Byzantine and Multi-writer K-quorums*

Amitanand S. Aiyer[1], Lorenzo Alvisi[1], and Rida A. Bazzi[2]

[1] Department of Computer Sciences,
The University of Texas at Austin
{anand,lorenzo}@cs.utexas.edu
[2] 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 $\big((2m-1)(k-1)+1\big)$ 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 $\big((2m-1)(k-1)+m\big)$.

1 Introduction

Quorum systems have been extensively studied, with applications that include mutual exclusion, coordination, and data replication in distributed systems [1–4]. systems [1–4]. 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

* 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 loose its consistency guarantees [3]. 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 operations 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-

[3] 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 $\big((2m-1)(k-1)+1\big)$-atomic semantics. We then present an m-writer protocol that provides $\big((2m-1)(k-1)+m\big)$-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 ρ in a given time interval (possibly infinite) in which a number of operations are started. The system's *availability for execution* ρ is the ratio of the number of operations that complete in a timely manner in ρ to the total number of operations in ρ. 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) α and mean time between failures (MTBF) β. 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[4].

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.

3.1 Protocol Correctness

We first prove the read protocol for single-writer Byzantine k-quorums, shown in Figure 3, only returns values that are :

- actually written by the writer (as opposed to an arbitrary value generated by a Byzantine server), and
- are not more than k writes old.

Lemma 1. *If the algorithm, in Figure 3 returns a value,* $Tuple[ts, \ldots, ts-k+1]$*, then the writer must have written* $Tuple[ts, \ldots, ts-k+1]$.

Proof: The algorithm returns a value, Tuple$[ts, \ldots, ts-k+1]$, only if it belongs to *Valid* $\cap$ *Consistent*. For Tuple$[ts, \ldots, ts-k+1]$ to be present in *Valid*, the latest of the

[4] 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 = ∅
2   static current_data[1..k];
3   while( true ) {
4     (msg, sender) = recieveMessage();
5
6     if( msg instanceof READ_REQUEST)
7       Reading ∪= {sender};
8       send current_data to sender.
9     else if( msg instance of STOP_READ )
10      Reading = Reading \ {sender};
11    else if( msg instance of WRITE )
12      // say msg is WRITE
              ⟨Tuple[ts_new, ..., ts_new − k + 1]⟩
13      if( ts_new.ts > current_data[1].ts )
14        current_data[1..k] =
                Tuple[ts_new, ..., ts_new − k + 1];
15        send ACK(ts_new) to sender;
16        forward current_data to all in Reading.
17      else
18        send ACK(ts_new) to sender;
19  }
```

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, ..., ts − k + 1] );
7       // E is the set of servers NOT used for the
              previous k − 1 writes
8       E = P \ ⋃_{j=ts−1}^{j=ts−1} W_j
9       Tuple[ts] = (v, ts, E, h);
10      delete Tuple[ts − k] to save space
11
12      Find a set PW, such that :
13          |PW ⋃ ⋃_{j=ts−k+1}^{j=ts−1} W_j| = Q_w
14      send WRITE⟨Tuple[ts, ..., ts − k + 1]⟩ to all
              servers in PW.
15
16      // wait for acknowledgements
17      W_ts = ∅
18      do
19        recv ACK(ts) from serv
20        W_ts = W_ts ∪ {serv}
21      until ( |⋃_{j=ts−k+1}^{j=ts} W_j| ≥ Q_w − f )
22      return
23  end
```

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

$k+1$ values – Tuple[ts] – has to be reported by at least $f+1$ different servers. Since at least one of these servers is correct, it follows that Tuple[ts] was written by the writer.

Moreover, since Tuple$[ts, \ldots, ts-k+1]$ also belongs to *Consistent*, the the hash in Tuple[ts] has to matches hash(Tuple$[ts-1, \ldots, ts-k+1]$). Therefore the history of the previous $k-1$ writes – Tuple$[ts-1, \ldots, ts-k+1]$– is also correct and should have been written by the writer. □

Lemma 2. *The set Fresh never contains a value that is more than k writes old.*
Proof: The intersection between a read and a write quorum consists of at least $3f+1$ servers. Hence, among the servers responding there are at least $3f+1$ servers who have "seen" one of the latest k writes. At least $2f+1$ of these are correct and have a timestamp greater than or equal to the k-th latest write that occurred before the read has begun. Since the timestamp at a correct server monotonically increases, the correct servers in the intersection will never return a value that is more than k writes old.

