

# Byzantine and Multi-writer $K$ -Quorums<sup>\*</sup>

Amitanand S. Aiyer<sup>1</sup>, Lorenzo Alvisi<sup>1</sup>, and Rida A. Bazzi<sup>2</sup>

<sup>1</sup> Department of Computer Sciences,  
The University of Texas at Austin

{anand, lorenzo}@cs.utexas.edu  
<sup>2</sup> Computer Science and Engineering Department,  
Arizona State University  
bazzi@asu.edu

**Abstract.** Single-writer  $k$ -quorum protocols achieve high availability without incurring the risk of read operations returning arbitrarily stale values: in particular, they guarantee that, even in the presence of an adversarial scheduler, any read operation will return the value written by one of the last  $k$  writes. In this paper, we expand our understanding of  $k$ -quorums in two directions: first, we present a single-writer  $k$ -quorum protocol that tolerates Byzantine server failures; second, we extend the single-writer  $k$ -quorum protocol to a multi-writer solution that applies to both the benign and Byzantine cases. For a system with  $m$  writers, we prove a lower bound of  $((2m - 1)(k - 1) + 1)$  on the staleness of any multi-writer protocol built over a single-writer  $k$ -quorum system and propose a multi-writer protocol that provides an almost matching staleness bound of  $((2m - 1)(k - 1) + m)$ .

## 1 Introduction

Quorum systems have been extensively studied, with applications that include mutual exclusion, coordination, and data replication in distributed systems [1,2,3,4]. A traditional, or strict, quorum system is simply a collection of servers organized in sets called quorums. Quorums are accessed either to write a new value to a *write quorum* or to read the values stored in a *read quorum*: in strict quorums, any read quorum intersects with a write quorum.

Important quality measures of quorum systems are *availability*, *fault tolerance*, *load*, and *quorum size*. lower size have measures are conflicting in strict quorum systems [5]. For instance, the majority quorum system provides the highest availability of all strict quorum systems when the failure probability of individual nodes is lower than 0.5, but it also suffers from high load and large quorum size—and this tension holds true in general [6]. When the failure probability of individual nodes is higher than 0.5, the quorum system with highest availability is the singleton, in which one node handles all requests in the system.

Probabilistic [7] and signed [8] quorum systems have been proposed to achieve high availability while guaranteeing system consistency (non-empty intersection of

---

<sup>\*</sup> This work was supported in part by NSF Cybertrust award 0430510, NSF award CNS 0509338, and a grant from the Texas Advanced Technology Program.

quorums) with high probability. These probabilistic constructions offer much better availability than the majority system at the cost of providing only probabilistic guarantees on quorum intersection. If a probabilistic quorum system is used to implement a distributed register with read and write operations, then, with high probability, a read operation will return the value most recently written.

To achieve a high probability of quorum intersection, probabilistic constructions assume, either implicitly (probabilistic quorum systems [7]) or explicitly (signed quorum systems [8]), that the network scheduler is not adversarial. If the scheduler is adversarial, both constructions can return arbitrarily old values, even if servers fail only by crashing. If instead servers can also be subject to Byzantine failures, the situation is a bit more complicated. Signed quorum systems are simply not defined under these circumstances; probabilistic Byzantine quorum systems [7] must instead be configured to prevent read operations from returning values fabricated by Byzantine servers. Note that returning a fabricated value can be much more problematic than returning an arbitrarily old value, especially if readers are required to write back what they read (as it is common to achieve strong consistency guarantees): in this case, the system can become *contaminated* and quickly lose its consistency guarantees<sup>1</sup>. Fortunately, the parameters of probabilistic quorums systems can be chosen to eliminate the possibility of contamination; unfortunately, doing so results in a loss of all the gains made in availability.

$k$ -quorum systems, which we have recently introduced [9], guarantee that a read operation will always return one of the last  $k$  written values – even if the scheduler is adversarial. If the scheduler is not adversarial and read quorums are chosen randomly, as is the case with probabilistic systems,  $k$ -quorums can guarantee a high probability of intersection with the quorum used by the latest write. In a sense,  $k$ -quorums have some of the best features of both strict systems and probabilistic constructions and they can be thought of as a middle ground between them. Like probabilistic constructions, they achieve high availability by performing their writes to small quorums, (called *partial-write-quorums*), and therefore weaken the intersection property of traditional strict quorum systems; unlike probabilistic constructions, however,  $k$ -quorums can still provide *deterministic* intersection guarantees: in particular, they require the set of servers contacted during  $k$  consecutive writes—the union of the corresponding partial write quorums—to form a traditional strict write quorum. Using this combination,  $k$ -quorum systems can bound the staleness of the value returned by a read, even in the presence of an adversarial scheduler: a read operation that contacts a random read quorum of servers is guaranteed to return one of the values written by the last  $k$  preceding writes; furthermore, during periods of synchrony the returned value will, with high probability, be the one written by the last preceding write.

In the absence of an adversarial scheduler, probabilistic systems can have higher availability than  $k$ -quorum systems.  $k$ -quorums make a tradeoff between safety and liveness. By allowing for lower availability than probabilistic systems, they guaran-

---

<sup>1</sup> This is not a problem if the returned values are simply old values because in that case timestamps can be used to prevent old values from overwriting newer values. Timestamps cannot be used with fabricated values because the timestamps of the fabricated values can themselves be fabricated.

tee a bound on the staleness of returned values even in the presence of an adversarial scheduler. In the absence of an adversarial scheduler,  $k$ -quorum systems have higher availability than strict quorum systems when the frequency of write operations is not high (in a sense well defined in [9]). In the same paper, we also propose  $k$ -consistency semantics and provide a single-writer implementation of  $k$ -atomic registers over servers subject to crash failures.

