Dynamic Byzantine Quorum Systems

Lorenzo Alvisi* Dahlia Malkhi† Evelyn Pierce‡ Michael K. Reiter§
Rebecca N. Wright¶

Abstract

Byzantine quorum systems [13] enhance the availability and efficiency of fault-tolerant replicated services when servers may suffer Byzantine failures. An important limitation of Byzantine quorum systems is their dependence on a static threshold limit on the number of server faults. The correctness of the system is only guaranteed if the actual number of faults is lower than the the threshold at all times. However, a threshold chosen for the worst case wastes expensive replication in the common situation where the number of faults averages well below the worst case.

In this paper, we present protocols for dynamically raising and lowering the resilience threshold of a quorum-based Byzantine fault-tolerant data service in response to current information on the number of server failures. Using such protocols, a system can operate in an efficient low-threshold mode with relatively small quorums in the absence of faults, increasing and decreasing the quorum size (and thus the tolerance) as faults appear and are dealt with, respectively.

*Department of Computer Sciences, University of Texas at Austin, USA. Email: `lorenzo@cs.utexas.edu`.

†School of Computer Science and Engineering, The Hebrew University of Jerusalem, Israel. Email: `dalia@cs.huji.ac.il`.

‡Department of Computer Sciences, University of Texas at Austin, USA. Email: `tumlin@cs.utexas.edu`.

§Bell Labs, Lucent Technologies, Murray Hill, NJ, USA. Email: `reiter@research.bell-labs.com`.

¶AT&T Labs–Research, Florham Park, NJ, USA. Email: `rwright@research.att.com`.

This document describes work partially supported by NSF CAREER award CCR-9734185, DARPA/SPAWAR grant N66001-98-8911, NSF CISE grant CDA-9624082, and United States Air Force contract F30602-99-C-0165. Any opinions, findings, conclusions or recommendations expressed in this document are those of the author(s) and do not necessarily reflect the views of any of these sponsoring organizations.

1 Introduction

Quorum systems are valuable tools for implementing highly available distributed shared memory. Mathematically, a quorum system is simply a set of sets (called *quorums*), each pair of which intersect. The principle behind their use in distributed data services is that, if a shared variable is stored at a set of servers, read and write operations need be performed only at a quorum of those servers. The intersection property of quorums ensures that each read has access to the most recently written value of the variable.

Traditionally, quorum systems have been used for *benign fault tolerance*, i.e., maintaining data availability in the presence of unresponsive servers (*crashes*). Recently, Byzantine quorum systems have been introduced to provide data availability even in the presence of arbitrary (*Byzantine*) faults [13]. The Byzantine fault model is attractively powerful in that it can be used to analyze a wide variety of faulty behaviors; for example, it has been proposed as a framework for modeling security problems such as intrusions and sabotage.

One important limitation of standard Byzantine fault-tolerance techniques, quorum-based or otherwise, is their dependence on a static, pessimistically defined *resilience threshold*, which limits the number of faults.[1] The designer must decide in advance what maximum number of simultaneous failures the system will tolerate and build the system to tolerate that number of faults, at the expense of keeping an appropriate number of separate up-to-date copies of each data item for this worst-case failure assumption. If the chosen threshold is higher than necessary, the excess replication is wasted, so that the system is unnecessarily inefficient and unwieldy. On the other hand, if the threshold is chosen too low, the correctness guarantees of the sys-

[1] Papers such as [13] consider generalized fault structures, offering a more general way of characterizing fault tolerance than a threshold. However, such structures remain static, and therefore necessarily worst-case.

Appears in *Proceedings of the International Conference on Dependable Systems and Networks 2000*

tem are nullified. Furthermore, even if a threshold is appropriately positioned for the worst case failure scenario, this scenario will usually be relatively rare; the degree of replication required will be wasted in the average case.

In this paper, we present a method of dynamically raising and lowering the resilience threshold of a quorum-based Byzantine fault-tolerant data service in response to estimates of the number of server failures. ([1] presents failure detection methods that might be used to obtain these estimates for such services.) The goal of our work is to design protocols that allow a quorum system to respond *at run time* to the presence or absence of detected faults. This flexibility comes at a cost: tolerating a given maximum number of faults requires more servers in our approach than in a static system. However, with a fixed number of servers, our protocols allow a system to operate in low-threshold mode with smaller quorums than a static approach would require for the same worst-case threshold. A natural way of using a dynamic quorum system is to increase the threshold when faults are detected, and decrease it again when the failures have been dealt with. The threshold could also be raised or lowered based on external evidence that the threat of an attack has increased or decreased, such as information in server logs or new information about the value of the data being stored.

The difficulty in dynamically adjusting Byzantine quorum systems can be exemplified as follows. Consider a threshold *masking quorum system* [13] of $n = 9$ replicated servers with quorums consisting of all sets of 6 servers. This guarantees that every pair of quorums intersect in 3 servers or more, and can tolerate a threshold $b = 1$ of Byzantine server failures while still guaranteeing that the majority of every quorum intersection is correct. Now, suppose that some client, detecting a possible failure in the system, wishes to reconfigure the quorum system to raise the resilience threshold to $b = 2$. This can be accomplished by making every set of 7 servers a quorum, thereby guaranteeing that every pair of new quorums intersect in at least 5 servers, a majority of which are correct. However, if the client informs an old quorum of 6 servers about the new configuration (Figure 1), or even a newly adjusted quorum of 7 servers (dashed line), then there is no guarantee that the intersection with the *old* quorum will contain a majority of correct servers. As a result, another client, which still uses a quorum of size 6, may obtain conflicting information by a collusion of 2 faulty servers.