Since there are at least $2f+1$ correct servers who never report a value more than k writes old. The $2f+1$ latest values received from different servers, *Fresh*, will never contain a value that is more than k writes old. □

Theorem 1. *The single-writer Byzantine k-quorum read protocol in Figure 3 never returns a value that is has not been written by the writer.*
Proof: Follows from Lemma 1 □

Theorem 2. *The single-writer Byzantine k-quorum read protocol in Figure 3 never returns a value that is more than k-writes old*
Proof: The read in Figure 3 only returns a value that belongs to $Valid \cap Consistent \cap Fresh$. From lemma 2, we know that the set *Fresh* can never contain a value that is

```
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 ≤ i ≤ |R|
10    CandidateValues = ∅
11    // receive values from all the servers in R
12    while ( |{i : received[i] ≠ null}| < |R| );
13    begin
14      receive Tuple[ts_S, ..., ts_S − k + 1] from server s;
15      received[s] = Tuple[ts_S, ..., ts_S − k + 1];
16      if( isValid( Tuple[ts_S, ..., 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, ..., ts_S − k + 1] from server s;
27      if (ts_S ≤ ts_highest )
28        received[s] = Tuple[ts_S, ..., 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, ..., ts_S − k] ∈ Received | ts_S is one of the 2f+1 largest time−stamped
          entries in Received received from different servers }
45    (2) Valid = { Tuple[ts_S, ..., ts_S − k] ∈ Received | Tuple[ts_S] occurs in the responses of at least
          f + 1 servers }
46    (3) Consistent = { Tuple[ts_S, ..., ts_S − k] ∈ Received | the hash, h, in Tuple[ts_S] matches
47          hash( Tuple[ts_S − 1, ..., ts_S − k] ) }
48    (4) if ( Valid ∩ Fresh ∩ Consistent ≠ ∅ )
49          value_chosen = v ∈ Valid ∩ Fresh ∩ Consistent, with the largest timestamp.
50  end
```

Fig. 3. K-quorum read protocol tolerating up to f Byzantine servers.

more than k writes old. Hence, a read will never return a value that is more than k writes old. □

In an asynchronous environment where there is no bound on the number of nodes failing no protocol can provide liveness guarantees always. If the network is behaving asynchronously, or if the required number of servers are not available then our protocols will just stall until the systems comes to a good configuration. We will now argue that if the required number of servers are accessible, and the network behaves synchronously then our protocols will 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*

Proof: If network is synchronous, and the non-Byzantine nodes recover, then the writer will be able to get find an accessible partial-write-quorum. On receiving the acknowledgements from all the servers in the partial-write-quorum, the writer 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*

Proof: New values from a server are allowed to overwrite old values only as long as the time stamp of the new value is $<= t_{highest}$. Therefore values cannot get overwritten indefinitely.

Consider the situation, after the writer completes the latest write before $t_{highest}$: say t_{latest} [5].

When this happens all correct servers in the intersection of the read and write quorum will have a timestamp ts such that $ts_{latest} - k + 1 \leq ts \leq ts_{latest}$. The correct servers will forward the values and, these values will not be overwritten by any other value[6].

