Probabilistic Opaque Quorum Systems

Michael G. Merideth[1] and Michael K. Reiter[2]

[1] Carnegie Mellon University, Pittsburgh, PA, USA
[2] University of North Carolina, Chapel Hill, NC, USA

Abstract. Byzantine-fault-tolerant service protocols like Q/U and FaB Paxos that optimistically order requests can provide increased efficiency and fault scalability. However, these protocols require $n \geq 5b+1$ servers (where b is the maximum number of faults tolerated), owing to their use of *opaque Byzantine quorum systems*; this is $2b$ more servers than required by some non-optimistic protocols. In this paper, we present a family of *probabilistic* opaque Byzantine quorum systems that require substantially fewer servers. Our analysis is novel in that it assumes Byzantine clients, anticipating that a faulty client may seek quorums that maximize the probability of error. Using this as motivation, we present an optional, novel protocol that allows probabilistic quorum systems to tolerate Byzantine clients. The protocol requires only one additional round of interaction between the client and the servers, and this round may be amortized over multiple operations. We consider actual error probabilities introduced by the probabilistic approach for concrete configurations of opaque quorum systems, and prove that the probability of error vanishes with as few as $n > 3.15b$ servers as n and b grow.

1 Introduction

For distributed systems consisting of a large number of servers, a Byzantine-fault-tolerant replication algorithm that requires all servers to communicate with each other for every client request can be prohibitively expensive. Therefore, for large systems, it is critical that the protocol have good *fault scalability* [1]—the property that performance does not (substantially) degrade as the system size is increased—by avoiding this communication.

Byzantine-fault-tolerant service protocols must assign a total order to requests to provide replicated state machine semantics [2]. To minimize the amount of communication between servers, protocols like Q/U [1] and FaB Paxos [3] use opaque quorum systems [4] to order requests *optimistically*. That is, servers independently choose an ordering, without steps that would be required to reach agreement with other servers; the steps are performed only if servers choose different orderings. Under the assumption that servers independently typically choose the same ordering, the optimistic approach can provide better fault scalability in the common case than protocols like BFT [5], which require that servers perform steps to agree upon an ordering *before* choosing it [1]. However, optimistic protocols have the disadvantage of requiring at least $5b + 1$ servers to tolerate b server faults, instead of as few as $3b + 1$ servers, and so they cannot tolerate as many faults for a given number of servers.

In this paper, we present *probabilistic opaque quorum systems* (POQS), a new type of probabilistic quorum system [6], in order to increase the fraction of faults that can be tolerated by an optimistic approach from fewer than $n/5$ to as many as $n/3.15$. A POQS provides the same properties as the strict opaque quorum systems used by, e.g., Q/U and FaB Paxos, but is probabilistic in the sense that quorums are not guaranteed to overlap in the number of servers required to ensure safety. However, we prove that this error probability is negligible for large system sizes (for a given ratio of b to n). Application domains that could give rise to systems of such scale include sensor networks and edge services.

Byzantine clients are problematic for all probabilistic quorum systems because the combination of high fault tolerance and low probability of error that can be achieved is based on the assumption that clients choose quorums uniformly at random (and independently of other quorums and the state of the system, e.g., the values held by each server, and the identities of faulty servers). This can be seen in our results that show: (i) that probabilistic opaque quorum systems can tolerate up to $n/3.15$ faults (compared with less than $n/5$ faults for strict opaque quorum systems) assuming that all quorums are selected uniformly at random, but that the maximum fault tolerance drops to $n/4.56$ faults if Byzantine clients are allowed to choose quorums according to their own goals; and (ii) that to achieve a specified error probability for a given degree of fault tolerance, substantially more servers are required if quorums are not selected uniformly at random.

Therefore, we present a protocol with which we constrain clients to using pseudo-randomly selected access sets (sets of servers contacted in order to find quorums, c.f., [7]) of a prescribed size. In the limit, we can set the sizes of access sets to be the sizes of quorums, thereby dictating that all clients use pseudo-randomly selected quorums, and providing a mechanism that guarantees, in practice, the behavior of clients that is assumed by probabilistic quorum systems. However, as shown in Section 4.3, the notion of restricted access sets allows us a range of options in trading off the low error probability and high fault tolerance of completely random quorum selection, for the guaranteed single-round access provided when there is an available quorum (one in which all servers respond) in every access set.

Our contributions are as follows:

- We present the first family of probabilistic opaque quorum system constructions. For each construction, we: (i) show that we are able to reduce the number of servers below the $5b + 1$ required by protocols that use strict opaque quorums, (ii) prove that it works with vanishing error probability as the system size grows, and (iii) evaluate the characteristics of its error probability over a variety of specific system sizes and configurations.
- We present the first analysis of a probabilistic quorum system that accounts for the behavior of Byzantine clients. We anticipate that a faulty client may choose quorums with the goal of maximizing the error probability, and show the effects that this may have.

– We present an access-restriction protocol that allows probabilistic quorum systems to tolerate faulty clients with the same degree of fault tolerance as if all clients were non-faulty. One aspect of the protocol is that servers work to propagate the values of established writes to each other in the background. Therefore, we provide analysis, unique to opaque quorum systems, of the number of servers that must propagate a value for it to be accepted by another server.

2 Related Work

Strict Opaque Quorum Systems. Opaque Byzantine quorum systems were introduced by Malkhi and Reiter [4] in two variants: one in which the number of non-faulty servers in a quorum is at least half of the quorum, and the other in which the number of non-faulty servers represents a strict majority of the quorum. The first construction makes it unnecessary for the client to know the sets of servers of which the system can tolerate failure (hence the term 'opaque'), while the second construction additionally makes it possible to create a protocol that does not use timestamps. The paper also proves that $5b$ is the lower bound on the number of servers for the first version; simply changing the inequality to a strict inequality proves $5b+1$ is the lower bound for the second. In this paper, we are concerned with the second variant.

