A Framework for Dynamic Byzantine Storage

Jean-Philippe Martin, Lorenzo Alvisi
Laboratory for Advanced Systems Research
The University of Texas at Austin
{jpmartin,lorenzo}@cs.utexas.edu *
Extended technical report

Abstract

We present a framework for transforming several quorum-based protocols so that they can dynamically adapt their failure threshold and server count, allowing them to be reconfigured in anticipation of possible failures or to replace servers as desired. We demonstrate this transformation on the dissemination quorum protocol. The resulting system provides confirmable wait-free atomic semantics while tolerating Byzantine failures from the clients or servers. The system can grow without bound to tolerate as many failures as desired. Finally, the protocol is optimal and fast: only the minimal number of servers —$3f + 1$— is needed to tolerate any f failures and, in the common case, reads require only one message round-trip.

1. Introduction

Quorum systems [5] are a valuable tool for building highly available distributed data services. These systems store a shared variable at a set of servers and perform read and write operations at some subset of these servers (a *quorum*). To access the shared variable, protocols define some intersection property for the quorums which, combined with the protocol description themselves, ensure that read and write operations obey precise consistency semantics. In particular, a shared register can provide, in order of increasing strength, *safe, regular,* or *atomic* semantics [11].

Malkhi and Reiter [13] have pioneered the study of *Byzantine* quorum systems (BQSs), in which servers may fail arbitrarily. Their *masking quorum systems* guarantee data integrity and availability despite compromised servers; they also introduce *dissemination quorum systems* that can be used by services that support *self-verifying data*, i.e., data that cannot be undetectably altered by a faulty server, such as data that have been digitally signed or associated with message authentication codes (MACs).

Traditional BQS protocols set two parameters—N, the set of servers in the quorum system, and f, the *resilience threshold* denoting the maximum number of servers that can be faulty[1]—and treat them as constants throughout the life of the system. The rigidity of these static protocols is clearly undesirable.

Fixing f forces the administrator to select a conservative value for the resilience threshold, one that can tolerate the worst case-failure scenario. Usually, this scenario will be relatively rare; however, since the value of f determines the size of the quorums, in the common case quorum operations are forced to access unnecessarily large sets, with obvious negative effects on performance.

Fixing N not only prevents the system administrator from retiring faulty or obsolete servers and substituting them with correct or new ones, but also greatly reduces the advantages of any technique designed to change f dynamically. For a given Byzantine quorum protocol, N must be chosen to accommodate the maximum value f_{max} of the resilience threshold, independent of the value of f that the system uses at a given point in time. Hence, in the common case the degree of replication required to tolerate f_{max} failures is wasted.

Alvisi et al. [2] take a first step towards addressing these limitations. They propose a protocol that, for a fixed N, can dynamically raise or lower f within a range $[f_{min} \cdots f_{max}]$ at run time without relying on any concurrency control mechanism (e.g., no locking). Improving on this result, Kong et al. [10] propose a protocol that can dynamically adjust f and, once faulty servers are detected, can ignore them to obtain quorums that exhibit better *load*[2], effectively shrinking N. The protocol however does not allow to add new servers to N. While other quorum-based systems such as Rambo [12], Rambo II [8],

* This work was supported in part by grants from the Texas Advanced Technology Program and Sandia National Laboratories and by an Alfred P. Sloan Fellowship.

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.

2 Given a quorum system S, the *load* of S is the access probability of the busiest quorum in S, minimized over all strategies.

and GeoQuorums [6] can adjust dynamically both f and N, they cannot tolerate Byzantine failures.

In this paper we propose a methodology for transforming static Byzantine quorum protocols into dynamic ones where both N and f can change, growing and shrinking as appropriate[3] during the life of the system. We have successfully applied our methodology to several Byzantine quorum protocols [9, 13, 14, 17, 18]. The common characteristic of these protocols is that they are based on the *Q-RPC* primitive [13]. A Q-RPC contacts a responsive quorum of servers and collects their answers, making it a natural building block for implementing quorum-based read and write operations. Our methodology is simple and non-intrusive: all that it requires to make a protocol dynamic is to substitute each call to Q-RPC with a call to a new primitive, called DQ-RPC for *dynamic* Q-RPC. DQ-RPC maintains the properties of Q-RPC that are critical for the correctness of Byzantine quorum protocols, even when N and f can change.

Defining DQ-RPC to minimize changes to existing protocols is challenging. The main difficulty comes from proving that read and write operations performed on the dynamic version of a protocol maintain the same consistency semantics of the operations performed on the static version of the same protocol. In the static case, these proofs rely on the intersection properties of the responsive quorums contacted by Q-RPCs while performing the read and write operations. Unfortunately, these proofs do not carry easily to DQ-RPC. When N changes, it is no longer possible to guarantee quorum intersection: given any two distinct times t_1 and t_2, the set of machines in N at t_1 and t_2 may be completely disjoint. We address this problem by taking a fresh look at what makes Q-RPC-based static protocols work.

Traditionally, the correctness of these protocols relies on properties of the quorums themselves, such as intersection. Instead, we focus our attention on the properties of the *data* that is retrieved by quorum operations such as Q-RPC. In particular, we identify two such properties, *soundness* and *timeliness*. Informally, soundness states that the data that clients gather from the servers was previously written; timeliness requires this data to be as recent as the last written value. We call these properties *transquorum* properties, because they do not explicitly depend on quorum intersection. We prove that transquorum properties are sufficient to guarantee the consistency semantics provided by each of the protocols that we consider. Now, all that is needed to complete our transition from static to dynamic protocols is to show an instance of a quorum operation that satisfies the transquorum properties even when f and N are allowed to change: we conclude the paper by showing that DQ-RPC is such an operation.

The rest of the paper is organized as follows. We cover related work and system model, respectively, in Section 2 and Section 3. We specify the transquorum properties in Section 4 and show in Section 5 that our DQ-RPC satisfies the transquorum properties before concluding.

2. Related work

Alvisi et al. [2] are the first to propose a dynamic BQS protocol. They let quorums grow and shrink depending on the value of f, which is allowed to range dynamically within an interval $[f_{min}, ..., f_{max}]$. This flexibility, however, comes at a cost: because their protocol does not allow to change N, it requires $2(f_{max} - f_{min})$ more servers than an equivalent static protocol to tolerate a maximum of f_{max} failures.

The Agile store [10] modifies the above protocol by introducing a special, fault-free node that monitors the set of servers in the quorum system. The monitor tries to determine which are faulty and to inform the clients, so that they can find a responsive quorums more quickly. In the Agile store servers can be removed from N, but not added. Therefore, if the monitor mistakenly identifies a node as faulty and removes it from N, the system's resilience is reduced: The system tolerates f_{max} Byzantine faulty servers only as long as the monitor never makes such mistakes.

The Rosebud project [19] shares several of our goals. Rosebud envisions a dynamic peer to peer system, where servers can fail arbitrarily, the set of servers can be modified at run-time, and clients use quorum operations to read and write variables. It is hard to compare our protocols to Rosebud, because the only Rosebud reference we have identified [19] does not give specific details of the protocols they intend to use to achieve their goals. Nonetheless, Rosebud, by requiring loosely synchronized clocks and assuming servers with a cryptographic co-processor, appears to make stronger assumptions than we do in this paper. Also, Rosebud's handling of view changes appears to differ from ours in at least two ways. First, when an operation in Rosebud detects that the set of servers is changing, it simply restarts; second, Rosebud allows N to change only at pre-set intervals. In contrast, we allow operations to continue even as N is changing, and we allow N (and f) to change at any time.

