Refined Quorum Systems

Rachid Guerraoui
IC, EPFL
CH-1015 Lausanne, Switzerland
rachid.guerraoui@epfl.ch

Marko Vukolić
IC, EPFL
CH-1015 Lausanne, Switzerland
marko.vukolic@epfl.ch

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 (just like traditional quorum systems), as well as *optimal best-case complexity* (unlike traditional quorum systems).

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.

Categories and Subject Descriptors

F.2.m [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*Miscellaneous*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Algorithms, Performance, Reliability, Theory

Keywords

Quorums, Consensus, Complexity, Shared-memory emulations, Arbitrary failures

1. INTRODUCTION

Quorum systems are powerful mathematical tools to reason about distributed implementations of shared objects including read/write storage (e.g., [3, 21, 29]) and consensus [6,10,24]. In particular, quorum systems have been used to reason about implementations that tolerate asynchrony and are optimally resilient to process failures. Originally, a quorum system was defined as a set of subsets that intersect [12], 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 [27]: these require larger intersections among subsets (i.e., quorums) [29].

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, specifically the *best-case* one. This is particularly important given the appealing nature of *optimistic* distributed object implementations, e.g., [1,5,8,9,14,15,26,30, 34,39]. In addition to being devised to tolerate worst-case conditions, namely a large number of failures, arbitrarily long periods of asynchrony and contention, 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"* is not captured by traditional quorum systems and it is natural to seek for a mathematical abstraction that captures it in precise yet general terms.

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 (subsets of S) 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 reason 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. 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 [20, 22, 29]. With such a general structure, we relax the often criticized assumption in practice, of independent and identically distributed failures, unlike [1, 5, 8, 9, 14, 15, 26, 30, 34, 39].

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.

- 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 [36]. 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 [18]) and (b) ensuring strong consistency (ideally atomicity [19, 23]). 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 [15, 31] (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 [24], 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 [25] (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 [25] 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 (but we might be subjective).

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. We conclude the paper by pointing out some open research directions. For space restrictions, we postpone the detailed model as well as the full algorithms and their proofs to the full paper [17].

2. REFINED QUORUM SYSTEMS

2.1 Definitions

A refined quorum system is expressed in the context of a non-empty set S of elements, and an *adversary structure* (or, simply, *adversary*) $\boldsymbol{B}$ defined as follows [20]. Let $\boldsymbol{B}$ be any set of subsets of S. $\boldsymbol{B}$ is an *adversary* (for S) if: $\forall B \in \boldsymbol{B}$: $B' \subseteq B \Rightarrow B' \in \boldsymbol{B}$.

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

Definition 1. ***Refined Quorum System.*** *We say that* $\boldsymbol{RQS}$ *is a* refined quorum system *for a set* S *and adversary* $\boldsymbol{B}$*, if* $\boldsymbol{RQS}$ *has two subsets* $\boldsymbol{QC_1} \subseteq \boldsymbol{QC_2} \subseteq \boldsymbol{RQS}$ *such that the following properties hold: (every* $\boldsymbol{QC_i}$ *is called a* quorum class*, and elements of* $\boldsymbol{QC_i}$ *are called* class i elements*)*

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

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

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

- $\forall Q_1, Q_1' \in \boldsymbol{QC_1}$, $\forall Q \in \boldsymbol{RQS}$, $\forall B_1, B_2 \in \boldsymbol{B}$: $Q_1 \cap Q_1' \cap Q \nsubseteq B_1 \cup B_2$.

Property 3. *The intersection of any class 2 element Q_2 and any element Q of $\boldsymbol{RQS}$ is:*
(a) not a subset of the union of any two elements of $\boldsymbol{B}$ (we say $P_{3a}(Q_2, Q)$ holds), or
(b) its intersection with every class 1 element[1] *is not an element of $\boldsymbol{B}$ (we say $P_{3b}(Q_2, Q)$ holds), i.e.,*

- $\forall Q_2 \in \boldsymbol{QC_2}$, $\forall Q \in \boldsymbol{RQS}$, $\forall B_1, B_2 \in \boldsymbol{B}$: $(Q_2 \cap Q \nsubseteq B_1 \cup B_2) \vee$ $\vee\, (\boldsymbol{QC_1} \neq \emptyset \wedge \forall Q_1 \in \boldsymbol{QC_1}$: $|Q_1 \cap Q_2 \cap Q| \notin \boldsymbol{B})$.

We simply call elements of a refined quorum system — *quorums*, and we sometimes refer to any quorum that is not a class 2 quorums as a *class 3* quorum (we write $\boldsymbol{QC_3} = \boldsymbol{RQS}$). Note that class 1 quorums are also class 2 quorums, which are also class 3 quorums. Notice also that, when $\boldsymbol{QC_1} = \boldsymbol{QC_2}$, Property 2 implies Property 3. Furthermore, when $k = 0$, Property 1 implies Property 3. Therefore, Property 3 is interesting on its own only if $k > 0$ and $\boldsymbol{QC_1} \neq \boldsymbol{QC_2}$.

To get an intuition of these properties, we instantiate them here in the context of a *k-bounded threshold adversary*, denoted $\boldsymbol{B_k}$. This is a special case of an adversary that contains all subsets of S with cardinality at most k (i.e., $\boldsymbol{B_k} = \{B | B \subseteq S \wedge |B| \leq k\}$). In this context, the RQS properties can be expressed as follows:

Property 1. Any two quorums intersect in at least $k + 1$ elements.

