex_2$ , except  $p_0, l_1$  and acceptors from  $Q_1$  that are correct in  $ex_2$  (such an execution is possible since  $p_l$  and  $l_1$  are not acceptors, and, hence,  $p_l$  and  $l_1$  are not in  $\overline{Q_1}$ ). Moreover, a correct proposer  $p_l$  proposes 0 in round 1 at the beginning of  $ex_2$  (i.e., at time  $t_0$ ). Since the system is synchronous in the first two rounds of a best-case execution  $ex_2$  (during  $[t_0, t_0 + 2\Delta]$ ), all round 1 and 2 messages exchanged among all correct processes are delivered in  $ex_2$ . By our assumption that  $A$  is  $(1, Q_1)$ -fast,  $l_1$  learns 0 by the end of round 2 (i.e., in two message-delays).
- Let  $ex_3$  be a partial execution in which all processes are correct *except* (1) acceptors from  $\overline{Q}$ , (2) learner  $l_1$  and (3) proposer  $p_1$ , which crash in  $ex_3$ . At the beginning of  $ex_3$ , at time  $t_0$ , both proposers  $p_0$  and  $p_1$  propose values 0 and 1, respectively. The messages sent in the first two rounds of  $ex_3$  are delivered as follows:
  - (Round 1 messages.) By the end of round 1: processes from  $Q_2 \cup \{l_1\}$  receive  $m1$ , while processes from  $\overline{Q_2} \cup \{l_2\}$  receive  $m0$ . Moreover, acceptors from  $B_2$  receive the message from  $p_0$  (i.e.,  $m0$ ) in round 2, while those from  $B_1$  receive  $m0$  in round 3. No other process receives  $m1$  (since  $p_1$  crashes in  $ex_3$ ).
  - (Round 2 messages) The following round 2 messages are delivered by the end of round 2: (a) from  $\{p_0, l_2\} \cup Q$  to  $p_0, l_2, Q \cap \overline{Q_2}$  and  $B_2$ , (b) from  $\{p_1, l_1\} \cup Q_2$  to  $p_1, l_1, Q_2 \cap \overline{Q}$  and  $B_1$ . Note that, at the end of round 1, processes from  $Q_2 \cup \{p_1, l_1\}$  (resp.,  $\overline{Q_2} \cap Q_1$ ) cannot distinguish  $ex_3$  from  $ex_1$  (resp.,  $ex_2$ ) and therefore play 1 (resp., 0) to all processes in round 2. Moreover, acceptors from  $B_1$  receive the round 2 messages sent by processes from  $(Q \cap \overline{Q_2}) \cup \{p_0, l_2\}$  in round 3. Finally, no other round 2 message is delivered in  $ex_3$ , (this is possible, since the only remaining round 2 messages are (a subset of) those sent by/to processes from  $\overline{Q} \cup \{p_1, l_1\}$  which fail by crashing in  $ex_3$ ). In particular, note that processes from  $\{p_0, l_2\} \cup B_2 \cup (Q \cap \overline{Q_2})$  never receive any round 2 message sent by processes from  $\{p_1, l_1\} \cup (Q_2 \cap \overline{Q})$ .



**Fig. 16.** Illustration of the partial executions used in the proof of Theorem 6. Only acceptors that belong to the set  $Q_2 \cup Q$  are depicted. For clarity, learner  $l_2$ , as well as messages received by proposers and those sent by learner  $l_1$  are not depicted.

Note that, in  $ex_3$ , by the end of round 2, all processes from  $\{p_0, p_1, l_1, l_2\} \cup Q_2 \cup Q$  receive all the messages sent in the first two rounds by (a) at least one proposer, (b) some quorum of acceptors, and (c) at least one learner. Processes from  $\{p_1, l_1\} \cup (Q_2 \cap \overline{Q}) \cup B_1$  cannot distinguish, at the end of round 2,  $ex_3$  from  $ex_1$ , while processes from  $\{p_0, l_2\} \cup (Q \cap \overline{Q_2}) \cup B_2$  cannot wait for any additional round 1 or round 2 message since these processes received all the messages sent by correct processes in first two rounds. Therefore, in  $ex_3$  no process from  $\{p_0, p_1, l_1, l_2\} \cup Q_2 \cup Q$  waits for any additional message before moving to round 3. At the beginning of round 3 processes from  $\overline{Q} \cup \{p_1, l_1\}$  fail by crashing such that no process receives any message sent by some of these processes in round 3. Furthermore, assume that in every round  $i \geq 3$ , all round  $i$  messages exchanged among correct processes are delivered by the end of round  $i$ . Since (a)  $p_0$  is correct in  $ex_3$  and proposes a value (note that  $p_0$  can be correct in  $ex_3$  since  $p_0 \notin \{p_1, l_1\} \cup \overline{Q}$ ), (b) there is a quorum of correct acceptors  $Q \in \mathbf{Q}^{(3)}$ , (c) the system is eventually synchronous, and (d)  $A$  implements  $(\mathbf{Q}^{(3)}, \mathbf{B})$ -consensus, eventually a correct learner  $l_2$  ( $l_2$  can be correct since,  $l_2 \notin \{p_1, l_1\} \cup \overline{Q}$ ) learns some value  $v \in \{0, 1\}$ , say at round  $K$ .

- Let  $ex_4$  be an execution identical to  $ex_3$ , except that, in  $ex_4$ :
  - Processes from  $\overline{Q} \cup \{p_1, l_1\}$  do not crash at the beginning of round 3, but due to asynchrony, no message sent in rounds 3 to  $K$  by some process from  $\overline{Q} \cup \{p_1, l_1\}$  is delivered before round  $K + 1$ , with the exception of round 3 messages sent by processes from  $\{p_1, l_1\} \cup (Q_2 \cap \overline{Q})$  to learner  $l_1$ .
  - Acceptors from  $B_2$  are Byzantine and in round 3 send the same message as in the round 3 of  $ex_1$  to  $l_1$  (i.e., as if they received round 2 messages (only) from  $Q_2 \cup \{p_1, l_1\}$ ). Otherwise, acceptors from  $B_2$  in rounds 3 to  $K$  respect the algorithm  $A$  and send the same messages as in  $ex_3$ .
  - In round 3 of  $ex_4$ , the messages sent by  $\{l_1, p_l\} \cup Q_2$  (including those from  $B_2$ ) are delivered to  $l_1$ , other messages sent to  $l_1$  are delayed until after round  $K$ . Since benign processes  $\{l_1, p_l\} \cup Q_2$  do not distinguish round 2 of  $ex_3$  (and, hence,  $ex_4$ ) from round 2 of  $ex_1$ , they send the same messages in round 3 of  $ex_4$  as in round 3 of  $ex_1$ .

Hence, at the end of round 3,  $l_1$  cannot distinguish  $ex_4$  from  $ex_1$  and learns 1. Other round 3 and later messages in  $ex_4$  are delivered as in  $ex_3$ . Since the proposer  $p_0$  is correct in  $ex_4$ , (note that  $p_0 \notin \{p_1, l_1\} \cup B_2 \cup \overline{Q}$ ) a correct learner  $l_2$  ( $l_2 \notin \{p_1, l_1\} \cup B_2 \cup \overline{Q}$ ) cannot distinguish  $ex_4$  and  $ex_3$ ,  $l_2$  learns a value  $v$  by the end of round  $K$ . Since both  $l_1$  and  $l_2$  are correct in  $ex_4$ , by the *Agreement* property,  $v$  must equal 1.

- Let  $ex_5$  be an execution in which all processes are correct except the acceptors from  $B_1$  that are Byzantine. At the beginning of  $ex_5$ ,  $p_0$  proposes 0, while  $p_1$  proposes 1. In round 1 of  $ex_5$  the message are delivered exactly as in round 1 of  $ex_3$ , except that (1) benign acceptors from  $Q_1$  (including those from  $Q_2 \cap \overline{Q} \cap Q_1$ ) receive  $m0$ , but not  $m1$  and (2) acceptors from  $B_0$  receive both  $m1$  and  $m0$ . In round 2 and later rounds,  $B_0$  plays 1 to processes other than  $l_1$ . Moreover all acceptors from  $Q_1$  (including those from  $B_0$ ) play 0 to  $l_1$  in round 2 (here, benign acceptors from  $Q_1$  obey the algorithm — they cannot distinguish round 1 in  $ex_5$  from that of  $ex_2$ ) and all such messages are delivered in round 2. Moreover, all other round 2 messages sent to  $l_1$  are delivered in round 3. Clearly, at the end of round 2,  $l_1$  cannot distinguish  $ex_5$  from  $ex_2$  (since  $l_1$  receives the same set of messages from the acceptors in round 2, and the same message from  $p_0$  in rounds 1 and 2 in both executions), and hence, learns 0. All other round 2 messages are delivered as in  $ex_3$ . Hence, just like in  $ex_3$ , in  $ex_5$  no process from  $\{p_0, l_2\} \cup (Q \cap \overline{Q_2}) \cup B_2$  receives any round 2 message from any process from  $\{p_1, l_1\} \cup (Q_2 \cap \overline{Q})$ . Starting from round 3 acceptors from  $B_1$  forge their state as if they received the round 2 messages as in  $ex_3$  and  $ex_1$ ,

i.e., as if all the processes from  $Q_2 \cup \{p_1, l_1\}$  played 1 to acceptors from  $B_1$  in round 2 of  $ex_5$  (which is actually the case, except for the processes from  $(Q_2 \cap \overline{Q} \cap Q_1) \cup \{l_1\}$ ), and acceptors from  $B_1$  received these messages (this is possible since the messages sent in the round 2 of the best-case execution  $ex_1$  are not authenticated). Moreover, no message sent in rounds 3 to  $K$  by some process from  $\overline{Q} \cup \{p_1, l_1\}$  is delivered before round  $K + 1$ . All other messages in round 3 and later are delivered as in  $ex_3$ . Since a correct learner  $l_2$  cannot distinguish  $ex_5$  and  $ex_3$  (note that  $p_0, l_2 \notin \{p_1, l_1\} \cup B_1 \cup \overline{Q}$ ),  $l_2$  learns a value  $v$  by the end of round  $K$ . Since  $v$  equals 1 (see  $ex_4$ ), and both  $l_1$  and  $l_2$  are correct, in  $ex_5$  *agreement* is violated.  $\square$

## 5 Related Work

In this section, we compare the techniques used in our atomic storage and consensus algorithms to existing algorithms; thus, we complement the comparison of our paper and RQS to previous work that we gave in Section 2.2.

Our atomic storage algorithm (Section 3.2) is inspired by techniques used in the regular [33] finite write (FW) terminating<sup>10</sup> storage algorithm of [2] and the wait-free atomic storage algorithm of [20]. Both of these algorithms are optimally resilient, like our algorithm, in terms of tolerating Byzantine servers (for tolerating Byzantine clients see, e.g., [3, 18, 23]). Unlike these (or other existing) algorithms, our atomic wait-free storage algorithm is the first to combine optimal resilience and best-case optimal latency in the non-threshold failure model.

However, our storage algorithm (deliberately) features unbounded worst case message-complexity and uses unbounded storage at servers (i.e., a server stores an entire history of a shared variable). Hence, it may seem that RQS have an undesirable side-effect of ruining other complexity metrics while optimizing the best-case time-complexity. However, achieving atomic semantics (and even a weaker, regular one), in a wait-free manner while precluding servers from storing the entire history, is unfeasible without using some non-trivial signaling scheme between the readers and the writer [2, 9]. Such schemes include, for example, the “Listeners” pattern of [42] (in which, roughly, concurrent readers subscribe at servers for updates on concurrent writes and, hence, not applicable to our storage model), the “freezing” technique of [20] (applicable to our model), bounded implementation techniques of [3], as well as techniques used in [5]. Similarly, optimizing Byzantine fault-tolerant storage (worst-case) complexity is not trivial [3, 21]. We opt not to address these issues in this paper since we believe that doing so would not contribute to better understanding of RQS. Our algorithms serve to illustrate how RQS can be used to build algorithms with optimal best case time complexity — they do not aim to optimize other complexity metrics. Integrating the above techniques here would not contribute to deeper understanding of RQS and might obfuscate the point of this paper.

As an illustration of why using RQS does not require servers to store an entire history of a shared variable, nor imposes unbounded message complexity, recall the simple variation of [4], that we described in our example in Section 1.2). The mentioned algorithm implements optimally (crash) resilient wait-free atomic storage. Moreover, the algorithm makes use of RQS; in the particular case of a system consisting of 5 servers, class 1 quorums are all subsets of 4 or more servers, whereas subsets containing 3 servers are class 2 quorums. Finally, all servers store 2 copies of the shared variable, and reads and writes complete in at most 2 rounds.

<sup>10</sup> In FW-terminating implementations, reads might not terminate in case there is an unbounded number of writes.

Our consensus algorithm (Section 4.2) is itself inspired by the PBFT algorithm [7], from which it borrows the idea of view-change mechanisms (with significant differences in choosing the proposal value during a view-change). Namely, like PBFT, our algorithm proceeds in sequence of “views”, which can be mapped to “ballots” in the Paxos algorithm [38]. Unlike PBFT, our algorithm does not implement state-machine replication, but rather a single-shot consensus instance.