Our previous paper left several important questions unanswered—in particular, it did not discuss how to handle Byzantine failures, nor how to provide a multi-writer/multi-reader construction with  $k$ -atomic semantics. The first question is particularly important in light of the contamination problem that can affect probabilistic Byzantine quorum systems. Answering these questions is harder than in strict quorum systems because the basic guarantees provided by  $k$ -quorum systems are relatively weak and hard to leverage. For example, in the presence of multiple writers it is hard for any single writer to guarantee that  $k$  consecutive writes (possibly performed by other writers) will constitute a quorum: because of the weaker  $k$  consistency semantics, a writer cannot accurately determine the set of servers to which the other writers are writing.

In this paper, we answer both questions. We begin by showing a protocol that implements single-writer  $k$ -atomic semantics and tolerates  $f$  Byzantine servers and any number of crash-and-recover failures as long as read and write quorums intersect in at least  $3f + 1$  servers. Like its crash-only counterpart, the protocol can provide better availability than strict quorum systems when writes are infrequent, and unlike probabilistic solutions, can bound the staleness of the values returned by read operations. Byzantine faults add another dimension to the comparison with probabilistic solutions: the cost, in terms of loss of availability, of preventing reads from returning a value that has never been written by a client, but has instead been generated by Byzantine servers out of thin air. We show that, for equally sized quorums, this cost is considerably higher for probabilistic constructions than for  $k$ -quorum systems.

We then investigate the question of  $k$ -atomic semantics in a multi-writer/multi-reader setting by asking whether it is possible to obtain a multi-writer solution by using a single writer solution as a building block—that is, by restricting read and write operations in the multi-writer case to use the read and write partial quorums of the single writer solution. This approach appears attractive, because, if successful, would result in a multi-writer system with availability very close to that of a single writer system.

We first show a lower bound on the price that any such system must pay in terms of consistency: we prove that no  $m$ -writer protocol based on a solution that achieves  $k$ -atomic semantics in the single writer case can provide better than  $((2m - 1)(k - 1) + 1)$ -atomic semantics. We then present an  $m$ -writer protocol that provides  $((2m - 1)(k - 1) + m)$ -atomic semantics, using a construction that, through a clever use of vector timestamps, allows readers and writers to disregard excessively old values.

## 2 System Model

We consider a system of  $n$  servers. Each server (or node) can crash and recover. We assume that servers have access to a stable storage mechanism that is persistent across crashes. We place no bound on the number of non-Byzantine failures and, when

considering Byzantine faults, we assume that there are no more than  $f$  Byzantine servers—all remaining servers can crash and recover.

*Network Model.* We consider an asynchronous network model that may indefinitely delay, or drop, messages. We require that the protocols provide staleness guarantees irrespective of network behavior.

For purposes of availability, we assume there will be periods of synchrony, during which, if enough servers are available, operations execute in a timely manner.

*Access Model.* A read or write operation needs to access a read or a (partial) write quorum in order to terminate successfully. If no quorum is available the operation has two options: it can either abort or remain pending until enough servers become available (not necessarily all at the same time). The operation can abort unless it has already taken actions that can potentially become visible to other clients.

Clients operations may have timeliness constraints. This does not contradict the asynchrony assumption we make about the network but simply reflects the expectation that operations should execute in a timely manner if the system is to be considered available. A client considers any operation that does not complete in time to have *failed*, independent of whether these operations abort or eventually complete. Note that an operation may be aborted and fail before being actually executed if the operation remains locally queued for too long after being issued.

We assume for simplicity that clients do not crash in between operations, although our protocols can be extended to tolerate client crash and recovery by incorporating a logging protocol.

Finally, we assume that writes are blocking. In other words, a writer will not start the next write until the current write has finished. While this assumption is not overly restrictive, we need to make it for a technical reason, as our protocols require a write operation to know exactly where the previously written values have been written to.

*Availability.* Informally, a system is *available* at time  $t$  if operations started at  $t$  execute in a timely manner. Consider an execution  $\rho$  in a given time interval (possibly infinite) in which a number of operations are started. The system's *availability for execution*  $\rho$  is the ratio of the number of operations that complete in a timely manner in  $\rho$  to the total number of operations in  $\rho$ . If the number of operations is infinite, then the system's availability is the limit of the ratio, if it exists.

The *read and write access patterns* are mappings from the natural numbers to the set of positive real numbers (denoting the duration between the requests). The *failure pattern* of a given node is a mapping from the positive real numbers (denoting global time) to  $\{up, down\}$ ; the system's failure pattern is a set of failure patterns, one for each node.

Given probability distributions on the access patterns (read or write) and failure patterns, the system's *availability* is the expected availability for all pairs of access patterns and failure patterns.

For the purposes of estimating availability, we assume that nodes crash and recover independently, with mean time to recover (MTTR)  $\alpha$  and mean time between failures (MTBF)  $\beta$ . We also assume the periods between two consecutive reads or writes to

be random variables with means MTBR and MTBW respectively and that MTBW is large compared to MTBF; in other words, writes are infrequent. We define the system's availability in periods in which the network is responsive; i.e. in periods in which the roundtrip delay is negligible compared to MTBF and MTTR. In other words, the availability we are interested in depends on whether nodes are up or down, and not on how slow is the network: indeed, in the presence of an adversarial network scheduler measuring availability becomes meaningless, since the scheduler could always cause it to be equal to zero. We assume that the time allowed for successful completion of an operation is negligible compared to MTBF and MTTR.