The constraints on strict opaque quorums have also been described in the context of consensus and state-machine-replication protocols, e.g., the Q/U [1] and FaB Paxos [3] protocols, though not explicitly as opaque quorums. Abd-El-Malek et al. [1] provide generic (not just threshold) opaque quorum system constraints that they prove sufficient for providing state-machine replication semantics where both writes and reads complete in a single (pipelined) phase when there is no write–write contention. Martin and Alvisi [3] use an opaque quorum system of acceptors in FaB Paxos, a two-phase consensus protocol (with a designated proposer) and three-phase state-machine-replication protocol requiring at least $5b+1$ servers.

Probabilistic Quorum Systems. A Probabilistic Quorum System (PQS), as presented by Malkhi et al. [6], can provide better availability and fault tolerance than provided by strict quorum systems; Table 1 compares probabilistic quorums with their strict quorum counterparts.[1] Malkhi et al. provide constructions for dissemination and masking quorums, and prove properties of load and availability for these constructions. They do not address

Table 1. Minimum servers needed for probabilistic and strict quorum variants

	prob.	strict	presented
Opaque	$3.15b+1$	$5b+1$	Here
Masking	$2.62b+1$	$4b+1$	[6]
Dissemination	$b+1$	$3b+1$	[6]

[1] The $2.62b$ lower bound for masking quorums is not shown in [6], but can be quickly derived using our results from Section 4.

opaque quorum systems, or the effects of concurrent or Byzantine writers; we address each of these. In addition, in Section 4, we borrow analysis techniques from [6], but our analysis is more general in the sense that clients are not all assumed to communicate only with quorums of servers. We also use a McDiarmid inequality [8] in our technical report [9] for bounding the error probability; this provides a simpler bounding technique for our purposes than do the Chernoff bounds used there. The technique that we present in Section 5 for restricting access to limited numbers of servers should be applicable to the constructions of Malkhi et al. equally well.

Other Work. Signed Quorum Systems [10] and k-quorums [11,12] also weaken the requirements of strict quorum systems but use different techniques; our technical report [9] has a more detailed discussion. There has been work on strict quorum systems that can tolerate Byzantine clients (e.g., [13,14]) but this is fundamentally unconcerned with the way in which quorums are chosen because such choices cannot impact the correctness of strict quorum systems.

3 System Model and Definitions

We assume a system with a set U of servers, $|U| = n$, and an arbitrary but bounded number of clients. Clients and servers can fail arbitrarily (i.e., Byzantine [15] faults). We assume that up to b servers can fail, and denote the set of faulty servers by B, where $B \subseteq U$. Any number of clients can fail. Failures are permanent. Clients and servers that do not fail are said to be *non-faulty*. We allow that faulty clients and servers may collude, and so we assume that faulty clients and servers all know the membership of B (although non-faulty clients and servers do not). We make the standard assumption that nodes are computationally bound such that they cannot subvert the effectiveness of cryptographic primitives.

Throughout the paper, we use $\mathsf{San\ Serif}$ font to denote random variables, uppercase *ITALICS* for set-valued constants, and lowercase *italics* for integer-valued constants.

3.1 Behavior of Clients

We abstractly describe client operations as either *writes* that alter the state of the service or *reads* that do not. Informally, a non-faulty client performs a write to update the state of the service such that its value (or a later one) will be observed with high probability by any subsequent operation; a write thus successfully performed is called “established” (we define established more precisely below). A non-faulty client performs a read to obtain the value of the latest established write, where “latest” refers to the value of the most recent write preceding this read in a linearization [16] of the execution. Therefore, we define the *correct* value for the read to return to be the value of this latest established write; other values are called *incorrect*. We assume that the read and write operations by non-faulty clients take the following forms:

- **Writes:** To perform a write, a non-faulty client selects a *write access set* $A_{\mathrm{wt}} \subseteq U$ of size a_{wt} uniformly at random and attempts to inform all servers in A_{wt} of the write value. Formally, the write is *established* once all non-faulty servers in some set $Q_{\mathrm{wt}} \subseteq A_{\mathrm{wt}}$ of size $q_{\mathrm{wt}} \leq a_{\mathrm{wt}}$ servers have *accepted* this write. (Intuitively, an access set is a set of servers contacted in order to find a live quorum, c.f., [7].) We refer to q_{wt} as the *write quorum size*; to any $Q_{\mathrm{wt}} \subseteq U$ of that size as a *write quorum*; and to $\mathcal{Q}_{\mathrm{wt}} = \{Q_{\mathrm{wt}} \subseteq U : |Q_{\mathrm{wt}}| = q_{\mathrm{wt}}\}$ as the *write quorum system.*
- **Reads:** To perform a read, a non-faulty client selects a *read access set* A_{rd} of size a_{rd} uniformly at random and attempts to contact each server in A_{rd} to learn the value that the server last accepted. We denote the minimum number of servers from which a non-faulty client must receive a response to complete the read successfully by $q_{\mathrm{rd}} \leq a_{\mathrm{rd}}$. We refer to q_{rd} as the *read quorum size*; to any $Q_{\mathrm{rd}} \subseteq U$ of that size as a *read quorum*; and to $\mathcal{Q}_{\mathrm{rd}} = \{Q_{\mathrm{rd}} \subseteq U : |Q_{\mathrm{rd}}| = q_{\mathrm{rd}}\}$ as the *read quorum system.*

In a read operation, we refer to each response received from a server in A_{rd} as a *vote* for a read value. We assume that votes for two read values that result from any two distinct write operations are distinguishable from each other, even if the corresponding write values are the same (this is discussed in Section 5). The read operation discerns the correct value from these votes in a protocol-specific way. It is possible in an optimistic protocol such as Q/U [1], for example, that the (at least q_{rd}) votes may reflect a write operation but not provide enough evidence to determine whether that write is established. In this case, the reader may itself establish, or *repair*, the write value before returning it, to ensure that a subsequent reader returns that value, as well (which is necessary to achieve linearizability). In such a protocol, the reader does so by copying its votes for that value to servers, in order to convince them to accept that write.