Several quorum-based protocols allow to change N and f, but only tolerate crash failures. Rambo and Rambo II [8, 12] provide the same interface as our protocols: read, write and reconfigure. They guarantee atomic semantics in an unreliable asynchronous network despite crash failures.

In GeoQuorums [6] the world is split into n focal points and servers are assigned to the nearest (geographically) fo-

3 We focus on the mechanisms necessary for supporting dynamic quorums. A discussion of the policies used to determine when to adjust N and f is outside the scope of this paper. Some examples of such policies are given in [3, 10].

4 Partial-atomic semantics guarantees that reads either satisfy atomic semantics or abort [18].

name	can tolerate (crash,Byz)	client failures	semantics	servers required
crash	$(f,0)$, without signatures	crash	atomic	$2f+1$
U-dissemination [17]	$(0,b)$, using signatures	crash	atomic	$3b+1$
hybrid-d [9]	(f,b), using signatures	crash	atomic	$2f+3b+1$
U-masking [18]	$(0,b)$, without signatures	correct	partial-atomic[4]	$4b+1$
hybrid-m [9]	(f,b), without signatures	correct	partial-atomic[4]	$2f+4b+1$
Phalanx [14]	$(0,b)$, without client signatures	Byzantine	partial-atomic[4]	$4b+1$
hybrid Phalanx	(f,b), without client signatures	Byzantine	partial-atomic[4]	$2f+4b+1$

Figure 1: List of quorum protocols that can be made dynamic using DQ-RPC

cal point. The system provides atomic semantics as long as no more than f focal points have no servers assigned to them. Servers can join and leave; however, neither n nor f can change with time.

Abraham et al. [1] target large systems, such as peer-to-peer, where it is important for clients to issue reads and writes without having to know the set of all servers, and it is important for servers to join and leave without having to contact all servers. Their *probabilistic quorums* meet these goals (for example, clients only need to know $O(\sqrt{n})$ servers), provide atomic semantics with high probability, and can tolerate crash failures of the servers.

View-oriented group communication systems provide a membership service whose task is to maintain a list of the currently active and connected members of a group [4]. The output of the membership service is called a *view*. If we consider the set of servers in the quorum system as a group, then in our protocol the membership service is trivially implemented by an administrator, who is solely responsible for steering the system from view to view (see Section 5.1).

An interesting property of our protocol is that it allows processes who are outside the quorum systems — i.e. the clients in our protocol—to query servers within the quorum system to learn the current view. Note that our clients do not learn about views from the membership service, but rather indirectly, through the servers. Nonetheless, our protocol guarantees that, despite Byzantine failures of some of the servers, a correct client will only accept views created by the administrator and will never accept as current a view that is obsolete (see Section 5.1).

3. System model

Our system consists of a set N of n servers. Servers can dynamically join and leave the system, i.e. both N and n can change during execution. To prevent Sibyl attacks [7], the identity of every server is verified before it is allowed to join the system. Servers can be either correct or faulty. A correct server follows its specification; a faulty server can arbitrarily deviate from its specification. The set of clients of the service is disjoint from N. Clients perform *read* and *write* operations on the variables stored in the quorum system. We assume that these operations return only when they complete (i.e. we consider confirmable operations [16]).

Our dynamic quorum protocols maintain the same assumptions about client failures of their static counterparts. Clients communicate with servers over point-to-point, asynchronous fair channels. A fair channel guarantees that a message sent an infinite number of times will reach its destination an infinite number of times. We allow channels to drop, reorder, and duplicate messages.

4. A new basis for determining correctness

The first step in our transition to dynamic quorum protocols is to establish the correctness of the static protocols we consider (shown in Figure 3) on a basis that does not rely on quorum intersection. To do so, we observe that at the heart of all these protocols lies the Q-RPC primitive [13]. This primitive takes a message as argument, sends that message to a quorum of responsive servers, and returns the response from each server in the quorum. Our approach to extend quorum protocols to the case where servers are added and removed (and thus quorums may not intersect anymore) is to define correctness in terms of the properties of the data returned by quorum-based operations such as Q-RPC. In this section, we first specify two properties that apply to the data returned by Q-RPC; then, we prove that these properties are sufficient to ensure correctness. In Section 5 we will show that it is possible to implement Q-RPC-like operations that guarantee these properties even when quorums do not intersect.

4.1. The transquorum properties

In the protocols listed in Figure 3, quorum-based operations such as Q-RPC are the fundamental primitives on top of which read and write operations are built. Not all Q-RPCs are created equal, however. Some Q-RPC operations change the state of the servers (e.g. when the message passed as an argument contains information that the servers should store), others do not. Some Q-RPCs need to return the latest data actually written in the system, others are content with returning data that is not obsolete, whether it was written or not. To capture this diversity, we introduce two properties, *timeliness* and *soundness*. We

```
READ
  1. $Q$ := Q-RPC("READ")
     // $Q$ is a set of $\langle ts, writer\_id, data\rangle_{writer}$
  2. reply $r$ := $\phi(Q)$ // returns largest valid value
  3. $Q$ := Q-RPC("WRITE",$r$)
  4. return $r.data$
WRITE($D$)
  1. $Q$ := Q-RPC("GET_TS")
  2. $ts := max\{Q.ts\} + 1$
  3. $m := \langle ts, writer\_id, D\rangle_{writer}$
  4. $Q$ := Q-RPC("WRITE",$m$)
```

```
READ
  1. $Q$ := TRANS-Q$_{\mathcal{R}}$("READ")
     // $Q$ is a set of $\langle ts, writer\_id, data\rangle_{writer}$
  2. reply $r$ := $\phi(Q)$ // returns largest valid value
  3. $Q$ := TRANS-Q$_{\mathcal{W}}$("WRITE",$r$)
  4. return $r.data$
WRITE($D$)
  1. $Q$ := TRANS-Q$_{\mathcal{T}}$("GET_TS")
  2. $ts := max\{Q.ts\} + 1$
  3. $m := \langle ts, writer\_id, D\rangle_{writer}$
  4. $Q$ := TRANS-Q$_{\mathcal{W}}$("WRITE",$m$)
```

Figure 2: U-dissemination protocol (fail-stop clients). On the left: Q-RPC. On the right: TRANS-Q.

call them *transquorum* properties because, as we will see in Section 5, they do not require quorum intersection to hold. Intuitively, timeliness says that any read value must be as recent as the last written value, while soundness says that any read value must have been written before. Note that not all Q-RPCs need to be both timely and sound. For example, Q-RPCs used to gather the current timestamps associated with the value stored by a quorum of servers do not need to be sound—all that is required is that the returned timestamps be no smaller than the timestamp of the last write.

We then define three sets $\mathcal{W}$, $\mathcal{R}$, and $\mathcal{T}$ of Q-RPC-like quorum operations. Each Q-RPC-like operation in a protocol belongs to zero or more of these sets.

Let $w \rightarrow r$ (w "happens before" r) indicate that the quorum operation w ended (returned) before the quorum operation r started (in real time). Further, let o be an ordering function that maps each quorum operation to an element of an ordered set $\mathcal{M}$. We define the transquorum properties as follows:

$$\text{(timeliness)} \quad \forall w \in \mathcal{W}, \forall r \in \mathcal{T}, o(r) \neq \bot : \quad w \rightarrow r \implies o(w) \leq o(r)$$

$$\text{(soundness)} \quad \forall r \in \mathcal{R}, o(r) \neq \bot : \quad \exists w \in \mathcal{W} \text{ s.t. } r \not\rightarrow w \wedge o(w) = o(r)$$

In this paper we always choose o so that when applied to a Q-RPC-like operation x, it returns both a timestamp and the data that is associated with x (i.e. either read or written). This allows us to use the timeliness property to ensure that readers get recent timestamps and the soundness property to ensure that reads get data that has been written.