Property 2. The intersection of any two class 1 quorums intersects with any quorum in at least $2k + 1$ elements.

Property 3. Any class 2 quorum intersects with any quorum in at least $2k+1$ elements or this intersection itself intersects with any class 1 quorum in at least $k + 1$ elements.

2.2 Examples

Example 1. Figure 1 depicts a simple illustration of a RQS for the *1-bounded threshold adversary* $\boldsymbol{B_1}$: 4 quorums are involved. As depicted by the example, the cardinality of a quorum is not always a good indication of its class: it is the intersection with others that matters. Quorum Q_1 contains 5 elements and is a class 1 quorum, while Q' contains 6 elements yet is only a class 3 quorum.

In the following, we give more illustrations of our refined quorum system notion by explaining how it extends traditional quorum systems. 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 $\boldsymbol{B}$ 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 $\boldsymbol{Q_i}$ the set of subsets of S that contains all

[1]Assuming there is at least one class 1 element, i.e., $\boldsymbol{QC_1} \neq \emptyset$.

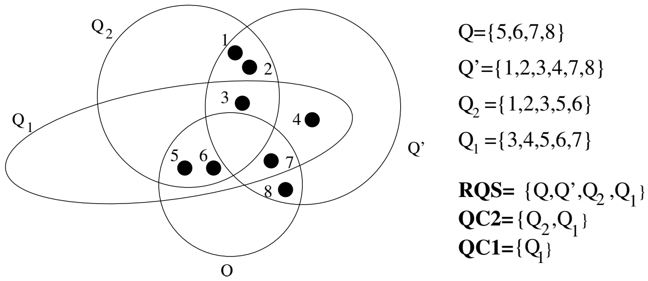


Figure 1: Example of a RQS for an adversary $\boldsymbol{B_k}$ ($k = 1$). Every pair of depicted sets intersects in at least $k + 1$ elements (satisfying Property 1). Q_1 intersects with every other set in at least $2k + 1$ elements (satisfying Property 2, for an intersection with itself). Moreover, $P_{3a}(Q_2, Q')$ and $P_{3a}(Q_2, Q_1)$ hold (since $|Q2 \cap Q'| = 2k + 1 = |Q2 \cap Q1|$) as well as $P_{3b}(Q_2, Q)$ (since $|Q_2 \cap Q \cap Q_1| = k + 1$). Hence, $\boldsymbol{RQS} = \{Q, Q', Q_2, Q_1\}$ is a refined quorum system, where Q_1 (resp., Q_2) is a class 1 (resp., class 2) quorum.

subsets of S that contain all but at most i elements of S, i.e., $\boldsymbol{Q_i} = \{P | P \subseteq S \wedge |P| \geq |S| - i\}$.

Example 2. Consider the case where: (a) $\boldsymbol{B} = \emptyset$, (b) $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \emptyset$ and (c) $\boldsymbol{RQS} = \boldsymbol{Q_{\lfloor (|S|-1)/2 \rfloor}}$ (i.e., every majority of S is a quorum). Property 1 is trivially satisfied. So are Properties 2 and 3, since $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \emptyset$. This quorum system is typically used when devising algorithms that tolerate (a minority of) crash-failures, e.g., [3,6,12,24,32,37].

Example 3. Consider the case of an adversary $\boldsymbol{B_{\lfloor (|S|-1)/3 \rfloor}}$, where (a) $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \emptyset$ and (b) $\boldsymbol{RQS} = \boldsymbol{Q_{\lfloor (|S|-1)/3 \rfloor}}$. In this case, each quorum contains more than two thirds of processes and satisfies Property 1. Properties 2 and 3 are also satisfied (since $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \emptyset$). Such a quorum system is typically used to tolerate (up to one third of) Byzantine failures, e.g., [5, 8, 31, 33].

Example 4. A refined quorum system for which $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \emptyset$ is a *disseminating quorum system* in the sense of [29]. In [29], disseminating quorum systems are used to build resilient distributed services that store authenticated (also called self-verifying) data (in short, such data is assumed to be unforgeable). On the other hand, a refined quorum system in which $\boldsymbol{QC_1} = \emptyset$ and $\boldsymbol{QC_2} = \boldsymbol{RQS}$ is a *masking quorum system* in the sense of [29]. 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 $\boldsymbol{QC_1} = \emptyset$. In the rest of the paper, we study the more general case where $\boldsymbol{QC_1} \neq \emptyset$. This is the case where RQS capture both the resilience and the best-case complexity dimensions of distributed algorithms.

Example 5. Consider the case of a refined quorum system where $\emptyset \neq \boldsymbol{QC_1} = \boldsymbol{QC_2}$. Such a RQS corresponds to the quorum system used in [26] for the specific case $\boldsymbol{B} = \emptyset$, 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 $\boldsymbol{B_k}$, where (a) $\boldsymbol{RQS} = \boldsymbol{Q_t}$, and (b) $\boldsymbol{QC_1} = \boldsymbol{QC_2} = \boldsymbol{Q_q}$ ($q \leq t$), 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 [25]. The special case of this RQS where $k = q = t$ (i.e., where $\boldsymbol{QC_1} = \boldsymbol{RQS}$) corresponds to the quorum system used in [1, 30], also with the goal of boosting consensus latency.