*Relaxed Consistency Semantics.* The semantics of shared objects that are implemented with quorum systems can be classified as *safe*, *regular* or *atomic* [10]. For applications that can tolerate some staleness, these notions of consistency are too strong and one can use define relaxed consistency semantics as follows [9]:

1. ***k*-safe:** A read that does not overlap with a write returns the result of one of the latest  $k$  completed writes. The result of a read overlapping a write is unspecified.
2. ***k*-regular:** A read that does not overlap with a write returns the result of one of the latest  $k$  completed writes. A read that overlaps with a write returns either the result of one of the latest  $k$  completed writes or the eventual result of one of the overlapping writes.
3. ***k*-atomic:** A read operation returns one of the values written by the last  $k$  preceding writes in an order consistent with real time (assuming there are  $k$  initial writes with the same initial value).

### 3 K-quorums for Byzantine Faults

We define a  $k$ -quorum construction that tolerates  $f$  Byzantine servers, while providing  $k$ -atomic semantics, as a triple  $(\mathcal{W}, \mathcal{R}, k)$ , where  $\mathcal{W}$  is the set of write quorums,  $\mathcal{R}$  is the set of read quorums, and  $k$  is a staleness parameter such that, for any  $R \in \mathcal{R}$ , and  $W \in \mathcal{W}$ ,  $|R \cap W| \geq 3f + 1$  and  $|R|, |W| \leq (n - f)$ .

*Server side protocol* Figure 1 shows the server-side protocol. Each server  $s$  maintains in the structure *current\_data* information about the last write the server knows of, as well as the  $k - 1$  writes that preceded it. READ\_REQUEST messages are handled using a “listeners” pattern [11]. The sender is added to  $s$ 's *Reading* set, which contains the identities of the clients with active read operations at  $s$ . A read operation  $r$  is active at  $s$  from when  $s$  receives  $r$ 's READ\_REQUEST to when it receives the corresponding STOP\_READ. On receipt of a WRITE message,  $s$  acknowledges the writer. Then, if the received information is more recent than the one stored in *current\_data*,  $s$  updates *current\_data* and forwards the update to all the clients in *Reading*; otherwise, it does nothing.

*Writer's Protocol.* Figure 2 shows the client-side write protocol. Each write operation affects only a small set of servers, called a *partial write quorum*, chosen by the writer so that the set of its last  $k$  partial write quorums forms a complete write quorum. The information sent to the servers contains not just a new value and timestamp, but also additional data that will help readers distinguish legitimate updates from values fabricated

by Byzantine servers. Specifically, the writer sends to each server in the partial write quorum,  $k$  tuples—one for each of its last  $k$  writes. The tuple for the  $i$ -th of these writes includes: i) the value  $v_i$ ; ii) the corresponding timestamp  $ts_i$ ; iii) the set  $E_i$  of servers that were not written to in the last  $k - 1$  writes preceding  $i$ ; and iv) a hash of the tuples of the  $k - 1$  writes preceding  $i$ . The write ends once the set of servers from which the writer has received an acknowledgment during the last  $k$  writes forms a complete write quorum<sup>2</sup>.

Thus, the value, timestamp,  $E$ , and hash information for write  $i$  are not only written to  $i$ 's partial write quorum, but will also be written to the partial write quorums used for the next  $k - 1$  writes. By the end of these  $k$  writes this information will be written to a complete write quorum which is guaranteed to intersect any read quorum in at least  $3f + 1$  servers.

*Reader's Protocol.* The reader contacts a read quorum of servers and collects from each of them the  $k$  tuples they are storing. The goal of the read operation is twofold: first, to identify a tuple  $t_i$  representing one of the last  $k$  writes, call it  $i$ , and return to the reader the corresponding value  $v_i$ ; second, to write back to an appropriate partial write quorum (one comprised of servers not in  $E_i$ ) both  $t_i$  and the  $k - 1$  tuples representing the writes that preceded  $i$ —this second step is necessary to achieve  $k$ -atomicity.

The read protocol computes three sets based on the received tuples. The *Valid* set contains, of the most recent tuples returned by each server in the read quorum, only those that are also returned by at least  $f$  other servers. The tuples in this set are legitimate: they cannot have been fabricated by Byzantine servers.

The *Consistent* set also contains a subset of the most recent tuples returned by each server  $s$  in the read quorum. For each tuple  $t_s$  in this set, the reader has verified that the hash of the  $k - 1$  preceding tuples returned by  $s$  is equal to the value of  $h$  stored in  $t_s$ .

The *Fresh* set contains the  $2f + 1$  most recent tuples that come from distinct servers. Since a complete write quorum intersects a read quorum in at least  $2f + 1$  correct servers, legitimate tuples in this set can only correspond to recent (i.e. not older than  $k$  latest) writes.

The intersection of these three sets includes only legitimate and recent tuples that can be safely written back, together with the  $k - 1$  tuples that precede them, to any appropriate partial write quorum. The reader can choose any of the tuples in this intersection: to minimize staleness, it is convenient to choose the one with the highest timestamp.

Because of space limitations, we must refer the reader to our extended technical report [12] for the proofs of the following theorems.

**Theorem 1.** *The single-writer Byzantine  $k$ -quorum read protocol in Figure 3 never returns a value that has not been written by the writer.*

**Theorem 2.** *The single-writer Byzantine  $k$ -quorum read protocol in Figure 3 never returns a value that is more than  $k$ -writes old.*

If the network is behaving asynchronously, or if the required number of servers is not available, then our protocols will just stall until the systems comes to a good con-

<sup>2</sup> Byzantine servers may never respond. The writer can address this problem by simply contacting  $f$  extra nodes for each write while still only waiting for a partial quorum of replies. For simplicity, we abstract from these details in giving the protocol's pseudocode.

```

1  static Reading = 0;
2  static current_data[1..k];
3  while( true ) {
4    (msg, sender) = receiveMessage();
5
6    if( msg instanceof READ_REQUEST)
7      Reading  $\cup$  = {sender};
8      send current_data to sender.
9    else if( msg instanceof STOP_READ )
10     Reading = Reading \ {sender};
11    else if( msg instanceof WRITE )
12     // say msg is WRITE
13     <math>\langle Tuple[ts_{new}, \dots, ts_{new} - k + 1] >
14     if( ts_{new}.ts > current_data[1].ts )
15     current_data[1..k] =
16     Tuple[ts_{new}, \dots, ts_{new} - k + 1];
17     send ACK(ts_{new}) to sender;
18     forward current_data to all in Reading.
19   else
20     send ACK(ts_{new}) to sender;
21 }