4.2. Proving correctness with transquorums

Transquorum properties are all that is needed to prove that the protocols listed in Figure 3 correctly provide the consistency semantics that they advertise. We present the complete set of proofs in the appendix. For conciseness, in the main body we limit ourselves to the first three protocols in the figure. All three protocols have the same client code, shown on the left in Figure 2 and all three guarantee atomic semantics. The server code is also identical: servers simply store the highest timesetamped data they see and send back to the client the data or its timestamp (in reply to READ or GET_TS requests, respectively). The protocols differ in the size of the quorums they use and in the degree of fault tolerance they provide: U-dissemination protocols [16] (a variant for fair channels of the dissemination protocol presented in [13]) can tolerate b Byzantine faulty servers, crash can tolerate f fail-stop faulty servers, and hybrid-d can tolerate both b Byzantine failures and f fail-stop failures ($f + b$ failures in total). To simplify our discussion, since the three client protocols are identical we will only discuss the U-dissemination protocol here; all we say also applies to the crash and hybrid-d protocols, except that the crash protocol does not use any signatures. Another simplification is that we show the transformation on the non-optimized version of the U-dissemination protocol. The technical report [15] shows how to shorten reads to a single message round-trip in the common case by skipping the write-back when it is not necessary.

4.2.1. Dissemination protocols with transquorums

To illustrate that we only rely on the transquorum properties and not on the specific implementation of Q-RPC, we replace all Q-RPC calls in the protocol (Figure 2) with an "abstract" function TRANS-Q that we postulate has the transquorum properties. TRANS-Q takes the same arguments and returns the same values as Q-RPC.

The U-dissemination protocol on the right of Figure 2 uses TRANS-Q as its low-level quorum communication primitive. We have annotated each call to indicate which set it belongs to ($\mathcal{R}, \mathcal{W}$, or $\mathcal{T}$).

We use the notation $\langle a\rangle_b$ to show that a is signed by b. Note that data is signed before being written, and verified before being read. The function $\phi(Q)$ returns the largest value in the set Q that has a valid signature using lexicographical ordering: since our values are triplets $(ts, writer\_id, D)$, ϕ selects the largest valid timestamp,

Operations of this form	are assigned this order	and this set
$r = $ TRANS-Q(“$READ$”)	$o(r) = \phi(r_{ret})$	$\mathcal{R}$
$w = $ TRANS-Q(“$WRITE$”$, ts, writer\_id, D$)	$o(w) = (w_{arg}.ts, w_{arg}.writer\_id, w_{arg}.D)$	$\mathcal{W}$
$t = $ TRANS-Q(“GET_TS”)	$o(t) = (max(t_{ret}) + 1, \perp, \perp)$[5]	$\mathcal{T}$

Figure 3: The o mapping

using *writer_id* and then *D* to break ties.

We assign each TRANS-Q quorum operation to one of the sets ($\mathcal{R}, \mathcal{W}$ or $\mathcal{T}$) and define the ordering $o(x)$ for each quorum operation x. Our assignment is shown in Figure 3. The assignment is fairly intuitive: operations that change the server state have been assigned to the $\mathcal{W}$ set and the ordering function consists either of what is being written, or of what the caller extracts from the set of responses to its query. More precisely, to define $o(x)$ we observe that any quorum operation x has two parts: the arguments passed to x and the value that x returns. We use the notation x_{arg} to refer to the arguments that were passed to the x operation, and x_{ret} to indicate the value returned by x (that value is always a set).

We want to show that the U-dissemination protocol with TRANS-Q operations offers atomic semantics. Informally, atomic semantics requires all readers to see the same ordering of the writes, and furthermore that this order be consistent with the order in which writes were made. Note that atomic semantics is concerned with *user-level* (or, simply, *user*) reads and writes, not to be confused with the *quorum-level operations* (or, simply, *quorum operations*) such as Q-RPC and TRANS-Q. We use lowercase letters to denote quorum-level operations, and capital letters to denote user-level operations (e.g. R or W). Similarly, we use the mapping o to denote the ordering constraint that the transquorum properties impose on quorum operations, and the mapping O to denote the ordering constraints imposed by the definition of atomic semantics on user read and write operations.

Atomic semantics can be defined precisely as follows.

Definition 1. *Every user read R returns the value that was written by the last user write W preceding R in the ordering “$<$”. “$<$” is a total order on user writes, and $W \to X \Longrightarrow W < X$ and $X \to W \Longrightarrow X < W$ for any user write W and user read or user write X.*

We use O, which maps every user read and write operation to an element of some ordered set $\mathcal{M}'$, to define completely the ordering relation“$<$”: $X < X' \iff O(X) < O(X')$.

We are now ready to prove our first theorem, showing that we can replace Q-RPC with any operation that satisfies the transquorum properties without compromising the semantics of the U-dissemination protocol. The proof is structured around the following three lemmas, which we prove in Appendix A.1.

Lemma 1. *Our ordering relation “$<$” is a total order on user writes; further, $W \to X \Longrightarrow W < X$ and $X \to W \Longrightarrow X < W$ for any user write W and user read or user write X.*

Lemma 2. *All user reads R return the value that was written by the last user write W preceding R in the “$<$” ordering.*

Combining the two lemmas gives our first theorem:

Theorem 1. *The U-dissemination protocol provides atomic semantics if (i) the TRANS-Q operations have the transquorums properties for the function o defined in Figure 3, and (ii) for all $r \in \mathcal{R} : o(r) \neq \perp$.*

5. Dynamic quorums

The transquorum properties allows us to reason about quorum protocols without being forced to use quorums that physically intersect. In this section, we leverage this result to build DQ-RPC, a quorum-level operation that satisfies the transquorum properties but also allows both the set of servers and the resilience threshold to be adjusted.

We must first introduce some way to describe how our system evolves over time, as N and f change.

5.1. Introducing views

We use the well-established term *view* to denote the set N that defines the quorum system at each point in time. Each view is characterized by a set of attributes, the most important of which are the view number t, the set of servers $N(t)$ and the resilience threshold $f(t)$. In general, view attributes include enough information to compute the quorum size $q(t)$. The responsibility to steer the system from view to view is left with an administrator, who can begin a view change by invoking the `newView` command.

When the administrator calls `newView`, the view information stored at the servers is updated. We say that a view t *starts* when a server receives a view change message for view t (for example because the administrator called `newView(`$t, \ldots$`)`). A view t *ends* when a quorum

5 We do not explicitly require this value to be larger than any timestamp previously sent by this client because we do not allow clients to issue multiple concurrent writes.

$q(t)$ of servers have processed a message indicating that some later view u is starting. After starting and before ending, the view is *active*. A view may start before the previous view ended, i.e. there may exist multiple active views at the same time; our protocol makes sure that the protocol semantics (e.g. atomic) is maintained despite view changes, even if client operations happen concurrently to them.

The `newView` function has the property that after `newView`(t) returns, all views older than t have ended and view t has started. At this point the administrator can safely turn off server machines that are not in view t.

Obviously, we must restrict who can call the `newView` command. In our system, this is solely the privilege of the administrator. If the administrator is malicious then we cannot provide any guarantee (for example, it could start a view containing no server to deny service to all clients). However, the system can tolerate crash failures of the administrator. This problem remains even if the administrator algorithm is run in a Byzantine fault tolerant manner, as long as that program takes its inputs from a person: the machine through which these inputs are transmitted must not have been tampered with. Since the determination of future values of f and the decision of adding computers to the system (possibly purchasing new ones as necessary) is best done by a person, we consider a single crash-only administrator machine for the remainder of this paper.