For this reason, the correctness requirements for POQS discussed in Section 4 treat not only the number of votes that a non-faulty reader observes for the correct value, but also the number of votes that a faulty client can gather for a *conflicting* value. A conflicting value is a specific type of incorrect value characterized by the property that a non-faulty server would accept either it or the correct value, but not both. Two values may conflict because, e.g., they both bear the same timestamp, or are "conditioned on" the same established write in the sense used in Q/U. We assume that this timestamp or similar information can be used to distinguish older (stale) values from newer values. Enabling a faulty client to obtain sufficiently many votes for a conflicting value would, e.g., enable it to convince other non-faulty servers to accept the conflicting value via the repair protocol, a possibility that must be avoided for correctness.

Consequently, an *error* is said to occur when a non-faulty client fails to return the correct value or a faulty client obtains sufficiently many votes for a conflicting value. This definition (or specifically "sufficiently many") will be made more precise in Section 4.4. The *error probability* then refers to the probability of an error when the client (non-faulty or faulty) reads from a read access set A_{rd} chosen uniformly at random. While we cannot force a faulty client to choose

A_{rd} uniformly at random, in Section 5 we demonstrate an access protocol that enables a faulty client to assemble votes for a value that can be verified by servers (and hence, e.g., to perform a repair in Q/U) only if A_{rd} was selected uniformly at random, which is good enough for our purposes. So, from here forward, we restrict our attention to read access sets chosen in this way.

3.2 Communication

The communication assumptions we adopt are common to prior works in probabilistic [6] and signed [10] quorum systems: we assume that each non-faulty client can successfully communicate with each non-faulty server with high probability, and hence with all non-faulty servers with roughly equal probability. This assumption is in place to ensure that the network does not significantly bias a non-faulty client's interactions with servers either toward faulty servers or toward different non-faulty servers than those with which another non-faulty client can interact. Put another way, we treat a server that can be reliably reached by none or only some non-faulty clients as a member of B.

This assumption enables us to refine the read protocol of Section 3.1 in a straightforward way so that non-faulty clients choose read quorums from an access set uniformly at random. (More precisely, a faulty server can bias quorum selection away from quorums containing it by not responding, but this decreases the error probability, and so we conservatively assume that non-faulty clients select read quorums at random from their access sets.) However, because a write is, by definition, established once all of the non-faulty servers in any write quorum within A_{wt} have accepted it, the write quorum at which a write is established contains all servers in $A_{\mathrm{wt}} \cap B$; i.e., only the the non-faulty servers within the write quorum are selected uniformly at random by a non-faulty client.

The access-restriction protocol of Section 5 requires no communication assumptions beyond those of the probabilistic quorums it supports.

4 Probabilistic Opaque Quorum Systems

In this section, we present a family of probabilistic opaque quorum systems. We begin by reviewing the properties of strict opaque quorum systems [4]. Define the following functions (where Q_{rd} and Q_{wt} are as defined in Section 3.1):

$$\mathbf{correct}(Q_{\mathrm{rd}}, Q_{\mathrm{wt}}) \quad : \quad |(Q_{\mathrm{rd}} \cap Q_{\mathrm{wt}}) \setminus B| \tag{1}$$

$$\mathbf{conflicting}(Q_{\mathrm{rd}}, Q_{\mathrm{wt}}) \quad : \quad |(Q_{\mathrm{rd}} \cap B) \cup (Q_{\mathrm{rd}} \setminus Q_{\mathrm{wt}})| \tag{2}$$

correct$(Q_{\mathrm{rd}}, Q_{\mathrm{wt}})$ returns the number of non-faulty servers in the intersection of a pair of read and write quorums, while **conflicting**$(Q_{\mathrm{rd}}, Q_{\mathrm{wt}})$ returns the other servers in the read quorum, all of which may return a conflicting value in some protocol execution. Let a read operation return a value that receives at least r votes. Then, the consistency property for strict opaque quorum systems is as follows:

$$\begin{aligned}&\textbf{O-Consistency}: \forall Q_{\mathrm{rd}} \in \mathcal{Q}_{\mathrm{rd}}, \forall Q_{\mathrm{wt}} \in \mathcal{Q}_{\mathrm{wt}}: \\ &\quad \textbf{correct}(Q_{\mathrm{rd}}, Q_{\mathrm{wt}}) \geq r > \textbf{conflicting}(Q_{\mathrm{rd}}, Q_{\mathrm{wt}}). \qquad (3)\end{aligned}$$

The property states that the number of non-faulty servers in the intersection of any read quorum and write quorum must represent a majority of the read quorum. Because of this and the fact that newer values can be distinguished from older values, the correct value—which, by definition, is established by being written to all of the non-faulty servers in a write quorum—can be distinguished from other values, even if some non-faulty servers (and all faulty servers) present conflicting or stale values. At a high level, O-Consistency guarantees:

P1 No two conflicting writes are both established.
P2 Every read observes sufficiently many votes for the correct value to identify it as such.
P3 No (non-faulty or faulty) reader obtains votes for a conflicting value sufficient to repair it successfully.

Given that the stated assumptions of a strict opaque quorum system hold, the system behaves correctly. In contrast to this, probabilistic opaque quorum systems (POQS) allow for a (small) possibility of error. Informally, this can be thought of as relaxing O-Consistency so that a variant of it holds for most—but not all—quorums. To ensure that the probability of an error happening is small, POQS are designed so that P1, P2, and P3 hold with high probability.

In the remainder of this section, we model the worst-case behavior of faulty clients (Section 4.1); derive a constraint (PO-Consistency, Section 4.2) that determines the maximum fraction of faulty servers that can be tolerated (Section 4.3) by POQS; and prove that the error probability goes to zero as n (and b) is increased if this constraint is satisfied (Section 4.4).