```

**Fig. 1.** K-quorum protocol for non-Byzantine servers

```

1  static ts := 0;
2  static Tuple[];
3  void Write(value v)
4  begin
5    ts := ts + 1;
6    h = hash( Tuple[ts - 1, \dots, ts - k + 1] );
7    // E is the set of servers NOT used for the
8    previous k - 1 writes
9    E = P \  $\bigcup_{j=ts-k+1}^{j=ts-1} W_j$ 
10   Tuple[ts] = (v, ts, E, h);
11   delete Tuple[ts - k] to save space
12
13   Find a set PW, such that :
14   |PW  $\cup \bigcup_{j=ts-k+1}^{j=ts-1} W_j$ | = Q_w
15   send WRITE(Tuple[ts, \dots, ts - k + 1]) to all
16   servers in PW.
17
18   // wait for acknowledgements
19   W_{ts} = 0
20   do
21     rcv ACK(ts) from serv
22     W_{ts} = W_{ts}  $\cup$  {serv}
23   until ( | $\bigcup_{j=ts-k+1}^{j=ts} W_j$ |  $\geq$  Q_w - f )
24   return
25 end

```

**Fig. 2.** K-quorum write protocol tolerating up to  $f$  Byzantine servers

figuration. If, during periods of synchrony, all non-Byzantine nodes recover and stay accessible, then our protocols eventually terminate.

**Theorem 3.** *If the network behaves synchronously and all non-Byzantine nodes recover and stay accessible, then the Byzantine  $k$ -quorum protocol for the writer in Figure 2 eventually terminates.*

**Theorem 4.** *If the network behaves synchronously and all non-Byzantine nodes recover and stay accessible, then the Byzantine  $k$ -quorum protocol for the reader in Figure 3 eventually terminates.*

**Theorem 5.** *The construction for Byzantine  $k$ -quorum systems shown in Figures 1, 2, 3 provides  $k$ -atomic semantics.*

### 3.1 Comparison to Probabilistic Quorum Systems

In the Byzantine version of probabilistic quorum systems— $(f, \epsilon)$ -masking quorum systems [7]—write operations remain virtually unchanged: values are simply written to a write quorum chosen according to a given access strategy. Read operations contact a read quorum, also chosen according to the access strategy, and return the highest timestamped value that is reported by more than  $p$  servers, where  $p$  is a safety parameter<sup>3</sup>. Choosing any value of  $p$  lower than  $f + 1$  can be hazardous as, under these circumstances, read operations may return a value that was never written by a client, but instead fabricated by Byzantine nodes. While the probability of an individual read

<sup>3</sup> The original paper [7] uses  $k$  to denote this safety parameter. We use  $p$  to avoid confusion with the staleness parameter of  $k$ -quorum systems. We also use  $f$  to denote the threshold on Byzantine faults instead of the original  $b$ .

```

1 // protocol for a reader
2 received[] // stores the responses from servers
3 CandidateValues // holds the set of candidate values
4 Read()
5 begin
6   choose a read quorum R.
7   send READ_REQUEST to servers in R.
8
9   received[i] = null,  $1 \leq i \leq |R|$ 
10  CandidateValues =  $\emptyset$ 
11  // receive values from all the servers in R
12  while (  $|\{i : received[i] \neq null\}| < |R|$  );
13  begin
14    receive Tuple[ $ts_S, \dots, ts_S - k + 1$ ] from server s;
15    received[s] = Tuple[ $ts_S, \dots, ts_S - k + 1$ ];
16    if ( isValid( Tuple[ $ts_S, \dots, ts_S - k + 1$ ] ) )
17      add Tuple[ $ts_S$ ] to the set CandidateValues
18  end
19
20  // try to choose a value
21  // if unsuccessful, wait for more responses.
22   $ts_{highest} = LargestTimestamp( received );$ 
23  tryChoosing( );
24  while( value_chosen == null )
25  begin
26    receive Tuple[ $ts_S, \dots, ts_S - k + 1$ ] from server s;
27    if (  $ts_S \leq ts_{highest}$  )
28      received[s] = Tuple[ $ts_S, \dots, ts_S - k + 1$ ];
29    tryChoosing( );
30  end
31
32  send STOP_READ to servers in R.
33
34  // write back the chosen value to a partial-write-quorum
35  Find a partial-write-quorum, PW, suitable for value_chosen.
36  send WRITE( $chosen\_value$ ) to PW
37  - wait for acks from PW
38
39  return value_chosen
40 end
41
42 void tryChoosing( )
43 begin
44  (1) Fresh = { Tuple[ $ts_S, \dots, ts_S - k$ ]  $\in Received$  |  $ts_S$  is one of the  $2f+1$  largest time-stamped
45    entries in  $Received$  received from different servers }
46  (2) Valid = { Tuple[ $ts_S, \dots, ts_S - k$ ]  $\in Received$  | Tuple[ $ts_S$ ] occurs in the responses of at least
47     $f + 1$  servers }
48  (3) Consistent = { Tuple[ $ts_S, \dots, ts_S - k$ ]  $\in Received$  | the hash, h, in Tuple[ $ts_S$ ] matches
49    hash( Tuple[ $ts_S - 1, \dots, ts_S - k$ ] ) }
50  (4) if (  $Valid \cap Fresh \cap Consistent \neq \emptyset$  )
51    value_chosen =  $v \in Valid \cap Fresh \cap Consistent$ , with the largest timestamp.
52 end

```

**Fig. 3.**  $k$ -quorum read protocol tolerating up to  $f$  Byzantine servers

operation returning a fabricated value can be low, if enough reads occur in the system, the probability that one of them will do so becomes significant, even in the absence of an adversarial scheduler. Byzantine  $k$ -quorums are immune from such dangers: read operations may return slightly stale values, but never fabricated values. This property allows for the safe use of write backs to achieve stronger consistency guarantees.