Example 6. Even more interesting is the general case where $\emptyset \neq \boldsymbol{QC_1} \neq \boldsymbol{QC_2} \subseteq \boldsymbol{RQS}$ (e.g., Fig. 1), especially when $\boldsymbol{RQS}$, $\boldsymbol{QC_1}$ and the adversary are defined as in Example 5, $\boldsymbol{QC_2} = \boldsymbol{Q_r}$, 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) elements. $\boldsymbol{RQS}$ 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., $\boldsymbol{RQS}$ 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 [9,15], and later in [39].

In the following, we describe two new algorithms that make full and explicit use of our abstract notion of RQS. The full algorithms and their correctness proofs are given in the full paper [17].

3. ATOMIC STORAGE

We show in this section how to use a refined quorum system to wait-free [18] implement the abstraction of a single-writer multi-reader (SWMR) atomic [23] 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 no-contention (best-case conditions). Our storage algorithm tolerates Byzantine servers yet does not rely on any data authentication primitive.[2]

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 the complexity of atomic storage implementations that are optimally resilient.

3.1 Preliminaries

A distributed storage (or, simply, storage) can be viewed a read/write abstraction implemented by a finite set of processes called servers, and a distinct, potentially unbounded, set of processes called *clients*. A process p (client or server) is modeled as an I/O automata [28]. A computation of a process p proceeds in *steps*. The steps include *communication* and *local* steps. For simplicity, we assume that a correct process performs local steps in negligible time w.r.t to communication steps. A *correct* process p is one that executes an infinite number of steps. 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 process is *Byzantine* if it is not benign. We define an *execution* of a storage algorithm in the vein of [28] (we discuss few model differences in the full paper [17]).

We assume asynchronous (no bound on communication delays), yet reliable point-to-point channels relating clients and servers (servers do not communicate among each other). We assume a global clock that is, however, not accessible to the processes. We say that the system is *synchronous* during time interval $[t, t']$ if for every two correct processes p_1 and p_2, every message sent by p_1 at time $t_1 \in [t, t']$ is received by a process p_2 at time $t_1 + \Delta \in [t, t']$, where the constant Δ ($\Delta > 0$) is known to all correct processes.

The set *clients* is a union of two distinct sets: a singleton *writer* and the set *readers*. 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 assume that the storage is initialized to a special value $\perp$, which is not a valid input of a write operation.

An atomic storage provides the illusion of sequential accesses by ensuring the linearizability of read/write operations [19, 23] (when there is no risk of ambiguity we say *operation* when we should be saying *operation execution*). We focus on *wait-free* [18] atomic storage algorithms in which all read/write operations invoked by a correct client eventually complete. We do not explicitly model the invocation and response steps of read/write operations. We simply say that an operation *op* invoked by the client c is *complete* if c takes a response step for *op*. 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 interval between the invocation and completion of *op*. No client c invokes a new operation before all operations previously invoked by c have completed.

3.2 Algorithm

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

Our algorithm is *round-based* (as in [2,3]), i.e., each operation *op* (read or write) proceeds in series of *communication round-trips* (or, simply, *rounds*). In such algorithms, in each round of *op*, (a) a client c sends a message to some (possibly to all) servers, (b) a server, on receiving a message from c, responds to c without waiting for any other message and (c) c receives a response from some subset of servers (the decision on when a client proceeds to the next round is algorithm-specific). No other messages are exchanged; in particular, servers can send only response messages to clients. (See the full paper [17] for more details).

We express the latency (complexity) of an operation *op* in terms of the number of rounds elapsed between the invocation and the completion of *op*.

[2] Although powerful, such primitives do not provide deterministic guarantees, and might require (1) an infrastructure for key management (for solutions based on symmetric cryptography, e.g., [4]), or (2) non-negligible complexity of data encryption (e.g., [35]). These typically introduce overhead that one would like to avoid, especially in the best-case scenarios.

```
write(v) is                                                   writer code
1:  inc(ts); QC'_2 := ∅; round(1)
2:  if some class 1 quorum replied then return(OK)
3:  QC'_2 := set of all class 2 quorums that replied in round 1
4:  round(2)
5:  if some quorum from QC'_2 replied then return(OK)
6:  QC'_2 := ∅; round(3); return(OK)

round(j) is
10:  send wr message containing ts, v, QC'_2 and j to all servers
11:  wait for (replies from some quorum) and (timeout (if j ∈ {1,2}))
------------------------------------------------------------------------
20:  BCD(⟨ts,v⟩,1,j) ::= ∃Q_1 ∈ QC_1, ∃Q_j ∈ QC_j, ∀s_i ∈ (Q_1 ∩ Q_j) :
                  history[i,ts,j] contains v and (if j = 2) Q_j
21:  BCD(⟨ts,v⟩,2,j) ::=
   ::= {Q_2 | (Q_2 ∈ QC'_2) ∧ (∃Q_j ∈ QC_j, ∀s_i ∈ (Q_2 ∩ Q_j) :
          history[i,ts,j] contains v}
read() is                                                     reader code
30:  repeat
31:    inc(round); send rd message containing round to all servers
32:    wait for (replies from some quorum) and (timeout (in round 1))
33:    if (round = 1) then
         QC'_2 := set of all class 2 quorums that replied in round 1
34:  until there is a safe timestamp/value pair ⟨ts,v⟩ such that all
          other pairs with the higher timestamp are invalid

40:  if ∃j ∈ {1,2,3} : BCD(⟨ts,v⟩,1,j) and (round = 1) then
       return(v)
41:  if ∃j ∈ {1,2,3} : BCD(⟨ts,v⟩,2,j) ≠ ∅ and (round = 1) then
42:    first_writeback_round(⟨ts,v⟩, BCD(⟨ts,v⟩,2,∗))
43:    if some quorum from BCD(⟨ts,v⟩,2,∗) replied then
         return(v)
44:    second_writeback_round(⟨ts,v⟩,∅); return(v)
45:  both_writeback_rounds(⟨ts,v⟩,∅); return(v)

50:  upon received read_ack containing round, history H_i from s_i do
51:    history[i,∗,∗] := H_i[∗,∗]
```