Our methods address this very problem. We describe protocols that guarantee *safe* shared variable

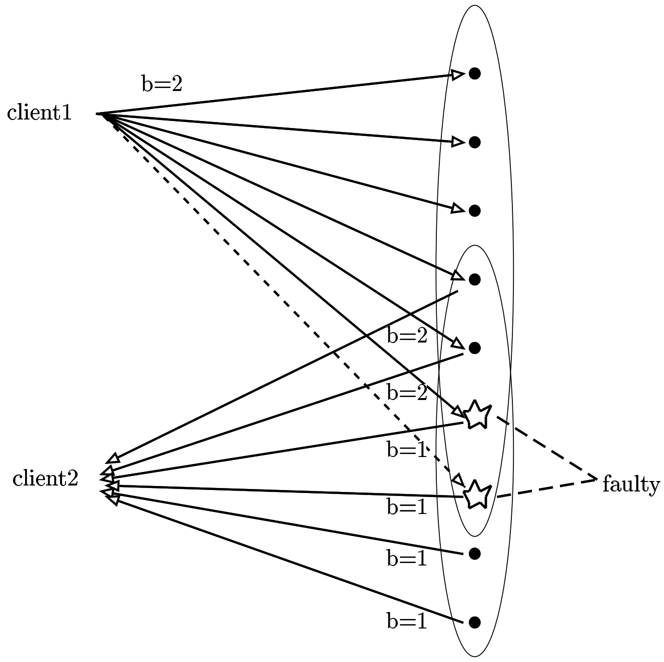

Figure 1. Byzantine failures in quorum threshold adjustment.

semantics [8] in the face of repeated configuration changes, and despite the use of arbitrarily stale quorum configurations by clients. Our work is the first of which we are aware that provides data replication with safe variable semantics in the face of a varying resilience threshold in a Byzantine environment, with no reliance on any concurrency control mechanism (e.g., no locking).

Our approach makes use of a *threshold variable* $\mathcal{B}$ which is coupled with the ordinary replicated variables that our system maintains, and is designated to maintain the current resilience threshold b. We show how to implement such a threshold variable with stronger semantics than safety in order to maintain the safety of the ordinary variables. Updates to $\mathcal{B}$ are made to *announce sets*, which are generally larger than ordinary quorums and are guaranteed to be observed by clients who may be using arbitrarily old thresholds. Ordinary variable operations are accordingly modified to take $\mathcal{B}$ into account. Maintaining safety of variables is complicated by the fact that we allow reads and writes to occur simultaneously with adjustments to $\mathcal{B}$. The challenge is to guarantee sufficient intersection of new quorums with previously written quorums (of unknown thresholds) without accessing too many servers in the normal case.

The structure of the remainder of the paper is as follows. We discuss related work in Section 1.1. In Section 2, we give our system model and preliminary definitions. In Section 3 we introduce the threshold adjustment framework and its requirements. In Section 4, we present the protocols for reading and modifying a dynamic threshold in threshold masking quorum systems. In Section 5, we present the enhanced protocols for reading and writing shared variables using the dynamic threshold. Section 6 generalizes some of the concepts of dynamic threshold adjustments to topographically defined quorum systems. We discuss performance and optimizations in Section 7 and conclude in Section 8.

1.1 Related Work

Our work builds on a substantial body of knowledge on quorum systems and their uses, which was initiated in [6, 16]; for a survey of quorum systems work, see [12]. More specifically, our methods are designed for Byzantine quorum-based replication, which was introduced in [13] and further investigated in various works [3, 15, 14].

Our consideration of Byzantine faults makes our treatment fundamentally different from most prior works on dynamic quorum reconfiguration with benign failures only. In [5, Ch. 8, *missing writes* protocol] and [7, 10], a quorum system is reconfigured by forming a write-quorum in such a way that following readers learn about the reconfiguration in the course of the normal read protocol. Byzantine faults complicate this scheme in that (undetectably) faulty servers might misinform a reader. As discussed earlier, this is especially significant when increasing the resilience threshold, since the reader, which may be using an out-of-date quorum system, could be misled by a new threshold of colluding Byzantine faulty servers.

Quorum *adaptation* for crash failures is discussed in [4]. In their approach, if a server u fails while u is the only server contained in the intersection of two quorums Q_i and Q_j, then the intersection property is maintained by removing u from Q_i and Q_j and by substituting another correct server. Our work differs from [4] in two significant ways. First, the fact that we treat Byzantine failures means that we must enable on-the-fly changes not only to the set of quorums, but also to the very intersection property that defines those quorums. Second, our clients (which we assume to be potentially numerous and transient) do not communicate between themselves and cannot inform one another of changes. Our work differs from [4, 9] as well, in that our protocols can take hints on the possible level of failures in the system in order to adjust the resilience threshold, rather than rely on specific detection of server failures.

