Byzantine Quorum Systems

Dahlia Malkhi Michael Reiter

AT&T Laboratories—Research, Murray Hill, NJ USA
{dalia,reiter}@research.att.com

October 25, 1996

Abstract

Quorum systems are well-known tools for ensuring the consistency and availability of replicated data despite the benign failure of data repositories. In this paper we consider the arbitrary (Byzantine) failure of data repositories and present the first study of quorum system requirements and constructions that ensure data availability and consistency despite these failures. We also consider the load associated with our quorum systems, i.e., the minimal access probability of the busiest server. For services subject to arbitrary failures, we demonstrate quorum systems over n servers with a load of $O(\frac{1}{\sqrt{n}})$, thus meeting the lower bound on load for benignly fault-tolerant quorum systems. We explore several variations of our quorum systems and extend our constructions to cope with arbitrary client failures.

Keywords: distributed computing, replication, fault-tolerance, Byzantine failures, quorums

1 Introduction

A well known way to enhance the availability and performance of a replicated service is by using *quorums*. A quorum system for a universe of servers is a collection of subsets of servers, each pair of which intersect. Intuitively, each quorum can operate on behalf of the system, thus increasing its availability and performance, while the intersection property guarantees that operations done on distinct quorums preserve consistency.

In this paper we consider the arbitrary (Byzantine) failure of clients and servers, and initiate the study of quorum systems in this model. Intuitively, a quorum system tolerant of Byzantine failures is a collection of subsets of servers, each pair of which intersect in a set containing sufficiently many *correct* servers to guarantee consistency of the replicated data as seen by clients. We provide the following contributions.

1. We define the class of *masking quorum systems*, with which data can be consistently replicated in a way that is resilient to the arbitrary failure of data repositories. We present several example constructions of such systems and show necessary and sufficient conditions for the existence of masking quorum systems under different failure assumptions.

2. We explore two variations of masking quorum systems. The first, called *dissemination quorum systems*, is suited for services that receive and distribute *self-verifying* information from correct clients (e.g., digitally signed values) that faulty servers can fail to redistribute but cannot undetectably alter. The second variation, called *opaque masking quorum systems*, is similar to regular masking quorums in that it makes no assumption of self-verifying data, but it differs in that clients do not need to know the failure scenarios for which the service was designed. This somewhat simplifies the client protocol and, in the case that the failures are maliciously induced, reveals less information to clients that could guide an adversary attempting to compromise the system.

3. We explore the *load* of each type of quorum system, where the load of a quorum system is the minimal access probability of the busiest server, minimizing over all strategies for picking quorums. We present a masking quorum system with the property that its load over a total of n servers is $O(\frac{1}{\sqrt{n}})$, thereby meeting the lower bound for the load of benignly-fault-tolerant quorum systems. As an added contribution, we also provide a proof of this lower bound for benignly-fault-tolerant quorum systems—and, *a fortiori*, Byzantine quorum systems—that is much simpler than previous proofs. For opaque masking quorum systems, we prove a lower bound of $\frac{1}{2}$ on the load, and present a construction that meets this lower bound and proves it tight.

4. For services that use masking quorums (opaque or not), we show how to deal with faulty clients in addition to faulty servers. The primary challenge raised by client failures is that there is no guarantee that clients will update quorums according to any specified protocol. Thus, a faulty client could leave the service in an inconsistent and irrecoverable state. We develop two *update* protocols, by which clients access the replicated service, that prevent clients from leaving the service in an inconsistent state. Both protocols have the desirable property that they involve only the quorum at which an access is attempted, while providing system-wide consistency properties.

In our treatment, we express assumptions about possible failures in the system in the form of a *fail-prone system* $\mathcal{B} = \{B_1, \ldots, B_k\}$ of servers, such that some B_i contains all the faulty servers.

This formulation includes typical failure assumptions that at most a threshold f of servers fail (e.g., the sets $B_1, \ldots, B_k$ could be all sets of f servers), but it also generalizes to allow less uniform failure scenarios. Our motivation for exploring this generalization stems from our experience in constructing secure distributed services [Rei96, MR96], i.e., distributed services that can tolerate the malicious corruption of some (typically, up to a threshold number of) component servers by an attacker. A criticism to assuming a simple threshold of corrupted servers is that server penetrations may not be independent. For example, servers in physical proximity to each other or in the same administrative domain may exhibit correlated probabilities of being captured, or servers with identical hardware and software platforms may have correlated probabilities of electronic penetration. By exploiting such correlations (i.e., knowledge of the collection $\mathcal{B}$), we can design quorum systems that more effectively mask faulty servers.

Our quorum systems, if used in conjunction with appropriate protocols and synchronization mechanisms, can be used to implement a wide range of data semantics. In this paper, however, we choose to demonstrate a variable supporting read and write operations with relatively weak semantics, in order to maintain focus on our quorum constructions. These semantics imply a *safe* variable [Lam86] in the case of a single reader and single writer, which a set of correct clients can use to build other abstractions, e.g., atomic, multi-writer multi-reader registers [Lam86, IS92, LTV96], concurrent timestamp systems [DS89, IL93], l-exclusion [DGS88, ADGM90], and atomic snapshot scan [ADGM93, And93]. Our quorum constructions can also be directly exploited in algorithms that employ "uniform" quorums for fault tolerance either explicitly or implicitly (by involving a threshold of processes), in order to tolerate non-uniform failure scenarios. Examples include algorithms for shared memory emulation [ABD95], randomized Byzantine agreement [Tou84], reliable Byzantine multicast [BT85, Rei94, MR96], and secure replicated data [HT88].

The rest of this paper is structured as follows. We begin in Section 2 with a description of related work. In Section 3 we present our system model and definitions. We present quorum systems for the replication of arbitrary data subject to arbitrary server failures in Section 4, and in Section 5 we present two variations of these systems. We then detail access protocols for replicated services that tolerate faulty clients in addition to faulty servers in Section 6. We conclude in Section 7.

2 Related work

Our work was influenced by the substantial body of literature on quorum systems for benign failures and applications that make use of them, e.g., [Gif79, Tho79, Mae85, GB85, Her86, ET89, CAA90, AE91, NW94, PW95]. In particular, our grid construction of Section 5.1 was influenced by grid-like constructions for benign failures (e.g., [CAA90]), and we borrow our definitions of *domination* and *load* from [GB85] and [NW94], respectively.

Quorum systems have been previously employed in the implementation of security mechanisms. Naor and Wool [NW96] described methods to construct an access-control service using quorums. Their constructions use cryptographic techniques to ensure that out-of-date (but correct) servers cannot grant access to unauthorized users. Agrawal and El Abbadi [AE90] and Mukkamala [Muk94] considered the confidentiality of replicated data despite the disclosure of the contents of a threshold of the (otherwise correct) repositories. Their constructions used quorums with increased intersection, combined with Rabin's dispersal scheme [Rab89], to enhance the confidentiality and availability of the data despite some servers crashing or their contents being observed. Our work differs from all of the above by considering arbitrarily faulty servers, and accommodating failure scenarios beyond a simple threshold of servers.

Herlihy and Tygar [HT88] applied quorums with increased intersection to the problem of pro-

tecting the confidentiality and integrity of replicated data against a threshold of arbitrarily faulty servers. In their constructions, replicated data is stored encrypted under a key that is shared among the servers using a threshold secret-sharing scheme [Sha79], and each client accesses a threshold number of servers to reconstruct the key prior to performing (encrypted) reads and writes. This construction exhibits one approach to make replicated data self-verifying via encryption, and consequently the quorum system they develop is a special case of our dissemination quorum systems, i.e., for a threshold of faulty servers.

Other techniques have been proposed for constructing distributed services that tolerate the arbitrary failure of some component servers, e.g., see [Sch90, Mar90, RB94]. Such techniques of which we are aware typically require that all servers receive and process all client accesses, and thus they exhibit high load. In many cases, our techniques could be used to more efficiently mask arbitrary failures.

3 Preliminaries

3.1 System model

We assume a *universe* U of servers, $|U| = n$, and an arbitrary number of clients that are distinct from the servers. A *quorum system* $\mathcal{Q} \subseteq 2^U$ is a set of subsets of U, any pair of which intersect. Each $Q \in \mathcal{Q}$ is called a *quorum*.

Servers (and clients) that obey their specifications are *correct*. A *faulty* server, however, may deviate from its specification arbitrarily. A *fail-prone system* $\mathcal{B} \subseteq 2^U$ is a set of subsets of U, none of which is contained in another, such that some $B \in \mathcal{B}$ contains all the faulty servers. The fail-prone system represents an assumption characterizing the failure scenarios that can occur, and could express typical assumptions that up to a threshold of servers fail, as well as less uniform assumptions.

In the remainder of this section, and throughout Sections 4 and 5, we assume that clients behave correctly. In Section 6 we will relax this assumption (and will be explicit when we do so).

We assume that any two correct processes (clients or servers) can communicate over an authenticated, reliable channel. That is, a correct process receives a message from another correct process if and only if the other correct process sent it. However, we do *not* assume known bounds on message transmission times; i.e., communication is asynchronous.