Finally, proofs of our storage and consensus algorithms (Appendices A and B), rely extensively on the notion of a *basic subset* (intuited in Section 3.2) which simply denotes a set of servers not belonging to adversary  $\mathbf{B}$ . Basic subsets are similar to *cores* defined by Junqueira and Marzullo [28,29]. They define cores (in broader context of a framework for tolerating dependent failures) as minimal sets of processes such that at least one process is correct in every execution. In contrast, a basic subset is *any* set of processes such that at least one process is *benign* (i.e., correct or crash-faulty) in every execution. Hence, all cores are basic subsets, yet not all basic subsets are cores.

## 6 Concluding Remarks

This paper introduces the notion of *refined quorum systems* (RQS) and argues that this is a useful notion to reason about optimally resilient and efficient distributed object implementations assuming general adversary structures. We show that refined quorum systems are necessary and sufficient (or, in a sense, minimal) for implementing an important class of *atomic* objects, namely atomic storage and consensus. This minimality holds when we indeed require atomicity and do not rely on authentication primitives to cope with Byzantine failures in best-case executions.

Roughly speaking, denoting the best possible latency of an object implementation by  $l_1$ <sup>11</sup> (i.e., 1 round in the case of storage, or 2 message delays in the case of (Byzantine and asynchronous [35]) consensus, and by  $l_2$  and  $l_3$ , incrementally, the next best possible latencies according to the corresponding metric, we proposed two RQS-based object implementations that achieve a latency of  $l_i$  whenever a quorum of class  $i$  is available and best-case conditions (namely, synchrony and no-contention) are met. Since Property 1 of RQS (defined on class 3 quorums) is anyway necessary for any resilient implementation of distributed storage and consensus in an asynchronous environment, there is no need for refining quorums further.

It might be important to notice here that the very notion of a refined quorum system helps highlight the information structure of optimally resilient and best-case efficient atomic object implementations (at least those implementing the abstractions of atomic storage or consensus). Basically, these implementations go through at most three “rounds” in best-case conditions and fall into a backup subprotocol in case of asynchrony or contention. A novel algorithmic scheme we used in both algorithms consists of appending the ids of (class 2) quorums, to written/proposed values. This is key to combining graceful degradation (i.e., achieving both latencies  $l_1$  and  $l_2$ ) with optimal resilience.

Our study opens several research directions. For example, it is intriguing to determine:

- the load and availability of RQS [44],
- how RQS can be optimally placed in the network [17],
- the extension of RQS with respect to asymmetric read and write quorums [43],
- how many RQS can be found given some adversary structure,

---

<sup>11</sup> This can be measured by the best possible latency in synchronous, uncontended and failure-free situations.

- how to devise algorithms that cope with unknown RQS/adversary structures, and
- how RQS can be expressed in frameworks for tolerating non-independent and identically distributed (non-IID) failures, other than general adversary structures (in particular, in the core/survivor framework of [29]).

Moreover, it would be interesting to carefully look into non-atomic semantics, e.g., regular or safe storage [33]. Recent results (in the threshold-based context) suggest that some (yet not all) properties of our RQS are necessary and sufficient even for achieving optimal best-case complexity of weaker object implementations. Namely [2, 21] suggest that Properties 1 and 3a of RQS are necessary and sufficient for characterizing non-atomic best-case efficient storage implementations. These properties correspond to the special case of RQS where  $QC_1 = \emptyset$ . Finally, it would also be interesting to look into atomic object implementations that use authentication in best-case executions. The lower bounds of [35], stated in the threshold-based context, suggest that Properties 1 and 2 are necessary and sufficient for characterizing best-case efficient and optimally resilient consensus implementations regardless of whether authentication is used in the best-case. These properties correspond to the special case of RQS where  $QC_2 = QC_1$ .

Finally, we note that the preliminary, conference version of this paper [22], was erroneous, notably in the way it stated Property 3 of RQS (please see Appendix C for more details).

## Acknowledgments

We are very grateful to anonymous Distributed Computing journal reviewers for pointing out a mistake in the conference version of this paper. Their very detailed comments significantly improved the paper. We also thank Hagit Attiya for her detailed comments on earlier drafts of this paper. Finally, we thank Christian Cachin, Christof Fetzer, Petr Kouznetsov, Ron R. Levy, Dahlia Malkhi, Eric Ruppert for their helpful comments and useful feedback.

## 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. In *Proceedings of the 20th ACM symposium on Operating systems principles*, pages 59–74, October 2005.
2. Ittai Abraham, Gregory V. Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.
3. Amitanand S. Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. Bounded wait-free implementation of optimally resilient byzantine storage without (unproven) cryptographic assumptions. In *Proceedings of the 21st International Symposium on Distributed Computing*, pages 7–19, September 2007.
4. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.
5. Rida Bazzi and Yin Ding. Non-skipping timestamps for Byzantine data storage systems. In *Proceedings of the 18th International Symposium on Distributed Computing*, volume 3274/2004 of *Lecture Notes in Computer Science*, pages 405–419, Oct 2004.
6. John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. UMAC: Fast and secure message authentication. *Lecture Notes in Computer Science*, 1666:216–233, 1999.
7. Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, February 1999.
8. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
9. Gregory Chockler, Rachid Guerraoui, and Idit Keidar. Amnesic distributed storage. In *Proceedings of the 21st International Symposium on Distributed Computing*, pages 139–151, September 2007.

10. James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementations*, November 2006.
11. Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the 23rd annual ACM symposium on Principles of distributed computing*, pages 236–245, July 2004.
12. Partha Dutta, Rachid Guerraoui, and Marko Vukolić. Best-case complexity of asynchronous Byzantine consensus. Technical Report 200499, Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland, 2005.
13. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.
14. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
15. Eli Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the 17th annual ACM symposium on Principles of distributed computing*, pages 143–152, June 1998.
16. David K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM symposium on Operating systems principles*, pages 150–162, December 1979.
17. Daniel Golovin, Anupam Gupta, Bruce M. Maggs, Florian Oprea, and Michael K. Reiter. Quorum placement in networks: Minimizing network congestion. In *Proceedings of the 15th annual ACM symposium on Principles of distributed computing*, pages 16–25, July 2006.
18. Garth Goodson, Jay Wylie, Gregory Ganger, and Michael Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 135–144, 2004.
19. Rachid Guerraoui. Indulgent algorithms (preliminary version). In *Proceedings of the 19th annual ACM symposium on Principles of distributed computing*, pages 289–297, July 2000.
20. Rachid Guerraoui, Ron R. Levy, and Marko Vukolić. Lucky read/write access to robust atomic storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 125–136, June 2006. The full version of this paper is available as a EPFL/LPD technical report (LPD-REPORT-2005-005) with the same title.
21. Rachid Guerraoui and Marko Vukolić. How Fast Can a Very Robust Read Be? In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing*, pages 248–257, July 2006.
22. Rachid Guerraoui and Marko Vukolić. Refined quorum systems. In *Proceedings of the 26th annual ACM symposium on Principles of distributed computing*, pages 119–128, New York, NY, USA, 2007. ACM.
23. James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Low-overhead byzantine fault-tolerant storage. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 73–86, New York, NY, USA, 2007. ACM.
24. Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
25. Maurice Herlihy and Jeannette Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.
26. Martin Hirt and Ueli Maurer. Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In *Proceedings of the 16th annual ACM symposium on Principles of distributed computing*, pages 25–34, 1997.
27. Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, 1998.
28. Flavio Junqueira and Keith Marzullo. A framework for the design of dependent-failure algorithms: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(17):2255–2269, 2007.
29. Flavio P. Junqueira and Keith Marzullo. Synchronous consensus for dependent process failures. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, pages 274–283, May 2003.
30. Idit Keidar and Alexander Shraer. Timeliness, failure-detectors, and consensus performance. In *Proceedings of the 25th annual ACM symposium on Principles of distributed computing*, pages 169–178, 2006.
31. Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles*, pages 45–58, New York, NY, USA, 2007. ACM.
32. Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

33. Leslie Lamport. On interprocess communication. *Distributed computing*, 1(1):77–101, May 1986.
34. Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.
35. Leslie Lamport. Lower bounds for asynchronous consensus. In *Future Directions in Distributed Computing*, Springer Verlag (LNCS), pages 22–23, 2003.
36. Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.
37. Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
38. Butler Lampson. The ABCD’s of paxos. In *Proceedings of the 20th annual ACM symposium on Principles of distributed computing*, page 13, New York, NY, USA, 2001. ACM.
39. Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.
40. Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.
41. Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.
42. Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal Byzantine storage. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 311–325, October 2002.
43. Jean-Phillipe Martin, Lorenzo Alvisi, and Michael Dahlin. Small Byzantine quorum systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 374–383, June 2002.
44. Moni Naor and Avishai Wool. The load, capacity and availability of quorum systems. In *Proceedings of the 35th IEEE Symposium on Foundations of Computer Science*, pages 214–225, 1994.
45. Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreements in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.
46. HariGovind V. Ramasamy and Christian Cachin. Parsimonious asynchronous Byzantine-fault-tolerant atomic broadcast. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, pages 88–102, December 2005.
47. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
48. Yasushi Saito, Svend Frolund, Alistair Veitch, Arif Merchant, and Susan Spence. Fab: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev.*, 38(5):48–58, 2004.
49. Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *Proceedings of the seventh symposium on Reliable distributed systems*, pages 93–100. IEEE Computer Society Press, 1988.
50. Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.
51. Marko Vukolić. *Abstractions for asynchronous distributed computing with malicious players*. PhD thesis no. 4241, Ecole polytechnique Fédérale de Lausanne (EPFL), 2008.
52. Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM symposium on Operating systems principles*, pages 253–267, 2003.
53. Piotr Zieliński. Optimistically terminating consensus. Technical Report UCAM-CL-TR-668, Cambridge University, Cambridge, UK, June 2006.

## A Correctness of the atomic storage algorithm

In this Appendix, we prove the correctness of our atomic storage algorithm of Section 3.2. For simplicity of presentation, we introduce the following notation and definitions:

- We say that the writer *attaches* timestamp  $ts$  to value  $val$ , if the writer invokes  $\text{write}(val)$  and  $ts$  equals the writer’s local variable  $ts$  after the writer executes line 1, Fig. 5 in  $\text{write}(val)$ ;
- We say that a (timestamp/value) pair  $c = \langle ts, val \rangle$  is *valid*, if the writer attached  $ts$  to  $val$ , or if  $c = \langle 0, \perp \rangle$  (otherwise,  $c$  is called *invalid*).
- If some reader  $r$  executes line 35, Fig. 7, and assigns some pair  $c = \langle ts, val \rangle$  to  $c_{sel}$ , we say  $r$  *selects* pair  $c$ ;
- We say that a server *responds to*, or *acks* a  $\text{wr}\langle ts, *, *, rnd \rangle$  (resp.,  $\text{rd}\langle tsr, rnd \rangle$ ) message from client, if a server sends a  $\text{wr\_ack}\langle ts, rnd \rangle$  (resp.,  $\text{rd\_ack}\langle tsr, rnd, * \rangle$ ) to the client.
- We say that a benign server  $s_i$  *stores* pair  $c = \langle c.ts, c.val \rangle$  (in slot  $rnd \in \{1, 2, 3\}$ ), if, at some point in time,  $\text{history}_i[c.ts, rnd].\text{pair} = c$ . If, additionally,  $Q \in \text{history}_i[c.ts, rnd].\text{sets}$  for some quorum  $Q$ , we say that  $s_i$  stores  $c$  (in slot  $rnd$ ) *with quorum*  $Q$ ;
- We denote by  $B_{ex}$  the set that contains all Byzantine servers in some execution  $ex$  (we assume  $B_{ex} \in \mathbf{B}$ ).

**Definition 5.** Consider a set of elements  $S$  and an adversary for  $S$ ,  $\mathbf{B}$ . We say that  $Q \subseteq S$  is a basic (resp., large) subset (of  $S$ ), if  $Q$  is not a subset of any element (resp., a union of any two elements) of an adversary structure, i.e.,  $Q \notin \mathbf{B}$  (resp.,  $\forall B_1, B_2 \in \mathbf{B}: Q \not\subseteq (B_1 \cup B_2)$ ).

We first prove atomicity and then we proceed to wait-freedom and complexity. To prove atomicity, we first prove few simple lemmas.

**Lemma 1. Size of basic sets.** In every execution of our storage algorithm, any basic subset of servers contains at least one benign server.

*Proof.* The lemma follows directly from the definition of a basic subset (Definition 5). □

**Lemma 2. Size of large sets.** In any execution  $ex$  of our storage algorithm, for any large subset  $T_2$  of servers, there is a basic subset  $T_1 \subseteq T_2$  that contains only benign servers.

*Proof.* By Definition 5, for any large subset  $T_2$ ,  $T_2 \setminus B_{ex}$  is a basic subset. By definition of  $B_{ex}$ ,  $T_2 \setminus B_{ex}$  contains only benign servers in  $ex$ . Hence the lemma. □

The following two lemmas follow directly from our assumption of benign readers and by trivial inspection of the read pseudocode in Fig. 7.

**Lemma 3. Returned values.** If read  $rd$  by reader  $r$  returns value  $val$ , then  $r$  selected pair  $c = \langle ts, val \rangle$ , for some timestamp  $ts$ .