4.1 Behavior of Faulty Clients

Because a faulty client can behave arbitrarily, we examine the way that a faulty client should choose quorums to maximize the chance of error. Throughout this section, let A_{wt} denote a write access set from which Q_{wt} (a quorum used for an established write) is selected by a faulty client, let A'_{wt} be a write access set used for a conflicting write by a faulty client, and let A_{rd} be a read access set from which Q_{rd}, a read quorum, is selected by a faulty client. Again, we assume that A_{wt}, A'_{wt}, and A_{rd} are selected uniformly at random, an assumption that can be enforced using the protocol of Section 5.

A faulty client can increase the error probability with a write in one of two ways: (i) by establishing a write at a write quorum that contains as many faulty servers as possible, or (ii) by performing the write of a conflicting value in a way that maximizes the number of non-faulty servers that accept it, i.e., by writing to all of $A'_{\mathrm{wt}} \setminus Q_{\mathrm{wt}}$. Since a faulty client may perform *both* such writes, we assume that this client has knowledge of A_{wt} and A'_{wt} simultaneously. However, it is important to note that a faulty client does not have knowledge of the read

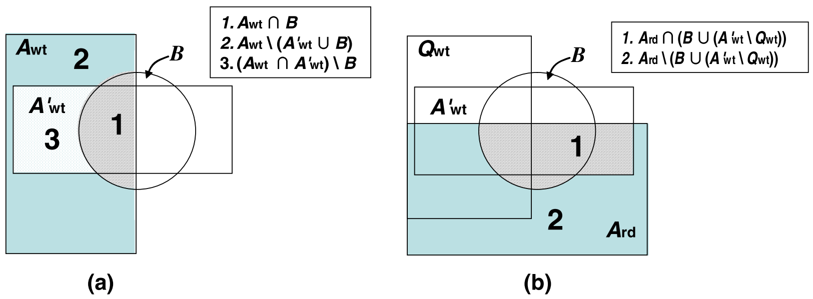
1. $A_{wt} \cap B$
2. $A_{wt} \setminus (A'_{wt} \cup B)$
3. $(A_{wt} \cap A'_{wt}) \setminus B$
A_{wt}
2
B
A'_{wt}
1
3
(a)
1. $A_{rd} \cap (B \cup (A'_{wt} \setminus Q_{wt}))$
2. $A_{rd} \setminus (B \cup (A'_{wt} \setminus Q_{wt}))$
Q_{wt}
B
A'_{wt}
1
2
A_{rd}
(b)

Fig. 1. The preference (1st, 2nd, 3rd) a faulty client gives to a server when choosing (a) Q_{wt}, or (b) Q_{rd}

access set A'_{rd} used by a non-faulty client—or specifically the non-faulty servers within it, i.e., $A'_{\mathrm{rd}} \setminus B$—and so Q_{wt} is chosen independently of $A'_{\mathrm{rd}} \setminus B$.[2]

Figure 1(a) shows the preferences that a faulty client gives to servers when choosing Q_{wt} to do both (i) and (ii). Goal (i) requires maximizing $|Q_{\mathrm{wt}} \cap B|$ to maximize the probability that P1 or P2 is violated; hence, first preference is given to the servers in $A_{\mathrm{wt}} \cap B$ in a write. Goal (ii) requires minimizing $|(Q_{\mathrm{wt}} \cap A'_{\mathrm{wt}}) \setminus B|$ to maximize the probability that P1 or P3 is violated; hence, the servers in $(A_{\mathrm{wt}} \cap A'_{\mathrm{wt}}) \setminus B$ are avoided to the extent possible.

A faulty client can increase the probability that P3 is violated by choosing a read quorum with the most faulty servers and non-faulty servers that share the same conflicting value. Figure 1(b) shows the preferences that a faulty client gives to servers to do so. Because a faulty client can collude with the servers in B, it can obtain replies from all servers in B that are also in A_{rd}, i.e., the servers in $A_{\mathrm{rd}} \cap B$. It can also wait for responses from all of the non-faulty servers in A_{rd} with the conflicting value, i.e., those in $A_{\mathrm{rd}} \cap (A'_{\mathrm{wt}} \setminus Q_{\mathrm{wt}})$. Only after receiving all such responses, and only if these responses number fewer than q_{rd}, must it choose responses from servers with other values.

4.2 Probabilistic Constraint

In this section, we present PO-Consistency, a constraint akin to O-Consistency specified in terms of expected values for POQS. As detailed below, let MinCorrect be a random variable for the minimum number of non-faulty servers that report the correct value in a randomly chosen read quorum taken by a non-faulty client. (Recall that an error is caused by MinCorrect being too small only for reads performed by a non-faulty client.) Also, let MaxConflicting be a random variable for the maximum number of servers that report a conflicting value in a read quorum taken from a randomly chosen read access set by a faulty client that seeks to maximize MaxConflicting. (Recall that an error is caused by MaxConflicting

[2] More precisely, with the access protocol in Section 5, A'_{rd} can be hidden unless, and until, that read access set is used for repair, at which point it is too late for faulty clients to choose Q_{wt} so as to induce an error in that read operation.

being too large even if the client is faulty.) Then the consistency property for POQS is:

$$\textbf{PO-Consistency} : \mathbb{E}\left[\mathsf{MinCorrect}\right] > \mathbb{E}\left[\mathsf{MaxConflicting}\right]. \tag{4}$$

As shown in Section 4.4, PO-Consistency allows us to choose a threshold, r, for the number of votes used to determine the result of a read operation, while ensuring that the error probability vanishes as we increase n (and b).

We now derive expressions for $\mathsf{MinCorrect}$ and $\mathsf{MaxConflicting}$. Recall that B is the set of up to b faulty servers. Let A_{wt} be a randomly chosen write access set, and let A_{rd} be a randomly chosen read access set. As stated in the system model, a write to A_{wt} is established once it has been accepted by all of the non-faulty servers in any Q_{wt}, a write quorum within A_{wt}. Therefore, we conservatively assume that the number of faulty servers in Q_{wt} is:

$$\mathsf{MalWrite} = |\mathsf{A}_{\mathrm{wt}} \cap B|. \tag{5}$$

Here, A_{wt} is a random variable taking on a write access set chosen uniformly at random from $\mathcal{A}_{\mathrm{wt}}$.