3.2 Access protocol

We consider a problem in which the clients perform read and write operations on a variable x that is replicated at each server in the universe U. For the purposes of this paper, we define correctness of this variable as follows; a more formal treatment of these concepts can be found in [Lam86]. We say that a read or write operation op_1 *precedes* an operation op_2 if op_1 terminates (in real time, at the client that initiated it) before op_2 starts (in real time, at the client initiating it). If op_1 does not precede op_2 and op_2 does not precede op_1, then they are called *concurrent*. Given a set of operations, a *serialization* of those operations is a total ordering on them that extends the precedence ordering among them. Then, our correctness condition requires that any read that is concurrent with no writes returns the last value written in some serialization of the preceding writes. In the case of a single-reader, single-writer variable, this will immediately imply *safe* semantics [Lam86].

Copies of the variable x are stored at each server, along with a timestamp value t. Timestamps are assigned by a client to each replica of the variable when the client writes the replica. Our protocols require that different clients choose different timestamps, and thus each client c chooses

its timestamps from some set T_c that does not intersect $T_{c'}$ for any other client c'. The timestamps in T_c can be formed, e.g., as integers appended with the name of c in the low-order bits. The read and write operations are implemented as follows.

Write: For a client c to write the value v, it queries each server in some quorum Q to obtain a set of value/timestamp pairs $A = \{\langle v_u, t_u \rangle\}_{u \in Q}$; chooses a timestamp $t \in T_c$ greater than the highest timestamp value in A and greater than any timestamp it has chosen in the past; and updates x and the associated timestamp at each server in Q to v and t, respectively.

Read: For a client to read x, it queries each server in some quorum Q to obtain a set of value/timestamp pairs $A = \{\langle v_u, t_u \rangle\}_{u \in Q}$. The client then applies a deterministic function *Result*() to A to obtain the result *Result*(A) of the read operation.

In the case of a write operation, each server updates its local variable and timestamp to the received values $\langle v, t \rangle$ only if t is greater than the timestamp currently associated with the variable.

Two points about this description deserve further discussion. First, the nature of the quorum sets Q and the function *Result*() are intentionally left unspecified; further clarification of these are the point of this paper. Second, this description is intended to require a client to obtain a set A containing value/timestamp pairs from *every* server in some quorum Q. That is, if a client is unable to gather a complete set A for a quorum, e.g., because some server in the quorum appears unresponsive, the client must try to perform the operation with a different quorum. This requirement stems from our lack of synchrony assumptions on the network: in general, the only way that a client can know that it has accessed every *correct* server in a quorum is to (apparently successfully) access every server in the quorum. Our framework guarantees the availability of a quorum at any moment, and thus by attempting the operation at multiple quorums, a client can eventually make progress. In some cases, the client can achieve progress by incrementally accessing servers until it obtains responses from a quorum of them.

3.3 Load

A measure of the inherent performance of a quorum system is its *load*. Naor and Wool [NW94] define the load of a quorum system as the probability of accessing the busiest server in the *best* case. More precisely, given a quorum system $\mathcal{Q}$, an *access strategy* w is a probability distribution on the elements of $\mathcal{Q}$; i.e., $\sum_{Q \in \mathcal{Q}} w(Q) = 1$. $w(Q)$ is the probability that quorum Q will be chosen when the service is accessed. Load is then defined as follows:

Definition 3.1 Let a strategy w be given for a quorum system $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ over a universe U. For an element $u \in U$, the load induced by w on u is $l_w(u) = \sum_{Q_i \ni u} w(Q_i)$. The load induced by a strategy w on a quorum system $\mathcal{Q}$ is

$$L_w(\mathcal{Q}) = \max_{u \in U} \{l_w(u)\}.$$

The *system load* on a quorum system $\mathcal{Q}$ is

$$L(\mathcal{Q}) = \min_{w} \{L_w(\mathcal{Q})\},$$

where the minimum is taken over all strategies. □

We reiterate that the load is a best case definition. The load of the quorum system will be achieved only if an optimal access strategy is used, and only in the case that no failures occur. A strength of this definition is that load is a property of a quorum system, and not of the protocol using it.

A comparison of the definition of load to other seemingly plausible definitions is given in [NW94]. They also explore a related notion, the *capacity* of a quorum system, which is the rate at which the system handles quorum accesses. They further prove that capacity is the inverse of load.

3.4 Size and Domination

Another natural measure of the performance of a quorum system is the *size* of its quorums. Intuitively, the size of a quorum represents the cost incurred by a client when accessing it. Thus, in general, smaller quorums are more desirable.

Of special interest are quorum systems that cannot be reduced in size (i.e., that no quorum in the system can be reduced in size). Such quorum systems are called *non-dominated*. More precisely, let a *coterie* be a quorum system $\mathcal{Q}$ with the property that if $Q_1, Q_2 \in \mathcal{Q}$, then $Q_1 \not\subset Q_2$. Suppose that $\mathcal{Q}, \mathcal{Q}'$ are two coteries, $\mathcal{Q} \neq \mathcal{Q}'$, such that for every $Q' \in \mathcal{Q}'$, there exists a $Q \in \mathcal{Q}$ such that $Q \subseteq Q'$. Then $\mathcal{Q}$ *dominates* $\mathcal{Q}'$. $\mathcal{Q}'$ is *dominated* if there exists a coterie $\mathcal{Q}$ that dominates it, and is *non-dominated* if no such coterie exists. In this paper we consider only coteries, although some of our constructions will be dominated.

4 Masking quorum systems

In this section we introduce *masking quorum systems*, which can be used to mask the arbitrarily faulty behavior of data repositories. To motivate our definition, suppose that the replicated variable x is written with quorum Q_1, and that subsequently x is read using quorum Q_2. If B is the set of arbitrarily faulty servers, then $(Q_1 \cap Q_2) \setminus B$ is the set of correct servers that possess the latest value for x. In order for the client to obtain this value, the client must be able to locate a value/timestamp pair returned by a set of servers that could not all be faulty. In addition, for availability we require that there be no set of faulty servers that can disable all quorums.

Definition 4.1 A quorum system $\mathcal{Q}$ is a *masking quorum system* for a fail-prone system $\mathcal{B}$ if the following properties are satisfied.

M1: $\forall Q_1, Q_2 \in \mathcal{Q}\ \forall B_1, B_2 \in \mathcal{B}: \quad (Q_1 \cap Q_2) \setminus B_1 \not\subseteq Q_2 \cap B_2$

M2: $\forall B \in \mathcal{B}\ \exists Q \in \mathcal{Q}: \quad B \cap Q = \emptyset$

□

It is not difficult to verify that a masking quorum system enables a client to obtain the correct answer from the service. The write operation is implemented as described in Section 3, and the read operation becomes:

Read: For a client to read a variable x, it queries each server in some quorum Q to obtain a set of value/timestamp pairs $A = \{\langle v_u, t_u \rangle\}_{u \in Q}$. The client computes the set

$$A' = \{\langle v, t \rangle \ : \ \exists B^+ \subseteq Q\ [\forall B \in \mathcal{B}\ [B^+ \not\subseteq B] \ \wedge \ \forall u \in B^+\ [v_u = v \wedge t_u = t]]\}.$$

The client then chooses the pair $\langle v, t \rangle$ in A' with the highest timestamp, and chooses v as the result of the read operation; if A' is empty, the client returns $\perp$ (a null value).

Lemma 4.1 A read operation that is concurrent with no write operations returns the value written by the last preceding write operation in some serialization of all preceding write operations.

Proof. Let W denote the set of write operations preceding the read. The read operation will return the value written in the write operation in W with the highest timestamp, since, by the construction of masking quorum systems, this value/timestamp pair will appear in A' and will have the highest timestamp in A' (any pair with a higher timestamp will be returned only by servers in some $B \in \mathcal{B}$). So, it suffices to argue that there is a serialization of the writes in W in which this write operation appears last, or in other words, that this write operation precedes no other write operation in W. This is immediate, however, as if it did precede another write operation in W, that write operation would have a higher timestamp. □

The above lemma implies that the protocol above implements a *single-writer single-reader safe* variable [Lam86]. From these, multi-writer multi-reader atomic variables can be built using well-known constructions [Lam86, IS92, LTV96].

A necessary and sufficient condition for the existence of a masking quorum system (and a construction for one, if it exists) for any given fail-prone system $\mathcal{B}$ is given in the following theorem:

Theorem 4.1 Let $\mathcal{B}$ be a fail-prone system for a universe U. Then there exists a masking quorum system for $\mathcal{B}$ iff $\mathcal{Q} = \{U \setminus B : B \in \mathcal{B}\}$ is a masking quorum system for $\mathcal{B}$.