Figure 2: Efficient atomic storage: simplified code of the writer and the readers

Let $\boldsymbol{P}$ be any set of subsets of S.

DEFINITION 2. $(m, \boldsymbol{P})$–**fast *storage algorithm*.** *Consider any synchronous and uncontended operation op invoked by a correct client. We say that a storage algorithm A is* $(m, \boldsymbol{P})$–fast *if in every execution of A in which some set* $P \in \boldsymbol{P}$ *contains only correct servers, op completes in at most m rounds, without using data authentication.*

Our storage algorithm is $(m, \boldsymbol{QC_m})$*–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 (simplified) pseudocode of the algorithm is given in Figure 2 (the full version is in [17]). The write (lines 1-11) consists of at most 3 rounds. In every round, the writer awaits replies from some quorum, and in the first two rounds, awaits for a timeout set to $2 * \Delta$. Upon reception of a wr message containing a timestamp ts, a value v, a set of quorums $\boldsymbol{QC'_2}$ and a round number j, a server s_i stores this data in its history matrix H_i, by storing v and $\boldsymbol{QC'_2}$ into the variable $H_i[ts, j]$.[3]

The read code is more involved. At its heart lies a *Best-Case Detector* abstraction (BCD, lines 20-21). Roughly, BCD examines quorum intersections to detect synchronous and uncontended reads.

A reader first invokes series of rounds (30-34) until some timestamp/value pair $c = \langle ts, v\rangle$ is *safe*, i.e., is confirmed in the history of some subset of servers $S' \notin \boldsymbol{B}$, such that all values v' with a $ts' > ts$ are *invalid*. A sufficient condition for a pair $\langle ts', v'\rangle$ to become *invalid* is that the last response of any quorum contains only values with timestamps smaller than ts' (or a pair $\langle ts', v'' \neq v\rangle$).[4] This holds at the end of the first round of any synchronous and uncontended read (in the following, we consider such a read rd). In this case, lines 30-34 are executed only once.

Then, the reader queries BCD. Assume the last write that precedes the read rd completed in j rounds writing v with timestamp ts. If the reader receives replies from a class 1 quorum in round 1, then $BCD(\langle ts, v\rangle, 1, j)$ holds (line 20) and rd completes at the end of round 1 (line 40). Else, if the reader receives replies from some class 2 quorum Q_2, then the set $X = BCD(\langle ts, v\rangle, 2, j)$ is non-empty (since $Q_2 \in X$) and rd proceeds to round 2. In round 2, the reader *writesback* (line 42) the information about the set X and the pair $\langle ts, v\rangle$ to servers and awaits for replies from some quorum from X in round 2. If no quorum from X replies, then the second round of writeback is invoked (line 44). Note that the read takes at most 2 rounds after line 34, so an uncontended and synchronous read rd always completes in at most 3 rounds.

3.3 Optimality

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

THEOREM 1. *If an algorithm A is* $(\boldsymbol{Q^{(3)}}, \boldsymbol{B})$–atomic*, then* $P1(\boldsymbol{Q^{(3)}})$ *holds.*

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

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

Together, Theorems 1–3 convey the optimal resilience and best-case complexity of our algorithm of Figure 2. Theorems 1 and 2 have been established for the special case of threshold-based quorums and with an implicit notion of quorums in [31] and [15], respectively. It is not difficult to extend these bounds to non-threshold quorums and the general adversary structure. Here we prove the novel Theorem 3, which is particularly interesting due to the unusual *or* condition that appears in Property 3 of RQS.

[3] To simplify our algorithm, we assume that servers store the entire history of the shared storage they are implementing. In fact, in our model, achieving atomic semantics (and even a weaker, regular [23] one) while precluding servers from storing the entire history, is unfeasible without using some non-trivial signalling scheme between readers and the writer [2, 7]. These techniques (e.g., [15]) are orthogonal to our latency efficient scheme based on RQS and integrating them here might obfuscate the point of this paper.

[4] The necessary condition for a value to be *invalid* is vital for wait-freedom and is more involved. See the full paper for details [17].