Q_{wt} also contains $q_{\mathrm{wt}} - \mathsf{MalWrite}$ non-faulty servers, not necessarily chosen at random, in addition to the $\mathsf{MalWrite}$ faulty servers. Let C_{wt} represent these non-faulty servers:

$$\mathsf{C}_{\mathrm{wt}} = \mathsf{Q}_{\mathrm{wt}} \setminus B, \tag{6}$$

$$|\mathsf{C}_{\mathrm{wt}}| = q_{\mathrm{wt}} - \mathsf{MalWrite}, \tag{7}$$

where Q_{wt} is a random variable taking on the write quorum at which the write is established, and C_{wt} is a random variable taking on the set of non-faulty servers within this write quorum. Then, the number of non-faulty servers that return the correct value in a read quorum selected by a non-faulty client is,

$$\mathsf{MinCorrect} = |\mathsf{Q}_{\mathrm{rd}} \cap \mathsf{C}_{\mathrm{wt}}|, \tag{8}$$

where Q_{rd} is a random variable taking on a read quorum chosen uniformly at random from A_{rd}, itself chosen uniformly at random from $\mathcal{A}_{\mathrm{rd}}$.

A faulty client may select its read quorum, Q_{rd}, to maximize the number of votes for a single conflicting value in an attempt to invalidate P3. Therefore, as described in Section 4.1, the client first chooses all faulty servers in A_{rd}. The number of such servers is,

$$\mathsf{Malevolent} = |\mathsf{A}_{\mathrm{rd}} \cap B|. \tag{9}$$

The faulty client also chooses the non-faulty servers that vote for the conflicting value that is most represented in A_{rd}; these servers are a subset of $(A_{\mathrm{rd}} \setminus (C_{\mathrm{wt}} \cup B))$. This conflicting value has an associated write access set A'_{wt} chosen uniformly at random from $\mathcal{A}_{\mathrm{wt}}$, and no vote from a non-faulty server not in A'_{wt} will be counted among those for this conflicting value (because votes for any two write operations are distinguishable from each other as discussed in

Section 3.1). Let $\mathsf{A}'_{\mathrm{wt}}$ be a random variable taking on A'_{wt}. Then, the number of non-faulty servers in A_{rd} that vote for this conflicting value is,

$$\mathsf{Conflicting} = |\mathsf{A}_{\mathrm{rd}} \cap (\mathsf{A}'_{\mathrm{wt}} \setminus (\mathsf{C}_{\mathrm{wt}} \cup B))|. \tag{10}$$

A faulty client can choose all of these servers for Q_{rd}. Therefore, since the sets of servers measured by Malevolent and Conflicting are disjoint (the former consists solely of faulty servers; the latter solely of non-faulty servers), the maximum number of instances of the same conflicting value that a faulty client will select for Q_{rd} is,

$$\mathsf{MaxConflicting} = \mathsf{Malevolent} + \mathsf{Conflicting}. \tag{11}$$

4.3 Minimum System Sizes

In this section, we consider PO-Consistency under various assumptions concerning the sizes of access sets and quorums in order to derive the maximum fraction of faults that can be tolerated with decreasing error probability as a function of n (and b). Our primary result is Theorem 1 which provides an upper bound on b for which PO-Consistency holds. It is derived using the expectations of MinCorrect and MaxConflicting (derived in our technical report [9]) that are computed using the worst-case behavior of faulty clients presented in Section 4.1.

Theorem 1. *PO-Consistency holds iff*

$$b < \frac{(a_{\mathrm{rd}}q_{\mathrm{wt}}n - 2a_{\mathrm{rd}}a_{\mathrm{wt}}n + a_{\mathrm{wt}}^2a_{\mathrm{rd}} + q_{\mathrm{rd}}q_{\mathrm{wt}}n)n}{n^2a_{\mathrm{rd}} - a_{\mathrm{rd}}a_{\mathrm{wt}}n + a_{\mathrm{wt}}^2a_{\mathrm{rd}} + q_{\mathrm{rd}}a_{\mathrm{wt}}n}.$$

As shown in Section 4.4, a construction exhibits decreasing error probability in the limit with increasing n if PO-Consistency holds. Therefore, the remainder of this section is concerned with interpreting the inequality in Theorem 1. Our analysis of this inequality is given in Table 2 and shows that the best bounds are provided when: (i) both types of quorums are as large as possible (while still ensuring an available quorum), i.e., $q_{\mathrm{rd}} = q_{\mathrm{wt}} = n - b$; and (ii), given (i), that access sets as small as possible. Our technical report [9] provides a more detailed analysis including an inequality for systems with no Byzantine clients.

Table 2. Lower bounds on n for various configurations

n >	$= n$	$= n - b$	$= n - 2b$
3.15b	-	a_{rd} q_{rd} a_{wt} q_{wt}	-
3.83b	a_{rd}	q_{rd} a_{wt} q_{wt}	-
4.00b	a_{wt}	a_{rd} q_{rd} q_{wt}	-
4.08b	-	a_{rd} a_{wt} q_{wt}	q_{rd}
4.56b	a_{rd} a_{wt}	q_{rd} q_{wt}	-
4.73b	a_{wt}	a_{rd} q_{wt}	q_{rd}
5.49b	-	a_{rd} q_{rd} a_{wt}	q_{wt}
6.07b	a_{rd}	q_{rd} a_{wt}	q_{wt}
6.19b	-	a_{rd} a_{wt}	q_{rd} q_{wt}