**Lemma 4. Values written by readers.** If some reader  $r$  sends a  $\text{wr}\langle ts, v, *, * \rangle$  to servers, then  $r$  selected  $\langle ts, v \rangle$ .

**Lemma 5. Validity of selected pairs.** If reader  $r$  selects pair  $c = \langle c.ts, c.val \rangle$  in some read, then  $c$  is valid.

*Proof.* Suppose by contradiction that some read selects pair  $c$ , such that  $c$  is invalid. Let  $rd$  be the first read (according to the global clock) to select an invalid pair  $c$  at time  $t$ . Therefore, up to time  $t$ , by Lemma 4, readers send only  $wr$  messages containing valid pairs and benign servers store only valid pairs.

Since  $r$  selects  $c$ , predicate  $safe(c)$  holds in  $rd$ , i.e., servers from a *basic* subset  $T$  have sent a  $rd\_ack$  message containing  $c$  in their  $history[c.ts, 1]$  or  $history[c.ts, 2]$  variables. By Lemma 1 at least one benign server  $s_i \in T$  stored an invalid pair  $c$  before time  $t$ . A contradiction.  $\square$

**Lemma 6. *Validity of stored pairs.*** *Benign servers store only valid pairs.*

*Proof.* Follows from Lemmas 4 and 5.  $\square$

**Lemma 7. *No ambiguity.*** *No two benign servers ever store different pairs with the same timestamp.*

*Proof.* By Lemma 6, the assumption that the writer is benign and the fact that the writer never attaches different timestamps to the same value.  $\square$

**Lemma 8. *Sticky values.*** *For any  $rnd \in \{1, 2, 3\}$  and benign server  $s_i$ , once  $history_i[ts, rnd] \neq \langle 0, \perp \rangle$  it is never modified.*

*Proof.* Immediate from the condition in line 4 of the server pseudocode (Fig. 6).  $\square$

**Lemma 9. *Sticky sets.*** *For any  $rnd \in \{1, 2, 3\}$  and benign server  $s_i$ , once  $s_i$  adds the id of some quorum  $Q$  to  $history_i[ts, rnd].sets$ ,  $s_i$  never removes  $Q$  from  $history_i[ts, rnd].sets$ .*

*Proof.* Trivially, by inspection of the server code (Fig. 6).  $\square$

Before proceeding with the proof, we define the following three global predicates, extensively used in the remainder of the proof.

1.  $V_1(c, Q)$  holds if and only if there is a basic subset  $T_Q \subseteq Q$  that contains *only* benign servers, such that every server  $s_i \in T_Q$  stores  $c$  in slot 1 (i.e.,  $\forall s_i \in T_Q : history_i[c.ts, 1].pair = c$ )
2.  $V_2(c, Q)$  holds if and only if there is a benign server  $s_i \in Q$  that stores  $c$  in slot 2 (i.e.,  $history_i[c.ts, 2].pair = c$ );
3.  $V_3(c, Q)$  holds if and only if there is a class 2 quorum  $Q_2$  and a set  $B$  that belongs to the adversary structure  $\mathbf{B}$ , such that  $P_{3b}(Q_2, Q, B)$  holds and every server  $s_i$  from  $Q_2 \cap Q \setminus B$  is benign and  $s_i$  stores  $c$  in slot 1 with quorum  $Q_2$  (i.e.,  $\forall s_i \in Q_2 \cap Q \setminus B : history_i[c.ts, 1] = \langle c, \mathbf{Set}_i \rangle \wedge Q_2 \in \mathbf{Set}_i$ ).

The following lemma relates the global predicate  $V_j(c, Q)$  to read predicate  $valid_j(c, Q)$  (for  $j \in \{1, 2, 3\}$ ).

**Lemma 10.** *Assume  $V_j(c, Q)$  holds at time  $t$  in some execution  $ex$ . Then in read  $rd$  invoked after  $t$ ,  $Q \in \mathbf{Responded}$  implies  $valid_j(c, Q)$ , for any  $j \in \{1, 2, 3\}$ .*

*Proof.* By Lemmas 8 and 9 once the predicate  $V_j(c, Q)$  (for  $j \in \{1, 2, 3\}$ ) becomes true in some execution  $ex$ , it remains always true in  $ex$ . Then, since  $Q \in \mathbf{Responded}$ , all benign servers from  $Q$  responded to at least one  $rd$  message in  $rd$  (lines 52-53, Fig. 7). Finally, since  $rd$  is invoked in  $ex$  after  $V_j(c, Q)$  becomes true in  $ex$ , it is straightforward to see that in  $rd$   $valid_j(c, Q)$  holds (for any  $j \in \{1, 2, 3\}$ ).  $\square$

The following lemma is crucial in proving atomicity.

**Lemma 11. *Locking the value.*** *If for every quorum  $Q$ , at least one of properties  $V_j(c, Q)$  holds by time  $t$  (for some  $j \in \{1, 2, 3\}$ ), for some pair  $c$ , then reader  $r$  cannot select pair  $c'$  in a complete read  $rd$  invoked after  $t$ , such that  $c'.ts < c.ts$ .*

*Proof.* Let  $t_1$  be the time when the first round of  $rd$  completes; by lines 26 and 52-53 of Fig. 7, for  $t \geq t_1$ ,  $\mathbf{Responded}$  contains at least one quorum  $Q_r$ . By reader pseudocode, reader  $r$  cannot select any pair in  $rd$  before  $t_1$ . Fix  $j \in \{1, 2, 3\}$ , such that  $V_j(c, Q_r)$  holds before  $rd$  is invoked.

Assume by contradiction that  $r$  can select  $c'$  in  $rd$ . Then,  $highCand(c')$  holds (line 9, Fig. 7) in  $rd$  after  $t_1$ . Since,  $highCand(c')$ , at least one of the following must hold: (a) after  $t_1$ , there is no server  $s_i$ , such that  $read(c, i)$  holds in  $rd$ , or (b)  $invalid(c)$  holds in  $rd$  after  $t_1$ . However, since  $Q_r \in \mathbf{Responded}$  (by time  $t_1$ ) and since  $V_j(c, Q_r)$  holds before  $rd$  is invoked, by Lemma 10,  $valid_j(c, Q_r)$  holds in  $rd$  after  $t_1$ . Moreover, by definition of  $highest.ts$  (line 29, Fig. 7), in  $rd$ ,  $highest.ts \geq c.ts$ . Therefore,  $invalid(c)$  cannot hold in  $rd$  after  $t_1$ . Moreover,  $valid_j(c, Q_r)$  trivially implies  $read(c, i)$  for some  $s_i \in Q_r$ . A contradiction.  $\square$

Now we prove atomicity of read operations with respect to write operations.

**Lemma 12. *read/write atomicity.*** *If read  $rd$  is complete and it follows some complete  $wr = write(v)$ , then  $rd$  does not return a value older than  $v$ .*

*Proof.* Having Lemma 3 in mind, it is sufficient to show that the reader in  $rd$  does not select  $c$ , such that  $c.ts < ts$ , where  $ts$  is the timestamp attached by the writer to  $v$ .

First, suppose that  $wr$  completes in a single round. Then, all benign servers of some class 1 quorum  $Q_1$  store pair  $\langle ts, v \rangle$  in slot 1. By Property 2 of RQS, and Definition 5, for every quorum  $Q$ ,  $Q_1 \cap Q$  is a large subset, and every large subset is a superset of a basic subset that contains only benign servers (by Lemma 2). Therefore, for every quorum  $Q$  there is a basic subset  $T_Q$  that contains only benign servers such that  $T_Q \subseteq Q$  and  $T_Q \subseteq Q_1$ . Hence, for every quorum  $Q$  property  $V_1(\langle ts, v \rangle, Q)$  holds by the end of  $wr$ . Finally, by Lemma 11,  $rd$  does not select  $c$ , such that  $c.ts < ts$ .

Now, suppose that  $wr$  completes in two or three rounds. Then, all benign servers of some quorum  $Q'$  store pair  $\langle ts, v \rangle$  in slot 2. Since any quorum intersects with any other quorum  $Q$  in a basic subset (by Property 1 of RQS, and Definition 5),  $Q' \cap Q$  is a basic subset that contains at least one benign server (by Lemma 1). Therefore, for every quorum  $Q$  there is a benign server  $s_{i_Q}$  such that  $s_{i_Q} \in Q$  and  $s_{i_Q} \in Q'$ . Hence, for every quorum  $Q$  property  $V_2(\langle ts, v \rangle, Q)$  holds by the end of  $wr$ . Finally, by Lemma 11,  $rd$  does not select  $c$ , such that  $c.ts < ts$ .

Now we proceed to proving atomicity of read operations. First, we prove one auxiliary lemma.

**Lemma 13. *Previous slots.*** *If a benign server stores  $c$  in slot 2 with class 2 quorum  $Q_2$ , then all benign servers from  $Q_2$  already stored  $c$  in slot 1. Similarly, if a benign server stores  $c$  in slot 3, then all benign servers from some quorum already stored  $c$  in slot 2.*

*Proof.* To prove the first part of the lemma, notice that, by code inspection, (a) benign servers can store  $c$  in slot 2 with  $Q_2$  only upon receiving a  $wr\langle c.ts, c.val, \mathbf{Set}, 2 \rangle$  message with  $Q_2 \in \mathbf{Set}$ , and (b) only the writer can send a  $wr\langle *, *, \mathbf{Set}, 2 \rangle$  message with non-empty  $\mathbf{Set}$ . Moreover, the writer sends such a message with  $Q_2 \in \mathbf{Set}$ , only in the second round of  $write(c.val)$  after the writer receives responses from all servers from  $Q_2$  in round 1 of  $write(c.val)$ . Hence, by the end of round 1 of  $write(c.val)$ , all benign servers from  $Q_2$  store  $c$  in slot 1.

The proof of the second part of the lemma closely follows that of the first part: (a) benign servers can store  $c$  in slot 3 with  $Q_2$  only upon receiving a  $wr\langle c.ts, c.val, *, 3 \rangle$  message, and (b) only the writer can send such a message and only in the third round of  $write(c.val)$  after the writer receives responses from all servers from some quorum in round 2 of  $write(c.val)$ . Hence, by the end of round 2 of  $write(c.val)$ , all benign servers from some quorum store  $c$  in slot 2.  $\square$

**Lemma 14. read atomicity.** *If read  $rd$  is complete and it follows some complete read  $rd'$  that returns  $v'$ , then  $rd$  does not return a value older than  $v'$ .*

*Proof.* Having Lemma 3 in mind, it is sufficient to show that the reader in  $rd$  does not select  $c$ , such that  $c.ts < ts'$ , where  $c' = \langle ts', v' \rangle$  was selected in  $rd'$ . Moreover, by Lemma 11, it is sufficient to show that, in any execution  $ex$  and for every quorum  $Q$ , at least one of the properties  $V_j(c', Q)$  (for  $j \in \{1, 2, 3\}$ ) holds by the time  $rd'$  completes.

We consider three exhaustive cases, where : (1)  $rd'$  completes in a single round, (2)  $rd'$  completes in 2 rounds, and (3)  $rd'$  completes in at least 3 rounds.

1. If  $rd'$  completes in a single round, then, at the end of the first round of  $rd'$ ,  $BCD(c', 1, rnd)$  holds, for some  $rnd \in \{1, 2, 3\}$ . We consider the following three exhaustive cases where: (a)  $BCD(c', 1, 1)$  holds, (b)  $BCD(c', 1, 2)$  holds, and (c)  $BCD(c', 1, 3)$  holds.
  - (a) In case  $BCD(c', 1, 1)$  holds in  $rd'$ , there are class 1 quorums  $Q_1$  and  $Q'_1$  (possibly  $Q_1 = Q'_1$ ), such that, by the end of round 1 of  $rd'$ , for all benign servers  $s_i \in Q_1 \cap Q'_1 = X$ ,  $history[i, ts', 1].pair = c'$  in  $rd'$ . Hence, by the end of round 1 of  $rd'$ , all benign servers  $s_i \in X$  stored  $c'$  in slot 1. By Property 2 of RQS and Definition 5, an intersection of a pair of class 1 quorums with any quorum  $Q$  is a large subset. Hence,  $X \cap Q$  is a large subset, and, by Lemma 2, every large subset is a superset of a basic subset that contains only benign servers. Therefore, for every quorum  $Q$  there is a basic subset  $T_Q$  that contains only benign servers such that  $T_Q \subseteq Q$  and  $T_Q \subseteq X$ . Hence, for every quorum  $Q$  property  $V_1(c', Q)$  holds by the time  $rd'$  completes.
  - (b) In case  $BCD(c', 1, 2)$  holds in  $rd'$ , there is class 1 quorum  $Q_1$  and class 2 quorum  $Q_2$ , such that, by the end of round 1 of  $rd'$ , for all benign servers  $s_i \in Q_1 \cap Q_2 = X$ ,  $history[i, ts', 2] = \langle c', QC''_2 \rangle$  and  $Q_2 \in QC''_2$  in  $rd'$ . Hence, by the end of round 1 of  $rd'$ , all benign servers  $s_i \in X$  stored  $c'$  in slot 2 with  $Q_2$ . Moreover, by Lemma 13, all benign servers from  $Q_2$  stored  $c'$  in slot 1. By Property 3 of RQS, in any execution  $ex$ , for every quorum  $Q$  at least one of the properties  $P_{3a}(Q_2, Q, B_{ex})$  and  $P_{3b}(Q_2, Q, B_{ex})$  holds. We now show that if  $P_{3a}(Q_2, Q, B_{ex})$  (resp.,  $P_{3b}(Q_2, Q, B_{ex})$ ) holds, then property  $V_1(c', Q)$  (resp.,  $V_2(c', Q)$ ) holds in  $ex$  by the time  $rd'$  completes:
    - i. In case  $P_{3a}(Q_2, Q, B_{ex})$  holds,  $T_Q = Q_2 \cap Q \setminus B_{ex}$  is a basic subset, that contains only benign servers (by Property 3a of RQS and definition of  $B_{ex}$ ). Hence, all (benign) servers in  $T_Q$  stored  $c'$  in slot 1, before  $rd'$  completes. Since  $T_Q \subseteq Q$ ,  $V_1(c', Q)$  holds in  $ex$  by the time  $rd'$  completes.