PROOF. Theorem 3 states that there is no $(\boldsymbol{Q^{(3)}}, \boldsymbol{B})$–*atomic* storage algorithm that is both $(1, \boldsymbol{Q^{(1)}})$–*fast* (for some $\boldsymbol{Q^{(1)}} \neq \emptyset$) and $(2, \boldsymbol{Q^{(2)}})$–*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 of servers S. Negating $P3(\boldsymbol{Q^{(1)}},\boldsymbol{Q^{(2)}},\boldsymbol{Q^{(3)}})$ yields (having in mind $\boldsymbol{Q^{(1)}} \neq \emptyset$):

$$\exists Q_2 \in \boldsymbol{Q^{(2)}},\ \exists Q \in \boldsymbol{Q^{(3)}},\ \exists Q_1 \in \boldsymbol{Q^{(1)}},\ \exists B_1, B_2 \in \boldsymbol{B}: \\ ((Q_2 \cap Q) \subseteq (B_1 \cup B_2) \wedge (Q_2 \cap Q \cap Q_1) \in \boldsymbol{B}.$$

In the following, we denote the set $Q_2 \cap Q \cap Q_1$ by B_0 (note that $B_0 \in \boldsymbol{B}$). Since $B_0 \subseteq Q_2 \cap Q$ and $\boldsymbol{B}$ is an adversary for S, we may assume, without loss of generality, that $B_0 \subseteq B_1$. 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 3) 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.

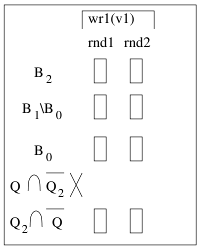
- Let ex_1 be the execution in which all servers from Q_2 are correct, while all others (i.e., those from $\overline{Q_2}$) fail by crashing at the beginning of the execution. Furthermore, let wr_1 be the write operation invoked at time t_1 by the correct writer w in ex_1 to write a value $v_1 \neq \perp$ in the storage. Moreover, assume that the system is synchronous in ex_1. Hence, wr_1 is synchronous and uncontended. Since A is $(2, \boldsymbol{Q^{(2)}})$–*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, messages sent by the writer to servers in $\overline{Q_2}$ are not delivered by the end of ex'_1 (we say these messages are in *transit* in ex'_1). Since the writer cannot distinguish ex'_1 from ex_1, wr_1 completes in ex'_1, in two communication rounds, at time t'_1.

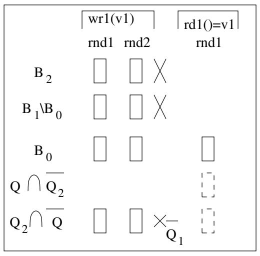
- Let the partial execution ex_2 extend ex'_1 such that: (1) servers from $\overline{Q_1}$ crash at t'_1, (2) rd_1 is a synchronous read operation invoked by the correct reader r_1 after t'_1, and (3) no other operation is invoked (hence, rd_1 is uncontended). Since A is $(1, \boldsymbol{Q^{(1)}})$–*fast*, 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.

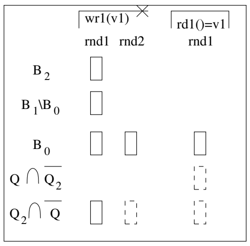
- Let ex''_2 be the partial execution identical to ex'_2 except that, in ex''_2: (1) the writer crashes during wr_1 and its



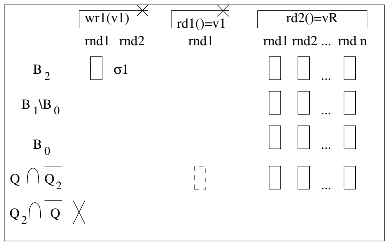
(a) ex_1



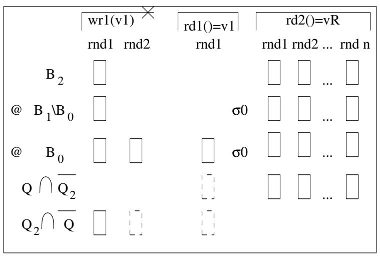
(b) ex_2



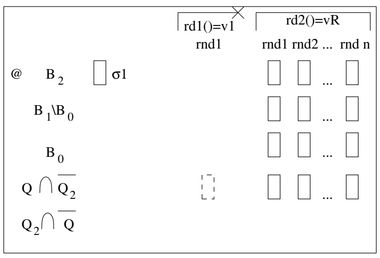
(c) ex''_2



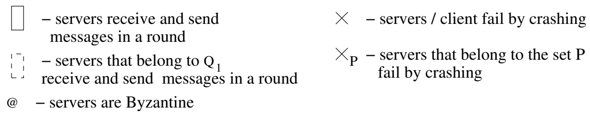
(d) ex_3



(e) ex_4



(f) ex_5



(g) Legend

Figure 3: 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.

round 2 messages are not received by any server from $\overline{Q_1} \cup \overline{Q_2}$ (i.e., only servers from $Q_1 \cup Q_2$ 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 the servers from the set B_2 (resp., $Q \cap \overline{Q_2} \cap Q_1$). We refer to the state of the servers that belong to the set B_2 *after* sending the reply to the round 1 message of wr_1 as to σ_1. In ex_3, all servers are correct except those from the set $\overline{Q}$ that fail by crashing at the beginning of the 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 Q are correct in ex_3 and A is a $(\boldsymbol{Q^{(3)}}, \boldsymbol{B})$–*atomic* storage algorithm, rd_2 eventually completes, at some point in time t_3, after n communication rounds and returns the 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 the servers from $\overline{Q}$ to r_2 are delayed until after t_3 (i.e., until after n^{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 σ_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 the 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 σ_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^{th} round of rd_2). The reader r_2 and the 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. □

4. CONSENSUS

In our storage algorithm, we assumed that (1) processes that access a RQS (the clients) might crash but cannot be Byzantine, and (2) processes that form a RQS (servers) do not communicate directly with each other. In this section, we illustrate the case where (1) the processes accessing a RQS might be Byzantine and (2) processes that form a RQS may directly communicate. Namely, we consider the *consensus* problem in the general framework of [24], composed of three sets of processes: *proposers*, *acceptors* and *learners*. Roughly, proposers propose values that are to be agreed upon by learners, where the role of acceptors is to help learners reach the decision. In this paper, as in [38], we assume that the set *acceptors* does not intersect with the set *proposers* $\cup$ *learners*.