Proof. Obviously, if $\mathcal{Q}$ is a masking quorum system for $\mathcal{B}$, then one exists. To show the converse, assume that $\mathcal{Q}$ is not a masking quorum. Since M2 holds in $\mathcal{Q}$ by construction, there exist $Q_1, Q_2 \in \mathcal{Q}$ and $B', B'' \in \mathcal{B}$, such that $Q_1 \cap Q_2 \setminus B' \subseteq B'' \cap Q_2$. Let $B_1 = U \setminus Q_1$ and $B_2 = U \setminus Q_2$. By the construction of $\mathcal{Q}$, we know that $B_1, B_2 \in \mathcal{B}$. By M2, any masking quorum system for $\mathcal{B}$ must contain quorums $Q_1' \subseteq Q_1$, $Q_2' \subseteq Q_2$. However, for any such Q_1', Q_2', it is the case that $Q_1' \cap Q_2' \setminus B' \subseteq Q_1 \cap Q_2 \setminus B' \subseteq B'' \cap Q_2 \subseteq B''$. Also, since $Q_1' \cap Q_2' \setminus B' \subseteq Q_2'$, we have $Q_1' \cap Q_2' \setminus B' \subseteq B'' \cap Q_2'$, thus violating M1. Therefore, there does not exist a masking quorum system for $\mathcal{B}$ under the assumption that $\mathcal{Q}$ is not a masking quorum system for $\mathcal{B}$. □

Corollary 4.1 Let $\mathcal{B}$ be a fail-prone system for a universe U. Then there exists a masking quorum system for $\mathcal{B}$ iff for all $B_1, B_2, B_3, B_4 \in \mathcal{B}$, $U \not\subseteq B_1 \cup B_2 \cup B_3 \cup B_4$. In particular, suppose that $\mathcal{B} = \{B \subseteq U : |B| = f\}$. Then, there exists a masking quorum system for $\mathcal{B}$ iff $n > 4f$.

Proof. By Theorem 4.1, there is a masking quorum for $\mathcal{B}$ iff $\mathcal{Q} = \{U \setminus B : B \in \mathcal{B}\}$ is a masking quorum for $\mathcal{B}$. By construction, $\mathcal{Q}$ is a masking quorum iff M1 holds for $\mathcal{Q}$, i.e., iff for all $B_1, B_2, B_3, B_4 \in \mathcal{B}$:

$$(U \setminus B_1) \cap (U \setminus B_2) \setminus B_3 \not\subseteq B_4 \;\Leftrightarrow\; U \setminus (B_1 \cup B_2) \not\subseteq B_3 \cup B_4 \;\Leftrightarrow\; U \not\subseteq B_1 \cup B_2 \cup B_3 \cup B_4.$$

□

The following theorem was proved in [NW94] for benign-failure quorum systems, and holds for masking quorums as well (as a result of M1). Let $c(\mathcal{Q})$ denote the size of the smallest quorum of $\mathcal{Q}$.

Theorem 4.2 If $\mathcal{Q}$ is a quorum system over a universe of n elements, then $L(\mathcal{Q}) \geq \max\{\frac{1}{c(\mathcal{Q})}, \frac{c(\mathcal{Q})}{n}\}$.

The proof of this theorem in [NW94] employs rather complex methods. Here we present a simpler proof of their theorem.

Proof. Let w be any strategy for the quorum system $\mathcal{Q}$, and fix $Q_1 \in \mathcal{Q}$ such that $|Q_1| = c(\mathcal{Q})$. Summing the loads induced by w on all the elements of Q_1 we obtain:

$$\sum_{u \in Q_1} l_w(u) \;=\; \sum_{u \in Q_1} \sum_{Q_i \ni u} w(Q_i) \;=\; \sum_{Q_i} \sum_{u \in (Q_1 \cap Q_i)} w(Q_i) \;\geq\; \sum_{Q_i} w(Q_i) \;=\; 1$$

Therefore, there exists some element in Q_1 that suffers a load of at least $\frac{1}{c(\mathcal{Q})}$.

Similarly, summing the total load induced by w on all of the elements of the universe, we get:

$$\sum_{u \in U} l_w(u) \;=\; \sum_{u \in U} \sum_{Q_i \ni u} w(Q_i) \;=\; \sum_{Q_i} \sum_{u \in Q_i} w(Q_i) \;\geq\; \sum_{Q_i} c(\mathcal{Q}) w(Q_i) \;=\; c(\mathcal{Q})$$

(Here, the inequality results from the minimality of $c(\mathcal{Q})$.) Therefore, there exists some element in U that suffers a load of at least $\frac{c(\mathcal{Q})}{n}$. □

Since any masking quorum system is a quorum system, we have, *a fortiori*:

Corollary 4.2 If $\mathcal{Q}$ is a masking quorum system over a universe of n elements, then $L(\mathcal{Q}) \geq \max\{\frac{1}{c(\mathcal{Q})}, \frac{c(\mathcal{Q})}{n}\}$ and thus $L(\mathcal{Q}) \geq \frac{1}{\sqrt{n}}$.

Below we give several examples of masking quorum systems and describe their properties. When we refer to a system as non-dominated, we mean within the class of masking quorum systems for the given fail-prone system $\mathcal{B}$.

Example 4.1 (*Threshold*) Suppose that $\mathcal{B} = \{B \subseteq U : |B| = f\}$, $n > 4f$. Note that this corresponds to the usual threshold assumption that up to f servers may fail. Then, the quorum system $\mathcal{Q} = \{Q \subseteq U : |Q| = \lceil \frac{n+2f+1}{2} \rceil\}$ is a masking quorum system for $\mathcal{B}$. M1 is satisfied because any $Q_1, Q_2 \in \mathcal{Q}$ will intersect in at least $2f+1$ elements. M2 holds because $\lceil \frac{n+2f+1}{2} \rceil \leq n \perp f$. This system is non-dominated, and a strategy that assigns equal probability to each quorum induces a load of $\frac{1}{n}\lceil \frac{n+2f+1}{2} \rceil$ on the system. By Corollary 4.2, this load is in fact the load of the system. □

The following example is interesting since its load decreases as a function of n, and since it demonstrates a method for ensuring system-wide consistency in the face of Byzantine failures while requiring the involvement of fewer than a majority of the correct servers.

Example 4.2 (*Grid quorums*) Suppose that the universe of servers is of size $n = k^2$ for some integer k and that $\mathcal{B} = \{B \subseteq U : |B| = f\}$, $3f + 1 \leq \sqrt{n}$. Arrange the universe into a $\sqrt{n} \times \sqrt{n}$ grid, as shown in Figure 1. Denote the rows and columns of the grid by R_i and C_i, respectively, where $1 \leq i \leq \sqrt{n}$. Then, the quorum system

$$\mathcal{Q} = \left\{ C_j \cup \bigcup_{i \in I} R_i : I, \{j\} \subseteq \{1 \ldots \sqrt{n}\}, |I| = 2f + 1 \right\}$$

is a masking quorum system for $\mathcal{B}$. M1 holds since every pair of quorums intersect in at least $2f+1$ elements (the column of one quorum intersects the $2f + 1$ rows of the other), and M2 holds since for any choice of f faulty elements in the grid, $2f + 1$ full rows and a column remain available. A strategy that assigns equal probability to each quorum induces a load of $\frac{(2f+2)\sqrt{n}-(2f+1)}{n}$, and again by Corollary 4.2, this is the load of the system. □

Note that by choosing $\mathcal{B} = \{\emptyset\}$ (i.e., $f = 0$) in the example above, the resulting construction has a load of $O(\frac{1}{\sqrt{n}})$, which asymptotically meets the bounds given in Corollary 4.2.

The grid construction above is dominated: e.g., all of the column elements above the top row in each quorum can be removed (and the quorum system remains dominated). The grid construction would work for some non-square grids as well, e.g., triangular grids [Lov73], or CWlog grids [PW95] (that contain 2^i rows of length 2^i each). As the square grid already possesses a load within a constant factor of the optimal, we do not pursue such constructions further.

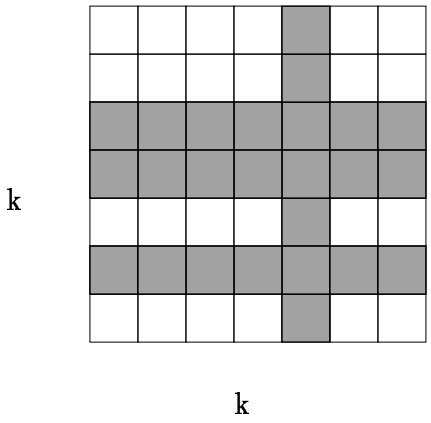

Figure 1: Grid construction, $k \times k = n$.

Example 4.3 (*Partition*) Suppose that $\mathcal{B} = \{B_1, \ldots, B_m\}$, $m > 4$, is a partition of U where $B_i \neq \emptyset$ for all i, $1 \leq i \leq m$. This choice of $\mathcal{B}$ could arise, for example, in a wide area network composed of multiple local clusters, each containing some B_i, and expresses the assumption that at any time, at most one cluster is faulty. Then, any collection of nonempty sets $\hat{B}_i \subseteq B_i$, $1 \leq i \leq m$, can be thought of as 'super-elements' in a universe of m, with a threshold assumption $f = 1$. Therefore, the following is a masking quorum system for $\mathcal{B}$:

$$\mathcal{Q} = \left\{ \bigcup_{i \in I} \hat{B}_i \ : \ I \subseteq \{1, \ldots, m\}, |I| = \lceil \tfrac{m+3}{2} \rceil \right\}$$

M1 is satisfied because the intersection of any two quorums contains elements from at least three sets in $\mathcal{B}$. M2 holds since there is no $B \in \mathcal{B}$ that intersects all quorums. $\mathcal{Q}$ is non-dominated iff $|\hat{B}_i| = 1$ for all i, $1 \leq i \leq m$. A strategy that assigns equal probability to each quorum induces a load of $\frac{1}{m}\lceil \frac{m+3}{2} \rceil$ on the system regardless of the size of each $\hat{B}_i$, and again Corollary 4.2 implies that this is the load of the system.