*Availability.* Although it is possible to tune probabilistic Byzantine quorum systems by choosing  $p > f$  so that they never return fabricated values, such a choice of  $p$  cannot guarantee that the read availability always increases with  $n$ : if  $p > \frac{q^2}{n}$ , then read availability actually tends to 0 as  $n$  increases, because even a reader able to contact a read-quorum is highly unlikely to receive at least  $p$  identical responses [7]. To ensure that, with high probability, there are at least  $f + 1$  identical responses in a read quorum, probabilistic Byzantine quorum systems would have to choose large quorum sets—requiring the size of the quorum  $q$  to be significantly larger than  $\sqrt{nf}$ . Thus, if the number of Byzantine failures  $f$  is large, then the quorum size for probabilistic quorum systems needs to be large in order to avoid fabricated values.

In summary, if probabilistic Byzantine systems are to have high availability when the scheduler is not adversarial, they run the risk of returning fabricated values, and if a value that is dependent on a fabricated value is written to the system, the system becomes contaminated. Also, if they are designed for high availability and the scheduler happens to be adversarial, probabilistic Byzantine systems can always be forced to return fabricated values.

Our system provides high availability for both reads and writes while guaranteeing that we always return one of the latest  $k$  values written to the system. There are two main reasons for the higher availability of  $k$ -quorums. First, each of their write operations also writes tuples for the preceding  $k - 1$  writes, causing a write to become visible at more locations than in a probabilistic quorum system with similar quorum sizes and load. Second,  $k$ -quorums reads are content to return one of the last  $k$  writes, not just the latest one. Read operations will therefore be likely to yield *Valid*, *Consistent*, and *Fresh* sets with a non-empty intersection. In  $(f, \epsilon)$ -masking quorums a read can return a legitimate value only if the read quorum intersects with a single write-quorum in more than  $p$  nodes. This is a much rarer case and the availability of probabilistic quorum systems is consequently lower.

*Probability of returning the latest value.* The definition of  $k$ -atomicity only bounds the worst-case staleness of a read. However, since the choice of read quorums is not dependent on any other quorums chosen earlier,  $k$ -quorums can also use a random access strategy to choose read quorums, as in [7]. A random access strategy guarantees that, when the network is not adversarial, a read which does not overlap with a write returns, with high probability, the latest written value.

Let  $r$  and  $w_p$  denote, respectively, the sizes of the read quorum and of the partial-write-quorums. We can use Chernoff bounds in a manner similar to [7] to establish the following theorem, whose proof is contained in our extended technical report [12].

**Theorem 6.** *If the read quorum is chosen uniformly at random, then at times when the network is non-adversarial, the probability that a read does not return the latest written value is at most  $e^{-\frac{w_p(r-f)}{2n} \left(1 - \frac{(f+1)n}{w_p(r-f)}\right)^2}$ .*

This probability can be high if  $f$  is small relative to  $n$ .

## 4 Multi Writer $k$ -Quorums

We now study the problem of building a multi-writer  $k$ -quorum system using single-writer  $k$ -quorum systems. This problem is interesting because the resulting multi-writer system will have almost the same availability as the underlying single-writer systems.

A single-writer multi-reader  $k$ -quorum system implements two operations.

1. *val sw-kread( wtr )*: returns one of the  $k$  latest written values, by the writer *wtr*.
2. *sw-kwrite( wtr, val )*: writes the value *val* to the  $k$ -quorum system. It can only be invoked by the writer *wtr*

We assume that the read and write availability of the single-writer  $k$ -quorum system is  $a_{sr} = 1 - \epsilon_{sr}$  and  $a_{sw} = 1 - \epsilon_{sw}$  respectively.

### 4.1 A Lower Bound

We show that using  $k$ -atomic single-writer systems as primitives for a multi-writer system with  $m$  writers, one cannot achieve more than  $((2m - 1)(k - 1) + 1)$ -atomic guarantees.

We assume that the multi-writer solution uses the single writer solution through the  $sw-kread$  and  $sw-kwrite$  functions. We use these functions as black boxes, and we assume that an invocation of  $sw-kread$  on a given register will return any one of the last  $k$  writes to that register.

Since we are interested in a multi-writer solution that has the same availability as the underlying single writer system, we should rule out solutions that require a write in the multi-writer system to invoke multiple write operations of the single writer system. In other words, a write operation in the multi-writer system should be able to successfully terminate if a read quorum and a partial write quorum of the single writer system are available. We require that a read quorum be available because otherwise writers would be forced to write independently of each other with no possibility for one writer to *see* other writes. We do not require that a read and a write quorum be available at the same time. So, without loss of generality, we assume that the implementation uses only  $m$  single-writer registers, one for each writer. The implementation of a write operation of a the multi-writer register can issue a write operation to the issuing writer’s register but not to the other writers’ registers; it can also issue read operations to any of the  $m$  registers. The read operations on the multi-writer register can only issue read operations on the single-writer registers.

In our lower bound proof, we assume that writers execute a full-information protocol in which every write includes all the history of the writer, including all the values it ever wrote and all the values it read from other writers. If the lower bound applies to a full-information protocol, then it will definitely apply to any other protocol, because a full-information protocol can simulate any other protocol by ignoring portions of the data read. Also, we assume that a reader and a writer read all single-reader registers in every operation, possibly multiple times; a protocol that does not read some registers can simply ignore the results of such read operations.