Some work in Byzantine agreement considers dynamic threshold changes during the run of the protocol. For example, Bar-Noy et al [2] present an optimal-round agreement protocol that lowers its operating threshold midway through the protocol in order to improve efficiency while still maintaining overall the resilience of the original threshold. Our work differs from theirs by considering general replicated data rather than agreement protocols, but more fundamentally in that our protocols allow the fault threshold to be raised or lowered as fault-tolerance needs change. In particular, they take a detected fault as a sign that therefore there are fewer remaining faults in the system, while we may take it as a sign that the initial threshold was too optimistic and should be increased.

2 Preliminaries

Our system consists of a set U of data servers such that the number $n = |U|$ of servers is fixed. During an execution of the system, a server receives input messages and responds to each with a (non-null) output message. The *correct response* of a server for each input message is defined by a function F that maps the preceding history of all inputs that server received since the beginning of time to an output message. A server that receives the sequence of inputs $m_1, \ldots, m_k$ should respond to m_k with $F(m_1, \ldots, m_k)$, the correct response. At any given moment a server may be either *correct* or *faulty*. A correct server responds to any request with the correct response. We allow Byzantine failures: a faulty server may respond with an arbitrary response (including no response). However, note that a correct response is defined irrespective of a server's prior faulty output behavior (or state modifications). In practice, since servers usually have a state that does not include the entire history, this means that for a once-failed server to be considered correct, it must be recovered to the state that it would have held if it had never failed.

The set of clients of the service is disjoint from U. Each client is assumed to have a FIFO channel to each server. In our system, clients may be numerous and transient. In particular, clients are not aware of each other, nor do they necessarily have the ability to communicate with one another. We restrict our attention in this work to server failures; clients and channels are assumed to be reliable.

We use a replicated data service model based on *quorum systems*, which are defined as follows:

Definition: A *quorum system* on a set U is a set $\mathcal{Q} \subseteq$

2^U such that $\forall Q_1, Q_2 \in \mathcal{Q}, Q_1 \cap Q_2 \neq \emptyset$. Elements of $\mathcal{Q}$ are called *quorums*. ■

In such a service, clients perform read and write operations on a variable by reading or writing its value at a quorum of servers. The intersection property ensures that any read operation observes the value of the most recent write operation; timestamps can be used to distinguish this value from older ones.

A useful type of quorum system for Byzantine fault tolerance is the *b-masking quorum system*:

Definition: A *b-masking quorum system* is a quorum system $\mathcal{Q}$ such that $\forall Q_1, Q_2 \in \mathcal{Q}, |Q_1 \cap Q_2| \geq 2b+1$. ■

This additional intersection property ensures that in spite of up to b faults in the system, any two quorums intersect in at least $b+1$ correct servers. This enables clients to determine the correct variable value using an algorithm that combines voting and timestamps. A b-masking quorum system is thus designed to tolerate failures in a system with a static resilience threshold b.

For the greater part of this paper we focus our attention on a particular type of b-masking system called a *threshold* quorum system. Threshold systems are defined as follows:

Definition: A *threshold* masking quorum system is a quorum system $\mathcal{Q}$ such that $\forall Q \in \mathcal{Q}, |Q| = \lceil (n+2b+1)/2 \rceil$. ■

For simplicity, we assume hereafter that n is odd, so that we can eliminate the ceiling operator from our calculations.

Ordinary (i.e., static) b-masking quorum systems support replicated variables in a Byzantine environment as follows. To emulate a shared variable V, each server u stores a "copy" V_u of V along with a timestamp variable $T_{V,u}$. A timestamp is assigned by a client to $T_{V,u}$ when the client writes V_u. Our protocols require that different clients choose different timestamps, and thus each client c chooses its timestamps from some set $\mathcal{T}_c$ that does not intersect $\mathcal{T}_{c'}$ for any other client c'. The timestamps in $\mathcal{T}_c$ can be formed, e.g., as integers appended with the name of c in the low-order bits. Note that faulty servers may return arbitrary values both for variables and for timestamps. The read and write operations are implemented as follows.

Write: For a client to write the value v to variable V, it queries servers in some quorum Q to obtain the timestamp t_u from $T_{V,u}$ at each $u \in Q$. The client then chooses a timestamp $t \in \mathcal{T}_c$ greater than the highest timestamp value in $\{t_u\}_{u \in Q}$ and greater than any timestamp it has chosen in the past, and updates V_u and $T_{V,u}$ at each server u in some quorum Q' to v and t, respectively.

Read: For a client to read a variable V, it queries servers in some quorum Q to obtain values v_u, t_u from variables $V_u, T_{V,u}$ at each $u \in Q$. From among all $\langle v_u, t_u \rangle$ pairs returned by at least $b+1$ servers in Q, the client chooses the pair $\langle v, t \rangle$ with the highest timestamp t, and then returns v as the result of the read operation. If there is no pair returned by at least $b+1$ servers, the result of the read operation is $\perp$ (a null value).

A server u that receives an update $\langle v, t \rangle$ during a write operation updates $V_u, T_{V,u}$ to $\langle v, t \rangle$, respectively, iff $t > T_{V,u}$. This pair of protocols guarantees *safe* [8] variable semantics, i.e., a read operation that does not overlap a write operation returns the value of the most recent write (the proof of this assertion can be found in [13]).

3 Threshold Adjustment

The read/write protocols above provide safe variable emulation in a Byzantine environment with a static resilience threshold. The goal of this work is to extend these protocols so as to allow dynamic adaptations of quorum systems to varying resilience thresholds. The challenge is to maintain safety of any replicated variable in the system while dynamically performing such changes, without stopping the normal operation of the system.