4.4 Bounding the Error Probability

Suppose a read operation always returns a value that receives more than r votes, where $\mathbb{E}[\mathsf{MaxConflicting}] \le r < \mathbb{E}[\mathsf{MinCorrect}]$. Then, the error probability, ϵ, is

$$\epsilon = \Pr(\mathsf{MaxConflicting} > r \vee \mathsf{MinCorrect} \leq r). \tag{12}$$

Theorem 2 states that if r is chosen so that

$$\begin{aligned} \mathbb{E}\left[\mathsf{MinCorrect}\right] - r = \theta(n) \quad \text{and} \\ r - \mathbb{E}\left[\mathsf{MaxConflicting}\right] = \theta(n) \end{aligned} \tag{13}$$

then ϵ decreases as a function of n, assuming that the ratio of each of b, a_{rd}, q_{rd}, a_{wt}, and q_{wt} to n remains constant. For example, r can be set equal to $(\mathbb{E}\left[\mathsf{MaxConflicting}\right] + \mathbb{E}\left[\mathsf{MinCorrect}\right])/2$.

Theorem 2. *Let* MinCorrect, MaxConflicting, *and r be defined as above (so PO-Consistency holds) and let the ratio of each of b, a_{rd}, q_{rd}, a_{wt}, and q_{wt} to n be fixed. Then,*

$$\epsilon = 2/e^{\Omega(n)} + 2/e^{\Omega(n)}.$$

5 Access-Restriction Protocol

Our analysis in the previous sections assumes that all access sets are chosen uniformly at random by all clients—even faulty clients. Therefore, here we present an access-restriction protocol that is used to enforce this. Recall from Section 3.1 that the need for read access sets to be selected uniformly at random is motivated by repair. As such, protocols that do not involve repair may not require this access-restriction protocol for read operations.

Our protocol must balance conflicting constraints. First, a client may be forced to discard a randomly chosen access set—and choose another—because a given access set (of size less than b servers more than a quorum) might not contain an available quorum. However, in order to support protocols like Q/U [1] that use opaque quorum systems for single-round writes, we cannot require additional rounds of communication for each operation. This precludes, for example, a protocol in which the servers collectively choose an access set at random and assign it to the client for every operation. As such, a client must be able to choose from multiple access sets without involving the servers for each. Yet, a faulty client should be prevented from discarding access sets in order to choose the one that has the highest probability of causing an error given the current system state. In addition, we should ensure that a faulty client does not benefit from waiting for the system state to change in order to use a previously chosen access set that becomes more advantageous as a result of the change.

In our protocol, the client obtains one or more random values, each called a Verifiable Random Value (VRV), with the participation of non-faulty servers. Each VRV determines a unique, verifiable, ordered sequence of random access sets that the client can use; the client has no control over the sequence. To deter a client from discarding earlier access sets in the sequence for potentially more favorable access sets later in the sequence, the protocol imposes an exponentially increasing cost (in terms of computation) for the ability to use later access sets.

The cost is implemented as a *client puzzle* [17]. We couple this with a facility for the propagation of the correct value in the background so that any advantages for a faulty client in the current system state are reduced if the client chooses to delay performing the operation while it explores later access sets. Finally, to deter a client from waiting for the system state to change, we tie the validity of a VRV (and its sequence of access sets) to the state of the system so that as execution proceeds, any unused access sets become invalid.

5.1 Obtaining a VRV

In order to get an access set, the client first must obtain a VRV from the servers. Servers implement a metering policy, in which each server responds to a request for a VRV only after a delay. The delay varies, such that it increases exponentially with the rate at which the client has requested VRVs during some recent interval of time—i.e., a client that has not requested a VRV recently will receive a VRV with little or no delay, whereas a client that has recently requested many VRVs will receive a VRV after a (potentially significant) delay. To offload work from servers to clients (e.g., for scalability), the servers can make it relatively more expensive (in terms of time) to ask for and receive a new VRV than to compute a given number of access sets (potentially for multiple operations) from a single VRV, using the mechanisms described below.

The VRV is characterized by the following properties:

– It can be created only with the consent of non-faulty servers;
– Its validity is tied to the state of the system, in the sense that as the system state evolves (possibly merely through the passage of time), eventually the VRV is invalidated;
– While it is valid, any non-faulty server can verify its validity and so will accept it.

The VRV must be created with the consent of non-faulty servers because otherwise faulty servers might collude to issue multiple VRVs to a faulty client with no delay. Therefore, l, the number of servers required for the issuance of a VRV, must be at least $b + 1$. However, of the non-faulty servers in the system, only those among the (at least $l - b$) used to issue a VRV will impose additional delay before issuing an additional VRV. Therefore, to minimize the time to get an additional VRV, a faulty client avoids involving servers that have issued VRVs recently. This strategy maximizes the number of VRVs to which the non-faulty server contributing to the fewest VRVs has contributed. Thus, once k VRVs have been issued, all $n - b$ non-faulty servers have contributed to the issuance of at least $\lfloor k(l - b)/(n - b) \rfloor$ of these k. Since all non-faulty servers have contributed to at least this many VRVs, and the delay is exponential in this number, the time $T(k)$ required for a client to obtain k VRVs is:

$$T(k) = \Omega\left(\exp\left\lfloor \frac{k(l-b)}{n-b} \right\rfloor\right)$$

In practice, $T(k)$ for a client decays during periods in which that client does not request additional VRVs, so that a client that does not request VRVs for a period can obtain one with small delay.

The validity of the VRV (and its sequence of access sets) is tied to the state of the system so that as execution proceeds, any unused access sets become invalid. To implement this, the replication protocol may provide some piece of data that varies with the state of the system—the *Object History Set* in Q/U [1] is an example of this—with which the servers can compute a VRV, but, in the absence of a suitable value from the protocol, the VRV can include a timestamp (assuming that the non-faulty servers have roughly synchronized clocks). The VRV consists of this value together with a digital signature created using a (l, n)-threshold signature scheme (e.g., [18]), i.e., so that any set of l servers can together create the signature, but smaller sets of servers cannot. The signature scheme must be *strongly unforgeable* [19], meaning that an adversary, given a VRV, is not able to find other valid VRVs. This is necessary because otherwise a faulty client would be able to generate variations of a valid VRV until finding one from which to select an access set that causes an error (see below).