- ii. In case  $P_{3b}(Q_2, Q, B_{ex})$  holds,  $X \cap Q = Q_1 \cap Q_2 \cap Q \not\subseteq B_{ex}$ , i.e.,  $X \cap Q$  contains at least one benign server  $s_i$ . Since  $s_i \in X$ ,  $s_i$  stores  $c'$  in slot 2 before  $rd'$  completes. Moreover, since  $s_i \in Q$ ,  $V_2(c', Q)$  holds in  $ex$  by the time  $rd'$  completes.
  - (c) In case  $BCD(c', 1, 3)$  holds in  $rd'$ , there is class 1 quorum  $Q_1$  and a quorum  $Q'$ , such that, at the end of round 1 of  $rd'$ , for all benign servers  $s_i \in Q_1 \cap Q = X$ ,  $history[i, ts', 3].pair = c'$ . Since, by Property 1 of RQS, any quorum intersection is a basic subset, by Lemma 1,  $X$  contains at least one benign server  $s_i$ . Hence, by the end of the round 1 of  $rd'$  benign server  $s_i$  store  $c'$  in slot 3. Moreover, by Lemma 13, there is a quorum  $Q''$  such that all benign servers from  $Q''$  stored  $c'$  in slot 2 by the end of round 1 of  $rd'$ .  
By Property 1 of RQS and Definition 5,  $Q'' \cap Q'$  is a basic subset for every quorum  $Q$ . Moreover, every basic subset contains at least one benign server (by Lemma 1). Therefore, for every quorum  $Q$  there is a benign server  $s_Q \in Q$  that stored  $c'$  in slot 2 before  $rd'$  completed. Hence, for every quorum  $Q$  property  $V_2(c', Q)$  holds by the end of  $rd'$ .
2. If  $rd'$  completes in exactly two rounds, then  $\exists rnd \in \{1, 2, 3\} : BCD(c', 2, rnd) \neq \emptyset$  (line 41, Fig. 7). We consider the following two exhaustive cases, where, in  $rd'$ : (a)  $BCD(c', 2, rnd) \neq \emptyset$  holds for  $rnd \in \{2, 3\}$ , and (b)  $BCD(c', 2, rnd) \neq \emptyset$  holds only for  $rnd = 1$ .
- (a) In this case, a client that invoked  $rd'$  received acks for its  $wr\langle ts', v', \emptyset, 2 \rangle$  message from at least a quorum  $Q'$  of servers (when executing the *writeback* procedure in line 42). Hence, by the time  $rd'$  completes, all benign servers from  $Q'$  store  $c'$  in slot 2.  
By Property 1 of RQS and Definition 5, for every quorum  $Q$ ,  $Q' \cap Q$  is a basic subset. Moreover, every basic subset contains at least one benign server (by Lemma 1). Therefore, for every quorum  $Q$  there is a benign server  $s_Q \in Q$  that stored  $c'$  in slot 2 before  $rd'$  completed. Hence, for every quorum  $Q$  property  $V_2(c', Q)$  holds by the end of  $rd'$ .
  - (b) In this case, since  $BCD(c', 2, 1) \neq \emptyset$ ,  $Q_2$  is an element of  $BCD(c', 2, 1)$  if: (i)  $Q_2$  is a class 2 quorum that responded in the first round of  $rd'$ , and (ii) there is a class 1 quorum  $Q_1$ , such that, for all servers from  $X = Q_1 \cap Q_2$ ,  $history[*, ts', 1].pair = c'$ .  
In the round 2 of  $rd'$  the reader sends the  $wr\langle ts', v', BCD(c', 2, 1), 1 \rangle$  to all servers. Since  $rd'$  completes in exactly two rounds, all servers from some (class 2) quorum from  $BCD(c', 2, 1)$  respond in the round 2 of  $rd'$  as well. Denote this quorum by  $Q'_2$ . Then, all benign servers from  $Q'_2$  store  $c'$  in slot 1 with  $Q'_2$  by the time  $rd'$  completes.  
Since  $Q'_2$  is a class 2 quorum, by Property 3 of RQS, for every quorum  $Q$ , at least one of the following two properties holds in any execution  $ex$ : (i)  $P_{3a}(Q'_2, Q, B_{ex})$ , or (ii)  $P_{3b}(Q'_2, Q, B_{ex})$ . We now show that if  $P_{3a}(Q'_2, Q, B_{ex})$  (resp.,  $P_{3b}(Q'_2, Q, B_{ex})$ ) holds, then property  $V_1(c', Q)$  (resp.,  $V_3(c', Q)$ ) holds in  $ex$  by the time  $rd'$  completes:
    - i. In case  $P_{3a}(Q'_2, Q, B_{ex})$  holds,  $T_Q = Q'_2 \cap Q \setminus B_{ex}$  is a basic subset that contains only benign servers (by Property 3a of RQS and definition of  $B_{ex}$ ). Since  $T_Q \subseteq Q'_2$ , all (benign) servers from  $T_Q$  stored  $c'$  in slot 1, before  $rd'$  completes. Since  $T_Q \subseteq Q$ ,  $V_1(c', Q)$  holds in  $ex$  by the time  $rd'$  completes.
    - ii. In case  $P_{3b}(Q'_2, Q, B_{ex})$  holds,  $Q'_2 \cap Q \setminus B_{ex}$  is a set that contains only benign objects  $s_i$ , such that every  $s_i$  stored  $c'$  in slot 1 with  $Q'_2$  before  $rd'$  completes. Hence,  $V_3(c', Q)$  holds in  $ex$  by the time  $rd'$  completes.
3. If  $rd'$  completes in more than two rounds, then a client that invoked  $rd'$  received acks for its  $wr\langle ts', v', \emptyset, 2 \rangle$  message from at least a quorum  $Q'$  of servers (when executing the *writeback* procedure in line 47, or line 49). Hence, by the time  $rd'$  completes, all benign servers from  $Q'$  store  $c'$  in slot 2. Applying Property 1 of RQS, it is not difficult to see that for every quorum

$Q, Q' \cap Q$  contains at least one benign server. Hence,  $V_1(c', Q)$  holds for every quorum  $Q$  by the time  $rd'$  completes.

**Theorem 7. Atomicity.** *The algorithm in Figures 5, 6 and 7 is atomic.*

*Proof.* By Lemmas 5, 12 and 14. □

We proceed to prove the wait-freedom property. In the remainder of the proof we denote by  $Q_c$  a quorum that contains only correct servers.

First we prove two important auxiliary lemmas that provide an intuition behind the liveness of the algorithm provided a quorum that contains only correct servers,  $Q_c$ . Roughly speaking, the two lemmas state that it is not possible that, for some timestamp value pair  $c$ ,  $valid_2(c, Q_c)$  (resp.,  $valid_3(c, Q_c)$ ) holds yet that  $safe(c)$  does not hold (notice that  $valid_1(c, Q)$  trivially implies  $safe(c)$ , for any quorum  $Q$ ).

**Lemma 15. Liveness of  $valid_2$  predicate.** *No server  $s_i \in Q_c$  stores  $c$  in slot 2, before there is a basic subset  $T \subseteq Q_c$  such that all  $s_j \in T$  stored  $c$  in slot 1.*

*Proof.* By the algorithm's pseudocode, (a) correct server  $s_i$  stores a pair  $c$  in slot 2 only after some client sends  $wr\langle c.ts, c.val, *, 2 \rangle$  to  $s_i$ . Moreover, before any client  $clnt$  sends  $wr\langle c.ts, c.val, *, 2 \rangle$  to servers,  $clnt$  has already received acks for its  $wr\langle c.ts, c.val, *, 1 \rangle$  message from some quorum  $Q$ , except in case of a writeback in line 42 of Fig. 7 (where  $clnt$  is a reader). In all other cases,  $Q \cap Q_c$  is a basic subset (by Property 1 of RQS and Definition 5) — hence the lemma.

In the case of a writeback in line 42 of Fig. 7, assume, by contradiction, that there is a read  $rd$  that issues a message  $wr\langle c.ts, c.val, \emptyset, 2 \rangle$ , such that there is no basic subset  $T \subseteq Q_c$ , such that all servers from  $T$  previously stored  $c$  in slot 1. Moreover, let  $rd$  be the first such read according to the global clock that executes the *writeback* procedure in line 42, Fig. 7, at time  $t$ .

In this case,  $BCD(c, 2, 2)$  or  $BCD(c, 2, 3)$  are not empty in line 42 of  $rd$ . If  $BCD(c, 2, 2)$  (resp.,  $BCD(c, 2, 3)$ ) are not empty, then there exist two class 2 quorum  $Q_2$  and  $Q'_2$  (resp., a class 2 quorum  $Q_2$  and a quorum  $Q$ ) such that, all benign servers from  $Q_2 \cap Q'_2 = X$  (resp.,  $Q_2 \cap Q = X$ ), stored  $c$  in slot 2 (resp., slot 3). By Property 1 of RQS, Definition 5 and Lemma 1, there is at least one benign server in  $X$ , i.e., by the end of round 1 of  $rd$  at least one benign server received  $wr\langle c.ts, c.val, *, 2 \rangle$  (resp.,  $wr\langle c.ts, c.val, *, 3 \rangle$ ) from some client  $clnt'$ . By algorithm pseudocode and by our assumption on  $rd$ , before time  $t$ , client  $clnt'$  sends a message  $wr\langle c.ts, c.val, *, 2 \rangle$  (resp.,  $wr\langle c.ts, c.val, *, 3 \rangle$ ) only upon  $clnt'$  received acks for its  $wr\langle c.ts, c.val, *, 1 \rangle$  message from some quorum  $Q'$  of servers (i.e., not in line 42 of Fig. 7). Note that  $Q' \cap Q_c = T_c$  is a desired basic subset. A contradiction.

**Lemma 16. Liveness of  $valid_3$  predicate.** *Let  $Q_2$  be any class 2 quorum and  $B$  any element of adversary structure  $\mathbf{B}$ , such that  $P_{3b}(Q_2, Q_c, B)$  holds. If every server  $s_i \in Q_2 \cap Q_c \setminus B$  stored  $c$  in slot 1 with  $Q_2$  (by time  $t$ ), then (not later than  $t$ ) there is a basic subset  $T$  such that  $T \subseteq Q_c$ , and all  $s_j \in T$  stored  $c$  in slot 1.*

*Proof.* Denote the set  $Q_2 \cap Q_c \setminus B$  by  $X$ . Obviously, if  $X \notin \mathbf{B}$ ,  $X$  is the desired basic subset and the lemma follows. Therefore, in the following, we assume  $X \in \mathbf{B}$ .

Since  $X \in \mathbf{B}$ , by Property 3 of RQS,  $P_{3a}(Q_2, Q_c, X)$  or  $P_{3b}(Q_2, Q_c, X)$  must hold. However, since  $Q_2 \cap Q_c \setminus B = X$ , we have  $Q_2 \cap Q_c \setminus X \subseteq B$ , i.e.,  $P_{3a}(Q_2, Q_c, X)$  does not hold.

The only step in the algorithm in which correct server  $s_i$  stores a pair  $c$  in slot 1 with some actual quorum id (i.e., when  $history_i[c.ts, 1].sets$  changes the state) is when  $s_i$  receives a message

sent by a reader in the writeback call in line 44, Fig. 7 (recall here that readers are benign). Since all servers from  $X$  (recall here that servers from  $X$  are correct, since  $X \subseteq Q_c$ ) stored  $c$  in slot 1 with  $Q_2$  (by time  $t$ ), then there is at least one read  $rd$  that executes the *writeback* procedure in line 44 and sends  $wr\langle c.ts, c.val, \mathbf{Set}, 1 \rangle$  (line 60), where  $Q_2 \in \mathbf{Set}$ . In this case,  $\mathbf{BCD}(c, 2, 1) = \mathbf{Set}$  is not empty (since  $Q_2 \in \mathbf{Set}$ ) in line 41 of  $rd$ , and the condition in line 42 is not satisfied. Since  $\mathbf{BCD}(c, 2, 1)$  is not empty in  $rd$ , then (line 2, Fig. 7) there exists class 1 quorum  $Q_1$  such that for every benign server  $s_j$  in  $Q_1 \cap Q_2$ ,  $history[j, c.ts, 1].pair = c$  holds in  $rd$  (notice that this holds before time  $t$ ). Therefore, every benign server from  $Q_1 \cap Q_2$  stored  $c$  in slot 1 before time  $t$ . Hence, if  $Y = Q_1 \cap Q_2 \cap Q_c \notin \mathbf{B}$ , the lemma follows ( $Y$  is the desired basic subset). Therefore, in the following, we assume  $Y \in \mathbf{B}$ .