If $m = k^2$ for some k, then a more efficient construction can be achieved by forming the grid construction from Example 4.2 on the 'super elements' $\{\hat{B}_i\}$, achieving a load of $\frac{4\sqrt{m}-3}{m}$. □

5 Variations

5.1 Dissemination quorum systems

As a special case of services which may employ quorums in a Byzantine environment, we now consider applications in which the service is a repository for self-verifying information, i.e., information that only clients can create and to which clients can detect any attempted modification

by a faulty server. A natural example is a database of *public key certificates* as found in many public key distribution systems (e.g., [CCIT88, TA91, LABW92, Ken93]). A public key certificate is a structure containing a name for a user and a public key, and represents the assertion that the indicated public key can be used to authenticate messages from the indicated user. This structure is digitally signed (e.g., [RSA78]) by a *certification authority* so that anyone with the public key of this authority can verify this assertion and, providing it trusts the authority, use the indicated public key to authenticate the indicated user. Due to this signature, it is not possible for a faulty server to undetectably modify a certificate it stores. However, a faulty server *can* undetectably suppress a change from propagating to clients, simply by ignoring an update from a certification authority. This could have the effect, e.g., of suppressing the revocation of a key that has been compromised.

As can be expected, the use of digital signatures to verify data improves the cost of accessing replicated data. To support such a service, we employ a *dissemination quorum system*, which has weaker requirements than masking quorums, but which nevertheless ensures that in applications like those above, self-verifying writes will be propagated to all subsequent read operations despite the arbitrary failure of some servers. To achieve this, it suffices for the intersection of every two quorums to not be contained in any set of potentially faulty servers (so that a written value can propagate to a read). And, supposing that operations are required to continue in the face of failures, there should be quorums that a faulty set cannot disable.

Definition 5.1 A quorum system $\mathcal{Q}$ is a *dissemination quorum system* for a fail-prone system $\mathcal{B}$ if the following properties are satisfied.

D1: $\forall Q_1, Q_2 \in \mathcal{Q}\ \forall B \in \mathcal{B}: \quad Q_1 \cap Q_2 \not\subseteq B$

D2: $\forall B \in \mathcal{B}\ \exists Q \in \mathcal{Q}: \quad B \cap Q = \emptyset$

□

A dissemination quorum system will suffice for propagating self-verifying information as in the application described above. The write operation is implemented as described in Section 3, and the read operation becomes:

Read: For a client to read a variable x, it queries each server in some quorum Q to obtain a set of value/timestamp pairs $A = \{<v_u, t_u>\}_{u \in Q}$. The client then discards those pairs that are not verifiable (e.g., using an appropriate digital signature verification algorithm) and chooses from the remaining pairs the pair $<v, t>$ with the largest timestamp. v is the result of the read operation.

It is important to note that timestamps must be included as part of the self-verifying information, so they cannot be undetectably altered by faulty servers. In the case of the application described above, existing standards for public key certificates (e.g., [CCIT88]) already require a real-time timestamp in the certificate.

The following two lemmata prove correctness of the above protocol using dissemination quorum systems:

Lemma 5.1 A read operation that is concurrent with no write operations returns the value written by the last preceding write operation in some serialization of all preceding write operations.

Proof. Let W denote the set of write operations preceding the read. The read operation will return the value written in the write operation in W with the highest timestamp, since the quorum in which that write completed intersects the quorum in which the read occurs in at least one correct server. So, it suffices to argue that there is a serialization of the writes in W in which this write operation appears last, or in other words, that this write operation precedes no other write operation in W. This is immediate, however, as if it did precede another write operation in W, that write operation would have a higher timestamp. □

In this case, we can also prove the following property.

Lemma 5.2 A read operation that is concurrent with one or more write operations returns either the value written by the last preceding write operation in some serialization of all preceding write operations, or any of the values being written in the concurrent write operations.

Proof. Due to the assumption of self-verifying data, the read operation must return a value written in a preceding or concurrent write operation. If the read operation does not return a value written in a concurrent write operation, then it must return the value written by the last preceding write operation in some serialization of all preceding write operations, by an argument similar to that for Lemma 4.1. □

The above lemmata imply that the protocol above implements a *single-writer single-reader regular* variable [Lam86]. Theorems analogous to the ones given for masking quorum systems above are easily derived for dissemination quorums. Below, we list these results without proof.

Theorem 5.1 Let $\mathcal{B}$ be a fail-prone system for a universe U. Then there exists a dissemination quorum system for $\mathcal{B}$ iff $\mathcal{Q} = \{U \setminus B : B \in \mathcal{B}\}$ is a dissemination quorum system for $\mathcal{B}$.

Corollary 5.1 Let $\mathcal{B}$ be a fail-prone system for a universe U. Then there exists a dissemination quorum system for $\mathcal{B}$ iff for all $B_1, B_2, B_3 \in \mathcal{B}$, $U \not\subseteq B_1 \cup B_2 \cup B_3$. In particular, suppose that $\mathcal{B} = \{B \subseteq U : |B| = f\}$. Then, there exists a dissemination quorum system for $\mathcal{B}$ iff $n > 3f$.

Corollary 5.2 If $\mathcal{Q}$ is a dissemination quorum system over a universe of n elements, then $L(\mathcal{Q}) \geq \max\{\frac{1}{c(\mathcal{Q})}, \frac{c(\mathcal{Q})}{n}\}$, and thus also $L(\mathcal{Q}) \geq \frac{1}{\sqrt{n}}$.

Below, we provide several example constructions of dissemination quorum systems.

Example 5.1 (*Threshold*) Suppose that $\mathcal{B} = \{B \subseteq U : |B| = f\}$, $n > 3f$. Note that this corresponds to the usual threshold assumption that up to f servers may fail. Then, the quorum system $\mathcal{Q} = \{Q \subseteq U : |Q| = \lceil \frac{n+f+1}{2} \rceil\}$ is a dissemination quorum system for $\mathcal{B}$. D1 is satisfied because any $Q_1, Q_2 \in \mathcal{Q}$ will intersect in at least $f+1$ elements. D2 holds because $\lceil \frac{n+f+1}{2} \rceil \leq n \perp f$. This system is non-dominated, and a strategy that assigns equal probability to each quorum induces a load of $\frac{1}{n}\lceil \frac{n+f+1}{2} \rceil$ on the system. By Corollary 5.2, this load is in fact the load of the system. □

Example 5.2 (*Grid*) Let the universe be arranged in a grid as in Example 4.2 above, and let $\mathcal{B} = \{B \subseteq U : |B| = f\}$, $2f + 1 \leq \sqrt{n}$. Then, the quorum system

$$\mathcal{Q} = \left\{ C_j \cup \bigcup_{i \in I} R_i : I, \{j\} \subseteq \{1 \ldots \sqrt{n}\}, |I| = f + 1 \right\}$$

is a dissemination quorum system for $\mathcal{B}$. D1 holds since every pair of quorums intersect in at least $f + 1$ elements, and D2 holds since for any choice of f faulty elements in the grid, $f + 1$ full rows and a column remain available. A strategy that assigns equal probability to each quorum induces a load of $\frac{(f+2)\sqrt{n}-(f+1)}{n}$, and by Corollary 5.2, this is the load of the system. □

Example 5.3 (*Partition*) Suppose that $\mathcal{B} = \{B_1, \ldots, B_{2k}\}$, $k > 1$, is a partition of U as in example 4.3 above. Then, for any collection of nonempty sets $\hat{B}_i \subseteq B_i$, $1 \leq i \leq 2k$, the following is a dissemination quorum system for $\mathcal{B}$:

$$\mathcal{Q} = \left\{ \bigcup_{i \in I} \hat{B}_i \ : \ I \subseteq \{1, \ldots, 2k\}, |I| = k + 1 \right\}$$

D1 is satisfied because the intersection of any two quorums contains elements from at least two sets in $\mathcal{B}$. D2 holds since there is no $B \in \mathcal{B}$ that intersects all quorums. $\mathcal{Q}$ is non-dominated iff $|\hat{B}_i| = 1$ for all i, $1 \leq i \leq 2k$. A strategy that assigns equal probability to each quorum induces a load of $\frac{k+1}{2k}$ on the system regardless of the size of each $\hat{B}_i$, and again Corollary 5.2 implies that this is the load of the system.