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. The algorithm is optimal in terms of resilience as well as complexity, matching the lower bounds of [25].[5] Our algorithm expedites the consensus decision under best-case conditions without using data authentication primitives; however, when best-case conditions are not met, data authentication primitives are used.

Many of the modeling concepts used in this section are borrowed from the previous one (Section 3.1). So we do no recall them here. 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: (a) *Validity*: If a benign learner learns a value v and all proposers are benign, then some proposer proposes v;[6] (b) *Agreement*: No two benign learners learn different values; and (c) *Termination*: If a correct proposer proposes a value, then eventually, every correct learner learns a value.

4.1 Algorithm

Our algorithm relies on acceptors forming a refined quorum system $\boldsymbol{RQS}$ for an adversary $\boldsymbol{B}$, such that $\boldsymbol{RQS}$ is known to all processes. Consensus *safety* (i.e., Validity and Agreement) is guaranteed as long as the set of Byzantine acceptors in any execution belongs to $\boldsymbol{B}$, while consensus *liveness* (i.e., Termination) is ensured if there is a correct quorum of acceptors $Q_c \in \boldsymbol{RQS}$ and if the system is eventually synchronous [10] (this is crucial to circumvent the impossibility of [11]). Any number of proposers and learners can be Byzantine. We denote by $\langle m \rangle$ an unauthenticated message, and by $\langle m \rangle_{\sigma_x}$ the message digitally signed [35] by process x.

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 $\boldsymbol{P}$ be any set of subsets of *acceptors*.

Definition 3. $(m, \boldsymbol{P})$–**fast** ***consensus algorithm***. *We say that a consensus algorithm is* $(m, \boldsymbol{P})$–fast *if in every best-case execution ex in which some set* $P \in \boldsymbol{P}$ *contains*

[5] The notion of complexity considered here is again best-complexity for this is considered practically appealing as we discussed earlier, and the worst-case complexity of a consensus algorithm that tolerates arbitrarily long periods of asynchrony is anyway unbounded.

[6] The *Validity* property, as stated in [25], "Only a value proposed by a proposer can be learned", is clearly impossible to ensure in the presence of Byzantine proposers.

only correct acceptors, all correct learners learn v in $m+1$ message delays, without using authenticated messages.

Our consensus algorithm is $(m, \boldsymbol{QC_m})$*–fast* for all $m \in \{1,2,3\}$. The algorithm consists of two modules: (1) a *Locking* module (Fig. 4) that ensures safety, and (2) an *Election* module[7] for liveness. The algorithm proceeds in a sequence of *views*. In every view, a single proposer is the *leader*, except in the initial view ($initView$) where all proposers can be seen as leaders (we assume that all acceptors are initially in $initView$). The *Locking* module consists of a *consult* (lines 1-6 and 11-20, Fig. 4) and an *update* phase (lines 7, 21-28 and 31-34 Fig. 4). In $initView$, the leader, on proposing a value, skips the consult phase and executes immediately the update phase.

Roughly, in a best-case execution, the update phase proceeds as follows: (1) the proposer p sends a message $m = \mathsf{prepare}$ containing its proposal value v and view number $view = initView$ to all acceptors (line 7, Fig. 4), (2) correct acceptor a_j, upon receiving m that contains the value v and view number $view$, performs some local computations (we say a_j *prepares* the value v in $view$ — line 22, Fig. 4), and then echoes v and $view$ using a update_1 message to all acceptors and learners (lines 23, Fig. 4), (3) correct acceptor a_j, upon receiving update_1 messages from some quorum Q performs again some local computations (we say a_j *1-updates* the value v in $view$ with quorum Q) and send update_2 messages along with the id of Q to all acceptors and learners (lines 25-28, Fig. 4), and (4) acceptors, similarly to step (3), upon receiving update_2 from some quorum, *2-update* the value v in $view$ and send update_3 to all acceptors (this is however done only once, and not per every quorum of received update messages, as is the case with 1-updating the value). If some class 1 (resp., class 2, class 3) quorum contains only correct acceptors, then all correct learners and acceptors *decide* v in $initView$ after two (resp., three, four) message delays (line 31 (resp., 32, 33), Fig. 4). Since learners learn a value upon deciding it (line 34), we ensure that our algorithm is $(m, \boldsymbol{QC_m})$*–fast* for all $m \in \{1,2,3\}$.