Since our system uses views to discretize time, so does our definition of faults. We say that a server is *correct* in some view t if it follows the protocol from the beginning of time until view t ends. Otherwise, it is faulty in view t. Note that a server may be correct in some view t and faulty in a later view u. However, faulty servers will never be considered correct again. If some server recovers from a failure (for example by reinstalling the operating system after a disk corruption), it takes on a new name before joining the system. The notion of resilience threshold is also parameterized using view numbers. For example, a static U-dissemination protocol requires a minimum of $n \geq 3f + 1$ servers: this requirement now becomes $|N(t)| \geq 3f(t) + 1$ for each view t. Our system assumes that between the start and the end of view t, at most $f(t)$ of the servers in $N(t)$ are faulty. Since views can overlap this means that sometimes a conjunction of such conditions must hold at the same time.

5.2. A simplified DQ-RPC

We begin with a simplified version of DQ-RPC that, while suffering from serious limitations, allows us to present more easily several of the key features of DQ-RPC—the full implementation of DQ-RPC is presented in Section 5.3.

The easiest way to implement DQ-RPC is to ensure that different views never overlap, i.e. that at any point in time there exists at most one active view. Since we know that the protocols in Figure 3 are correct for a static quorum system, we can simply make sure to evolve the system through, as it were, a sequence of static quorum systems. We can do so as follows.

- Replies from servers are tagged with a view number
- Once a client accumulates $q(t)$ responses tagged with view t, the DQ-RPC returns these responses.

Our simplified DQ-RPC has two outputs: a view t (that we call DQ-RPC's *current* view) and a quorum of $q(t)$ responses. Pseudocode for the simplified DQ-RPC is shown in Figure 4. The function $q(n, f)$ computes the quorum size based on the number of servers n and the resilience threshold f. The $activeServers()$ function gives the list of servers in active views (views that have started but not ended – there may be more than one). The variable $replies$ keeps track of all replies from active servers. Simplified DQ-RPC loops until it gets $q(|N(t)|, f(t))$ messages tagged with the same view t (vouched for by message mt). We write $m.tag$ for the view meta-information tagged onto message m. These tags contain three fields: the set of servers N, the resilience threshold f and the view number t. If we assume that clients have some external, infallible way to know which servers are in an active view (the `activeServers` function) then the above simple scheme is sufficient: DQ-RPC sends its messages to servers in an active view and it makes sure that it only picks active views as its current view[6].

Showing how DQ-RPC can determine which views are active is the subject of the rest of this section.

5.2.1. View changes To determine whether a view is active, it is important to specify how the system starts (and ends) views.

To initiate a view change, the administrator's computer first tells a quorum of machines on the old view that their view has ended. These machines immediately stop accepting client requests. Clients can thus no longer read from the old view since they will not be able to gather a quorum of responses. The administrator then performs a user-level read on the machines from the old view to obtain some value v. Finally, the administrator tells all the machines in the new view that the new view is starting, and provides them with the initial value v. At this point, the machines in the new view start accepting client requests.

Naturally, it is not always possible for the administrator to make sure it has contacted all the new machines: if some server is faulty then it could choose not to acknowledge, causing the administrator to block forever. In our simplified DQ-RPC we remove this problem by simply assuming that the administrator has some way to contact all the

6 It is necessary to pick an active view: after some DQ-RPC writes data to the latest view, reads to a view that has ended would return old data since different views may have no servers in common.

```
Simplified-DQ-RPC($D$)

1. repeat
2.         send $D$ to activeServers()
3.         gather responses in $replies$
4.         $replies$ := $\{r \in replies : r.sender \in$ activeServers$()\}$
5. until $\exists t, mt : mt \in replies \wedge mt.tag.t = t$
      $\wedge |\{r \in replies : r.tag.t = t\}| \geq q(|mt.tag.N|, mt.tag.f)$
6. return $\{r \in replies : r.tag.t = t\}$ // $t$ is the current view associated with this operation
```

Figure 4: Simplified Dynamic quorum RPC

servers. We will see in Section 5.3 how the full DQ-RPC ensures that all view changes terminate.

A delicate point to consider when performing a view change is that, after view t ends, so does the constraint that at most $f(t)$ of the machines in view t can be faulty. For example, if the view was changed to remove some decommissioned servers, it is natural to expect that the semantics of the system from then on does not depend on the behavior of the decommissioned servers.

And yet, the decommissioned machines know something about the previous state of the system. If they all became faulty (as it may happen, since they are no longer under the administrator's watchful eye) they would be able to respond to queries from clients that are not yet aware of the new servers and fool them into accepting stale data, violating atomic semantics. This situation is depicted in Figure 5. To prevent the system from depending on servers that have been decommissioned, the view change protocol must ensure that no client can read or write to a view after that view has ended. Our *forgetting* protocol enforces this property.

Safe View Certification through "Forgetting" The simplified DQ-RPC requires the client to receive a quorum of responses with view t's tag before it returns that value and considers view t current. If the servers are correct, then this ensures that no DQ-RPC chooses t as current after t ends (recall that views end once a quorum of their servers have left the view).

The forgetting protocol ensures that this property holds despite Byzantine failure of the servers. Clients tag their queries with a nonce e. Server i tags its response with two pieces of information: 1) server i's view certificate $\langle i, meta, pub\rangle_{admin}$, signed by the administrator, and 2) a signature for the nonce $\langle e\rangle_{priv}$, proving that server i posses the private key associated with the public key in the view certificate. The key pair $pub, priv$ is picked by the administrator. In the certificate, $meta$ contains the meta information for the view, namely the view number t, the set of servers N and the resilience threshold f. The quorum size q can be computed from these parameters.

When servers leave view t, they discard the view certificate and private key that they associated with that view. The challenge is to ensure that even if they become faulty later, they cannot recover that private key and thus cannot vouch for a view that they left. We now discuss how our protocol addresses this issue.

The private key is only transmitted when the administrator informs the server of the new view. Our network model allows the channel to duplicate and delay this message, which may therefore be received after the server has left the view. To prevent the decommissioned server from recovering the private key we encrypt the message using a secret key that changes for every view.

The administrator's view change message for view t to server i contains the following:

$$(\text{NEW\_VIEW}, t, oldN,$$
$$encrypt\ ((\langle i, meta, pub\rangle_{admin}, priv), k_i^t))$$

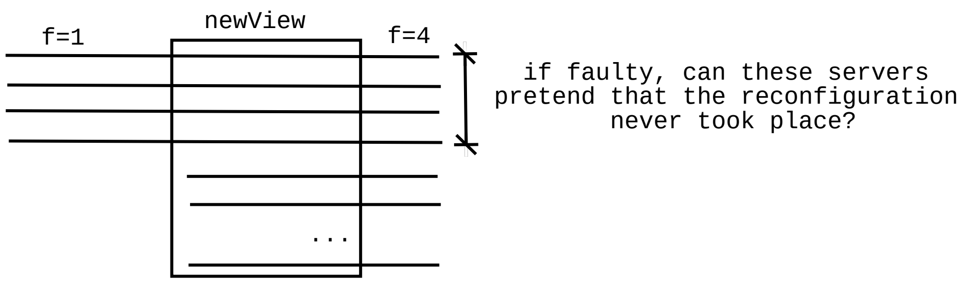
We use the notation $encrypt(x, k)$ for the result of encrypting data x using the secret key k. The view key k_i^t is shared by the administrator and server i for view t. It is computed from the previous view's key using a one-way hash function: $k_i^t := h(k_i^{t-1})$. The administrator and server i are given k_i^0 at system initialization.