To accommodate changes in the threshold setting, we introduce a new replicated variable $\mathcal{B}$ that contains the current threshold setting. The variable $\mathcal{B}$ can be written with integral values in the range $[b_{min}, b_{max}]$. The threshold variable has an associated timestamp $T_{\mathcal{B}}$ that follows the same rules as the timestamps of other variables: every update to the variable is stamped with a unique timestamp that is greater than any timestamp used in a previously completed operation.

A client that wishes to change the resilience threshold for the system must first write the up-to-date threshold into $\mathcal{B}$, and then continue performing operations with the new resilience threshold accounted for. Intuitively, a client can update $\mathcal{B}$ in much the same way that it updates any other shared variable. There is one significant complication, however: because the threshold is dynamic, different clients may have different memories of its value depending on how recently they have accessed the quorum system. It is therefore necessary that new threshold values be written to a set of servers whose intersection with *all possible quorums* (i.e., defined by any $b \in [b_{min}, b_{max}]$) is sufficiently large to allow clients to determine unambiguously the correct current threshold during any given operation; the client can then continue or restart the operation accordingly. Hence, in a *threshold write* operation, $\mathcal{B}$ is updated at all servers in an *announce set*. A few issues need to be addressed in order to specify threshold adjustment fully. First, we need to specify the intersection requirement between the announce set and all ordinary quorums in such a way that threshold adjustments will be noticed by all potential clients. Second, we need to specify threshold write and read protocols. Third, we need to modify our read/write protocols to account for threshold adjustments. These will be the topic of discussion in the remainder of this paper.

Since the system for which we design our protocols has a dynamically changing resilience threshold, a standard threshold constraint, e.g., "the number of faulty servers in the system at any given time does not exceed b," does not suffice for our purposes. Rather, in order to guarantee correctness, we need a statement that the dynamically written threshold values are correct. To this end, we adopt the following assumption for the remainder of the paper.

Assumption 1 *Let o be any operation, i.e., a threshold read, threshold write, variable read, or variable write. Let b be the minimum among (i) the value written in the last write to $\mathcal{B}$ preceding o (in some serialization of all preceding writes) and (ii) the values written to $\mathcal{B}$ in any threshold writes that are concurrent with o. Then, no quorum access issued within o returns more than b faulty responses (i.e., no more than b servers are "currently" faulty).*

In practice, Assumption 1 amounts to requiring that the threshold should be changed proactively and at a reasonable rate.

The goal of our threshold adjustment protocols is to maintain safety of all replicated variables despite possible modifications to the resilience threshold. For the purpose of safety, we treat the associated threshold variable $\mathcal{B}$ as an integral part of any variable V. Accordingly, the modified safety condition that our protocols will satisfy is the following.

Safety: A read operation on V that overlaps no writes to V or threshold write operations to $\mathcal{B}$ returns the value of the most recent write to V that precedes this read, in some serialization of all write operations preceding it.

4 Quorum Adjustment in Threshold Systems

In this section we present and discuss protocols that allow clients to read and adjust the fault tolerance of a threshold masking quorum system. An important property of our protocols is that they require no direct interaction among clients; all information is passed through shared variables.

4.1 Basic Protocol

We wish to design protocols in which the resilience threshold of the underlying quorum system can be dynamically adjusted to any value within some range $[b_{min}, b_{max}]$. The simplest way to ensure that clients always use the correct threshold is to require clients to read the threshold before any read or write, and to adopt a threshold value only if at least $b_{max}+1$ servers in a quorum agree on this threshold and its associated timestamp. By Assumption 1 and the definition of b_{max}, no more than b_{max} responses to any query will be faulty. Therefore, if the announce set for a threshold change intersects every possible quorum (i.e., every set of size $(n+2b+1)/2$ for $b_{min} \leq b \leq b_{max}$), in at least $2b_{max}+1$ servers, it follows that the response to any query will include at least $b_{max}+1$ notifications of the change. (One advantage of this approach is that clients need not maintain their own copy of the current value of the threshold b. In particular, new clients can join without having to initialize a copy of the threshold.)

We take as announce sets all sets of size $n-b_{max}$, i.e., the largest number of servers guaranteed to be available under any threshold setting. This value will be sufficient provided that the intersection between any quorum and any announce set is of size at least $2b_{max}+1$. That is, we need:

$$2b_{max} + 1 \leq ((n + 2b_{min} + 1)/2) + (n - b_{max}) - n$$

It follows that $n \geq 6b_{max} - 2b_{min} + 1$, and thus we need at least $6b_{max} - 2b_{min} + 1$ servers to provide a dynamic threshold quorum system whose threshold ranges from b_{min} to b_{max}. We take this as an assumption for the remainder of the paper.

Assumption 2 $n \geq 6b_{max} - 2b_{min} + 1$.

Note that Assumption 2 is a generalization of the $4b+1$ servers required by a static threshold system, i.e., one where $b_{min} = b_{max}$ [13].