Since the intersection between a read and a write quorum contains at least $2f + 1$ correct servers, eventually the reader will receive at least $2f + 1$ values that have their timestamp in the range $[ts_{latest} - k + 1, ts_{latest}]$. Consider the $(f + 1)$-th largest timestamped value, v_c, received from the correct servers.

v_c is reported by a correct server, so its hashes match and v_c will be present in *Consistent*. Since there are no more than f faulty servers, who may report higher timestamps, v_c will be present in $Fresh$. Also, since all the $2f + 1$ highest timestamped values from correct servers lie in the range $[ts_{latest} - k + 1, ts_{latest}]$, it follows that the $(f + 1)$-th value from a correct server will be contained in the history of the first f highest timestamped values from correct servers. Hence v_c is also present in $Valid$. Therefore tryChoosing will set $value_chosen$ to a non-null value and the algorithm will terminate. □

K-Atomic Semantics To prove that the protocols achieve k-atomic semantics, we show a linearized schedule of reads and writes such that every read returns one of the k previously written values. We define

Definition 1. *Written-time: Let written-time denote the global time instance, when a value that is being written reaches a partial-write-quorum.*

We will order the reads and writes in a manner similar to [9].

- All writes are ordered as if they instantaneously take place at their written-time.
- A read which returns a value (v, ts, E, h), which was written at time-stamp t, can be scheduled any time between
  1. The written-time, τ_t of the value returned, (v, t, E, h).
  2. and, before the written-time of the next k^{th} write, $(v', t+k, E', h')$. i.e. before τ_{t+k}.

It is easy to see that, such an ordering satisfies the requirements of k-atomic semantics. We need to show that such a ordering can be done in a manner consistent with local history.

[5] If the writer does not write a value with a timestamp $> t_{highest}$, then t_{latest} would be the timestamp of the last write written. Otherwise t_{latest} can be taken as $t_{highest}$.

[6] if $t_{latest} <= t_{highest}$ then writer has not written any value with a time stamp $> t_{latest}$, so these values never get overwritten. Otherwise, if $t_{latest} == t_{highest}$, then the values do not get overwritten because we discard values with timestamps $> t_{highest}$.

The scheduling of writes is trivial, because written-time of a write occurs between the time a write has begun and before the write ends.

We now show, by contradiction, that reads can also be scheduled. Assume, if possible, that the read interval does not overlap with the interval $[\tau_t, \tau_{t+k})$. There are two cases:

1. Read finishes before τ_t: This scenario is not possible, because a read has to write-back the value. Therefore a read can end only after the written-time of the value it returns.
2. Read begins after τ_{t+k}: From Lemma 2, any read that starts after τ_{t+k} cannot return a value as old as τ_t, which is a contradiction. $\Rightarrow\Leftarrow$

3.2 Comparison to Probabilistic Quorum Systems

In the Byzantine version of probabilistic quorum systems—(f, ϵ)-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[7]. 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 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.

[7] 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.

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, ϵ)-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.

We now try to bound the probability that during times when the network is not adversarial, a read returns the latest value. For these calculations, we assume that the write operations do not overlap with any other operations.

From our optimized protocol, it is clear that if the read quorum intersects with the partial-write-quorum used for the latest write in at least $f+1$ correct nodes, then the read will return the latest write.

We now use Chernoff bounds to bound the probability that the read does not return the latest value. Let r and w_p denote the size of the read quorum and the size of partial-write-quorums used.

Theorem 5. *If the read quorum is chosen uniformly at random, the probability that a read does not return the latest written value is*

$$\leq e^{-\frac{w_p(r-f)}{2n}(1-\frac{(f+1)n}{w_p(r-f)})^2}$$

Proof: If the reader receives the latest written value from at least $f+1$ different servers, then the read will return that value. Hence the probability that the read returns the latest value is no less than the probability that the read quorum intersects with the partial-write-quorum used for the latest write in at least $f+1$ correct servers.

The number of correct servers in the read quorum is at least $r-f$. If these servers are chosen at random, then the number of servers in the intersection of the partial write quorum of size w_p and these $r-f$ servers follows a hypergeometric distribution with a mean of

$$\frac{w_p(r-f)}{n}$$

[12] shows that the tail bounds for hypergeometric distribution is no more than the tail bounds for a sum of independent Bernoulli variables with the same mean. Hence, we can use Chernoff bounds to provide an upper bound on the probability that the number of servers in the intersection is less than or equal to f.

Therefore

$$Pr[intersection < f+1] \leq e^{-\frac{w_p(r-f)}{2n}(1-\frac{(f+1)n}{w_p(r-f)})^2}$$

□

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 $\big((2m-1)(k-1)+1\big)$-atomic guarantees.