In a best-case execution, the *Election* module, responsible for view changes, does not change the view before all correct learners learn v. However, if the proposer p is Byzantine, or the system is asynchronous, then the *Election* module might designate a different proposer p_w to be the leader for the new view w (line 8, Fig. 4). Process p_w starts the consult phase for view number w by sending the $\mathsf{new\_view}$ message to acceptors (line 2, Fig. 4). Then, p_w waits for a quorum Q of *valid* signed $\mathsf{new\_view\_ack}$ messages (or, simply, *acks* — line 3) containing the last prepared and *step*-updated values (where $step \in \{1,2\}$) and the corresponding view numbers. We say that an ack is *valid* if the following conditions holds (for $step \in \{1,2\}$) (1) if an acceptor a_j reports that it *step*-updated some value v in $view'$, then a_j also reports that it prepared v in $view'$, or some other value in a view higher than $view'$, and (2) the last values *step*-updated in some view $view'$ are accompanied with a set of corresponding signed update_{step} messages from some subset of acceptors that is not an element of $\boldsymbol{B}$ (obtained in lines 13-17, Fig. 4).[8] Then, the leader evaluates acks from

[7] The *Election* module is based on well-known leader election techniques, and is similar to that of [5]. For space limitations, the pseudocode of the *Election* module is postponed to the full paper [17].

[8] This technique can be seen as a generalization of the "lazy

---

at every proposer p_j:

Initialization:
$view := initView$; $viewProof, vProof := nil$; $\boldsymbol{faulty} := \emptyset$

propose(v) **is**
1: **if** $(view \neq initView)$ **then** % *consult* phase
2: send $\mathsf{new\_view}\langle view, viewProof\rangle$ to $acceptors$
3: **wait for** valid acks from some quorum $Q \in \boldsymbol{RQS} \setminus \boldsymbol{faulty}$
4: $vProof :=$ set of received acks from Q
5: $(v, abort) := \mathbf{choose}(v, vProof, Q)$
6: **if** $abort$ **then** $\boldsymbol{faulty} := \boldsymbol{faulty} \cup \{Q\}$; **goto** 3
7: send $\mathsf{prepare}\langle v, view, vProof, Q\rangle$ to $acceptors$ % *update* phase

upon p_j is elected by *Election* module
8: obtain new $view$, $viewProof$ from *Election* module; **propose**(v)

---

at every acceptor a_j:

% *consult phase* (lines 11-20)
upon received $\mathsf{new\_view}\langle view, viewProof\rangle$ from p_i
11: **if** $(view > view_{a_j})$ **and** $(viewProof$ matches $view)$ **then**
12: $view_{a_j} := view$
13: $v_{step} :=$ the last value *step*-updated by a_j (for $step \in \{1,2\}$)
14: **forall** $view'$ such that v_{step} is *step*-updated in $view'$ **and** **and** $Proof(v_{step}, view')$ is empty **do**
15: send $\mathsf{sign\_req}\langle m' = \mathsf{update}_{step}\langle v_{step}, view', *\rangle\rangle$ to some \ quorum with which a_j *step*-updated v_{step} in $view'$
16: **wait for** acks with a valid signature from \ some subset of $acceptors$, $T_{m'}$, such that $T_{m'} \notin \boldsymbol{B}$
17: $Proof(v_{step}, view') :=$ received signatures from $T_{m'}$
18: **end do**
19: send $\mathsf{new\_view\_ack}_{\sigma_{a_j}}$ message to p_i, containing \ $view_{a_j}$, the last prepared and *step*-updated values with proofs

upon received $\mathsf{sign\_req}\langle m'\rangle$ from a_i
20: **if** $m' \in old$ **then** send $\mathsf{sign\_ack}\langle m'\rangle_{\sigma_{a_j}}$ to a_i

% *update phase* (lines 21-28 and 31-34)
upon received $m = \mathsf{prepare}\langle v, view_{a_j}, vProof, Q\rangle$ from p_i
21: **if** no value is prepared in $view_{a_j}$ **and** **and** $((p_i$ is leader **and** v matches $\mathbf{choose}(v, vProof, Q))$ **or** **or** $view_{a_j} = initView)$ **then**
22: $prepare(v, view_{a_j})$ % we say: a_j prepares v in $view_{a_j}$
23: send $m_1 = \mathsf{update}_1\langle v, view_{a_j}, \emptyset\rangle$ to $acceptors \cup learners$
24: $old := old \cup m_1$

upon received $m = \mathsf{update}_{step}\langle v, view_{a_j}, *\rangle$ from some quorum Q **and** **and** a_j prepared v in $view_{a_j}$ (for some $step \in \{1,2\}$)
25: $step\text{-}update(v, view_{a_j}, Q)$ % we say a_j *step*-updates v in $view_{a_j}$ (with quorum Q)
26: **if** $step = 1$ **or** m is the first update_2 msg. received in $view_{a_j}$ **then**
27: send $m_{step+1} = \mathsf{update}_{step+1}\langle v, view_{a_j}, Q\rangle$ to $acceptors \cup learners$
28: $old := old \cup m_{step+1}$

---

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

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

upon received the same $\mathsf{update}_3\langle v, view, *\rangle$ from $Q \in \boldsymbol{RQS}$
33: **if** x has not yet decided **then decide**(v)

---

at every learner l_j:
34: **upon** l_j decides v **learn**(v)

Figure 4: Simplified pseudocode of the *Locking* module of the consensus algorithm

Q using the *choose*() function (line 5, Fig. 4). 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 prepare (and step-update) only v.* The details of this function are postponed to the full paper [17].