For a writer  $wtr$ , we denote with  $v_{wtr,i}$  the  $i$ ’th value written by  $wtr$ . If a client reads  $v_{x,i}$ , then it will also read  $v_{x,j}$ ,  $j \leq i$ . We denote with  $ts_{wtr}$  a vector timestamp that captures the writer’s knowledge of values written to the system.  $ts_{wtr}[u]$  is the largest  $i$  for which  $wtr$  has read a value  $v_{u,i}$ . In what follows, we will simply denote values with their indices. So, we will say that a writer writes a vector timestamp instead of writing values whose indices are less than or equal to the indices in the vector timestamp.

We now describe a scenario where a reader would return a value that happens to be  $((2m - 1)(k - 1) + 1)$  writes old.

Consider a multi-writer read operation, where the timestamps for all the  $m$  values that the reader receives are similar—specifically, the timestamps

$$Rcvd = \left\{ \begin{array}{l} \langle k - 1, 0, 0, \dots, 0 \rangle, \\ \langle 0, k - 1, 0, \dots, 0 \rangle, \\ \langle 0, 0, k - 1, \dots, 0 \rangle, \\ \vdots \\ \langle 0, 0, 0, \dots, k - 1 \rangle \end{array} \right\}$$

	Writer 1	Writer 2		Writer m
Phase 0	$\langle 0, ?, ?, \dots, ? \rangle$	$\langle ?, 0, ?, \dots, ? \rangle$		$\langle ?, ?, ?, \dots, 0 \rangle$
Phase 1	$\langle 1, 0, 0, \dots, 0 \rangle$ $\langle 2, 0, 0, \dots, 0 \rangle$ $\langle 3, 0, 0, \dots, 0 \rangle$ $\vdots$ $\langle k-1, 0, 0, \dots, 0 \rangle$			
TIME	Phase 2	$\langle 0, 1, 0, \dots, 0 \rangle$ $\langle 0, 2, 0, \dots, 0 \rangle$ $\langle 0, 3, 0, \dots, 0 \rangle$ $\vdots$ $\langle 0, k-1, 0, \dots, 0 \rangle$		
				$\langle 0, 0, 0, \dots, 1 \rangle$ $\langle 0, 0, 0, \dots, 2 \rangle$ $\langle 0, 0, 0, \dots, 3 \rangle$ $\vdots$ $\langle 0, 0, 0, \dots, k-1 \rangle$
Phase 3	k-1 more writes	k-1 more writes	k-1 more writes	k-1 more writes

Read Occurs Now

Fig. 4. Write ordering in the multi-writer  $k$  quorum system

where the timestamp for the value received from the  $i$ -th writer contains information up to the  $(k - 1)$ -th write by that writer, but only contains information about the 0-th write for all remaining writers.

Since all the  $m$  timestamp values are similar, the reader would have no reason to choose one value over the other. Let us assume, without loss of generality, that the reader who reads such a set of timestamps returns the value with the timestamp

$$\langle k - 1, 0, 0, \dots, 0 \rangle$$

written by the first writer.

We now show a set of writes to the system wherein the value returned would be  $((2m - 1)(k - 1) + 1)$  writes old. The writes to the system occur in 4 phases.

In phase 0, each of the  $m$  writers performs a write operation such that the writer's entry in the corresponding timestamp reads 0. For the sake of this discussion, the non-positive values stored in the other entries of the timestamp are irrelevant. We refer to this write as the 0-th write.

In phase 1, writer 1 – whose value is being returned by the read – performs  $(k - 1)$  writes. During each of these writes, the reads of the  $k$ -atomic register of other writers returns their 0-th write. The timestamp vector associated with each of these writes is shown in Figure 4.

In phase 2, each of the remaining  $(m - 1)$  writers perform  $(k - 1)$  writes. Since the underlying single-writer system only provides  $k$ -atomic semantics, also during this phase all reads to the underlying single-writer system returns the 0-th write for that writer. Hence the timestamp vector associated with these writes would be as shown in Figure 4.

At the end of phase 2, each writer has performed  $k - 1$  writes. The total number of writes performed in this phase is  $(m - 1)(k - 1)$ .

Finally, in phase 3, each writer performs another  $k - 1$  writes. There are a total of  $m(k - 1)$  writes in this phase. The exact timestamps associated with these writes are not important.

At the end of phase 3, the multi-writer read takes place. Since the underlying single-writer system only provides  $k$ -atomic semantics, all the reads to the underlying single-writer system during the read are only guaranteed to return a value which is not any older than the  $(k - 1)$ -th write. Thus  $Rcvd$  could be the set of values received by the reader where the reader chooses

$$\langle k - 1, 0, 0, \dots, 0 \rangle$$

which is  $(1 + (m - 1)(k - 1) + m(k - 1))$  writes old.

## 4.2 Multiple Writer Construction

We present a construction for a  $m$ -writer, multi-reader register with relaxed atomic semantics using single-writer, multi-reader registers with relaxed atomic semantics. Using  $k$ -atomic registers, our construction provides  $((2m - 1)(k - 1) + m)$ -atomic semantics, which is almost optimal.

The single-writer registers can be constructed using the  $k$ -quorum protocols from [9], if servers are subject to crash and recover failures, or using the construction from Section 3 if servers are subject to Byzantine failures. In particular, using the single-writer  $k$ -atomic register implementation for Byzantine failures described in Section 3, we obtain an  $m$ -writer  $((2m - 1)(k - 1) + m)$ -atomic register for Byzantine failures.