We assume that the 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 $\boldsymbol{ts}_{wtr}$ a vector timestamp that captures the writer's knowledge of values written to the system. $\boldsymbol{ts}_{wtr}[u]$ 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 $\big((2m-1)(k-1)+1\big)$ 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

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

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 timestamp returns the value with the timestamp

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

written by the first writer.

We now show a set of writes to the system wherein the value returned would be $\big((2m-1)(k-1)+1\big)$ 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.

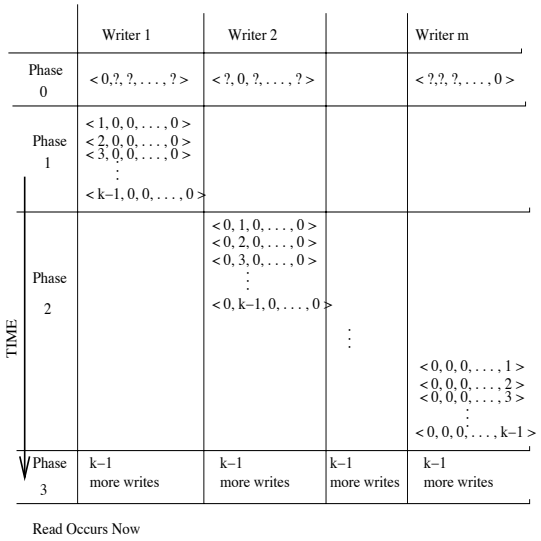

Fig. 4. Write ordering in the multi-writer k quorum system

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

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

which is $\big(1 + (m-1)(k-1) + m(k-1)\big)$ 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 $\big((2m-1)(k-1)+m\big)$-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 $\big((2m-1)(k-1)+m\big)$-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.

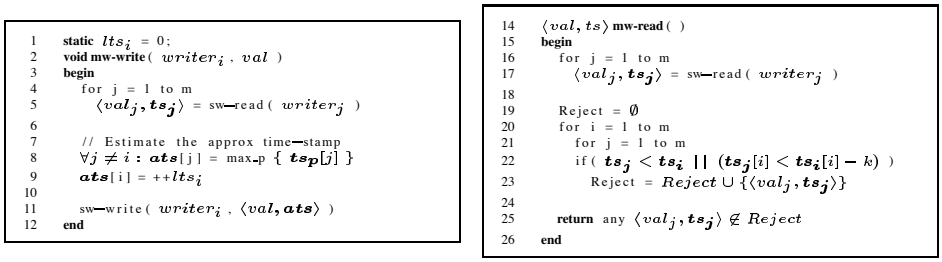

Fig. 5. Multi-writer K-quorum protocols

At a given time, let $\boldsymbol{gts}$ be defined by

$$\forall i : \boldsymbol{gts}[i] = lts_i$$

where the equality holds at the time of interest. The vector $\boldsymbol{gts}$ 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 $\boldsymbol{ats}$, 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 was started. Let $\boldsymbol{gts}^{\boldsymbol{beg}}$ and $\boldsymbol{gts}^{\boldsymbol{end}}$ denote the global timestamps at the start and end of the write. Then,

$$\boldsymbol{ats}[i] = \boldsymbol{gts}^{\boldsymbol{end}}[i]$$

$$\boldsymbol{ats}[j] > \boldsymbol{gts}^{\boldsymbol{beg}}[j] - k$$

$$\boldsymbol{gts}^{\boldsymbol{end}} \geq \boldsymbol{gts}^{\boldsymbol{beg}}$$

The writer then writes the value, val, along with the timestamp $\boldsymbol{ats}$ 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 $\big((2m-1)(k-1)+m\big)$ writes old. Finally, after rejecting old values, the reader returns any value that has not been rejected.

Protocol Correctness We now analyze the protocol in Figure 5 to give a bound on the staleness.