5.2 Choosing an Access Set

As motivated above: (i) the VRV determines a sequence of valid access sets; and (ii) a client puzzle must make it exponentially harder to use later access sets in the sequence than earlier ones. In addition, it is desirable for our protocol to satisfy the following requirements:

- Each VRV must determine only a single valid sequence of access sets. This is to prevent a faulty client from choosing a preferred sequence.
- The puzzle solutions must be easy to verify, so that verification costs do not limit the scalability of the system in terms of the number of requests.
- There must be a solution to each puzzle. Otherwise a non-faulty client might be unable to use any access set.
- No server can know the solution to the puzzle beforehand due to the Byzantine fault model. Otherwise, a faulty client could avoid the exponential work by asking a faulty server for the solution.

In our protocol, the sequence of access sets is determined as follows. Let v be a VRV, let $\mathbf{g}$ be a hash function modeled as a random oracle [20], and let **access_set** be a deterministic operation that, given a seed value, selects an access set of the specified size from the set of all access sets of that size in a uniform fashion. Let the first seed, s_1, be $\mathbf{g}(v)$, and the i'th seed, s_i, be $\mathbf{g}(s_{i-1})$. Then the i'th access set is $\mathbf{access\_set}(s_i)$. Our technical report [9] contains an example specification of **access_set**.

In order to use the i'th access set, the client must solve a puzzle of suitable difficulty. This puzzle must be non-interactive [21] to avoid additional rounds of communication. There are many suitable candidate puzzle functions [21].

5.3 Server Verification

Upon receiving a write request for the i'th access set, each non-faulty server in the chosen access set must verify that it is a member of the access set; for a repair request it must verify that the relevant votes are from servers in the access set of the read operation that gave rise to the repair. In addition, in either case, before accepting the value, each server must verify that the VRV is valid, that the access set corresponds to the i'th access set of the sequence, and that the client has provided a valid solution to a puzzle of the appropriate difficulty level to use the i'th access set.

While the client can obtain additional access sets from the VRV, each access set used is treated as a different operation by servers as stated in Section 3.1; e.g., a write operation using one access set, and then using another access set, is treated as two different writes,[3] so that a faulty client cannot "accumulate" more than a_{wt} servers for its operation through the use of multiple write access sets.

5.4 Background Propagation

As described above, servers work to propagate the values of established writes to each other in the background. Our main contribution in this area is our analysis of the threshold number of servers that must propagate a value for it to be accepted by another server. While related Byzantine diffusion protocols (e.g., [22]) use the number $b+1$, we require a larger number because opaque quorum systems allow that some non-faulty servers may accept conflicting values. We assume an appropriate propagation algorithm (e.g., a variant of an epidemic algorithm [23] such as [22]). At a high level, a non-faulty server has two responsibilities. First, having accepted a write value and returned a response to the client, it periodically informs other servers that it has accepted the value. Second, if it has not yet accepted a value upon learning that a threshold number, p, of servers have accepted the value, it accepts the value. Faulty servers are all assumed to have access to any conflicting value directly without propagation, so we assume no additional constraints on their behavior.

Lemma 1. *Let $n < 2q_{\mathrm{wt}} - 2b$ and $p = n - q_{\mathrm{wt}} + b + 1$. Then an established value will be accepted and propagated by at least p non-faulty servers, and no conflicting value can be propagated by p servers (faulty or non-faulty).*

For example, if $q_{\mathrm{wt}} = n - b$ and $n > 4b$ we set $p = 2b + 1$. Since the established value will be accepted by at least p non-faulty servers, it will propagate. No conflicting value will propagate.

If the conditions of Lemma 1 do not hold, we must allow for some probability of error during propagation. We set p so that it is between the expectations of the minimum number of non-faulty servers that accept an established write ($\mathsf{PCorrect}$), and the maximum number of servers that propagate a conflicting value ($\mathsf{PConflicting}$).

[3] Typically, a Byzantine-fault-tolerant write protocol must already be resilient to partial writes, which is how these writes using different access sets might appear to the service.

Lemma 2. *PO-Consistency* $\Rightarrow \mathbb{E}[\mathsf{PCorrect}] > \mathbb{E}[\mathsf{PConflicting}]$.

Lemma 2 shows that we can set p as described for any system in which PO-Consistency holds.

6 Evaluation

In this section, we analyze error probabilities for concrete system sizes. In addition to validating our results from Section 4, this shows that an access restriction protocol like that of Section 5 can provide significant advantages in terms of worst-case error probabilities.

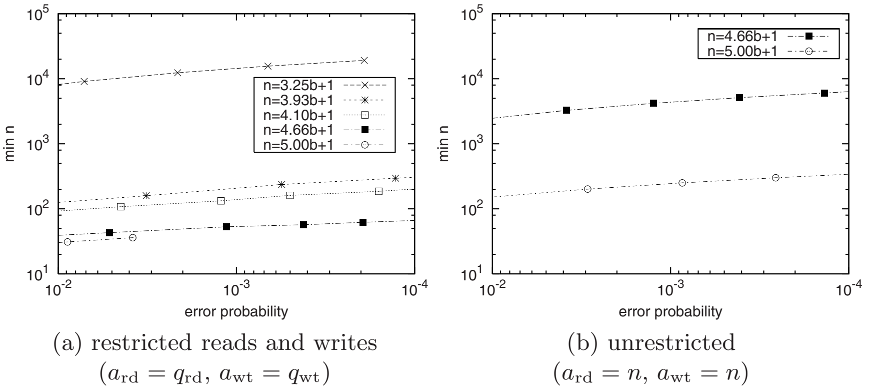

Fig. 2. Number of servers required to achieve given calculated worst-case error probability