An improved dissemination quorum system is achieved by forming the grid construction of example 5.2 above on the 'super-elements' $\{\hat{B}_i\}$, achieving a load of $\frac{3\sqrt{2k}-2}{2k}$. □

5.2 Opaque masking quorum systems

Masking quorums impose a requirement that clients know the fail-prone system $\mathcal{B}$, while there may be reasons that clients should not be required to know this. First, it somewhat complicates the client's read protocol. Second, by revealing the failure scenarios for which the system was designed, the system also reveals the failure scenarios to which it is vulnerable, which could be exploited by an attacker to guide an active attack against the system. By not revealing the fail-prone system to clients, and indeed giving each client only a small fraction of the possible quorums, the system can somewhat obscure (though perhaps not secure in any formal sense) the failure scenarios to which it is vulnerable, especially in the absence of client collusion.

In this section we describe one way to modify the masking quorum definition of Section 4 to be *opaque*, i.e., to eliminate the need for clients to know $\mathcal{B}$. In the absence of the client knowing $\mathcal{B}$, the only method of which we are aware for the client to reduce a set of replies from servers to a single reply from the service is via *voting*, i.e., choosing the reply that occurs most often. In order for this reply to be the correct one, however, we must strengthen the requirements on our quorum systems. Specifically, suppose that the variable x is written with quorum Q_1, and that subsequently x is read with quorum Q_2. If B is the set of arbitrarily faulty servers, then $(Q_1 \cap Q_2) \setminus B$ is the set of correct servers that possess the latest value for x (see Figure 2). In order for the client to obtain this value by vote, this set must be larger than the set of faulty servers that are allowed to respond, i.e., $Q_2 \cap B$. Moreover, since these faulty servers can "team up" with the out-of-date but correct servers in an effort to suppress the write operation, the number of correct, up-to-date servers that reply must be no less than the number of faulty or out-of-date servers that can reply, i.e., $(Q_2 \cap B) \cup (Q_2 \setminus Q_1)$.

Definition 5.2 A quorum system $\mathcal{Q}$ is an *opaque masking quorum system* for a fail-prone system $\mathcal{B}$ if the following properties are satisfied.

O1: $\forall Q_1, Q_2 \in \mathcal{Q} \ \forall B \in \mathcal{B} : \quad |(Q_1 \cap Q_2) \setminus B| \geq |(Q_2 \cap B) \cup (Q_2 \setminus Q_1)|$

O2: $\forall Q_1, Q_2 \in \mathcal{Q} \ \forall B \in \mathcal{B} : \quad |(Q_1 \cap Q_2) \setminus B| > |Q_2 \cap B|$

O3: $\forall B \in \mathcal{B} \ \exists Q \in \mathcal{Q} : \quad B \cap Q = \emptyset$

□

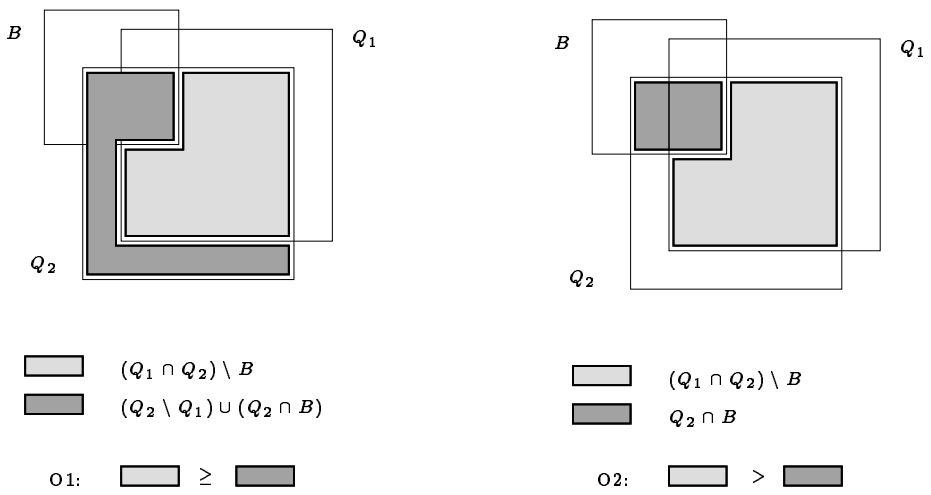

Figure 2: O1 and O2

Note that O1 admits the possibility of equality in size between $(Q_1 \cap Q_2) \setminus B$ and $(Q_2 \cap B) \cup (Q_2 \setminus Q_1)$. Equality is sufficient since, in the case that the faulty servers "team up" with the correct but out-of-date servers in Q_2, the value returned from $(Q_1 \cap Q_2) \setminus B$ will have a higher timestamp than that returned by $(Q_2 \cap B) \cup (Q_2 \setminus Q_1)$. Therefore, in the case of a tie, a reader can choose the value with the higher timestamp. It is interesting to note that a strong inequality in O1 would permit a correct implementation of a single-reader singer-writer safe variable that does not use timestamps (by taking the majority value in a read operation).

It is not difficult to verify that an opaque masking quorum system enables a client to obtain the correct answer from the service. The write operation is implemented as described in Section 1, and the read operation becomes:

Read: For a client to read a variable x, it queries each server in some quorum Q to obtain a set of value/timestamp pairs $A = \{\langle v_u, t_u \rangle\}_{u \in Q}$. The client chooses the pair $\langle v, t \rangle$ that appears most often in A, and if there are multiple such values, the one with the highest timestamp. The value v is the result of the read operation.

Opaque masking quorum systems, combined with the access protocol described previously, provide the same semantics as regular masking quorum systems. The proof is almost identical to that for regular masking quorums.

Lemma 5.3 A read operation that is concurrent with no write operations returns the value written by the last preceding write operation in some serialization of all preceding write operations.

Below we give several examples of opaque masking quorum systems (or just "opaque quorum systems") and describe their properties. When we refer to a system as non-dominated, we mean within the class of opaque quorum systems for the given fail-prone system $\mathcal{B}$.

Example 5.4 (*Threshold*) Suppose that $\mathcal{B} = \{B \subseteq U : |B| = f\}$ where $n \geq 5f$ and $f > 0$. Then, the quorum system $\mathcal{Q} = \{Q \subseteq U : |Q| = \lceil \frac{2n+2f}{3} \rceil\}$ is an opaque quorum system for $\mathcal{B}$. O1 is satisfied because for any $Q_1, Q_2 \in \mathcal{Q}$ and any $B \in \mathcal{B}$,

$$|(Q_1 \cap Q_2) \setminus B| \geq \left\lceil \frac{n+4f}{3} \right\rceil \perp f = f + \left\lceil \frac{n-2f}{3} \right\rceil \geq f + (\lceil \frac{2n+2f}{3} \rceil \perp \lceil \frac{n+4f}{3} \rceil) \geq |(Q_2 \cap B) \cup (Q_2 \setminus Q_1)|.$$

Similarly, O2 is satisfied because

$$|(Q_1 \cap Q_2) \setminus B| \geq \lceil \tfrac{n+4f}{3} \rceil \perp f \geq 3f \perp f > f \geq |Q_2 \cap B|.$$

O3 holds since $\lceil \frac{2n+2f}{3} \rceil \leq n \perp f$. $\mathcal{Q}$ is non-dominated, and a strategy that assigns equal probability to each quorum induces a load of $\frac{1}{n}\lceil \frac{2n+2f}{3} \rceil$ on the system. Corollary 4.2 implies that this load is in fact the load of the system. □

The next theorem proves a resilience bound for opaque quorum systems.

Theorem 5.2 Suppose that $\mathcal{B} = \{B \subseteq U : |B| = f\}$. There exists an opaque quorum system for $\mathcal{B}$ iff $n \geq 5f$.

Proof. That $n \geq 5f$ is sufficient is already proved in Example 5.1 above. Now suppose that $\mathcal{Q}$ is an opaque quorum system for $\mathcal{B}$. Fix any $Q_1 \in \mathcal{Q}$ such that $|Q_1| \leq n \perp f$ (Q_1 exists by O3); note that $|Q_1| > f$ by O2. Choose $B_1 \subseteq Q_1$, $|B_1| = f$, and some $Q_2 \in \mathcal{Q}$ such that $Q_2 \subseteq U \setminus B_1$ (Q_2 exists by O3). Then $|Q_1 \cap Q_2| \leq n \perp 2f$. By O2, $|Q_1 \cap Q_2| \geq f$, and therefore there is some $B_2 \in \mathcal{B}$ such that $B_2 \subseteq Q_1 \cap Q_2$. Then

$$|(Q_2 \cap Q_1) \setminus B_2| = |Q_2 \cap Q_1| \perp |B_2| \leq n \perp 3f \quad (1)$$

$$|(Q_1 \setminus Q_2) \cup (Q_1 \cap B_2)| = |Q_1 \setminus Q_2| + |B_2| \geq |B_1| + |B_2| = 2f \quad (2)$$

By O1, (1) $\geq$ (2), therefore $n \perp 3f \geq 2f$, and we have $n \geq 5f$. □