The protocol for raising or lowering the threshold using an announce set of size $n - b_{max}$ is as follows:

Threshold write: For a client to set $\mathcal{B}$ to a new threshold value b, it queries servers in some announce set A (of size $n - b_{max}$) to obtain values b_u, t_u from variables $\mathcal{B}_u, T_{\mathcal{B},u}$ at each $u \in A$. It then chooses a timestamp $t \in \mathcal{T}_c$ greater than the largest timestamp in $\{t_u\}_{u \in Q}$, and greater than any timestamp it has chosen in the past. Finally, at each server u in some announce set A', it updates $\mathcal{B}_u$ and $T_{\mathcal{B},u}$ to b and t, respectively. (Note that this is exactly the static variable write protocol except that an announce set is used instead of a standard quorum.)

Threshold read: For a client to read the current threshold value b from $\mathcal{B}$, it queries servers in some quorum Q of size $(n + 2b_{min} + 1)/2$ to obtain values b_u, t_u from variables $\mathcal{B}_u, T_{\mathcal{B},u}$ at each $u \in Q$. Of the $\langle b_u, t_u \rangle$ pairs returned by at least $b_{max} + 1$ servers in Q, it selects the pair $\langle b, t \rangle$ with the highest timestamp t, provided that it is not *countermanded* as defined below. If there is no such pair, or if $\langle b, t \rangle$ is countermanded, then it sets b to $\perp$ (undefined).

Definition: A threshold/timestamp pair $\langle b, t \rangle$ is *countermanded* in a given query if at least $b_{max} + 1$ servers return threshold timestamps (not necessarily identical) greater than t. A threshold value b is countermanded if all the pairs it appears in are countermanded. ■

The purpose of this definition is made clear by the following theorem:

Theorem 1 *If b is older than the most recently completed threshold write at the time of a threshold read, then it will be countermanded in that read.*

Proof: If no threshold write operations are taking place concurrently with the threshold read, then the result follows immediately from Assumption 1, the intersection property between announce sets and quorums, and the fact that $b_{max} \geq b$. Furthermore, for any b, the number of correct servers whose threshold timestamp exceeds that of b is monotonically nondecreasing over the course of a threshold write. Therefore the result holds during threshold writes as well. ■

4.1.1 Correctness

The correctness of the threshold variable follows from the following theorem and subsequent corollary to Theorem 1.

Theorem 2 *In any threshold read that does not overlap a threshold write, the most recently written threshold value (in the serialization consistent with timestamp order of the writes) is returned.*

This theorem is easily seen to be implied by the following two lemmas:

Lemma 1 *For any such threshold read, the most recently written threshold/timestamp pair is returned by at least $b_{max} + 1$ servers.*

Proof: Let b be the most recently written threshold value. The announce set for this threshold intersects all possible quorums in at least $2b_{max} + 1$ servers by Assumption 2. Because b was set in the most recent threshold write, and the current threshold read does not overlap any threshold writes, the variable $\mathcal{B}$ has not been overwritten at any correct servers in this set. By Assumption 1 and the fact that $b \leq b_{max}$, at least $b_{max} + 1$ servers in any possible quorum will return the threshold b along with the most recent timestamp. ■

Lemma 2 *For any such threshold read, the most recently written threshold is not countermanded.*

Proof: The most recently written threshold has the highest (nonforged) timestamp. Therefore if the correct threshold is b, then by Assumption 1, no more than b servers may forge higher timestamps in their response to the query. Since $b < b_{max} + 1$, the value b is not countermanded. ■

Corollary 1 *A threshold read that overlaps one or more threshold writes will not return a threshold older than the value in the most recently completed threshold write.*

Proof: This follows from Theorem 1 and the fact that the read does not return a countermanded value. ■

Remark: A consequence of this theorem and corollary is that the protocol given above implements a weakened version of *regular* variable semantics [8] for the threshold variable; i.e., a query that overlaps one or more writes will return either the value of the most recently completed write, the value of one of the writes which it overlaps, or $\perp$. This is a stronger guarantee than that provided by safe semantics.

5 Variables Implemented with Dynamic Threshold Systems

In a quorum system whose resilience threshold is dynamic, a change to the quorum structure may require some attention to the variables that make use of that threshold. Specifically, an increase in the threshold may compromise the integrity of previously written variables unless some specific corrective action is taken.

Suppose, for example, that the threshold of a system is increased from b to $b+3$. Once this operation is complete, clients performing read and write operations will learn of the new threshold and perform those operations on quorums of the new size. Unfortunately, values that have been written under the previous threshold will appear only at an *old* quorum of servers. The intersection between an old quorum and a new one is only guaranteed to be $2b+4$, not the $2b+7$ required for tolerating an additional 3 faults.

More generally, the main difficulty is that if a variable was last written when the threshold was smaller than the current threshold b, then reading from a quorum of size $(n+2b+1)/2$ does not suffice to ensure that $b+1$ correct servers will respond with the latest value. Rather, to ensure that $b+1$ correct servers will respond with the latest value, it may be necessary to increase the quorum used during a read operation to $(n+2b+1)/2+b-b_{min}$.

A similar problem occurs when a writer accesses a quorum of servers in order to determine a timestamp for the write. The writer needs to access a quorum that guarantees intersection in one correct server with the most recently written quorum. The difficulty is that the latter could have a quorum size that corresponds to an arbitrarily old threshold. If the current threshold b can be determined, then a quorum of size $(n+2b+1)/2-b_{min}$ suffices to intersect in $b+1$ with any other quorum, and hence, in at least one correct server. However, if the threshold cannot be determined, which can happen when a write operation overlaps a threshold-write operation, then a (potentially larger) quorum of size $(n+2b_{max}+1)/2-b_{min}$ needs to be accessed in order to determine a correct timestamp for the write operation.