Lemma 3. *If a writer w_i performs a write, beginning at the (global) time $\boldsymbol{gts^{beg}}$ and ending at $\boldsymbol{gts^{end}}$, with a (approximate) timestamp $\boldsymbol{t}$, then*

$$\boldsymbol{t} \leq \boldsymbol{gts^{end}}; \quad \boldsymbol{t}[i] = \boldsymbol{gts^{end}}[i]; \text{ and}$$

$$\forall j : \boldsymbol{t}[j] \geq \boldsymbol{gts^{beg}}[j] - k + 1$$

Proof: k-quorum implementation of the single-writer system for writer j, guarantees that any sw-read for writer j will return one of the k latest values written by the writer j. Thus, during the initial read phase, the writer will read one of the last k timestamp values used by writer j. Hence $\boldsymbol{gts^{end}}[j] \geq \boldsymbol{t}[j] \geq \boldsymbol{gts^{beg}}[j] - k + 1$ for all j.

Moreover, the writer w_i always sets the i^{th} coordinate of the computed vector timestamp to his local virtual timestamp lts_i (line 9). Therefore $\boldsymbol{t}[i] = \boldsymbol{gts^{end}}[i]$. □

Lemma 4. *Let $\langle val_j, \boldsymbol{ts_j} \rangle$ be one of the m values read in lines 16-17. If a writer, say s, has performed $2k$ writes after $\langle val_j, \boldsymbol{ts_j} \rangle$ has been written (and before the read starts) then $\langle val_j, \boldsymbol{ts_j} \rangle$ will be rejected in lines 19-23.*

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

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

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

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

$$\boldsymbol{ts_s}[s] = \boldsymbol{gts_s^{end}}[s] > \boldsymbol{gts_{read}^{beg}}[s] - k$$

$$\Rightarrow \boldsymbol{ts_j}[s] \leq \boldsymbol{gts_j^{end}}[s] \leq \boldsymbol{gts_{read}^{beg}}[s] - 2k$$

$$< \boldsymbol{gts_s^{end}}[s] - k = \boldsymbol{ts_s}[s] - k$$

Hence $\langle val_j, \boldsymbol{ts_j} \rangle$ will be added to Reject in line 23. □

Theorem 6. *The multi-writer read protocol never returns a value that is more than $\big((2m-1)(k-1)+m\big)$ writes old.*

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

The writer j cannot have written more than $k-1$ writes after $\langle val_j, \boldsymbol{ts_j} \rangle$ (and before the read begins). From Lemma 4 it follows that each of the remaining $(m-1)$ writers could have written no more than $2k-1$ writes after the write for $\langle val_j, \boldsymbol{ts_j} \rangle$ (and before the read begins).

Hence, $\langle val_j, \boldsymbol{ts_j} \rangle$ can be at most $\big(1+(k-1)+(m-1)(2k-1)\big)$ writes old. □

Lemma 5. *At least one of the m received values remains un-rejected.*

Proof: There are two rules that we apply for rejecting a value $\langle val_j, \boldsymbol{ts_j} \rangle$

1. rule (i): $\exists i : \boldsymbol{ts_j} < \boldsymbol{ts_i}$
2. rule (ii): $\exists i : \big(\boldsymbol{ts_j}[i] < \boldsymbol{ts_i}[i] - k\big)$

To show that at least one value remains un-rejected by both, we first show a value that is not rejected by rule (ii). Then we argue that for any value that survives rule (ii), but gets rejected by rule (i) we have another value that survives rule (ii). Thus we have at least one value that remains un-rejected, because just applying rule (i) cannot reject all values.

Among all the m values received, consider the value whose write started last. Let the value be $\langle val_l, \boldsymbol{ts_l} \rangle$.

Consider the write for any other value $\langle val_j, \boldsymbol{ts_j} \rangle$. Since this write has started before $\boldsymbol{gts_l^{beg}}$, it follows that

$$\boldsymbol{gts_l^{beg}}[j] \geq \boldsymbol{gts_j^{beg}}[j] = \boldsymbol{ts_j}[j] - 1$$

Since, when writer l performs a read to estimate $ts_l[j]$ it is guaranteed to receive a value no older than k writes,

$$\boldsymbol{ts_l}[j] \geq \boldsymbol{gts_l^{beg}}[j] - k + 1$$

$$\Rightarrow \boldsymbol{ts_l}[j] \geq \boldsymbol{ts_j}[j] - k + 1$$

Thus $\langle val_l, \boldsymbol{ts_l} \rangle$ will not be rejected by rule (ii).