We have showed that (by time  $t$ ), every server  $s_i$  from the set  $Z = X \cup Y$  stored  $c$  in slot 1 and that  $Z \subseteq Q_c$  (since  $X \subseteq Q_c$  and  $Y \subseteq Q_c$ ). We now show that  $Z$  is the desired basic subset, i.e., we show that it is not possible that  $Z \in \mathbf{B}$ .

Assume by contradiction that  $Z \in \mathbf{B}$ . Then, by Property 3 of RQS,  $P_{3a}(Q_2, Q_c, Z)$  or  $P_{3b}(Q_2, Q_c, Z)$  must hold. However, since (i)  $P_{3a}(Q_2, Q_c, X)$  does not hold and (ii)  $X \subseteq Z$ ,  $P_{3a}(Q_2, Q_c, Z)$  cannot hold either (we have  $Q_2 \cap Q_c \setminus Z \subseteq B$ ). Moreover, since (i)  $P_{3b}(Q_2, Q_c, Y)$  does not hold (since  $Q_1 \cap Q_2 \cap Q_c = Y$ ) and (ii)  $Y \subseteq Z$ ,  $P_{3b}(Q_2, Q_c, Z)$  cannot hold either (we have  $Q_1 \cap Q_2 \cap Q_c \subseteq Z$ ). A contradiction.  $\square$

**Theorem 8. (Wait-freedom.)** *The algorithm in Figures 5, 6 and 7 is wait-free.*

*Proof.* The argument for the wait-freedom of a write operation is straightforward; in every round of a write, the writer waits for acks from at least one quorum, so the writer is guaranteed to receive the awaited acks eventually, since we assume existence of quorum  $Q_c$  that contains only correct servers. The timer that the writer awaits eventually expires and write eventually completes.

The argument for the wait-freedom of a read operation is more involved. We show that any read operation invoked by a correct client does not block in line 34, Fig. 7; the remainder of the proof is straightforward. We distinguish two cases: (1) the case where there is an infinite (unbounded) number of write operations in the execution, and (2) the case where the writer issues a finite number of write operations in the execution.

1. In this case, there is an infinite number of writes. Suppose, by contradiction, that  $rd$  never completes. Let  $ts$  equal  $highest\_ts$  computed at the end of round 1 of  $rd$ , in line 29, Fig. 7. Since the writer issues an unbounded number of writes, the writer will also issue a write with a timestamp  $ts$ , writing some value  $v$ . Since all benign servers from some quorum  $Q$  store  $\langle ts, v \rangle$  in slot 1 at some time  $t$  and, since by Property 1 of RQS  $Q \cap Q_c$  is a basic subset,  $safe(\langle ts, v \rangle)$  holds after  $rd$  receives at least one ack from every server from  $Q_c$  sent after  $t$ . Moreover, for all other pairs  $c$  with  $c.ts > ts$ ,  $invalid(c)$  will also hold (since  $c.ts > highest\_ts = ts$ ) and, hence,  $highCand(\langle ts, v \rangle)$  also eventually holds. Hence  $\langle ts, v \rangle$  is eventually in  $C$  and  $rd$  terminates. A contradiction.
2. In this case, there is a write operation with the highest timestamp. Let  $wr$  denote the last complete write operation that writes  $v$  with timestamp  $ts$  (or  $v = \perp$ ,  $ts = 0$  if there is none). We denote by  $wr'$  a possible later (incomplete) write that writes  $v'$  with  $ts'$ . Assume, by contradiction, that read  $rd$  never returns a value. First consider the case, where  $ts < highest\_ts$  (where  $highest\_ts$  is computed in line 29, Fig 7). Then,  $rd$  invokes rounds on all correct servers, sending rd messages infinitely many times. We distinguish two cases: (a) there is no basic subset  $T \subseteq Q_c$  such that all servers from  $T$  ever

stores  $c' = \langle ts', v' \rangle$  in slot 1, and (b) there is a time  $t$  at which all servers from some basic subset  $T \subseteq Q_c$  store  $c'$  in slot 1. In case (a), let  $t$  be the time at which the last correct server stores  $c'$  in slot 1. Moreover, let  $t' > t$  be the time at which  $rd$  receives at least one response from every server from  $Q_c$  sent after  $t$  (in both cases (a) and (b)).

- (a) Since  $wr$  completed, there is a quorum  $Q$  such that all benign servers from  $Q$  have stored  $\langle ts, v \rangle$  in slot 1. By Property 1 of RQS,  $Q \cap Q_c = T_v$  is a basic subset. Hence, from time  $t'$  onward,  $rd$  received at least one ack from all servers from  $T_v$  sent after  $wr$  completed and, hence,  $safe(\langle ts, v \rangle)$  holds.

Moreover, by Lemma 15 and assumption (a),  $V_2(c, Q_c)$  never holds for some pair  $c$  such that  $c.ts > ts$ . Similarly, by Lemma 16 and assumption (a),  $V_3(c, Q_c)$  never holds for such a pair  $c$ . Therefore, for every timestamp-value pair  $c$ , such that  $c.ts > ts$ ,  $valid_2(c, Q_c)$  and  $valid_3(c, Q_c)$  cannot hold in  $rd$ . Finally, by our assumption (a) no  $T \subseteq Q_c$  stores  $c$  in slot 1, such that  $c.ts > ts$ . Therefore, after time  $t'$ , for any value  $c$ , such that  $c.ts > ts$ ,  $valid_1(c, Q_c)$  does not hold. Hence, at the next iteration,  $invalid(c)$  holds for all  $c$  such that  $c.ts > ts$  and, therefore,  $highCand(\langle ts, v \rangle)$  holds; moreover, since  $safe(\langle ts, v \rangle)$  also holds,  $\langle ts, v \rangle$  is in set  $C$  in line 33 and  $rd$  returns, a contradiction.

- (b) In this case, after  $t'$ , there is a basic subset  $T \subseteq Q_c$  for which  $history[*, ts', 1].pair = \langle ts', v' \rangle$ . Hence,  $safe(\langle ts', v' \rangle)$  holds after  $t'$ . It is not difficult to see, since no subsequent valid value is present in the system (since  $wr'$  is the last write invoked), that for every timestamp/value pair  $c''$  such that  $c''.ts > c'.ts \vee (c''.ts = c'.ts \wedge c''.val \neq c'.val)$  none of the predicates  $valid_1(c'', Q_c)$ ,  $valid_2(c'', Q_c)$  or  $valid_3(c'', Q_c)$  holds, i.e.,  $invalid(c'')$  holds. Hence,  $highCand(\langle ts', v' \rangle)$  also holds. Thus, in the next iteration,  $\langle ts', v' \rangle \in C$  and  $read$  returns: a contradiction.

Consider now the case, where  $ts \geq highest\_ts$ . Since write  $wr$  (with timestamp  $ts$ ) completed, then a write with a timestamp  $highest\_ts$  also completed. It is not difficult to see (along the lines of the proof of case (1)) that  $rd$  returns the value written with timestamp  $highest\_ts$ .  $\square$

**Theorem 9. (Best-Case Latency.)** *The storage algorithm in Figures 5, 6 and 7 is  $(m, QC_m)$ -fast for all  $m \in \{1, 2, 3\}$ .*

*Proof.* For write operation, the proof is straightforward. For read, it is important to show that whenever the read is synchronous and uncontended, lines 20-35 are executed only once. This proof is given in the following. The rest of the proof is straightforward, by using the output of BCD (lines 1-2, Fig. 7).

Since there is no contention, let  $wr$  writing timestamp value pair  $c = \langle ts, v \rangle$  be the last (complete) write that precedes the read  $rd$ . Regardless of whether  $wr$  completed in 1, 2, or 3 rounds,  $wr$  wrote  $c = \langle ts, v \rangle$  into some quorum of servers  $Q$ . Moreover, no benign server stores any value with a higher timestamp than  $ts$  by Lemma 6. Since  $rd$  is synchronous, a quorum  $Q_c$  that contains only correct server will respond in the first round of  $rd$ . By Property 1 of RQS and Definition 5  $Q_c \cap Q = T_c$  is a basic subset that contains only correct servers, and, hence,  $safe(c)$  holds at the end of round 1 of  $rd$ . It is not difficult to see that for any value  $c'.val$  with  $c'.ts > ts$ , none of the predicates  $valid_1(c', Q_c)$ ,  $valid_2(c', Q_c)$  and  $valid_3(c', Q_c)$  will hold. Hence, for any such timestamp/value pair  $invalid(c')$  holds. Hence, at the end of round 1 of  $rd$ ,  $highCand(c)$  also holds and hence  $c \in C$  in line 33, Fig. 7.  $\square$

## B Correctness of the consensus algorithm

In this Appendix we prove the correctness of our consensus algorithm of Section 4.2. First, we give few definitions.

**Definition 6 (Value decided in a view).** We say that a value  $v$  is Decided-2, Decided-3 or Decided-4 in view  $w$ , if there is a benign process (acceptor or learner)  $p$  that eventually decides a value by receiving (respectively):

- (Decided-2)  $\text{update}_1\langle v, w, * \rangle$  messages from a class 1 quorum (line 51, Fig. 15).
- (Decided-3)  $\text{update}_2\langle v, w, Q_2 \rangle$  messages from a class 2 quorum  $Q_2$  (line 52, Fig. 15).
- (Decided-4)  $\text{update}_3\langle v, w, * \rangle$  messages from some quorum (line 53, Fig. 15).

We also say that a value  $v$  is decided in view  $w$ , if some benign process  $p$  Decided- $m$   $v$  in view  $w$  (where  $m \in \{2, 3, 4\}$ ).

**Definition 7 (Prepares).** We say that an acceptor  $a_i$  prepares a value  $v$  in view  $w$ , if it eventually receives  $\text{prepare}\langle v, w, *, * \rangle$  and executes lines 31-33, Fig. 15.

**Definition 8 (Updates).** We say that a benign acceptor  $a_i$  updates a value  $v$  in view  $w$ , if it eventually receives  $\text{update}_{\text{step}}\langle v, w, * \rangle$  for some  $\text{step} \in \{1, 2\}$  and executes lines 34-38, Fig. 15. More precisely, we say  $a_i$  1-updates (resp., 2-updates)  $v$  in  $w$  if  $\text{step} = 1$  (resp.,  $\text{step} = 2$ ).

**Definition 9 (Accepts).** We say that a benign acceptor  $a_i$  accepts a value  $v$  in view  $w$ , if it prepares or updates  $v$  in view  $w$ .

We also make use of the Definition 5 of Appendix A (definition of *basic* and *large* subsets).

In addition, we introduce the following notation:

- We say that an invocation of function  $\text{choose}(*, v\text{Proof}, Q)$  is *valid* if  $v\text{Proof}$  consists of valid `new_view_ack` messages sent by acceptors from quorum  $Q$  (with slight abuse of language, we also simply say  $\text{choose}(*, v\text{Proof}, Q)$  is *valid*);
- We denote all Byzantine acceptors in execution  $ex$  by  $B_{ex}$ . We assume  $B_{ex} \in \mathbf{B}$  for any execution  $ex$ .

**Lemma 17. Size of basic sets.** In every execution of our consensus algorithm, any basic subset contains at least one benign acceptor.

*Proof.* The lemma follows directly from the definition of a basic subset (Definition 5). □

**Lemma 18. Size of large sets.** In every execution  $ex$  of our consensus algorithm. for any large subset  $T_2$ , there is a basic subset  $T_1 \subseteq T_2$  that contains only benign acceptors.

*Proof.* By Definition 5, for any large subset  $T_2$ ,  $T_2 \setminus B_{ex}$  is a basic subset. By definition of  $B_{ex}$ ,  $T_2 \setminus B_{ex}$  contains only benign acceptors. Hence the lemma. □

We first prove *Validity*.

**Lemma 19. Validity of the choose function.** If valid  $\text{choose}(v, v\text{Proof}, Q)$  returns  $v$  such that  $v$  is a candidate with view  $w$ , then at least one benign acceptor  $a_i$  prepared  $v$  in  $w$ .

*Proof.* Assume  $Cand_2(v, w)$  holds (line 1, Fig. 13). In this case, every acceptor  $a_j$  from the set  $X = (Q_1 \cap Q) \setminus B$  (where  $B$  is not a basic subset and  $Q_1$  is a class 1 quorum) reported that it prepared  $v$  in  $w$ . Note that, by Property 2 of RQS,  $Q_1 \cap Q$  is a large subset. By Lemma 18,  $X$  contains at least one benign acceptor.