When correct servers leave a view t, they discard view t's certificate, private key $priv$ and view key k_i^t. As a result they will be unable to vouch for view t later even if they become faulty and gather information from duplicated network messages. This ensures that client following the simplified DQ-RPC protocol will not pick view t as its current view after t ends.

5.2.2. Finding the current view In the previous section we have seen how clients can identify old views. We now need to make sure that the clients will be able to find the current view, too.

If the set of servers that the client contacts to perform its DQ-RPC intersects with the current view in one correct server i, then the client will receive up to date view information from i and will be able to find the current view.

If that is not the case, then the client can consult well-known sites to which the administrator publishes the list of the servers in the current view. Our certified tags ensure safety: even if the information the client retrieves from one of these sites is obsolete, the client will never pick as current a view that has ended. Therefore it suffices that the



newView
f=1
f=4
if faulty, can these servers pretend that the reconfiguration never took place?
...

Figure 5: Example of view change

client eventually learn of an active view from one of the well-known sites.

In the case of a local network, clients could also broadcast a query to find the servers currently in N. This solution has the advantage of simplicity but it only works if all servers are in the same subnet.

5.2.3. Summary Clients only accept responses if they all have valid tags for the same view. Until they accept a response, clients keep re-sending their request (for read or write) to the servers. Clients use the information in the tags to locate the most recent servers, and periodically check well-known servers if the servers do not respond or do not have valid tags. Tags are valid if their view certificate has a valid signature from the administrator and the tag includes a signature of the client-supplied nonce that matches the public key in the certificate.

Replacing Q-RPC with this simplified DQ-RPC in a dissemination quorum protocol from Figure 3 results in a dynamic protocol that maintains all the properties listed in the figure.

However, simplified DQ-RPC has two significant limitations. First, it requires the administrator's `newView` command to wait for a reply from all the servers in the new view, which may never happen if some servers in the new view are faulty. Second, it does not let DQ-RPCs (and, implicitly, user-level read and write operations issued by clients) complete during a view change: instead the operations are delayed until the view change has completed. We address both limitations in the next section.

5.3. The full DQ-RPC for dissemination quorums

The full DQ-RPC for dissemination quorums follows the same pattern as its simplified version: it sends the message repeatedly until it gets a consistent set of answers, and picks a current view in addition to returning the quorum of responses. DQ-RPC uses the technique described in the previous section to determine whom to send to, but it can decide on a response sooner than the simplified DQ-RPC because it can identify consistent answers without requiring all the responses to be tagged with the same view. The full DQ-RPC also runs a different view change protocol that terminates despite faulty servers.

```
DQ-RPC($msg$)
1. Sender $sdr$ := new Sender($msg$)
2. static ViewTracker $g\_vt$ := new ViewTracker
3. repeat
4.   sender.sendTo($g\_vt$.get().N)
5.   $(Q, t)$ := $g\_vt$.consistentQuorum($sdr$.getReplies())
6.   if running for too long then $g\_vt$.consult()
7. until $Q \neq \emptyset$
   // $t$ is the current view associated with this operation
8. return $Q$          // sender stops sending at this point
```

Figure 6: Dynamic quorum RPC

We split the implementation of DQ-RPC into three parts. The main DQ-RPC body (Figure 6) takes a message and sends it repeatedly to the servers believed to constitute the current view. The client's current view changes with the responses that it gets; if no responses are received for a while then DQ-RPC consults well-known sources for a list of possible servers (line 6). The repetitive sending is handled by the Sender object, and the determination of the current view is done by the ViewTracker object (Figure 9). The client exits when it receives a quorum of consistent answers. In the simplified protocol, answers were consistent if they all had the same tag. In this section we develop a more efficient notion of consistent responses.

The Sender is given a message and a destination and it repeatedly sends the message to the destination. The destination can be changed using the `sendTo` method and the replies are accessed through `getReplies` (The code for the Sender object can be found in [15]). The code for the Sender object is shown in Figure 7.

The ViewTracker acts like a filter: Sender must go through it to read messages. The ViewTracker looks at the messages and keeps track of the most recent view certificate it sees. As we saw in the forgetting protocol, messages are tagged with a signed view certificate and a signed nonce. Messages that do not have a correct signature for the nonce are not considered as vouching for the view (line 3 of ViewTracker.`consistentQuorum`). However, even if the nonce signature is invalid, ViewTracker will use valid view certificates to learn which servers are part of the latest view (line 5). The most recent view certificate can be accessed through the `get` method. The ViewTracker can also get new candidates from well-known servers with the `consult` method. Finally, the ViewTracker has the responsibility of deciding when a set of answers is consistent, through the `consistentQuorum` method.

5.3.1. Introducing generations Our dynamic protocols only require the minimal number of servers [16] to tolerate f faults: $3f + 1$. The price for this minimal replication is that every time new servers are added, the data must be copied to them.

When more machines are available, it is possible to use the additional replicas to speed up view changes. We offer this capability through the new *spread* parameter. When the spread parameter m is non-zero, quorum operations involve more servers than strictly necessary. This margin allows the quorums to still intersect when a few new servers are added, allowing these view changes to proceed quickly. As a result, there are now two different kinds of view changes: one in which data must be copied and one in which no copy is necessary. In the second case we say that the old and new views belong to the same *generation*. Each view is tagged with a generation number g that is incremented at each generation change.

These two parameters, m and g, are stored in the view meta-data alongside with N, f and t.

The additional servers do not necessarily need to be used to speed up view changes. Using a smaller m with a given n makes the quorums smaller and reduces the load on the system. The parameter m therefore allows the administrator to trade-off low load and quick view changes.

Intra-Generation: When Quorums Still Intersect When clients write using the DQ-RPC operation, their message is received by a quorum of responsive servers. The size of the quorum depends on the parameters of the current view t (recall that t is also determined in the course of a DQ-RPC). The quorum size depends on the failure assumptions made by the protocol. For a U-dissemination Byzantine protocol that tolerates b faulty servers, the quorum size is $q(n, b, m) = \lceil (n + b + 1)/2 + m/4 \rceil$.

In the absence of view changes, our quorums intersect in $b+1+m/2$ servers. If m new (blank) servers are added to the system, then our quorums intersect in $b + 1$ servers, which is still sufficient for correctness: one of the servers is correct and the reader will recognize the signature on the correct data. Thus, up to m servers can be added to the system before data must be copied to any of the new servers.

Similarly, if m of the servers that were part of a write quorum are removed, new quorums will still intersect in $b+1$ servers and the system will behave correctly. Finally, if b is increased or reduced by up to m (causing the quorums to grow or shrink accordingly), new quorums will still intersect the old ones in $b + 1$ servers.

More generally, if after a write a servers are added, d servers are removed, b is modified by c, and m is reduced to m_{min} then the quorums will still intersect sufficiently as long as $a + d + c \leq m_{min}$. If a view change would break this inequality then the value must be copied to some of the new servers before the view change completes: we say that the old and new views are in different generations.

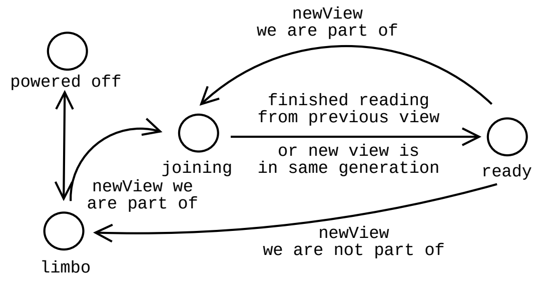


Figure 8: Server transitions for the dissemination protocol

5.3.2. View changes: closing the generation gap The copying of data across generations is done as part of the view change protocol. Unlike the view change protocol that is associated with simplified DQ-RPC, the full view change protocol terminates.