We address these issues in the protocol below.

5.1 The Protocol

The protocol for reading and writing a variable V using dynamic thresholds is as follows:

Read: For a client to read variable V, it performs the following steps:

1. Perform a threshold read using the protocol in Section 4 to obtain current threshold b. If $b=\perp$, return $\perp$ as the result of the read.
2. Query servers in a quorum Q of size $(n+2b+1)/2$ to obtain values v_u, t_u from variables $V_u, T_{V,u}$ at each $u \in Q$.
3. Of the $\langle v_u, t_u\rangle$ pairs returned by at least $b_{min}+1$ servers in Q, consider the pair $\langle v, t\rangle$ with the highest value of t. If no such pair exists, return $\perp$ as the result of the read. If $\langle v, t\rangle$ appears in $b+1$ identical responses, return v as the result of the read.
4. Otherwise (i.e., $\langle v, t\rangle$ appears at least $b_{min}+1$ times but fewer than $b+1$), query servers in an additional set C of size $b-b_{min}$, to obtain values v_u, t_u from variables $V_u, T_{V,u}$ at each $u \in Q'$, where $Q' = Q \cup C$ contains $(n+2b+1)/2+b-b_{min}$ servers.
5. Of the $\langle v_u, t_u\rangle$ pairs that appear in at least $b+1$ responses from Q', select the pair $\langle v', t'\rangle$ with the highest value of t' and return v' as the result of the read. If no such pair exists, return $\perp$ as the result of the read.

Write: For a client to write value v to variable V, it performs the following steps:

1. Perform threshold read using the protocol in Section 4 to obtain the current threshold b. If $b=\perp$, then use $b=b_{max}$.
2. Query servers in a quorum Q of size $(n+2b+1)/2-b_{min}$ to obtain timestamp t_u from $T_{V,u}$ at each $u \in Q$.
3. Create a new timestamp $t \in \mathcal{T}_c$ such that t is larger than any timestamp in $\{t_u\}_{u\in Q}$ and any timestamp used before by this client.
4. Write v and t to V_u and $T_{V,u}$, respectively, at each server u in a quorum of size $(n+2b+1)/2$.

Note that in a steady system state, when reads and writes obtain the up-to-date threshold b, they perform operations simply by accessing ordinary b-masking quorums, i.e., of size $(n+2b+1)/2$. Following adjustments to the threshold, though, operations may incur the higher costs of accessing larger quorums.

5.1.1 Correctness

The following theorem proves the correctness of the above protocol—namely, that it maintains safety of the variable V:

Theorem 3 *A read operation that overlaps no write operations to V or threshold write operations to $\mathcal{B}$ returns the value of the most recent write to V that precedes this read, in some serialization of all write operations preceding it.*

Proof: Let W denote the set of all write operations preceding the read. By Theorem 2, Step 1 of the read operation obtains the most recently written threshold b or a concurrently written one. Therefore, by Assumption 1, any quorum of responses obtained in the read contains at most b faulty responses. Consider the write operation in W with the highest timestamp. Since this write completed at $(n + 2b_{min} + 1)/2$ or more servers, its value and timestamp appear in at least $b_{min} + 1$ of the responses returned in Step 2 of the read protocol. It is then returned in Step 3 if it appeared in $b+1$ of the responses from Step 2, or otherwise will be returned in Step 5 since reading from $(n + 2b + 1)/2 + b - b_{min}$ (Step 4) intersects any previous write quorum in at least $2b + 1$ servers.

It is left to show that there is a serialization of the writes in W in which the write with the highest timestamp is last, i.e., that a write operation w_2 that follows a write operation w_1 uses a higher timestamp. This follows from the facts that if w_2 uses the threshold b', then at most b' faulty responses to its query in Step 2 are returned, and that $(n + 2b' + 1)/2 - b_{min}$ servers must intersect the quorum written in w_1 in $b' + 1$ servers (and thus at least one correct server). ■

6 Other b-Masking Quorum Systems

In this section, we briefly discuss how to employ two additional b-masking quorum systems in an environment with a dynamically varying resilience threshold, and how to set an appropriate announce set for changing the threshold value. Thus, the utility of our methods is not limited to the threshold construction.

6.1 BoostFPP quorum system

BoostFPP masking quorum systems [15] are constructed as a composition of two quorum systems. The first is a quorum system based on a finite projective plane (FPP), suggested originally by [11]. In the FPP quorum system, there are $q^2 + q + 1$ elements and quorums of size $q + 1$ (corresponding to the hyperplanes of the FPP), where $q = p^r \geq 2$ for some prime p and integer r. Each pair of distinct quorums in FPP intersect in exactly one element. The second quorum system is a threshold b-masking quorum system with some system size $s \geq 4b + 1$. The composition of the two systems is made by replacing each element of the FPP with a distinct copy of a threshold system. That is, the universe for a boostFPP system is $U = \bigcup_{i=1}^{q^2+q+1} U_i$ where each U_i is a set of s servers, and $U_i \cap U_j = \emptyset$ for any $i \neq j$. Each U_i is called a "super element". A quorum is selected by first selecting a quorum of super elements in the FPP, say $U_{i_1}, \ldots, U_{i_{q+1}}$, and then selecting $\lceil (s+2b+1)/2 \rceil$ servers from each U_{i_j}. A boostFPP is a b-masking quorum system since every pair of quorums of super elements intersect in at least one super element, say U_i, while the selection of threshold quorums within U_i guarantees intersection of $2b + 1$ elements.