If $\langle val_i, \boldsymbol{ts_i} \rangle$ is a value that is not rejected by rule (ii) but is rejected because $\boldsymbol{ts_j} > \boldsymbol{ts_i}$ then $\langle val_j, \boldsymbol{ts_j} \rangle$ cannot be rejected by rule (ii).

Hence at least one value remains un-rejected. □

Theorem 7. *The multi-writer protocol described in Figure 5 provides* $\big((2m-1)(k-1)+m\big)$*-atomic semantics.*

Proof: To prove that the multi-writer protocols achieve $\big((2m-1)(k-1)+m\big)$-atomic semantics, we show a serialized schedule of reads and writes where each read returns one of the values written by the last $\big((2m-1)(k-1)+m\big)$ writes.

Since the underlying single-writer k-quorum system provides k-atomic semantics, all reads and writes to any such underlying system can be serialized such that all reads return one of the last k values written to the system.

For proving $\big((2m-1)(k-1)+m\big)$-atomicity for the multi-writer system, we schedule the writes to take place at the same instance as it is scheduled to take place in the single-writer system.

Let τ_i denote the time instance when the i^{th} write is scheduled to occur. We schedule the reads as follows: Any read that returns a value v, written during the i^{th} write to the system, would be scheduled to occur some time between τ_i and $\tau_{i+\big((2m-1)(k-1)+m\big)}$.

It is easy to see that, such an ordering satisfies the requirements of $\big((2m-1)(k-1)+m\big)$-atomic semantics. We need to show that such a ordering can be done in a manner consistent with local history.

The scheduling of writes is trivial, because from the k-atomicity property of the underlying single-writer system, it follows that the time instance when the write is scheduled to occur lies between the time when the sw-write(line 11) begins and ends.

We now show, by contradiction, that reads can also be scheduled. Assume if possible that the read interval does not overlap with the interval $\left[\tau_i, \tau_{i+\left((2m-1)(k-1)+m\right)}\right)$. There are two cases:

1. Read finishes before τ_i: Since the mw-read only returns a value that has been read from sw-read in line 5, if mw-read were to finish before τ_i that would contradict the assumption that the underlying single-writer implementation satisfies k-atomicity.
2. Read begins after $\tau_{i+\left((2m-1)(k-1)+m\right)}$: From Lemma **??** any read that starts after $\tau_{i+\left((2m-1)(k-1)+m\right)}$ cannot return a value which is more than $\left((2m-1)(k-1)+m\right)$ writes old. Hence the read could not possibly have returned v from the i^{th} write; this is a contradiction. $\Rightarrow\Leftarrow$

□

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.[8] 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 ϵ_{sr}. Hence the probability of the multi-writer write succeeding is at least $a_{sr}a_{sw}$.

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

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

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

[8] The partial-write-quorums could still be different, if the writers have chosen different partial-write-quorums in the past.

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

Theorem 8. *The probability that the multiple-writer system does not return the latest value is at most* $m\delta_{sw}$

Proof: Consider the latest write v. Without loss of generality, assume it was written by *writer-i*. The reader may not return the latest values if either (i) the reader does not read the latest written value or (ii) the reader reads the value, but chooses a value from some other writer whose timestamp is concurrent with the latest value.

The probability that the reader does not read the latest written value is at most δ_{sw}.

For any other writer, *writer-j* the probability that while writing v, *writer-i* did not read the latest write written by *writer-j* is at most δ_{sw}. Hence the probability that there exists a write, whose timestamp is concurrent with v's timestamp is at most $(m-1)\delta_{sw}$. Hence the result. □

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 $\big((2m-1)(k-1)+1\big)$ 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 $\big((2m-1)(k-1)+m\big)$-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 partioned 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. Hoeffding, W.: Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association **58**(301) (1963) 13–30