View changes are initiated by the administrator when some machines need to be added, removed or moved, or when the resilience f or the spread m have to be changed. The `newView` method first determines whether the new view will be in the same generation as the previous one, using the relation in Section 5.3.1. It then computes the key pairs and certificates for the new view. Finally the administrator encodes the certificates using the appropriate shared key and sends them to all servers in t, re-sending when appropriate and waiting for a quorum of responses.

Servers switch states according to the diagram in Figure 8. When they receive a new view message for a new generation (and they are part of that generation), servers piggyback that message on top of a read they perform on a quorum from the old view. They then update their value with what they read (if it is newer than the value they currently store) and update their view certificate. If they are part of the new view but there is no generation change then the servers just update their view information as per the forgetting protocol. If they are not part of the new view then the servers update their certificates too. In that case they will not be able to vouch for the new view since they have no valid view certificate for it, but they will still be able to direct clients to the current servers.

Servers are in the *limbo* state initially and after leaving the view. They are in the *joining* state while they copy information from the older view, and they are in the *ready* state otherwise. Servers process client requests in all three

Sender variables

$m\_message$	the message that is to be transmitted
$m\_destinations$	the set of destination addresses
m_thread	a resend thread (initially not running)
$m\_replies$	set of (sender, reply, meta) triples
$m\_recentReply$	set of senders who sent a reply in the most recent view
m_delay	delay until the next retransmission (initially 1 second)

```
constructor Sender($msg$)
1. $m\_message$ := $msg$

Sender.sendTo($Dests$)
1. $m\_destinations$ := $Dests$
2. if $m\_thread$ is not running then
3.     create $m\_thread$        // the worker thread $m\_thread$ will call run()

Sender.run()
1. $m\_message.nonce$ = a new nonce
2. while (true)
3.     $m\_recentReply$ = senders in $m\_replies$ with meta that has the same view number as $g\_vt.get()$
4.     asynchronously send $m\_message$ to each element of $m\_destinations$ - $m\_recentReply$
5.     wait $m\_delay$ seconds
6.     $m\_delay$ := $m\_delay$ * 2
7.     while (($j$,$r$,$s$):=g_vt.receive($m\_message.nonce$))      // received valid reply ($r$,$s$) from server $j$
8.         if $j \in m\_destinations$ then
9.             $m\_replies := m\_replies \cup (j, r, s)$

Sender.getReplies()
1. return $m\_replies$
```

Figure 7: Sender class for dynamic quorum RPC

states. Servers in the *joining* state use the view certificate for the old view (if they have it) until they are *ready*.

The administrator's `newView` waits for a quorum of new servers to acknowledge the view change and then it posts the new view to the well-known locations and returns. At this point, the administrator knows that the data stored in the machines that were removed from the view are not needed anymore and therefore the old machines can be powered off safely.

There may still be some machines in the *joining* stage at this point. These machines do not prevent operations from completing because DQ-RPC operations only need $f + 1$ servers in the new generation to complete, and any dissemination quorum contains at least $f + 1$ correct servers.

When `newView` returns, the old view has ended and the new view has started and *matured*, meaning that at least one correct server is done processing the view change message for it. This means that reads and writes to the new view will succeed and reads and writes to the old view will be redirected to the new view (either by the old servers or after consultation of the well-known locations).

The protocol as presented here requires the administrator to be correct. If the administrator crashes after sending the new view message to a single faulty new server, the new server can cause the servers in the old view to join the limbo state without informing the new servers that they are supposed to start serving. In Appendix B we show a variant that tolerates crashes in the sense that if the administrator machine crashes at any point during the view change and never recovers then read and write operations will still succeed even though it is not possible to change views anymore. The basic idea is that the servers in the old view make sure that a quorum of servers in the new view is informed of the view change before they let the old view end.

5.3.3. DQ-RPC satisfies transquorums for dissemination quorums

We now prove our final theorem:

Theorem 2. *U-dissemination, crash and hybrid-d based on DQ-RPC provide atomic semantics.*

The complete proof is in Appendix B.1. We present the main lemmas below.

```
($meta$) **ViewTracker.get()**
  // returns the latest view meta-data
  1. return $m\_maxMeta$

**ViewTracker.consult**
  // ask well-known servers for the latest meta-data
  1. Choose a server $j$ at random from the list of well-known view publishers
  2. Send $(\text{CONSULT}, m\_maxMeta)$ to $j$

($sender, reply, meta$) **ViewTracker.receive**($nonce$)
  // used by the Sender object when gathering replies
  1. if there is no message waiting, then return false
  2. receive $(msg, meta)$ from $sender$
  3. if not validCertificate($meta$) then return false
  4. if $meta.t > m\_maxMeta.t$ then
  5.         $m\_maxMeta := meta$
  6. if $msg ==$ CONSULT-ACK then goto 1
  7. return $(sender, msg, meta)$

($messages, view$) **ViewTracker.consistentQuorum**($messageTriples$)
  // returns a consistent quorum of messages (if any) and the current view
  1. $msgInQuorun := \{m \in messageTriples : m.sender \in m\_maxMeta.N\}$
  2. if $|msgInQuorun| < q(|m\_maxMeta.N|, m\_maxMeta.f, m\_maxMeta.m)$ then return $(\emptyset, \perp)$
       // fail if there is no consistent quorum of messages
  3. $validMessages := \{m \in msgInQuorun :$ validTag$(m)\}$
  4. $recentMessages := \{m \in validMessages : m.meta.g == m\_maxMeta.g\}$
  5. if $|recentMessages| < m\_maxMeta.f + 1$ then return $(\emptyset, \perp)$    // fail if the view is not mature
  6. return $(msgInQuorun, m\_maxMeta)$

**ViewTracker.consult**    // consults well-known servers for the latest meta-data
  1. Choose a server $j$ at random from the list of well-known view publishers
  2. Send $(\text{CONSULT}, m\_maxMeta)$ to $j$
```

Figure 9: Definition of the ViewTracker object

Lemma 3. *The view t chosen by a DQ-RPC operation is concurrent with the DQ-RPC operation.*

Lemma 4. *The DQ-RPC protocol in Figure 6 provides the transquorum properties for the ordering function o of Figure 3.*

Lemma 5. *When using DQ-RPC for the U-dissemination, crash or hybrid-d protocol, no $\mathcal{R}$ operation returns $\perp$.*

5.4. Optimizations

Our protocol can be optimized in several ways. For space reasons, we defer the detailed explanation of the optimizations to Section B.2 in the appendix and only briefly go over each optimization here.

The first group of optimizations makes both the dissemination and the masking protocols faster. Reads can proceed in a single round-trip in the common case where no write is concurrent with the read by skipping the writeback in this case. Both the DQ-RPC operation and `newView` can be improved for speed, at the cost of a little complexity. New servers can pre-fetch the data immediately when they are added to the system instead of waiting until the next generation change, allowing generation changes to complete faster.

The second group of optimizations concerns the view meta-data that is exchanged between the servers and the clients. This communication can be reduced by omitting the view information in some cases, using asymmetric quorums for faster reads, and sending differences instead of the whole data.

Third, we can improve resilience. The administrator program is less likely to crash if it runs on a replicated state machine instead of a single computer. Also, our system adapts well to proactive recovery since clients can use the system even when servers are constantly added and removed in the background.

We have successfully applied the methodology presented in this paper to masking quorum system storing generic data. The DQ-RPC protocol remains the same in that case but the `newView` operation needs to be adapted: the details can be found in Appendix C.