To employ boostFPP with a variable resilience threshold $b_{min} \leq b \leq b_{max}$, we leave the FPP construction of super elements unmodified, and change only the selection of servers within each super-element. That is, we require that $s \geq 6b_{max} - 2b_{min} + 1$ and for each U_i and any threshold b, we select quorums in U_i as in the threshold system, e.g., an ordinary quorum has $\lceil (s + 2b + 1)/2 \rceil$ servers in each U_i, and an announce set comprises of $s - b_{max}$ servers from each such U_i.

It is easily seen that such selections guarantee the required intersection size between announce sets and ordinary quorums, as well as between read and write quorums and pairs of write quorums, as in the threshold system case.

6.2 M-grid quorum system

An M-Grid masking quorum system is described in [15]. For any resilience threshold b, where $b \leq (\sqrt{n}-1)/2$, M-Grid is constructed as follows: The universe of n servers is arranged as a $\sqrt{n} \times \sqrt{n}$ grid. A quorum in an M-Grid consists of any choice of $\sqrt{b+1}$ rows and $\sqrt{b+1}$ columns. Formally, 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 is

$$\text{M-Grid}(b) = \left\{ \bigcup_{j \in J} C_j \cup \bigcup_{i \in I} R_i : J, I \subseteq \{1 \ldots \sqrt{n}\}, |J| = |I| = \sqrt{b+1} \right\}$$

M-Grid maintains the requirement of b-masking quorum systems as follows: If a pair of quorums overlap in a full row or column, then there are $\sqrt{n} \geq 2b+1$ elements in their intersection. Otherwise, their intersection contains the crossing points of all rows of one quorum with columns of the other, and vice versa, and hence contains at least $2\sqrt{b+1}\sqrt{b+1} > 2b + 1$ elements.

To make use of M-Grid quorum systems with a variable resilience threshold $b_{min} \leq b \leq b_{max}$, we need to require that $b_{max} \leq (\sqrt{n}-1)/2$. The grid arrangement remains static for all quorum systems, but the number of rows/columns in each quorum will depend on b, the current resilience threshold. For the purpose of setting the threshold variable $\mathcal{B}$, we use announce sets comprising of $\left\lceil \frac{(b_{max}+1)}{\sqrt{b_{min}+1}} \right\rceil$ rows and $\left\lceil \frac{(b_{max}+1)}{\sqrt{b_{min}+1}} \right\rceil$ columns, which guarantees that they intersect any quorum ever used in $2b_{max}+1$ servers. With these announce sets, we use the same threshold write and threshold read protocols as for the threshold b-masking system. Unfortunately, the read and write protocols cannot use ordinary size quorums, since in general, quorums in M-Grid(b) may not intersect quorums in M-Grid(b') in $b+b_{min}+1$ elements as required. Hence, for Step 2 of the read protocol, we need to use quorums comprising of $\max\{\sqrt{b+1}, \frac{b+b_{min}+1}{2\sqrt{b_{min}+1}}\}$ rows and columns to guarantee intersection of $b_{min}+1$ correct servers with any previously written quorum, and intersection of $2b+1$ with other quorums using the b threshold (the normal case). In Step 4 of the read protocol, we use enlarged quorums of $\frac{b+1}{\sqrt{b_{min}+1}}$ rows and $\frac{b+1}{\sqrt{b_{min}+1}}$ columns, guaranteeing intersection in $b+1$ correct servers. For Step 2 of the write protocol, a quorum comprising of $\frac{b+1}{2\sqrt{b_{min}+1}}$ rows and $\frac{b+1}{2\sqrt{b_{min}+1}}$ columns is queried for timestamps, guaranteeing intersection in $b+1$ servers (and hence, one correct) with any previously written quorum. Finally, it suffices to send updates to ordinary b-quorums containing $\sqrt{b+1}$ rows and $\sqrt{b+1}$ columns.

The proof of correctness is essentially identical to the threshold system case, simply making use of the intersection size statements for this construction.

7 Discussion

7.1 Comparison to static quorums

When deploying a system in practice, the maximum anticipated number of failures b_{max} is typically calculated as a function of the total number of servers n, e.g., based on an analysis of the probability of each individual server failing. A disadvantage of the approach in this paper, as compared to a static quorum system deployment, is that it can accommodate fewer values of b_{max} for a given n: ours requires $n \geq 6b_{max} - 2b_{min} + 1$ servers, as opposed to only $n \geq 4b_{max}+1$ in the static case. However, for those configurations of n and b_{max} where our dynamic approach is possible, our approach performs better than a static quorum system in the common case, where there are no Byzantine failures and the system runs with a threshold of (or close to) b_{min}.