Assume now  $Cand_3(v, w, char)$  holds (for  $char \in \{a', b'\}$ ). From lines 2-3, Fig. 13, it follows that all acceptors from the set  $X = (Q_2 \cap Q) \setminus B$ , (where  $B$  is not a basic subset and  $Q_2$  is a class 2 quorum) reported that they updated  $v$  in  $w$  (i.e.,  $\forall a_j \in X : vProof[a_j].Update[1] = v$  and  $w \in vProof[a_j].Update_{view}[1]$ ). Note that, by Property 1 of RQS,  $Q_2 \cap Q$  is a basic subset. Hence, by Definition 5,  $X$  is a non-empty set. In this case,  $vProof[a_j].Update_{proof}[1, w]$  contains at least a basic subset of signed  $update_1\langle v, w, * \rangle$  messages. By Lemma 17 at least one of these signed messages comes from a benign acceptor  $a_i$  that indeed prepared  $v$  in view  $w$ .

The argument for the case where  $Cand_4(v, w)$  holds (line 5, Fig. 13) is very similar to the case where predicate  $Cand_3(v, w, char)$  holds.  $\square$

**Theorem 10. (Validity)** *If a benign learner learns a value  $v$  and all proposers are benign, then some proposer proposed  $v$ .*

*Proof.* A benign learner learns a value  $v$  by receiving (1)  $update_*$  messages (lines 51-53 and 60, Fig. 15), or (2) by receiving a basic subset of  $decision$  messages (line 101, Fig. 15). In case (b), by Lemma 17 and line 40, Fig. 15 at least one benign acceptor decided  $v$  before the learner learned  $v$ .

Hence, in both cases, if a learner learns  $v$ , then  $v$  was accepted in some view  $w$  (prepared or updated) by benign acceptors from some quorum of acceptors. Since any quorum is a basic subset, there is at least one such benign acceptor  $a_j$  (by Property 1 of RQS, and Lemma 17). Note that  $a_j$  updates  $v$  in  $w$  only upon  $a_j$  prepares  $v$  in  $w$ . We prove the following statement using induction on view numbers: *if a benign acceptor prepares  $v$  in view  $w$ , then some proposer proposed  $v$ .*

*Base Step:* ( $w = initView$ )

Benign acceptors prepare value  $v$  in  $initView$  only if they receive a  $prepare\langle v, initView, *, * \rangle$  message from some proposer. Since all proposers are benign, no proposer sends a  $prepare$  message containing  $v$  unless it proposes  $v$ . Hence, if some benign acceptor accepts  $v$ ,  $v$  was indeed proposed by some proposer.

*Inductive Hypothesis (IH):* For every view  $w, w' > w \geq initView$ , if a benign acceptor accepts  $v$  in  $w$ , then some proposer proposed  $v$ .

*Inductive Step:* We prove that the statement is true for view  $w'$ . In view  $w'$ , acceptor accept only values returned by valid  $choose(*, vProof, Q)$ . If  $choose(*, vProof, Q)$  returns a candidate value  $v$ , by Lemma 19, some benign acceptor prepared  $v$  in view  $w, w < w'$ , and by IH,  $v$  was proposed by some proposer. If  $choose(*, Writeproof)$  returns  $v$  in line 21, Figure 13, then  $v$  is the initial proposal value of the leader of  $w'$ . We conclude that  $v$  was proposed by some proposer.  $\square$

Now we prove *Agreement*.

**Lemma 20.** *After sending a `new_view_ack` message for view  $w$ , a benign acceptor cannot accept a value  $v$  with view number  $w' < w$ .*

*Proof.* By line 21, Fig. 15 a benign acceptor cannot prepare a value with  $w' < w$ . Moreover, a benign acceptor  $a_i$  updates a value  $v$  in some view  $w''$  only after  $a_j$  prepares  $v$  in  $w''$ . Hence the lemma.  $\square$

**Lemma 21.** *If two values  $v$  and  $v'$  are decided in view  $w$ , then  $v = v'$ .*

*Proof.* Suppose  $v \neq v'$ . From Def. 6, all acceptors from some quorum  $Q$  (resp.,  $Q'$ ) sent  $\text{update}_m\langle v, w \rangle$  (resp.,  $\text{update}_{m'}\langle v', w \rangle$ ) message, for some  $m, m' \in \{1, 2, 3\}$ . Hence, all benign acceptors from  $Q$  (resp.,  $Q'$ ) prepare  $v$  (resp.,  $v'$ ) in  $w$ . By Property 1 of RQS and Definition 5,  $Q \cap Q'$  is a basic subset, which contains at least one benign acceptor  $a_i$  (by Lemma 17). That is, there exists a benign acceptor that prepared different values in the same view. A contradiction.  $\square$

**Lemma 22. Unique Cand2( $v, w$ ).** *There are no two different values  $v$  and  $v'$  such that, in valid  $\text{choose}(*, v\text{Proof}, Q)$ , both  $\text{Cand}_2(v', w)$  and  $\text{Cand}_2(v, w)$  hold, for the same  $w$ .*

*Proof.* Assume by contradiction that such values  $v$  and  $v'$  exist. By definition of the predicate  $\text{Cand}_2()$  (line 1, Fig. 13), there are sets  $X = (Q_1 \cap Q) \setminus B$  and  $X' = (Q'_1 \cap Q) \setminus B'$ , such that (1)  $B, B' \subseteq \text{acceptors}$  and  $B$  and  $B'$  are not basic subsets, (2)  $Q_1$  and  $Q'_1$  are class 1 quorums, and (3) all acceptors from the set  $X$  (resp.,  $X'$ ) prepared  $v$  (resp.,  $v'$ ) in  $w$ . By Property 2 of RQS,  $Q_1 \cap Q'_1 \cap Q$  is a large subset. Applying Definition 5 we conclude that  $X \cap X'$  is a non-empty set. Hence, there is an acceptor  $a_j \in Q$  such that  $v\text{Proof}[a_j].\text{Prep} = v$  and  $v\text{Proof}[a_j].\text{Prep} = v'$ . Hence,  $v = v'$ . A contradiction.  $\square$

**Lemma 23. Cand3( $v, w, 'a'$ )/Cand4( $v, w$ ).** *If for some value  $v$   $\text{Cand}_3(v, w, 'a')$  or  $\text{Cand}_4(v, w)$  hold in valid  $\text{choose}(*, v\text{Proof}, Q)$ , then all benign acceptors from some quorum  $Q'$  prepared  $v$  in  $w$ .*

*Proof.* Assume first  $\text{Cand}_3(v, w, 'a')$  holds. Then there is a set  $X = (Q_2 \cap Q) \setminus B'$  such that  $B'$  is not a basic subset and  $Q_2$  is a class 2 quorum and  $P_{3a}(Q_2, Q, B')$  holds. By Property 3a of RQS,  $X$  is a basic subset. By Lemma 17 there is at least one benign acceptor in  $X$  that updated  $v$  in  $w$ . Therefore, by lines 34-38 in Fig. 15, all benign acceptors from some quorum  $Q'$  prepared  $v'$  in  $w$ .

Assume now  $\text{Cand}_4(v, w)$  holds. Then there exists an acceptor  $a_j \in Q$  such that:

- $v\text{Proof}[a_j].\text{Update}[2] = v'$ ,
- $w \in v\text{Proof}[a_j].\text{Update}_{\text{view}}[2]$ , and
- $v\text{Proof}[a_j].\text{Update}_{\text{proof}}[2, w]$  contains a basic subset of signatures of  $\text{update}_2\langle v, w, * \rangle$  messages including at least one from a benign acceptor  $a_b$ .

Hence a benign acceptor  $a_b$  updated  $v'$  in  $w$ . Therefore, all benign acceptors from some quorum  $Q'$  prepared  $v'$  in  $w$ .  $\square$

**Lemma 24. Impossible candidates after decision.** *If value  $v$  is decided in some view  $w$ , then, in any valid  $\text{choose}(*, v\text{Proof}, Q)$ , for some  $v' \neq v$ , neither  $\text{Cand}_3(v', w, 'a')$  nor  $\text{Cand}_4(v', w)$  can hold.*

*Proof.* In case  $v$  was decided in view  $w$ , by Definitions 6, 7 and 8, all benign acceptors from a quorum  $Q$  prepared  $v$  in  $w$ .

Assume, by contradiction, that such value  $v' \neq v$  exists, such that  $\text{Cand}_3(v', w, 'a')$  or  $\text{Cand}_4(v', w)$  hold and  $v' \neq v$ . Then, by Lemma 23, all benign acceptors from some quorum  $Q'$  prepared  $v'$  in  $w$ . By Property 1 of RQS,  $Y = Q \cap Q'$  is a basic subset that contains at least one benign acceptor which prepared both  $v$  and  $v'$  in  $w$ . A contradiction.  $\square$

**Lemma 25.** *If  $w$  is the lowest view number in which some value  $v$  is Decided-2, then no benign acceptor  $a_i$  prepares any value  $v'$ ,  $v' \neq v$  in any view higher than  $w$ .*

*Proof.* We prove this lemma by induction on view numbers.

*Base Step:* First, we prove the lemma for view  $w + 1$ . A benign acceptor  $a_i$  prepares a value  $v'$  in some view  $W > w$  only if the valid  $choose(*, vProof, Q)$  function in view  $W$  returns  $v'$ , without setting the *abort* flag. Therefore, it is sufficient to prove that for a valid  $choose(*, vProof, Q)$  in view  $w + 1$  returns  $v$ , or *abort* flag is set.

By Definitions 6 and 7, all benign acceptors from a class 1 quorum  $Q_1$  prepared  $v$  in  $w$ . By definition of  $B_{ex}$ , set  $X = (Q_1 \cap Q) \setminus B_{ex}$  that contains only benign acceptors. By Lemma 20, every acceptor  $a_i \in X$  prepared  $v$  in  $w$ , before replying with the `new_view_ack` message to the leader of the view  $w + 1$ . In the meantime, no acceptor  $a_j \in X$  prepared any other value, as this would mean that  $a_j$  would be in a higher view than  $w + 1$  when replying with `new_view_ack` for the view  $w + 1$ , which is impossible. Therefore,  $Cand_2(v, w)$  (line 1, Fig. 13) holds in  $choose(*, vProof, Q)$ , for any  $Q$ . Notice that for every acceptor  $a_j \in X$ ,  $w \in vProof[a_j].Prep_{view}$ .

By Lemma 19, it is not difficult to see that there is no value  $v'$  such that  $Cand_2(v', w')$ ,  $Cand_3(v', w', *)$  or  $Cand_4(v', w')$  holds for some  $w' > w$ .

By Lemma 22, there is no value  $v' \neq v$  such that  $Cand_2(v', w)$  holds.

By Lemma 24, there is no value  $v' \neq v$  such that (i)  $Cand_3(v', w, 'a')$  holds or (ii)  $Cand_4(v', w)$  holds.

Finally, in the following part of the proof, we show that it is not possible that both  $Cand_3(v', w, 'b')$  and  $Valid_3(v', w, 'b')$  hold for  $v' \neq v$ .

Assume, by contradiction, that there is such a value  $v' \neq v$  such that both  $Cand_3(v', w, 'b')$  and  $Valid_3(v', w, 'b')$ . Since  $Cand_3(v', w, 'b')$  holds, there are class 2 quorum  $Q_2$  and  $B \in \mathbf{B}$  such that  $C_3(v', w, 'b', Q_2, B)$  holds (line 2, Fig. 13). Moreover, since  $Valid_3(v', w, 'b')$  holds, all acceptors from  $Y = Q_2 \cap Q$  claim they prepared  $v'$  in  $w$  (line 2, Fig. 13) or do not have  $w$  in  $vProof[a_j].Prep_{view}$ . Since we know that for all (benign) servers from  $a_j \in X = (Q_1 \cap Q) \setminus B_{ex}$ ,  $v \in vProof[a_j].Prep$  and  $w \in vProof[a_j].Prep_{view}$ , we conclude  $X \cap Y = \emptyset$ . Hence, we have  $Q_1 \cap Q_2 \cap Q \setminus B_{ex} = \emptyset$ , i.e.,  $P_{3b}(Q_2, Q, B_{ex})$  does not hold. By Property 3 of RQS,  $P_{3a}(Q_2, Q, B_{ex})$  must hold.

Moreover, since  $C_3(v', w, 'b', Q_2, B)$ , all acceptors from  $Z = Q_2 \cap Q \setminus B$  claim that all acceptors from  $Q_2$  prepared  $v'$  in  $w$ . We distinguish two cases: (1) all acceptors from  $Z$  are Byzantine, i.e.,  $Z \subseteq B_{ex}$ , and (2) there is a benign server in  $Z$ .

In case (1), since  $Z \subseteq B_{ex}$  and  $P_{3a}(Q_2, Q, B_{ex})$  holds, we have  $P_{3a}(Q_2, Q, Z)$ . Hence  $Q_2 \cap Q \setminus Z \notin \mathbf{B}$ . However, by definition of  $Z$ ,  $Q_2 \cap Q \setminus Z \subseteq B \in \mathbf{B}$ . A contradiction.

In case (2), since  $Z$  contains at least one benign acceptor  $s_i$ , then all benign acceptors from  $Q_2$  prepared  $v'$  in  $w$ . By Property 2 of RQS,  $Q_1 \cap Q_2 \setminus B_{ex}$  is a basic subset that contains only benign acceptors, that all prepared both  $v$  and  $v'$  in  $w$ . A contradiction.