6. Conclusions

We present a methodology that easily transforms several existing Byzantine protocols for static quorum systems [9, 13, 14, 17, 18] into corresponding protocols that operate correctly when the administrator is allowed to add or remove servers from the quorum system, as well as to change its resilience threshold. Performing the transformation does not require extensive changes to the protocols: all that is required is to replace calls to the Q-RPC primitive used in static protocols with calls to DQ-RPC, a new primitive that in the static case behaves like Q-RPC but can handle operations across quorums that may not intersect while still guaranteeing consistency. Our methodology is based on a novel approach for proving the correctness of Byzantine quorum protocols: through our

```
mainLoop()
  1. receive $message$ from machine $i$
  2. $response$ := result of calling the non-private function with the same name as the first element of
     $message$, if any ($\perp$ otherwise).
  3. reply to $i$ with $(m\_viewCert, \langle message.nonce\rangle_{m\_priv}, response)$

write($ts$,$D$)
  1. if ($m\_ts$<$ts$) then ($m\_ts$,$m\_D$) := ($ts$,$D$)
  2. return "WRITE-ACK"

read()
  1. return $(m\_ts, m\_D)$

newView($t$, $oldN$, $encryptedBody$)
  // encryptedBody is of the form $encrypt\ ((\langle i, meta, pub\rangle_{admin}, priv), k_i^t)$
  1. $newK$ := $h^{t-m\_meta.t}(m\_k)$
  2. $(cert, priv) := decrypt(encryptedBody, newK)$
  3. if ($cert.meta.N$ does not include this server) then
  4.         // limbo
  5.         $(m\_cert, m\_priv, m\_k) := (cert, priv, newK)$
  6.         return "OK"
  7. if ($cert.meta.g$ == $m\_cert.meta.g$) then
  8.         // intra-generation view change (ready state)
  9.         $(m\_cert, m\_priv, m\_k) := (cert, priv, newK)$
 10.         return "OK"
 11. // inter-generation view change (joining state)
 12. $(newTS, newD)$ := $\phi$(Q-RPC("READ+NEWVIEW", $cert$)) to the servers in $oldN$
 13. if $m\_ts < newTS$ then ($m\_ts$,$m\_D$) := ($ts$,$D$)
 14. $(m\_cert, m\_priv, m\_k) := (cert, priv, newK)$ // ready state now
 15. return "OK"

read+NewView($cert$)
  1. if ($m\_cert.meta.t < cert.meta.t \wedge cert$ has a valid signature) then
  2.         $(m\_cert, m\_priv) := (cert, \perp)$
  3. return $(m\_ts, m\_D)$
```

Figure 10: Server protocol for dissemination quorums

transquorum properties, we specify the characteristics of quorum-level primitives (such as Q-RPC) that are crucial to the correctness of Byzantine quorum protocols and proceed to show that it is possible to design primitives, such as DQ-RPC, that implement these properties even when quorums don't intersect. We hope that designers of new quorum protocols will be able to leverage this insight to easily make their own protocols dynamic.

7. Acknowledgments

The authors would like to thank Eunjin Jung and Jeff Napper for several interesting conversations and feedback on the paper presentation.

References

[1] I. Abraham and D. Malkhi. Probabilistic quorums for dynamic systems. In *Proc. 17th Intl. Symp. on Distributed Computing (DISC)*, Oct. 2003.

[2] L. Alvisi, D. Malkhi, E. Pierce, M. Reiter, and R. Wright. Dynamic Byzantine quorum systems. In *Proc. of the Intl. Conference on Dependable Systems and Networks (DSN)*, June 2000.

[3] L. Alvisi, D. Malkhi, E. Pierce, and M. K. Reiter. Fault detection for byzantine quorum systems. *IEEE Trans. Parallel Distrib. Syst.*, 12(9):996–1007, 2001.

[4] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):427–469, 2001.

[5] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys (CSUR) Volume 17, Issue 3*, pages 341–370, Sept. 1985.

[6] S. Dolev, S. Gilbert, N. Lynch, A. Shvartsman, and J. Welch. Geoquorums: Implementing atomic memory in mobile ad hoc networks. In *Proc. 17th Intl. Symp. on Distributed Computing (DISC)*, Oct. 2003.

[7] J.R. Douceur. The sybil attack. In *Proc. of the IPTPS02 Workshop*,

March 2002.

[8] S. Gilbert, N. Lynch, and A. Shvartsman. Rambo II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proc. 17th Intl. Symp. on Distributed Computing (DISC)*, pages 259–268, June 2003.

[9] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient consistency for erasure-coded data via versioning servers. Technical Report CMU-CS-03-127, Carnegie Mellon University, 2003.

[10] L. Kong, A. Subbiah, M. Ahamad, and D.M. Blough. A reconfigurable byzantine quorum approach for the agile store. In *Proc. 22nd Intl. Symp. on Reliable Distributed Systems (SRDS)*, Oct. 2003.

[11] L. Lamport. On interprocess communications. *Distributed Computing*, pages 77–101, 1986.

[12] N. Lynch and A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proc. 16th Intl. Symp. on Distributed Computing (DISC)*, pages 173–190, Oct. 2002.

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

[14] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Proc. 17th IEEE Symp. on Reliable Distributed Systems (SRDS)*, Oct 1998.

[15] J-P. Martin and L. Alvisi. A framework for dynamic byzantine storage. Technical Report TR04-08, The University of Texas at Austin, 2004.

[16] J-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proc. 16th Intl. Symp. on Distributed Computing (DISC)*, pages 311–325, Oct. 2002.

[17] J-P. Martin, L. Alvisi, and M. Dahlin. Small Byzantine quorum systems. In *Proc. of the Intl. Conference on Dependable Systems and Networks (DSN)*, pages 374–383, June 2002.

[18] E. Pierce and L. Alvisi. A framework for semantic reasoning about byzantine quorum systems. In *Brief Announcements, Proc. 20th Symp. on Principles of Distributed Computing (PODC)*, pages 317–319, Aug. 2001.

[19] R. Rodrigues, B. Liskov, and L. Shrira. The design of a robust peer-to-peer system. In *Tenth ACM SIGOPS European Workshop*, Sept. 2002.

[20] P. Thambidurai, YK. Park, and K. S. Trivedi. Interactive consistency with multiple failure modes. *9th Intl. Conference on Distributed Computing Systems*, June 1988.

[21] A. Venkataramani, R. Kokku, and M. Dahlin. TCP-Nice: A mechanism for background transfers. In *5th Symp. on Operating Systems Design and Implementation*, Dec 2003.

A. Dissemination Protocol

A.1. U-Dissemination provides atomic semantics

We show that transquorums suffice to prove that the U-dissemination protocol of Figure 2 is atomic.

Theorem 3. *The U-dissemination protocol provides atomic semantics if (i) the TRANS-Q operations have the transquorums properties for the function o defined in Figure 3, and (ii) for all $r \in \mathcal{R} : o(r) \neq \bot$.*

Lemma 6. *Our ordering relation "$<$" has the following properties: (i) $W \to X \implies W < X$ and (ii) $X \to W \implies X < W$ for any user write W and user read or user write X.*

Proof. Because we assume that user reads and writes are confirmable, $R \to W$ means that R returns before W starts.

We build the ordering O for user reads and writes from the ordering o of the quorum operations that are invoked within these user operations.

The simplest choice would be to set $O(X) = o(x)$ for some quorum operation x that is called as part of the user-level operation X. Unfortunately we need to take one extra step to make sure that user reads get ordered after the user write whose value they read.