Example 5.5 (*Partition*) Suppose that $\mathcal{B} = \{B_1, \ldots, B_{3k}\}$, $k > 1$, is a partition of U where $B_i \neq \emptyset$ for all i, $1 \leq i \leq 3k$. Choose any collection of sets $\hat{B}_i \subseteq B_i$, $1 \leq i \leq 3k$, such that $|\hat{B}_i| = c$ for a fixed constant $c > 0$. Then,

$$\mathcal{Q} = \left\{ \bigcup_{i \in I} \hat{B}_i \ : \ I \subseteq \{1, \ldots, 3k\}, |I| = 2k+1 \right\}$$

is an opaque quorum system for $\mathcal{B}$. O1 is satisfied since for any $Q_1, Q_2 \in \mathcal{Q}$ and $B \in \mathcal{B}$,

$$|(Q_1 \cap Q_2) \setminus B| \geq (k+2)c \perp c = (k+1)c > kc = c + ((2k+1)c \perp (k+2)c) \geq |(Q_2 \cap B) \cup (Q_2 \setminus Q_1)|.$$

Similarly, O2 is satisfied since

$$|(Q_1 \cap Q_2) \setminus B| \geq (k+2)c \perp c = (k+1)c > kc \geq c \geq |Q_2 \cap B|.$$

Finally, O3 is satisfied since no $B \in \mathcal{B}$ intersects all quorums. $\mathcal{Q}$ is non-dominated iff $c = 1$. A strategy that assigns equal probability to each quorum induces a load of $\frac{2k+1}{3k}$ on the system regardless of the value of c, and again Corollary 4.2 (considering each $\hat{B}_i$ as a single element of a universe of size $3k$) implies that this is the load of the system. □

Unlike the case for regular masking quorum systems, an open problem is to find a technique for testing whether, given a fail-prone system $\mathcal{B}$, there exists an opaque quorum system for $\mathcal{B}$ (other than an exhaustive search of all subsets of 2^U).

In the constructions in Examples 5.3, 5.4 and 5.5 the resulting quorum systems exhibited loads that at best were constant as a function of n. In the case of masking quorum systems, we were able to exhibit quorum systems whose load decreased as a function of n, namely the grid quorums. A natural question is whether there exists an opaque quorum system for any fail-prone system $\mathcal{B}$ that has load that decreases as a function of n. In this section, we answer this question in the negative: we show a lower bound of $\frac{1}{2}$ of the load for any opaque quorum system construction, regardless of the fail-prone system.

Theorem 5.3 The load of any opaque quorum system is at least $\frac{1}{2}$.

Proof. O1 implies that for any $Q_1, Q_2 \in \mathcal{Q}$, $|Q_1 \cap Q_2| \geq |Q_1 \setminus Q_2|$, and thus $|Q_1 \cap Q_2| \geq \frac{|Q_1|}{2}$. Let w be any strategy for the quorum system $\mathcal{Q}$, and fix any $Q_1 \in \mathcal{Q}$. Then, the total load induced by w on the elements of Q_1 is:

$$\sum_{u \in Q_1} l_w(u) \;=\; \sum_{u \in Q_1} \sum_{Q_i \ni u} w(Q_i) \;=\; \sum_{Q_i} \sum_{u \in Q_1 \cap Q_i} w(Q_i) \;\geq\; \sum_{Q_i} \frac{|Q_1|}{2} w(Q_i) \;=\; \frac{|Q_1|}{2}$$

Therefore, there must be some server in Q_1 that suffers a load at least $\frac{1}{2}$. □

We now present a generic construction of an opaque quorum system for $\mathcal{B} = \{\emptyset\}$ and increasingly large universe sizes n, that has a load that tends to $\frac{1}{2}$ as n grows.[1] We give this construction primarily to show that the lower bound of $\frac{1}{2}$ is tight; due to the requirement that $\mathcal{B} = \{\emptyset\}$, this construction is not of practical use for coping with Byzantine failures.

Example 5.6 Suppose that the universe of servers is $U = \{u_1, \ldots, u_n\}$ where $n = 2^\ell$ for some $\ell > 2$, and that $\mathcal{B} = \{\emptyset\}$. Consider the $n \times n$ Hadamard matrix $H(\ell)$, constructed recursively as follows:[2]

$$H(1) \;=\; \begin{bmatrix} \perp 1 & \perp 1 \\ \perp 1 & 1 \end{bmatrix}$$

$$H(k) \;=\; \begin{bmatrix} H(k \perp 1) & H(k \perp 1) \\ H(k \perp 1) & \perp H(k \perp 1) \end{bmatrix}$$

$H(\ell)$ has the property that $H(\ell)H(\ell)^T = nI$, where I is the $n \times n$ identity matrix. Using well-known inductive arguments [Hal86, Ch. 14], it can be shown that (i) the first row and column consist entirely of $\perp 1$'s, (ii) the i-th row and i-th column, for each $i \geq 2$, has 1's in $\frac{n}{2}$ positions (and similarly for $\perp 1$'s), and (iii) any two rows (and any two columns) $i, j \geq 2$ have identical elements in $\frac{n}{2}$ positions, i.e., 1's in $\frac{n}{4}$ common positions and $\perp 1$'s in $\frac{n}{4}$ common positions.

We treat the rows of $H(\ell)$ as indicators of subsets of U. That is, let $Q_i = \{u_j : H(\ell)[i,j] = 1\}$ be the set defined by the i-th row, $1 \leq i \leq n$. Note that $Q_1 = \emptyset$ and that u_1 is not included in any Q_i. We claim that the system $\mathcal{Q} = \{Q_2, ..., Q_n\}$ is an opaque quorum system for $\mathcal{B}$. Using properties (i)–(iii) above, we have that $|Q_i| = \frac{n}{2}$ for each $i \geq 2$; that each u_i, $i \geq 2$, is in exactly $\frac{n}{2}$ of the sets $Q_2, \ldots, Q_n$; and that for any $i, j \geq 2$, if $i \neq j$ then $|Q_i \cap Q_j| = \frac{n}{4}$. From these, the O1 and O2 requirements can be quickly verified, and a load of $\frac{\frac{n}{2}}{n-1}$ can be achieved, e.g., with a strategy that assigns equal probability to each quorum. □

6 Faulty clients

So far, we have been concerned with providing a consistent service to a set of correct clients. In this section, we extend our treatment to address faulty clients in addition to faulty servers, as would be required if servers are allowed to act as (or on behalf of) clients. Since updates may now be generated by faulty clients, we can make no assumption of self-verifying data, and thus use masking

[1] This construction slightly improves on a similar idea suggested to us by Andrew Odlyzko.

[2] The more common form of Hadamard matrices is -1 times the form we use. However, the relevant characteristics of Hadamard matrices are invariant to multiplication by -1.

quorum systems (Section 4) in our read and write protocols. We focus on ensuring the consistency of the data stored at the replicated service as seen by correct clients only. Since a faulty client may not complete a write operation at a quorum of servers, or may even write different values to different servers, in this section we modify the write protocol to include an *update* protocol implemented by the servers that prevents clients from leaving the service in an inconsistent state. This update protocol could be implemented using well-known agreement protocols (e.g., [LSP82, BT85]), but only if the given fail-prone system $\mathcal{B}$ has the property that each $B \in \mathcal{B}$ is of size less than $|U|/3$, and only by involving all of the servers in the system. We describe two protocols that are correct for any fail-prone system $\mathcal{B}$ for which a masking quorum exists, and that involve only a quorum of correct servers to complete an update operation. While we do not explicitly treat load in this section, this latter property is essential for the load measure that we have defined to be useful.

The write protocol

This section describes the protocol by which clients write the variable x replicated at each server. We replace the write operation of Section 3 by the following procedure:

Write: For a client c to write the value v, it queries each server in some quorum Q to obtain a set of value/timestamp pairs $A = \{<v_u, t_u>\}_{u \in Q}$; chooses a timestamp $t \in T_c$ greater than the highest timestamp value in A and greater than any timestamp it has chosen in the past; and performs *InitiateUpdate*(Q, v, t).

Note that writing the pair $<v, t>$ to the quorum Q is performed by executing the operation *InitiateUpdate*(Q, v, t). Servers execute corresponding events *DeliverUpdate*(c, v, t) where c is a client. If a correct server executes *DeliverUpdate*(c, v, t), and if $t \in T_c$ and is greater than the timestamp currently stored with the variable, then the server updates the value of the variable and its timestamp to v and t, respectively. Regardless of whether it updates the variable, it sends an acknowledgment message to c.

Correctness of this protocol depends on the following relationships among *InitiateUpdate* executions at clients and *DeliverUpdate* events at servers. How to implement the *InitiateUpdate* and *DeliverUpdate* primitives to satisfy these relationships is the topic of this section.

Integrity: If c is correct, then a correct server executes *DeliverUpdate*(c, v, t) only if c executed *InitiateUpdate*(Q, v, t) for some $Q \in \mathcal{Q}$.

Agreement: If a correct server executes *DeliverUpdate*(c, v, t) and a correct server executes *DeliverUpdate*(c, v', t), then $v = v'$.

Propagation: If a correct server executes *DeliverUpdate*(c, v, t), then eventually there exists a quorum $Q \in \mathcal{Q}$ such that every correct server in Q executes *DeliverUpdate*(c, v, t).