**The Construction.** The multi-write construction uses  $m$  instances of the single-writer  $k$ -atomic registers, one for each writer  $w_i$ .

It uses approximate vector timestamps to compare writes from different writers. Each writer  $w_i$ ,  $1 \leq i \leq m$ , maintains a local virtual clock  $lts_i$ , which is incremented by 1 for each write so that its value equals the number of writes performed by writer  $w_i$ .

At a given time, let  $gts$  be defined by

$$\forall i : gts[i] = lts_i$$

where the equality holds at the time of interest. The vector  $gts$  represents the global vector timestamp and it may not be known to any of the clients or servers in the system. The read and write protocols are shown in Figure 5.

*Write Operation.* To perform a write operation, the writer first performs a read to obtain the timestamp information about all the writers (lines 4-5). Since the registers used are  $k$ -atomic, each of the received timestamp information is guaranteed to be no more than  $k$  writes old for any writer.

A writer  $wtr_i$  executing a write would calculate (lines 8-9) an approximate vector timestamp  $ats$ , whose  $i$ -th entry is equal to  $lts_i$  and whose remaining entries can be at most  $k$  older than the local time stamps of the entries at the time the write operation

<pre> 1  static <math>lts_i = 0</math>; 2  void mw-write( <math>writer_i, val</math> ) 3  begin 4    for <math>j = 1</math> to <math>m</math> 5      <math>\langle val_j, ts_j \rangle = sw-read( writer_j )</math> 6 7    // Estimate the approx time-stamp 8    <math>\forall j \neq i : ats[j] = \max_p \{ ts_p[j] \}</math> 9    <math>ats[i] = ++lts_i</math> 10 11   sw-write( <math>writer_i, \langle val, ats \rangle</math> ) 12  end </pre>	<pre> 14  <math>\langle val, ts \rangle mw-read( )</math> 15  begin 16    for <math>j = 1</math> to <math>m</math> 17      <math>\langle val_j, ts_j \rangle = sw-read( writer_j )</math> 18 19    Reject = <math>\emptyset</math> 20    for <math>i = 1</math> to <math>m</math> 21      for <math>j = 1</math> to <math>m</math> 22        if ( <math>ts_j &lt; ts_i</math>    <math>(ts_j[i] &lt; ts_i[i] - k)</math> ) 23          Reject = <math>Reject \cup \{ \langle val_j, ts_j \rangle \}</math> 24 25    return any <math>\langle val_j, ts_j \rangle \notin Reject</math> 26  end </pre>
---	--

Fig. 5. Multi-writer K-quorum protocols

was started. Let  $gts^{beg}$  and  $gts^{end}$  denote the global timestamps at the start and end of the write. Then,

$$\begin{aligned}
 ats[i] &= gts^{end}[i] \\
 ats[j] &> gts^{beg}[j] - k \\
 gts^{end} &\geq gts^{beg}
 \end{aligned}$$

The writer then writes the value,  $val$ , along with the timestamp  $ats$  to the single-writer  $k$ -atomic system for the writer.

*Read operation.* To perform a multi-writer read operation, a reader reads from all the  $m$  single-writer  $k$ -quorum systems. Because of the  $k$ -atomicity of the underlying single-writer implementation, each of these  $m$  responses is guaranteed to be one of the  $k$  latest values written by each writer. However, if some writer has not written for a long time, then the value could be very old when considering *all* the writes in the system. Finding the latest value among these  $m$  values is difficult because the approximate timestamps are not totally ordered.

The reader uses elimination rules (lines 19-23) to reject values that can be inferred to be older than other values. This elimination is guaranteed to reject any value that is more than  $((2m - 1)(k - 1) + m)$  writes old. Finally, after rejecting old values, the reader returns any value that has not been rejected.

**Lemma 1.** *If a writer  $w_i$  performs a write, beginning at the (global) time  $gts^{beg}$  and ending at  $gts^{end}$ , with a (approximate) timestamp  $t$ , then*

$$\begin{aligned}
 t &\leq gts^{end}; \quad t[i] = gts^{end}[i]; \text{ and} \\
 \forall j : t[j] &\geq gts^{beg}[j] - k + 1
 \end{aligned}$$

**Lemma 2.** *Let  $\langle val_j, ts_j \rangle$  be one of the  $m$  values read in lines 16-17. If a writer, say  $s$ , has performed  $2k$  writes after  $\langle val_j, ts_j \rangle$  has been written (and before the read starts) then  $\langle val_j, ts_j \rangle$  will be rejected in lines 19-23.*

**Proof:** Let  $gts_j^{beg}, gts_j^{end}$  and  $gts_s^{beg}, gts_s^{end}$  denote the global timestamp at the beginning and end of the writes for  $\langle val_j, ts_j \rangle$  and  $\langle val_s, ts_s \rangle$ . Also, let  $gts_{read}^{beg}$  be the timestamp when the read is started.

Since writer  $s$  has performed at least  $2k - 1$  writes after writing  $\langle val_j, ts_j \rangle$  we have

$$gts_{read}^{beg}[s] \geq gts_j^{end}[s] + 2k$$

Also, from the  $k$ -atomic properties of the single writer system, we know that

$$\begin{aligned} ts_s[s] &= gts_s^{end}[s] > gts_{read}^{beg}[s] - k \\ \Rightarrow ts_j[s] &\leq gts_j^{end}[s] \leq gts_{read}^{beg}[s] - 2k \\ &< gts_s^{end}[s] - k = ts_s[s] - k \end{aligned}$$

Hence  $\langle val_j, ts_j \rangle$  will be added to Reject in line 23.  $\square$

**Theorem 7.** *The multi-writer read protocol never returns a value that is more than  $((2m - 1)(k - 1) + m)$  writes old.*

**Proof:** Let  $\langle val_j, ts_j \rangle$  be the value returned by the read protocol.

The writer  $j$  cannot have written more than  $k - 1$  writes after  $\langle val_j, ts_j \rangle$  (and before the read begins). From Lemma 2 it follows that each of the remaining  $(m - 1)$  writers could have written no more than  $2k - 1$  writes after the write for  $\langle val_j, ts_j \rangle$  (and before the read begins).