By inspection of *choose()* pseudocode, *choose()* returns  $v$  or *abort* flag is set.

*Inductive Hypothesis (IH):* Assume that no benign acceptor  $a_i$  prepares any value different from  $v$  in any view from  $w + 1$  to  $w + k$ . We prove that no benign acceptor  $a_i$  can prepare any value different from  $v$  in the view  $w + k + 1$ .

*Inductive Step:* It is sufficient to prove that for a valid *choose*( $*$ ,  $vProof$ ,  $Q$ ) in view  $w + k + 1$  returns  $v$ , or *abort* flag is set.

By Definitions 6 and 7, all benign acceptors from a class 1 quorum  $Q_1$  prepared  $v$  in  $w$ . By IH, all benign acceptors from  $Q_1$  can prepare only  $v$  in views  $w + 1$  to  $w + k$ . Set  $X = (Q_1 \cap Q) \setminus B$  contains only benign acceptors; by Lemma 20, no acceptor  $a_i \in X$  prepares a value in a higher view than  $w + k$  before sending a *new\_view\_ack* message to the leader of the view  $w + k + 1$ . Hence, by definition of predicate  $Cand_2()$ ,  $Cand_2(v, w)$  holds in *choose*( $*$ ,  $vProof$ ,  $Q$ ), for any  $Q$ . Notice that for every acceptor  $a_j \in X$ ,  $w \in vProof[a_j].Prepview$ .

By Lemma 22, there is no other value  $v' \neq v$  such that  $Cand_2(v', w)$  holds.

By Lemma 19 and IH, it is not difficult to see that there is no value  $v' \neq v$  such that  $Cand_2(v', w')$ ,  $Cand_3(v', w', *)$  or  $Cand_4(v', w')$  holds for some  $w' > w$ .

By Lemma 24, there is no value  $v' \neq v$  such that  $Cand_3(v', w, 'a')$  holds or  $Cand_4(v', w)$  holds.

Finally, exactly as in the Base Step, it can be shown that it is not possible that both  $Cand_3(v', w, 'b')$  and  $Valid_3(v', w, 'b')$  hold for  $v' \neq v$ .

By inspection of *choose()* pseudocode, *choose()* returns  $v$  or *abort* flag is set. □

Similarly to Lemma 25, we prove the following two lemmas using the properties of RQS and induction on view numbers.

**Lemma 26.** *If  $w$  is the lowest view number in which some value  $v$  is Decided-3, then no benign acceptor  $a_i$  prepares any value  $v'$ ,  $v' \neq v$  in any view higher than  $w$ .*

*Proof.* We prove this lemma by induction on view numbers.

*Base Step:* First, we prove the lemma for view  $w + 1$ . It is sufficient to prove that any valid *choose*( $*$ ,  $vProof$ ,  $Q$ ) in view  $w + 1$  returns  $v$ , or *abort* flag is set.

By Definitions 6 and 8, all benign acceptors from a class 2 quorum  $Q_2$  updated-1 (and prepared)  $v$  in  $w$ . By definition of  $B_{ex}$ , set  $X = (Q_2 \cap Q) \setminus B_{ex}$  contains only benign acceptors. By Lemma 20, every acceptor  $a_i \in X$  prepared  $v$  in  $w$ , before replying with the *new\_view\_ack* message to the leader of the view  $w + 1$ . In the meantime, no acceptor  $a_j \in X$  prepared any other value, as this would mean that  $a_j$  would be in the higher view then  $w + 1$  when replying with *new\_view\_ack* for the view  $w + 1$ , which is impossible. Therefore, for some  $char \in \{'a', 'b'\}$ ,  $Cand_3(v, w, char, Q_2, B_{ex})$  and  $Cand_3(v, w, char)$  (lines 2-3, Fig. 13) hold in *choose*( $*$ ,  $vProof$ ,  $Q$ ), for any  $Q$ .

By Lemma 19, it is not difficult to see that there is no value  $v'$  such that  $Cand_2(v', w')$ ,  $Cand_3(v', w', *)$  or  $Cand_4(v', w')$  holds for some  $w' > w$ .

By Lemma 24, there is no value  $v' \neq v$  such that  $Cand_3(v', w, 'a')$  holds or  $Cand_4(v', w)$  holds.

We distinguish two cases: (a)  $Cand_3(v', w', 'a')$ , and (b)  $Cand_3(v', w', 'b')$  holds. By inspection of *choose()* pseudocode, in case (a) *choose()* returns  $v$ , whereas in case (b) either *choose()* returns  $v$  or *abort* flag is set.

*Inductive Hypothesis (IH):* Assume that no benign acceptor  $a_i$  prepares any value different from  $v$  in any view from  $w + 1$  to  $w + k$ . We prove that no benign acceptor  $a_i$  can prepare any value

different from  $v$  in the view  $w + k + 1$ .

*Inductive Step:* It is sufficient to prove that any valid  $choose(*, vProof, Q)$  in view  $w + k + 1$  returns  $v$ , or *abort* flag is set.

By Definitions 6 and 8, all benign acceptors from a class 2 quorum  $Q_2$  update-2  $v$  in  $w$ . By IH, all benign acceptors from  $Q_2$  can update-2 only  $v$  in views  $w + 1$  to  $w + k$ . All acceptors from  $X = (Q_2 \cap Q) \setminus B_{ex}$  are benign and, by Lemma 20, no acceptor  $a_i \in X$  prepares (nor 1-updates) a value in a higher view than  $w + k$  before sending a `new_view_ack` message to the leader of the view  $w + k + 1$ . Hence, for every  $a_i \in X$   $vProof[a_i].Update[1] = v$  and  $w \in vProof[a_i].Update_{view}[1]$ . Hence, by definition of predicate  $Cand_3()$ , for some  $char \in \{‘a’, ‘b’\}$ ,  $Cand_3(v, w, char, Q_2, B_{ex})$  and  $Cand_3(v, w, char)$  hold in  $choose(*, vProof, Q)$ , for any  $Q$ .

By Lemma 19 and IH, it is not difficult to see that there is no value  $v' \neq v$  such that  $Cand_2(v', w')$ ,  $Cand_3(v', w', *)$  or  $Cand_4(v', w')$  holds for some  $w' > w$ .

By Lemma 24, there is no value  $v' \neq v$  such that  $Cand_3(v', w, ‘a’)$  holds or  $Cand_4(v', w)$  holds.

We distinguish two cases: (a)  $Cand_3(v', w', ‘a’)$ , and (b)  $Cand_3(v', w', ‘b’)$  holds. By inspection of  $choose()$  pseudocode, in case (a)  $choose()$  returns  $v$ , whereas in case (b) it either returns  $v$  or *abort* flag is set.  $\square$

**Lemma 27.** *If  $w$  is the lowest view number in which some value  $v$  is Decided-4, then no benign acceptor  $a_i$  prepares any value  $v'$ ,  $v' \neq v$  in any view higher than  $w$ .*

*Proof.* We prove this lemma by induction on view numbers.

*Base Step:* First, we prove the lemma for view  $w + 1$ . It is sufficient to prove that any valid  $choose(*, vProof, Q)$  in view  $w + 1$  returns  $v$ .

By Definitions 6 and 8, all benign acceptors from some quorum  $Q_3$  updated-2  $v$  in  $w$ . By Property 1 of RQS  $X = Q_3 \cap Q$  is a basic subset that contains at least one benign acceptor  $a_i$  (by Lemma 17). By Lemma 20,  $a_i$  updated-2  $v$  in  $w$ , before replying with the `new_view_ack` message to the leader of the view  $w + 1$ . In the meantime,  $a_j$  did not prepare (nor update-2) any other value, as this would mean that  $a_j$  would be in the higher view than  $w + 1$  when replying with `new_view_ack` for the view  $w + 1$ , which is impossible. Therefore,  $Cand_4(v, w)$  (line 5, Fig. 13) holds in  $choose(*, vProof, Q)$ , for any  $Q$ .

By Lemma 19, it is not difficult to see that there is no value  $v'$  such that  $Cand_2(v', w')$ ,  $Cand_3(v', w', *)$  or  $Cand_4(v', w')$  holds for some  $w' > w$ .

By Lemma 24, there is no value  $v' \neq v$  such that  $Cand_3(v', w, ‘a’)$  holds or  $Cand_4(v', w)$  holds.

By inspection of  $choose()$  pseudocode,  $choose()$  returns  $v$ .

*Inductive Hypothesis (IH):* Assume that no benign acceptor  $a_i$  prepares any value different from  $v$  in any view from  $w + 1$  to  $w + k$ . We prove that no benign acceptor  $a_i$  can prepare any value different from  $v$  in the view  $w + k + 1$ .

*Inductive Step:* It is sufficient to prove that any valid  $choose(*, vProof, Q)$  in view  $w + k + 1$  returns  $v$ .

By Definitions 6 and 8, all benign acceptors from some quorum  $Q_3$  updated-2  $v$  in  $w$ . By IH, all benign acceptors from  $Q_3$  can prepare (and, hence, update-2) only  $v$  in views  $w + 1$  to  $w + k$ . Moreover,  $X = Q_3 \cap Q$  contains at least one benign acceptor  $a_j$  (by Property 1 of RQS and

Lemma 17). Hence, by definition of predicate  $Cand_4()$ ,  $Cand_4(v, w')$  holds in  $choose(*, vProof, Q)$ , for any  $Q$ , for some  $w', w + k \geq w' \geq w$ .

By Lemma 19 and IH, it is not difficult to see that there is no value  $v' \neq v$  such that  $Cand_2(v', w'')$ ,  $Cand_3(v', w'', *)$  or  $Cand_4(v', w'')$  holds for some  $w'' > w$ .

By Lemma 24, there is no value  $v' \neq v$  such that  $Cand_3(v', w, 'a')$  holds or  $Cand_4(v', w)$  holds. By inspection of  $choose()$  pseudocode,  $choose()$  returns  $v$ .  $\square$

**Theorem 11. (Agreement)** *No two benign learners learn different values.*

*Proof.* If a benign learner learns a value then a value was decided in some view (by some benign process). Indeed, a benign learner learns a value  $v$  by receiving (1) **update**<sub>\*</sub> messages (lines 51-53, and 60 Fig. 15), or (2) by receiving a basic subset of **decision** messages (line 101, Fig. 15). In case (b), by Lemma 17 at least one benign acceptor decided  $v$  before the learner learned  $v$ .

It is not difficult to see that, if some value  $v'$  is decided in view  $w$ , then some benign acceptor prepared  $v'$  in  $w$ . The theorem follows from Lemmas 21, 25, 26 and 27.  $\square$

It is straightforward to show that our algorithm is  $(m, \mathbf{QC}_m)$ -fast, for  $m \in \{1, 2, 3\}$ .

The following two lemmas are critical for ensuring *Termination* property.

The first lemma proves that our algorithm does not block in lines 3-8, Fig. 15, in case some quorum contains only correct acceptors.

**Lemma 28.** *The abort flag is never set in valid  $choose(*, vProof, Q)$ .*

*Proof.* It is sufficient to prove that if  $choose(*, vProof, Q)$  sets *abort* flag, then  $Q$  contains at least one Byzantine acceptor. We consider two exhaustive cases.

Case (a):  $choose()$  aborts in line 16, as there are two values  $v$  and  $v' \neq v$  such that both  $Cand_3(v, w, 'b')$  and  $Cand_3(v', w, 'b')$  hold (for  $w = view_{max}$ ). In this case, by definition of predicate  $Cand_3()$  (lines 2-3, Fig. 13) there are acceptors  $a_i, a_j \in Q$  and class 2 quorums  $Q_2$  and  $Q'_2$  such that: (1)  $a_i$  claims that all (benign) acceptors from  $Q_2$  prepared  $v$  in  $w$ , (2)  $a_j$  claims that all (benign) acceptors from  $Q'_2$  prepared  $v'$  in  $w$ . By Property 1 of RQS  $Q_2 \cap Q'_2$  is a basic subset, that by Lemma 17 contains at least one benign acceptor  $a_x$ . Hence,  $a_i$  claims that a benign acceptor  $a_x$  prepared  $v$  in  $w$ , while  $a_j$  claims that  $a_x$  prepared  $v' \neq v$  in  $w$ . Hence, at least one acceptor from the set  $\{a_j, a_i\} \subset Q$  is Byzantine.

Case (b):  $choose()$  aborts in line 18, as  $Cand_3(v, w, 'b')$  holds but  $Valid_3(v, w, 'b')$  does not hold. Assume, by contradiction, that  $Q$  contains only benign acceptors. Notice that, if a benign acceptor  $a_j \in Q$  prepares  $v$  in view  $w$ , then the following predicate  $P$  (extracted from definition of  $Valid_3(v, w, 'b')$ , line 4, Fig. 13)

$$((vProof[a_j].Prep = v) \wedge (w \in vProof[a_j].Prep_{view})) \vee (w' \in vProof[a_j].Prep_{view} \Rightarrow w' > w)$$

must hold in any valid  $choose(*, vProof, Q)$  for view higher than  $w$ . To see this, consider lines 31-33, Fig. 15, and notice that when benign acceptor  $a_j$  prepares  $v$  in  $w$ ,  $Prep = v$  and  $w \in Prep_{view}$  (at  $a_j$ ). If this ceases to hold at some point, then  $a_j$  prepared a different value in a view higher than  $w$ , and  $Prep_{view}$  contains only view numbers higher than  $w$  (line 32, Fig. 15).