Validity: If a correct client c executes *InitiateUpdate*(Q, v, t) and all servers in Q are correct, then eventually a correct server executes *DeliverUpdate*(c, v, t).

Note that by Validity, if a correct client executes *InitiateUpdate*(Q, v, t) but Q contains a faulty server, then there is no guarantee that *DeliverUpdate*(c, v, t) will occur at any correct server; i.e., the write operation may have no effect. A correct server acknowledges each *DeliverUpdate*(c, v, t) execution as described above to inform c that *DeliverUpdate*(c, v, t) was indeed executed. If the client receives acknowledgments from a set B^+ of servers, such that $\forall B \in \mathcal{B} : B^+ \not\subseteq B$, then it is

certain that its write will be applied at all correct servers in some quorum Q (by Propagation). If the client receives acknowledgements from no such set B^+ of servers, then it must attempt the write operation again with a different quorum. By M2, some quorum with correct servers exists at any moment, and thus by repeatedly trying, a client can eventually make progress. In some cases, the client can achieve progress by incrementally accessing servers until it obtains acknowledgements from a quorum of them.

In order to argue correctness for this protocol, we have to adapt the definition of operation precedence to allow for the behavior of a faulty client. The reason is that it is unclear how to define when an operation at a faulty client starts or, in the case of a read, when it terminates, as the client can behave outside the specification of any protocol. We now simply say that a write operation that writes v with timestamp $t \in T_c$ *terminates* when all correct servers in some quorum have executed $DeliverUpdate(c, v, t)$. Let op_1 be any write operation and op_2 be any operation (read or write) executed by a correct client. We say that op_1 *precedes* op_2 if and only if op_1 terminates before op_2 starts (in real time, at the correct client); otherwise, and if op_2 does not precede op_1 and op_1 eventually terminates, then op_1 is *concurrent* with op_2. Note that by this definition, no operations precede an operation involving a faulty client, and every operation by a faulty client either precedes or is concurrent with every other operation. Given this change in definition, a proof very similar to that of Lemma 4.1 suffices to prove the following:

Lemma 6.1 A correct process' read operation that is concurrent with no write operations returns the value written by the last preceding write operation in some serialization of all preceding write operations.

The update operation: without signatures

The remaining protocol to describe is the update protocol for masking quorum systems that satisfies Integrity, Agreement, Propagation, and Validity. We first present an update protocol that does not use digital signatures. The protocol is shown in Figure 3.

Lemma 6.2(*Integrity*) If c is correct, then a correct server executes $DeliverUpdate(c, v, t)$ only if c executed $InitiateUpdate(Q, v, t)$ for some Q.

Proof. The first $\langle \mathsf{ready}, Q, c, v, t \rangle$ message from a correct server is sent only after it receives $\langle \mathsf{echo}, Q, c, v, t \rangle$ from each member of Q. Moreover, a correct member sends $\langle \mathsf{echo}, Q, c, v, t \rangle$ only if it receives $\langle \mathsf{update}, Q, v, t \rangle$ for some Q from c over an authenticated channel, i.e., only if c executed $InitiateUpdate(Q, v, t)$. □

Lemma 6.3(*Agreement*) If a correct server executes $DeliverUpdate(c, v, t)$ and a correct server executes $DeliverUpdate(c, v', t)$, then $v = v'$.

Proof. As argued in the previous lemma, for a correct server to execute $DeliverUpdate(c, v, t)$, $\langle \mathsf{echo}, Q, c, v, t \rangle$ must have been sent by all servers in Q. Similarly, $\langle \mathsf{echo}, Q', c, v', t \rangle$ must have been sent by all servers in Q'. Since every two quorums intersect in (at least) one correct member, and since any correct server sends $\langle \mathsf{echo}, *, c, \hat{v}, t \rangle$ for at most one value $\hat{v}$, v must be identical to v'. □

Lemma 6.4 If $\mathcal{Q}$ is a masking quorum system over a universe U with respect to a fail-prone system $\mathcal{B}$, then $\forall Q \in \mathcal{Q}\ \forall B_1, B_2, B_3 \in \mathcal{B}, Q \not\subseteq B_1 \cup B_2 \cup B_3$.

1. If a client executes *InitiateUpdate*(Q, v, t), then it sends <update, Q, v, t> to each member of Q.

2. If a server receives <update, Q, v, t> from a client c, and if the server has not previously received from c a message <update, Q', v', t'> where either $t' = t$ and $v' \neq v$ or $t' > t$, then the server sends <echo, Q, c, v, t> to each member of Q.

3. If a server receives identical echo messages <echo, Q, c, v, t> from every server in Q, then it sends <ready, Q, c, v, t> to each member of Q.

4. If a server receives identical ready messages <ready, Q, c, v, t> from a set B^+ of servers, such that $B^+ \not\subseteq B$ for all $B \in \mathcal{B}$, then it sends <ready, Q, c, v, t> to every member of Q if it has not done so already.

5. If a server receives identical ready messages <ready, Q, c, v, t> from a set Q^- of servers, such that for some $B \in \mathcal{B}$, $Q^- = Q \setminus B$, it executes *DeliverUpdate*(c, v, t).

Figure 3: An update protocol that does not use signatures

Proof. Assume otherwise for a contradiction, i.e., that there is a $Q \in \mathcal{Q}$ and $B_1, B_2, B_3 \in \mathcal{B}$ such that $Q \subseteq B_1 \cup B_2 \cup B_3$. By M2, there exists $Q' \in \mathcal{Q}$, $Q' \cap B_1 = \emptyset$. Then, $Q \cap Q' \subseteq B_2 \cup B_3$ and thus $(Q \cap Q') \setminus B_2 \subseteq B_3 \cap Q'$, contradicting M1. □

Lemma 6.5(*Propagation*) If a correct server executes *DeliverUpdate*(c, v, t), then eventually there exists a quorum $Q \in \mathcal{Q}$ such that every correct server in Q executes *DeliverUpdate*(c, v, t).

Proof. According to the protocol, the correct server that executed *DeliverUpdate*(c, v, t) received a message <ready, Q, c, v, t> from each server in $Q^- = Q \setminus B$ for some $Q \in \mathcal{Q}$ and $B \in \mathcal{B}$. Since, for some $B' \in \mathcal{B}$, (at least) all the members in $Q^- \setminus B'$ are correct, every correct member of Q receives <ready, Q, c, v, t> from each of the members of $B^+ = Q^- \setminus B'$. Since, $\forall B'' \in \mathcal{B}$, $Q^- \setminus B' \not\subseteq B''$ (by Lemma 6.4), the ready messages from B^+ cause each correct member of Q to send such a ready message. Consequently, *DeliverUpdate*(c, v, t) is executed by all of the correct members of Q. □

Lemma 6.6(*Validity*) If a correct client c executes *InitiateUpdate*(Q, v, t) and all servers in Q are correct, then eventually a correct server executes *DeliverUpdate*(c, v, t).

Proof. Since both the client and all of the members of Q are correct, <update, Q, v, t> will be received and echoed by every member in Q. Consequently, all the servers in Q will send <ready, Q, c, v, t> messages to the members of Q, and will eventually execute *DeliverUpdate*(c, v, t). □

The update operation: with signatures

The update protocol that we describe in this section requires that each server be able to *digitally sign* a message so that any party can reliably authenticate the message as having originated from that server, even if the message is received only via an (untrusted) intermediary. Digital signature algorithms are well-known (e.g., [RSA78]). In the remainder of this section, we use m_u to denote

the message m signed by server u. The use of digital signatures results in an Update protocol that exchanges significantly less messages than the protocol in Figure 3. This protocol is shown in Figure 4.

---

1. If a client executes $InitiateUpdate(Q, v, t)$, then it sends $\langle \mathsf{update}, Q, v, t \rangle$ to each member of Q.

2. If a server u receives $\langle \mathsf{update}, Q, v, t \rangle$ from a client c, and if u has not previously received from c a message $\langle \mathsf{update}, Q', v', t' \rangle$ where either $t' = t$ and $v' \neq v$ or $t' > t$, then u sends $\langle \mathsf{echo}, Q, c, v, t \rangle_u$ to c.

3. If a client c receives $\langle \mathsf{echo}, Q, c, v, t \rangle_u$ from every member of Q, then it sends $\{\langle \mathsf{echo}, Q, c, v, t \rangle_u\}_{u \in Q}$ to each member of Q.

4. If a server $u \in Q$ receives $\{\langle \mathsf{echo}, Q, c, v, t \rangle_u\}_{u \in Q}$, then it executes $DeliverUpdate(c, v, t)$, and sends $\{\langle \mathsf{echo}, Q, c, v, t \rangle_u\}_{u \in Q}$ to each member of Q if it has not already done so.

Figure 4: A update protocol that uses signatures

Lemma 6.7(*Integrity*) If c is correct, then a correct server executes $DeliverUpdate(c, v, t)$ only if c executed $InitiateUpdate(Q, v, t)$ for some Q.