Hence,  $\langle val_j, ts_j \rangle$  can be at most  $(1 + (k - 1) + (m - 1)(2k - 1))$  writes old.  $\square$

**Lemma 3.** *At least one of the  $m$  received values is not rejected.*

**Theorem 8.** *The multi-writer protocol described in Figure 5 provides  $((2m - 1)(k - 1) + m)$ -atomic semantics.*

*Availability of a Multi-writer System* We now estimate the availability of the multi-writer system, assuming that the underlying single-writer  $k$ -quorum system has a read and write availability of  $a_{sr} = 1 - \epsilon_{sr}$  and  $a_{sw} = 1 - \epsilon_{sw}$  respectively.

Each multi-writer write operation involves reading from all the  $m$  single-writer  $k$ -quorum systems and writing to one single-writer system. Hence the write availability of the multi-writer system,  $a_{mw}$ , is at least  $(a_{sr})^m a_{sw}$ . This is a conservative estimate because we are assuming that, when the network is synchronous, we treat finding a read quorum and finding a partial-write-quorum as independent events. In practice, however, the fact that a particular number of servers (size of read quorum) are up and accessible only increases the probability of being able to find an accessible partial-write-quorum.

Moreover, If the  $m$  underlying single-writer  $k$ -quorum systems are implemented over the same strict quorum system, then the potential read quorums that can be used for all the  $m$  systems will be the same.<sup>4</sup> Thus, we can use the same read quorum to perform all the  $m$  read operations. In this case, either all reads are available with probability  $a_{sr}$  or all reads fail with probability  $\epsilon_{sr}$ . Hence the probability of the multi-writer write succeeding is at least  $a_{sr} a_{sw}$ .

$$a_{mw} \geq a_{sr} a_{sw} \geq 1 - \epsilon_{sr} - \epsilon_{sw}$$

<sup>4</sup> The partial-write-quorums could still be different, if the writers have chosen different partial-write-quorums in the past.

To perform a multi-writer read, our read protocol performs  $m$  reads from the  $m$  single writer  $k$ -quorum implementations. Thus, along similar lines, we can argue that the availability  $a_{mr}$  is at least  $a_{sr}^m$ . Using the same underlying strict quorum system for all the  $m$  single-writer systems, we can achieve an availability of

$$a_{mr} = a_{sr} = 1 - \epsilon_{sr}$$

*Probabilistic freshness guarantees* We now estimate the probability that our multi-writer implementation of  $k$ -quorums provides the latest value, when all the writes that occur are non-overlapping.

Let  $\delta_{sw}$  denote the probability that a sw-read does not return the latest value written to the single-writer system. Let  $\delta_{mw}$  denote the probability that the multi-writer system does not return the latest value written to the system.

**Theorem 9.** *If the operations are non-overlapping, the probability that the multiple-writer system does not return the latest value is at most  $m\delta_{sw}$*

Once again, the proof can be found in our extended technical report [12].

## 5 Conclusion and Future Work

In this paper we expand our understanding of  $k$ -quorum systems in three key directions [9].

First, we present a single-writer  $k$ -quorum construction that tolerates Byzantine failures. Second, we prove a lower bound of  $((2m - 1)(k - 1) + 1)$  on the staleness for a  $m$  writer solution built over a single-writer  $k$ -quorum solution.

Finally, we demonstrate a technique to build multiple-writer multiple-reader  $k$ -quorum protocols using a single-writer multiple-reader protocol to achieve  $((2m - 1)(k - 1) + m)$ -atomic semantics.

One limitation of our approach is that it improves availability only when writes are infrequent. Also, we have restricted our study of multi-writer solutions to those that built over a single-writer  $k$ -quorum system; it may be possible that a direct implementation can achieve a better staleness guarantee.

## References

1. Raynal, M., Beeson, D.: Algorithms for mutual exclusion. MIT Press, Cambridge, MA, USA (1986)
2. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Proc. of the Third Symposium on Operating Systems Design and Implementation, USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS (1999)
3. Susan Davidson, H.G.M., Skeen, D.: Consistency in partitioned network. Computing Survey 17(3) (1985)
4. Herlihy, M.: Replication methods for abstract data types. Technical Report TR-319, MIT/LCS (1984)
5. Naor, M., Wool, A.: The load, capacity, and availability of quorum systems. SIAM Journal on Computing 27(2) (1998) 423–447

6. Peleg, D., Wool, A.: The availability of quorum systems. *Inf. Comput.* **123**(2) (1995) 210–223
7. Malkhi, D., Reiter, M.K., Wool, A., Wright, R.N.: Probabilistic quorum systems. *Inf. Comput.* **170**(2) (2001) 184–206
8. Yu, H.: Signed quorum systems. In: *Proc. 23rd PODC*, ACM Press (2004) 246–255
9. Aiyer, A., Alvisi, L., Bazzi, R.A.: On the availability of non-strict quorum systems. In: *DISC '05*, London, UK, Springer-Verlag (2005) 48–62
10. Lamport, L.: On interprocess communication. part i: Basic formalism. *Distributed Computing* **1**(2) (1986) 77–101
11. Martin, J.P., Alvisi, L., Dahlin, M.: Minimal byzantine storage. In: *DISC '02*, London, UK, Springer-Verlag (2002) 311–325
12. Aiyer, A., Alvisi, L., Bazzi, R.A.: Byzantine and multi-writer k-quorums. Number TR-06-37 (2006)