Figure 2 plots the total number of nodes required to achieve a given calculated error probability for two configurations that tolerate faulty clients where $q_{\mathrm{wt}} = q_{\mathrm{rd}} = n - b$: the *restricted* configuration ($a_{\mathrm{rd}} = q_{\mathrm{rd}}$, $a_{\mathrm{wt}} = q_{\mathrm{wt}}$) and the *unrestricted* configuration ($a_{\mathrm{rd}} = n$, $a_{\mathrm{wt}} = n$). Since the unrestricted configuration (Figure 2(b)) does not require the access-restriction protocol of Section 5, yet yields better maximum ratios of b to n than the other configurations listed in Table 2 in which $q_{\mathrm{wt}} = a_{\mathrm{wt}} - b$ or $q_{\mathrm{rd}} = a_{\mathrm{rd}} - b$ from Section 4.3, we do not evaluate the error probabilities for those configurations here. In all cases, the error probabilities are worst-case in that they reflect the situation in which all b nodes are in fact faulty. For each configuration, we provide plots for different ratios of n to b, ranging from the maximum b for a given configuration, to $n = 5b+1$, as a comparison with strict opaque quorum systems. Our technical report [9] provides details of our calculations, as well as calculations for additional configurations.

In the figure, we see that to decrease the worst-case error probability, we can either keep the same function of b in terms of n while increasing n, or hold n fixed while decreasing the number of faults the system can tolerate. In addition, we see that configurations that tolerate a larger b also provide better error probabilities for a given b. Overall, we find that our constructions can tolerate significantly more than $b = n/5$ faulty servers, while providing error probabilities in the range of 10^{-2} to 10^{-4} for systems with fewer than 50 servers to hundreds of servers. Coupled with the dissemination of correct values between servers (off the critical path), as described in Section 5.4, the error probability decreases between writes.

7 Conclusion

First, have presented probabilistic opaque quorum systems (POQS), a new type of opaque quorum system that we have shown can tolerate up to $n/3.15$ Byzantine servers (compared with $n/5$ Byzantine servers for strict opaque quorum systems) with high probability, while preserving the properties that make opaque quorums useful for optimistic Byzantine-fault-tolerant service protocols. Second, we have presented an optional, novel access-restriction protocol for POQS that provides the ability for servers to constrain clients so that they use randomly selected access sets for operations. With POQS, we expect to create probabilistic optimistic Byzantine fault-tolerant service protocols that tolerate substantially more faults than current optimistic protocols. While strict opaque quorums systems may be more appropriate for smaller systems that require no chance of error, a POQS can provide increased fault tolerance for a given number of nodes, with a worst-case error probability that is bounded and that decreases as the system scales.

Acknowledgments. This work was partially supported by NSF grant CCF-0424422.

References

1. Abd-El-Malek, M., Ganger, G.R., Goodson, G.R., Reiter, M.K., Wylie, J.J.: Fault-scalable Byzantine fault-tolerant services. In: Symposium on Operating Systems Principles (October 2005)
2. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys 22(4), 299–319 (1990)
3. Martin, J.P., Alvisi, L.: Fast Byzantine consensus. IEEE Transactions on Dependable and Secure Computing 3(3), 202–215 (2006)
4. Malkhi, D., Reiter, M.: Byzantine quorum systems. Distributed Computing 11(4), 203–213 (1998)
5. Castro, M., Liskov, B.: Practical Byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems 20(4), 398–461 (2002)
6. Malkhi, D., Reiter, M.K., Wool, A., Wright, R.N.: Probabilistic quorum systems. Information and Computation 170(2), 184–206 (2001)
7. Bazzi, R.A.: Access cost for asynchronous Byzantine quorum systems. Distributed Computing 14(1), 41–48 (2001)
8. McDiarmid, C.: Concentration for independent permutations. Combinatorics, Probability and Computing 11(2), 163–178 (2002)
9. Merideth, M.G., Reiter, M.K.: Probabilistic opaque quorum systems. Technical Report CMU-CS-07-117, CMU School of Computer Science (March 2007)

10. Yu, H.: Signed quorum systems. Distributed Computing 18(4), 307–323 (2006)
11. Aiyer, A.S., Alvisi, L., Bazzi, R.A.: On the availability of non-strict quorum systems. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 48–62. Springer, Heidelberg (2005)
12. Aiyer, A.S., Alvisi, L., Bazzi, R.A.: Byzantine and multi-writer k-quorums. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 443–458. Springer, Heidelberg (2006)
13. Liskov, B., Rodrigues, R.: Tolerating Byzantine faulty clients in a quorum system. In: International Conference on Distributed Computing Systems (2006)
14. Cachin, C., Tessaro, S.: Optimal resilience for erasure-coded Byzantine distributed storage. In: International Conference on Dependable Systems and Networks (2006)
15. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems 4(3), 382–401 (1982)
16. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
17. Juels, A., Brainard, J.: Client puzzles: A cryptographic countermeasure against connection depletion attacks. In: Network and Distributed Systems Security Symposium, pp. 151–165 (1999)
18. Shoup, V., Gennaro, R.: Securing threshold cryptosystems against chosen ciphertext attack. Journal of Cryptology 15(2), 75–96 (2002)
19. An, J.H., Dodis, Y., Rabin, T.: On the security of joint signature and encryption. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 83–107. Springer, Heidelberg (2002)
20. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: Conference on Computer and Communications Security, pp. 62–73 (1993)
21. Jakobsson, M., Juels, A.: Proofs of work and bread pudding protocols. In: Communications and Multimedia Security, pp. 258–272 (1999)
22. Malkhi, D., Mansour, Y., Reiter, M.K.: Diffusion without false rumors: On propagating updates in a Byzantine environment. Theoretical Computer Science 299(1–3), 289–306 (2003)
23. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: Principles of Distributed Computing, pp. 1–12 (August 1987)