4.2 Optimality

We say that an algorithm A implements $(\boldsymbol{Q},\boldsymbol{B})$*–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 $\boldsymbol{B}$, as well as Termination in case the system is eventually synchronous and there is a set $Q \in \boldsymbol{Q}$ that contains only correct acceptors. In addition, analogously to Section 3.3, we define the properties $P1(\boldsymbol{Q^{(3)}})$, $P2(\boldsymbol{Q^{(1)}},\boldsymbol{Q^{(3)}})$ and $P3(\boldsymbol{Q^{(1)}},\boldsymbol{Q^{(2)}},\boldsymbol{Q^{(3)}})$, where $\boldsymbol{Q^{(i)}}$ is some set of subsets of *acceptors*. The following theorems capture the minimality of our RQS, assuming $|proposers| \geq 2$ and $|learners| \geq 3$.[9]

THEOREM 4. *If an algorithm A implements $(\boldsymbol{Q^{(3)}},\boldsymbol{B})$–*consensus*, then $P1(\boldsymbol{Q^{(3)}})$ holds.*

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

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

In the special threshold case, where (a) $\boldsymbol{B}{=}\boldsymbol{B_k}$, (b) all elements of $\boldsymbol{Q^{(1)}}$ (resp., $\boldsymbol{Q^{(3)}}$) contain all but at most q (resp., t) acceptors, and (c) $q = t-2k$, Theorems 4–5 reduce to the lower bounds stated in [25]. In the full paper [17] we prove the novel optimality result established by Theorem 6.

5. 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[10] (i.e., 1 round in the case of storage, or 2 message delays in the case of (Byzantine and asynchronous [25]) 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: (a) the load and availability of RQS [32], (b) how RQS can be optimally placed in the network [13], (c) how many RQS can be found given some adversary structure, (d) how to devise algorithms that cope with unknown RQS/adversary structures, and (e) 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 [22]).

Moreover, it would be interesting to carefully look into non-atomic semantics, e.g., regular or safe storage [23]. 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, 16] 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 $\boldsymbol{QC_1} = \emptyset$. Finally, it would also be interesting to look into atomic object implementations that use authentication in best-case executions. The lower bounds of [25], 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 $\boldsymbol{QC_2} = \boldsymbol{QC_1}$.

Acknowledgments

We thank Hagit Attiya, Christian Cachin, Christof Fetzer, Petr Kouznetsov, Ron R. Levy, Dahlia Malkhi, Eric Ruppert and anonymous reviewers for their very helpful comments.

proof obtaining" technique of [5].

[9] We exclude here the special cases where $|proposers| = 1$, $|learners| \leq 2$ or $acceptors \cap (proposers \cup learners) \neq \emptyset$). These have to be addressed separately.

[10] This can be measured by the best possible latency in synchronous, uncontended and failure-free situations.

6. REFERENCES

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 59–74, New York, NY, USA, 2005. ACM Press.

[2] I. Abraham, G. V. Chockler, I. Keidar, and D. Malkhi. Byzantine disk paxos: optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[4] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. *Lecture Notes in Computer Science*, 1666:216–233, 1999.

[5] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, USA, February 1999.

[6] T. D. Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[7] G. Chockler, R. Guerraoui, and I. Keidar. On the space requirements of robust storage implementations. In *Dagstuhl Seminar From Security to Dependability*, September 2006.

[8] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementations*, Seattle, Washington, November 2006.

[9] P. Dutta, R. Guerraoui, and M. 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.

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

[11] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[12] D. K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM Press.

[13] D. Golovin, A. Gupta, B. M. Maggs, F. Oprea, and M. K. Reiter. Quorum placement in networks: Minimizing network congestion. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 16–25, New York, NY, USA, 2006. ACM Press.

[14] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, p. 135–144, 2004.

[15] R. Guerraoui, R. R. Levy, and M. Vukolić. Lucky read/write access to robust atomic storage. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 125–136, Washington, DC, USA, 2006. IEEE Computer Society.

[16] R. Guerraoui and M. Vukolić. How Fast Can a Very Robust Read Be? In *25th ACM Symposium on Principles of Distributed Computing (PODC'06)*, 2006.

[17] R. Guerraoui and M. Vukolić. Refined quorum systems. Technical Report LPD-REPORT-2007-002, Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland, February 2007.

[18] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

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

[20] M. Hirt and U. Maurer. Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 25–34, New York, NY, USA, 1997. ACM Press.

[21] P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, 1998.

[22] F. P. Junqueira, K. Marzullo. Synchronous Consensus for Dependent Process Failures. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03)*, pages 274 - 283, Providence, RI, USA, May 2003.

[23] L. Lamport. On interprocess communication. *Distributed computing*, 1(1):77–101, May 1986.

[24] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[25] L. Lamport. Lower bounds for asynchronous consensus. In *Future Directions in Distributed Computing*, Springer Verlag (LNCS), pages 22–23, 2003.

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

[27] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[28] N. A. Lynch and M. R.Tuttle. An introduction to I/O automata. *CWI Quarterly*, 2(3):219–246, 1989.

[29] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[30] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

[31] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 311–325. Springer-Verlag, 2002.

[32] M. Naor and A. 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.

[33] M. Pease, R. Shostak, and L. Lamport. Reaching agreements in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[34] H. V. Ramasamy and C. Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS 2005)*, Lecture Notes in Computer Science, pages 88–102, December 2005.

[35] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[36] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. Fab: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev.*, 38(5):48–58, 2004.

[37] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[38] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 253–267, New York, NY, USA, 2003.

[39] P. Zieliński. Optimistically terminating consensus. Technical Report UCAM-CL-TR-668, Cambridge University, Cambridge, UK, June 2006.