Since  $Cand_3(v, w, 'b')$  holds, but  $Valid_3(v, w, 'b')$  does not hold, there is class 2 quorum  $Q_2$  and  $B \in \mathbf{B}$ , such that  $C_3(v, w, 'b', Q_2, B)$  holds, i.e., all acceptors in  $X = Q_2 \cap Q \setminus B$  claim that all acceptors from  $Q_2$  prepared  $v$  in view  $w$ . However, there is an acceptor  $a_i$  in  $Q_2 \cap Q$  (since  $Valid_3(v, w, 'b')$  does not hold) for which the above predicate  $P$  does not hold, i.e.,  $a_i$  never prepared  $v$  in  $w$ . Obviously, some acceptor from  $a_i \cup X \subseteq Q$  is Byzantine. A contradiction.  $\square$

The second lemma proves that our algorithm does not block in lines 23-27, Fig. 15, in case some quorum contains only correct acceptors. In the following proof, we explicitly make use of the assumption of an eventually synchronous system, i.e., of an existence of a global stabilization time  $GST$  after which the system is synchronous.

**Lemma 29. (*Availability of signatures.*)** *If a correct acceptor  $a_j$  issues a  $sign\_req\langle Update[step], w, step \rangle$  message (line 24, Fig. 15) after  $GST$ , then  $a_j$  eventually receives signed  $update_{step}\langle Update[step], w, * \rangle$  messages from some basic subset of acceptors.*

*Proof.* A correct acceptor  $a_j$  issues a  $sign\_req\langle Update[step], w, step \rangle$ , only if (at  $a_j$ )  $w \in Update_{view}[step]$ , i.e., only if  $a_j$  updated a value  $v = Update[step]$  in  $w$ . In other words, before issuing a  $sign\_req$  message,  $a_j$  received  $update_{step}\langle v, w, * \rangle$  messages from some quorum  $Q$  and executed lines 34-38, Fig. 15. In particular  $a_j$  adds the identifier of the quorum  $Q$  to the  $Update_Q[step, w]$  set (line 37, Fig. 15). Without loss of generality, we can assume that  $a_j$  sent a  $sign\_req\langle v, w, step \rangle$  message to acceptors from quorum  $Q$ .

Let  $Q_c$  be the quorum that contains only correct acceptors. By Property 1 of RQS,  $T = Q \cap Q_c$  is a basic subset. Since  $T \subseteq Q_c$ ,  $T$  contains only correct acceptors. Since after  $GST$  the system is synchronous,  $a_j$  eventually receives the desired set of signatures.  $\square$

We also need the following two simple lemmas. In the remainder of the paper, we denote by  $Q_c$  the quorum that contains only correct acceptors.

**Lemma 30.** *If some process receives **decision** messages with the same value  $v$  from some quorum of acceptors  $Q$ , then every correct learner learns a value.*

*Proof.* Suppose, by contradiction, that some correct learner  $l_k$  never learns a value.

By Property 1 of RQS and assumption on  $Q_c$ ,  $T = Q_c \cap Q$  is a basic subset of correct acceptors that decided a value  $v$ . Denote by  $t$  the time after which all acceptors from  $T$  have decided  $v$ . By lines 102-103, Fig. 15, and our assumption that  $l_k$  never learns the value,  $l_k$  sends an infinite number of **decision\_pull** messages to all acceptors. Those messages sent after  $max(t, GST)$  are received by all acceptors from  $T$  who send **decision** messages to  $l_k$ . These messages are received by  $l_k$  and, by line 101 Fig. 15,  $l_k$  learns  $v$  — a contradiction.  $\square$

Towards proving *Termination*, we first show that our algorithm (or, more precisely, its *Locking* module) satisfies a weaker property we call *Eventual Obstruction-Free Termination* (EOFT), defined as follows.

**Definition 10. (*Eventual Obstruction-Free Termination.*)** *Assume a correct proposer  $p_k$  proposes a value at time  $t_p$ , after  $GST$  ( $t_p > GST$ ) with the view number  $view_{high}$  such that: (a)  $p_k$  is the leader of  $view_{high}$ , (b)  $p_k$  has a valid *viewProof* for  $view_{high}$ , (c) no value with a view number higher than  $view_{high}$  is proposed up to time  $t$ , and (d) no proposer proposes a value (with a valid *viewProof*) for the view higher than  $view_{high}$  by  $t_p + D_{OF}$ , where  $D_{OF} = 7\Delta$ . Then, every correct learner eventually learns a value.*

**Lemma 31. (EOFT.)** *The Locking module of our consensus algorithm satisfies Eventual Obstruction-Free Termination property.*

*Proof.* The following proof relies on our assume that, after  $GST$ , a correct acceptor takes any step in negligible time and that every message by a correct process  $p$  to a correct process  $q$  is received within  $\Delta$ .

By our assumption of an eventually synchronous system, all acceptors from  $Q_c$  receive the `new_view` message for  $view_{high}$  sent by  $p_k$ . Moreover, by assumptions (a)-(d) of Definition 10 we conclude that the condition in line 21 (Fig. 15) is satisfied for every acceptor from  $Q_c$ , which then proceeds to execute lines 22-28. By Lemma 29, this part of the code is non-blocking. In case some acceptor from  $Q_c$  sends some `sign_req` message (line 24), it will do so by  $t_p + \Delta$ , and similarly send `sign_ack` messages (line 29) by  $t_p + 2\Delta$ . Hence, all acceptors from  $Q_c$  execute line 28 of Fig. 15 and send the `new_view_ack` messages to  $p_k$  by  $t_p + 3\Delta$ . Denote the set of these `new_view_ack` messages sent by quorum  $Q_c$  (and received by the proposer  $p_k$ ) by  $vProof$ . By Lemma 28, the  $choose(*, vProof, Q_c)$  does not abort, but rather returns some value  $v$ . Therefore, at latest by  $t_p + 4\Delta$ ,  $p_k$  sends the `prepare` $\langle v, view_{high}, vProof, Q_c \rangle$  message to all acceptors — hence, all acceptors from  $Q_c$  receive this message. By assumptions (a) and (d) of Definition 10, we conclude that the condition in line 31, Fig. 15 is satisfied and that all acceptors from  $Q_c$  prepare  $v$  in  $view_{high}$  and send the `update` $_1 \langle v, view_{high}, \emptyset \rangle$  message to all acceptors (and learners) by  $t_p + 5\Delta$ . By assumption (d) of Definition 10, given that all acceptors from  $Q_c$  prepare  $v$  in  $view_{high}$ , we conclude that all acceptors from  $Q_c$  send an `update` $_2 \langle v, view_{high}, Q_c \rangle$  (by  $t_p + 6\Delta$ ), and, later, an `update` $_3 \langle v, view_{high}, Q_c \rangle$  (by  $t_p + 7\Delta = t_p + D_{OF}$ ) to every acceptor and learner. As soon as a correct learner receives a `update` $_3 \langle v, view_{high}, Q_c \rangle$  message from every acceptor of  $Q_c$  it learns a value (lines 53 and 101, fig. 15), unless it already learned a value.  $\square$

We are now ready to prove *Termination*. Basically, what is left to show is that, in every execution in which a correct proposer proposes a value, eventually, the *Election* module ensures that the assumptions of Definition 10 eventually hold.

**Theorem 12. (Termination)** *If a correct proposer proposes a value, then eventually, every correct learner learns a value.*

*Proof.* Suppose, by contradiction, that some correct learner  $l_k$  never learns a value even if some correct proposer, say  $p_k$ , proposes a value.

Note that, if  $p_k$  proposes a value, by (1) lines 0 and 101-103, Fig. 14 and (2) the assumption of an eventually synchronous system, either (a) all correct acceptors eventually trigger their `suspectTimeout` (line 0, Fig. 14), or (b)  $p_k$  receives a `decision` message from some quorum  $Q$  of acceptors and halts (line 104, Fig. 14). In the latter case (case (b)), by Lemma 30, every correct learner eventually learns a value — a contradiction.

We now focus on the case (a), where all correct acceptors eventually trigger `suspectTimeout` (line 0, Fig. 14) — we denote this time by  $t_{trigger}$ . Let  $GST' = \max(GST, t_{trigger})$ .

We distinguish two sub-cases: (i) when no correct acceptor stops its `suspectTimeout` permanently (i.e., no correct acceptor executes the line 7, Fig. 14), and (ii) when some correct acceptor stops its `suspectTimeout` permanently.

We first consider case (i). We define functions  $view_{min}(t) = \min\{nextView_{a_i} | a_i \in Q_c\}$  at time  $t$ , and, similarly,  $view_{max}(t) = \max\{nextView_{a_i} | a_i \in Q_c\}$ . It is not difficult to see (lines 1-5, Fig. 14), that, in case (i), at every correct acceptor  $a_j$ , variable  $nextView_{a_j}$  is: (1) monotonically increasing,

(2) unbounded, and (3) non-skipping (it always increments by one). Hence, every correct acceptor sends an infinite number of `view_change` messages, for every view number from 1 (i.e.,  $initView + 1$ ) to  $\infty$ . Moreover, functions  $view_{min}(t)$  and  $view_{max}(t)$  are also monotonically increasing and unbounded.

Let  $viewGST' = view_{max}(GST')$ . Hence, every correct proposer receives all `view_change` messages sent by acceptors from  $Q_c$  for view numbers  $viewGST' + 1$  and higher. Therefore, every correct proposer  $p_k$  proposes a value in every view  $w_k \geq viewGST' + 1$ , such that  $k = (w_k \bmod |proposers|)$ .

Note that a correct acceptor  $a_i$ , on sending a `view_change` message with a view number  $w$ , at some time  $t_w$ , triggers a timer equal to  $initTimeout * 2^w$  (lines 1-5, Fig. 14). Upon expiration of this timeout,  $a_i$  sends the subsequent `view_change` message. Hence, the time between  $a_i$  sends the `view_change` messages for view numbers  $w$  and  $w + 1$  is at most  $initTimeout * 2^w$ .

Let  $t$  be any point time in time after  $GST'$ . Let  $view(t)$  be the first view in which  $p_k$  proposes a value, such that  $view(t) > view_{max}(t) + 1$ . Note that no acceptor from  $Q_c$  sends a `view_change` message for a view higher than  $view(t)$  before  $T_{OF}(t) = t + initTimeout * 2^{view(t)}$ . By Property 1 of RQS and the proposer code of an *Election* module, we conclude that no proposer can propose a value with a valid *viewProof* and the view number higher than  $view(t)$  before  $T_{OF}(t)$ .

On the other hand, all acceptors from the quorum  $Q_c$  will send the `view_change` message for the  $view(t)$  at latest by  $T_{vc}(t) = t + InitTimeout * (2^{view_{min}(t)} + 2^{view_{min}(t)+1} + \dots + 2^{view(t)-1})$ . These will be received by  $p_k$ , which will propose a value with a view number  $view(t)$  at latest by  $T_{prop}(t) = T_{vc}(t) + \Delta$ .

Therefore,  $p_k$  proposes a value with  $view(t)$  at  $T_{prop}(t) > GST$ , and (a)  $p_k$  is the leader of  $view(t)$ , (b)  $p_k$  has a valid *viewProof* for  $view(t)$  (`view_change` messages from  $Q_c$ ), (c) no value with a view number higher than  $view(t)$  is proposed up to time  $T_{prop}(t)$ , and (d) no proposer proposes a value (with a valid *viewProof*) for the view higher than  $view(t)$  by  $T_{OF}(t)$ . Hence, in order to apply Lemma 31 and reach contradiction, we need to show that there exist  $t'$ , such that  $T_{OF}(t') - T_{prop}(t') > D_{OF}$  (where  $D_{OF} = 7\Delta$ ).

Since  $T_{OF}(t') - T_{prop}(t') = initTimeout * (2^{view(t')} - (2^{view_{min}(t')} + 2^{view_{min}(t')+1} + \dots + 2^{view(t')-1})) - \Delta$ ,  $T_{OF}(t') - T_{prop}(t') > D_{OF} \Leftrightarrow 2^{view_{min}(t')-1} > (D_{OF} + \Delta) / initTimeout = c$ , where  $c$  is a constant. Since  $view_{min}(t)$  is monotonically increasing and unbounded, such  $t'$  exists.

In case (ii) the contradiction follows directly from Lemma 30. □

## C Errata

There was an omission in the conference version of this paper [22] (propagated to [51]), related to the proofs of optimality of atomic storage and consensus algorithms. This caused an error in the statement of Property 3 of RQS.

In the context of Definition 2, the definition in [22] stated that, for a given class 2 quorum  $Q_2$  and quorum  $Q$ ,  $P_{3a}(Q_2, Q, B)$  holds for all  $B \in \mathbf{B}$ , or  $P_{3b}(Q_2, Q, B)$  holds for all  $B \in \mathbf{B}$ . Consequently, the algorithms presented in [22], stated as optimal, are actually not. On the other hand, algorithms presented in this paper are optimal.

We would like to thank the anonymous Distributed Computing journal reviewers for pointing out the above mentioned omission, which allowed us to correct the mistakes from [22, 51].