Proof. For a correct server to execute $DeliverUpdate(c, v, t)$, it must receive $\{\langle \mathsf{echo}, Q, c, v, t \rangle_u\}_{u \in Q}$ for some quorum Q. But each correct server $u \in Q$ sends $\langle \mathsf{echo}, Q, c, v, t \rangle_u$ only after receiving $\langle \mathsf{update}, Q, v, t \rangle$ from c (on an authenticated channel), i.e., only if c executed $InitiateUpdate(Q, v, t)$. □

Lemma 6.8(*Agreement*) If a correct server executes $DeliverUpdate(c, v, t)$ and a correct server executes $DeliverUpdate(c, v', t)$, then $v = v'$.

Proof. As argued in the previous lemma, for a correct server to execute $DeliverUpdate(c, v, t)$, $\langle \mathsf{echo}, Q, c, v, t \rangle_u$ must have been sent by all $u \in Q$. Similarly, $\langle \mathsf{echo}, Q', c, v', t \rangle_u$ must have been sent by all $u \in Q'$. Since every two quorums intersect in (at least) one correct member, and since any correct server u sends $\langle \mathsf{echo}, *, c, \hat{v}, t \rangle_u$ for at most one value $\hat{v}$, v must be identical to v'. □

Lemma 6.9(*Propagation*) If a correct server executes $DeliverUpdate(c, v, t)$, then eventually there exists a quorum $Q \in \mathcal{Q}$ such that every correct server in Q executes $DeliverUpdate(c, v, t)$.

Proof. The correct server that executed $DeliverUpdate(c, v, t)$ received $\{\langle \mathsf{echo}, Q, c, v, t \rangle_u\}_{u \in Q}$ for some quorum Q. By forwarding this to all servers in Q in step 4 of the protocol, it will cause all correct servers in Q to execute $DeliverUpdate(c, v, t)$. □

Lemma 6.10(*Validity*) If a correct client c executes $InitiateUpdate(Q, v, t)$ and all servers in Q are correct, then eventually a correct server executes $DeliverUpdate(c, v, t)$.

Proof. Since both the client and all of the members of Q are correct, $\langle \mathsf{update}, Q, v, t \rangle$ will be received and echoed by every member in Q, and the client will forward these echos to all members of Q. Hence each member of Q will execute $DeliverUpdate(c, v, t)$. □

7 Conclusions

The literature contains an abundance of protocols that use quorums for accessing replicated data. This approach is appealing for constructing replicated services as it allows for increasing the availability and efficiency of the service while maintaining its consistency. Our work extends this successful approach to environments where both the servers and the clients of a service may deviate from their prescribed behavior in arbitrary ways. We introduced a new class of quorum systems, namely *masking* quorum systems, and devised protocols that use these quorums to enhance the availability of systems prone to Byzantine failures. We also explored two variations of our quorum systems, namely *dissemination* and *opaque masking* quorums, and for all of these classes of quorums we provided various constructions and analyzed the load they impose on the system.

Our work leaves a number of intriguing open challenges and directions for future work. One is to characterize the average performance of our quorum constructions and their load in less-than-ideal scenarios, e.g., when failures occur. More generally, in this work we described only quorum systems that are uniform, in the sense that any quorum is possible for both read and write operations. In practice it may be beneficial to employ quorum systems with distinguished *read quorums* and *write quorums*, with consistency requirements imposed only between pairs consisting of at least one write quorum. Although this does not seem to improve our lower bounds on the overall load that can be achieved, it may allow greater flexibility in trading between the availability of reads and writes.

Acknowledgments

We are grateful to Andrew Odlyzko for suggesting the use of Hadamard matrices to construct opaque masking quorum systems with an asymptotic load of $\frac{1}{2}$. We also thank Yehuda Afek and Michael Merritt for helpful discussions, and Rebecca Wright for many helpful comments on earlier versions of this paper. An insightful comment by Rida Bazzi led to a substantial improvement over a previous version of this paper.

References

[ADGM93] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM* 40(4):873–890, September 1993.

[ADGM90] Y. Afek, D. Dolev, E. Gafni, M. Merritt and N. Shavit. A bounded first-in first-enabled-solution to the l-exclusion problem. In *Proceedings of the International Workshop on Distributed Algorithms*, LNCS 486, Springer-Verlag, 1990.

[AE90] D. Agrawal and A. El Abbadi. Integrating security with fault-tolerant distributed databases. *Computer Journal* 33(1):71–78, February 1990.

[AE91] D. Agrawal and A. El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems* 9(1):1–20, 1991.

[And93] J. H. Anderson. Composite registers. *Distributed Computing* 6(3):141–154, 1993.

[ABD95] H. Attiya, A. Bar-Noy and D. Dolev. Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM* 42(1):124–142, January 1995.

[BHG87] P. A. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency control and recovery in database systems.* Addison-wesley, 1987.

[BT85] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM* 32(4):824–840, October 1985.

[CAA90] S. Y. Cheung, M. H. Ammar, and M. Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. In *Proceedings of the 6th IEEE International Conference on Data Engineering*, pages 438–445, 1990.

[CCIT88] International Telegraph and Telephone Consultative Committee (CCITT). The Directory – Authentication Framework, Recommendation X.509, 1988.

[DGS88] D. Dolev, E. Gafni and N. Shavit. Toward a non-atomic era: l-exclusion as a test case. In *Proceedings of the twentieth Annual ACM Symposium on Theory of Computing*, pages 78–92, May 1988.

[DS89] D. Dolev and N. Shavit. Bounded concurrent time-stamp systems are constructible. *SIAM Journal of Computing*, to appear. Also: *Proceedings of the 21st Symposium on the Theory of Computing*, pages 454–466, 1989.

[ET89] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems* 14(2):264–290, June 1989.

[GB85] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM* 32(4):841–860, October 1985.

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating Systems Principles*, pages 150–162, 1979.

[Hal86] M. Hall, Jr. *Combinatorial Theory.* 2nd Ed. Wiley-Interscience Series in Discrete Mathematics, 1986.

[Her86] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems* 4(1):32–53, February 1986.

[HT88] M. P. Herlihy and J. D. Tygar. How to make replicated data secure. In *Advances in Cryptology—CRYPTO '87 Proceedings* (Lecture Notes in Computer Science 293), pages 379–391, Springer-Verlag, 1988.

[IL93] A. Israeli and M. Li. Bounded time-stamps. *Distributed Computing* 6(4):205–209.

[Ken93] S. T. Kent. Internet privacy enhanced mail. *Communications of the ACM* 36(8):48–60, August 1993.

[Kum91] A. Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Transactions on Computers* 40(9):996–1004, 1991.

[IS92] A. Israeli and A. Shaham. Optimal multi-write multi-reader atomic register. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 71–82, 1992.

[LSP82] L. Lamport, R. Shostak and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems* 4(3):382–401, July 1982.

[LABW92] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems* 10(4):265–310, November 1992.

[Lam86] L. Lamport. On interprocess communication (part II: algorithms). *Distributed Computing* 1:86–101, 1986.

[Lov73] L. Lovász. Coverings and colorings of hypergraphs. In *Proceedings of the 4th Southeastern Conference on Combinatorics, Graph Theory, and Computing*, pages 3–12, 1973.

[LTV96] M. Li, J. Tromp and P. M. B. Vitanyi. How to share concurrent wait-free variables. In *Journal of the ACM*, to be published.

[Mae85] M. Maekawa. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems* 3(2):145–159, 1985.

[Mar90] K. Marzullo. Tolerating failures of continuous-valued sensors. *ACM Transactions on Computer Systems* 8(4):284–304, November 1990.

[MR96] D. Malki and M. Reiter. A high-throughput secure reliable multicast protocol. In *Proceedings of the 9th IEEE Computer Security Foundations Workshop*, pages 9–17, June 1996.

[Muk94] R. Mukkamala. Storage efficient and secure replicated distributed databases. *IEEE Transactions on Knowledge and Data Engineering* 6(2):337–341, April 1994.

[NW94] M. Naor and A. Wool. The load, capacity, and availability of quorum systems. In *Proceedings of the 35th IEEE Symposium on Foundations of Computer Science*, pages 214–225, 1994.

[NW96] M. Naor and A. Wool. Access control and signatures via quorum secret sharing. In *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, pages 157–168, March 1996.

[PW95] D. Peleg and A. Wool. Crumbling walls: A class of high availability quorum systems. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 120–129, August 1995.

[Rab89] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM* 36(2):335–348, 1989.

[Rei94] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80, November 1994.

[Rei96] M. K. Reiter. Distributing trust with the Rampart toolkit. *Communications of the ACM* 39(4):71–74, April 1996.

[RB94] M. K. Reiter and K. P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems* 16(3):986–1009, May 1994.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* 21(2):120–126, February 1978.

[Sha79] A. Shamir. How to share a secret. *Communications of the ACM* 22(11):612–613, November 1979.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22(4):299–319, December 1990.

[TA91] J. J. Tardo and K. Alagappan. SPX: Global authentication using public key certificates. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pages 232–244, May 1991.

[Tho79] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems* 4(2):180–209, 1979.

[Tou84] S. Toueg. Randomized Byzantine agreement. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178, August 1984.