More specifically, the measure of efficiency that we consider is quorum size, since this determines the number of servers a client must access in order to perform an operation. Let $Q(b)$ denote the quorum size for a static quorum system with threshold b; e.g., in the threshold system, $Q(b) = (n+2b+1)/2$. For all of the quorum constructions we have described—threshold, boostFPP, and M-Grid—variable read and write operations access quorums of only size $Q(b_{min})$ while the system runs with a threshold of b_{min} and there are no threshold write operations. This compares favorably to the $Q(b_{max})$-sized quorums that a static system would use.

That said, when the threshold is raised to some $b > b_{min}$, penalties can be experienced that exceed the costs of accessing a static system. For example, in the threshold quorum construction, the threshold write itself accesses $n - b_{max}$ servers; variable read operations may access quorums of size $Q(2b - b_{min})$; and variable writes may access quorums of size $Q(b_{max})$ when concurrent with threshold write operations. Similarly, our variable read and write protocols for M-Grid employ quorums comprised of $O(b)$ rows and columns, so their cost is on the order of $Q(b^2)$. Also, because read operations return $\perp$ if the threshold read does, the likelihood that a read operation returns $\perp$ is increased.

In the performance discussion above, we have ignored the number of communication rounds performed by our protocols, and indeed, have not attempted to optimize for it. An obvious direction for optimization is to couple together the threshold reading with the value or timestamp reading at the beginning of variable read or write operations. For brevity, we do not include these in the exposition here.

7.2 An alternative approach

One of the strongest restrictions on the work we have presented here is that of Assumption 1. At the cost of making writes pessimistically static (and thus more expensive) we can weaken this restriction while still maintaining safety. Specifically, we can replace it with:

Assumption 3 *Any operation that is concurrent with no threshold writes receives no more than b faulty responses in any quorum access, where b is the current value of $\mathcal{B}$. In addition, no quorum access in any operation (even one concurrent with threshold writes) returns more than b_{max} faulty responses.*

A consequence of the second half of this assumption is that a write quorum performs correctly if it intersects all other write quorums in at least $b_{max}+1$ servers. This is accomplished by any write quorum size between $(n+b_{max}+1)/2$ and $n-b_{max}$. However, there remains the requirement of ensuring that every read quorum intersects every write quorum in at least $2b+1$ servers, where b is the current threshold during the read.[2] If we use a write quorum of the smallest

[2] Since the safety property applies to reads that do not overlap threshold adjustment operations, b is well defined.

size, reads are *more* expensive than in the static case for values of b that are sufficiently close to b_{max}. If, on the other hand, we use write quorums of size $n - b_{max}$, it becomes unnecessary for read operations to be aware of the current threshold at all; we have shown above that a read quorum based on b_{min} intersects such a write quorum in at least $2b_{max} + 1$ servers. This is a potentially useful tradeoff for systems in which reads are much more frequent than writes. It is, however, a static system and as such is uninteresting from the point of view of this work.

A more interesting approach is to set the write quorum to the same size as in a static b_{max} system, i.e. $(n + 2b_{max} + 1)/2$. In this case, a read quorum can be sure of a sufficient intersection with the previous write quorum if it is of size $(n + 4b - 2b_{max} + 1)$, where b is the current threshold. If $b = b_{max}$, then this is exactly the same as a static b_{max} read quorum; if $b < b_{max}$ it is an improvement even over a static read quorum for b, let alone b_{max}. For systems in which write operations are sufficiently less frequent than read operations, this alternative way of using a varying threshold may be attractive.

8 Conclusion

In this paper, we have presented protocols for reading and adjusting the Byzantine fault tolerance level of a threshold masking quorum system, and shown how these protocols can be extended to other types of b-masking quorum systems, specifically M-Grid and boostFPP systems. In doing so, we have preserved the safe variable semantics provided by such systems.

References

[1] L. Alvisi, D. Malkhi, E. Pierce, and M. Reiter. Fault detection for Byzantine quorum systems. In *Proceedings of the 7th International Working Conference on Dependable Computing for Critical Applications*, pages 357–372, January 1999.

[2] A. Bar-Noy, D. Dolev, C. Dwork, and R. Strong. Shifting gears: Changing algorithms on the fly to expedite Byzantine agreement. *Information and Computation*, 97(2):205–233, 1992.

[3] R. A. Bazzi. Synchronous Byzantine quorum systems. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, pages 259–266, August 1997.

[4] M. Bearden and R. Bianchini. A fault-tolerant algorithm for decentralized online quorum adaptation. In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing (FTCS 98)*, pages 262–271, June 1998.

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

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

[7] M. Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Transactions on Database Systems*, 12(2), June 1987.

[8] L. Lamport. On interprocess communications (part ii: algorithms). *Distributed Computing*, 1:86–101, 1986.

[9] E. Lotem, I. Keidar, and D. Dolev. Dynamic voting for consistent primary components. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing (PODC)*, August 1997.

[10] N. Lynch and A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 20th Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, June 1997. Seattle, Washington.

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

[12] D. Malkhi. *Quorum Systems.* in *The Encyclopedia of Distributed Computing.* Joseph Urban and Partha Dasgupta editors, Kluwer Academic Publishers, To be published.

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

[14] D. Malkhi, M. Reiter, A. Wool, and R. N. Wright. Probabilistic Byzantine quorum systems. Technical Report 98.7, AT&T Research, 1998.

[15] D. Malkhi, M. K. Reiter, and A. Wool. The load and availability of Byzantine quorum systems. In *Proceedings 16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 249–257, August 1997. To appear in Siam Journal of Computing.

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