Let $last\_o(X)$ be the value assigned through the o mapping to the last quorum operation in the (read or write) user-level operation X. For a user write operation W, we define $O(W)$ to be the pair $(last\_o(W), 0)$. For a user read operation R, we define $O(R)$ to be the pair $(last\_o(R), 1)$. The second element in these pairs ensures that user read operations are ordered after the user write whose value they return.

We now show that condition (i) holds. Suppose first that $W \to X$. W ends with a $\mathcal{W}$ operation and X starts with an quorum operation $t \in \mathcal{T}$, so, by timeliness, $last\_o(W) \leq o(t)$. If X is a user read then its second element is 1, so $O(W) < O(X)$. If X instead of a user write then $last\_o(X) > o(t)$ and therefore $O(W) < O(X)$ still holds.

To show (ii), suppose now that $X \to W$. The last quorum operation of X (regardless of whether it is a user read or write) is a $\mathcal{W}$ operation. The first quorum operation of W, t, is a $\mathcal{T}$ operation. By timeliness, $last\_o(X) \leq o(t)$. The write then fills in the last two elements of $o(t)$ with its $writer\_id$ and data D, resulting in a value that is strictly larger than $o(t)$. This value is then passed to the last quorum operation in W, ensuring that $last\_o(W) > o(t)$. Since $last\_o(X) \leq o(t)$ and $o(t) < last\_o(W)$, it follows that $O(X) < O(W)$. □

Lemma 7. *Our ordering relation "$<$" is a total order on user writes.*

Proof. By construction, any two reads W_1 and W_2 can be compared. To show that $<$ is a total order, we need to also show that $O(W_1) = O(W_2) \Longrightarrow W_1 = W_2$.

Assume $O(W_1) = O(W_2)$. W_1 and W_2 cannot be performed by different writers, because the o value includes the writer_id, which is different for each writer. And if W_1 and W_2 are performed by the same writer they cannot be distinct. If they were, because user writes are confirmable it would either follow that $W_1 \rightarrow W_2$ (and, by (1) $O(W_1) < O(W_2)$, or, symmetrically, that $W_2 \rightarrow W_1$ and $O(W_2) < O(W_1)$. $\square$

Lemma 8. *All user reads R return a value that was written by some user write W, and $R \not\rightarrow W$.*

Proof. The value that is returned by read R is the third element in the first quorum operation $r \in \mathcal{R}$ in R. By soundness, this value was passed to some quorum operation in $\mathcal{W}$. Both user-level write and user-level read operations call a $\mathcal{W}$ quorum operation, but we observe that the user-level read calls it with a value that it first gets from a $\mathcal{R}$ quorum operation. We can therefore use soundness repeatedly to show that there must have been some quorum operation $w \in \mathcal{W}$ inside a user-level write operation W such that $o(r) = o(w)$ and $r \not\rightarrow w$. This proves the first part of the lemma. Since r did not happen before w, it is also impossible that R happened before W (since the latter would imply the former). This proves the second part of the lemma. $\square$

Lemma 9. *All reads R return the value that was written by the last write W preceding R in the "$<$" ordering.*

Proof. Lemma 8 tells us that the value returned by R was written by some write W. By construction of O, we have $O(W) = (last\_o(W), 0) = (last\_o(R), 0) < (last\_o(R), 1) = O(R)$, so W precedes R in the "$<$" ordering. By definition, for any write W' that is ordered after W in $<$" we know that $O(W') > O(W)$. Since the pairs for $O(W)$ and $O(W'$ have the same second element, the first element in $O(W')$ must be larger than the first in $O(W)$. Hence, since $O(R)$ and $O(W)$ have the same first element, it follows that W' is ordered in "$<$" after R. Therefore W is the last write preceding R in the "$<$" ordering. $\square$

The four lemmas together prove our first theorem, showing that we can replace Q-RPC with any operation that satisfies the transquorum properties without compromising the semantics of the U-dissemination protocol.

A.2. Quorum Intersection implies Transquorums

We have shown that U-dissemination provides atomic semantics for any TRANS-Q operation that has the transquorums property. The proof also follows for the hybrid dissemination protocol [20] since it follows the same schema. In this section, we show that the traditional implementation of Q-RPC (using quorum intersection) satisfies the transquorums property.

Both the u-dissemination and the crash protocol are special cases of the hybrid dissemination protocol. All three use quorums of size $\lceil \frac{n+b+1}{2} \rceil$ and requiree at least $2f+3b+1$ servers to tolerate f crash failures and b Byzantine failures from the servers (a total of $f + b$ failures). Any number of clients may crash. In the case of the U-dissemination protocol, f is zero. In the case of the crash protocol, b is zero.

The client protocol is shown in Figure 2. Servers store the highest-timestamped value they have received that has a valid signature (except for the crash protocol in which signatures are not necessary).

There must be at least $2f + 3b + 1$ servers. Servers do not communicate with each other; clients use the Q-RPC operation to communicate with servers. The Q-RPC operation sends a given message to a responsive quorum of servers.

Any two quorums intersect in $2q - n = b+1$ servers. At least one of these servers, s, is not Byzantine faulty (and has not crashed).

We use the same ordering o as Section 4.2.1, namely $\mathcal{W}$ calls are ordered according to their arguments, and $\mathcal{R}$ and $\mathcal{T}$ calls are ordered according to their return value. No $\mathcal{R}$ quorum operation ever returns $\perp$, so we do not need to consider that case. We prove the timeliness and soundness conditions separately.

Lemma 10 (timeliness). *For the quorum size and ordering described above:* $\forall w \in \mathcal{W} \forall r \in \mathcal{T} : w \rightarrow r \Longrightarrow o(w) \leq o(r)$

Proof. The quorum to which the value was written with w intersects with the quorum from which r reads in one non-Byzantine server that has not crashed. That server will report the timestamp that was written in w; since the server is not Byzantine faulty that data has a valid signature. The ϕ function will therefore return a value that is at least as large as $o(w)$. The result of that function is equal to $o(r)$. $\square$

Lemma 11 (soundness). *For the quorum size and ordering described above:* $\forall r \in \mathcal{R} : \exists w \in \mathcal{W}$ *s.t.* $r \not\rightarrow w \wedge o(w) = o(r)$

Proof. Values selected through ϕ(Q-RPC$_{\mathcal{R}}$) have a valid signature (by definition of ϕ). We know that valid values returned by $\mathcal{R}$ must come from a $\mathcal{W}$ operation since only $\mathcal{W}$ quorum operations introduce new values. Since these signatures cannot be faked, it follows that the $\mathcal{W}$ quorum operation w did not happen after after r. □

This proves that the dissemination protocols in Figure 3 are atomic when using the traditional Q-RPC.

B. Fault-Tolerant Dissemination View Change

Let $s_i := encrypt(|i, N, f, m, t, g, pub\rangle_{adm}, priv, k_i^t)$. The administrator sends $\langle(N, f, m, t, g), s_0 \dots s_{n-1}\rangle_{admin}$ to a responsive quorum of new servers and then a responsive quorum of old servers.

New servers forward that message to the old servers, causing them to end the old view. The old servers acknowledge right away but they also start a new thread with which they send that message back to a responsive quorum of new servers. The new servers proceed as before (Figure 8), namely they wait for an acknowledgement from a quorum of old servers before joining the ready state in which they acknowledge to the administrator and tag their responses with the new view.

As a result, if a single correct old server ends view t then eventually a quorum of new servers will have received the message for the new view $t+1$. That is enough to guarantee that view $t+1$ has matured, so reads in the new view will go through. If on the other hand no old correct server ends view t then reads in view t will go through. Since in the event of an administrator crash the old servers are not turned off, in both cases the system will continue to process reads and writes and provide atomic semantics. If the view change does not include a generation change then the server transitions directly to the ready state.

The careful reader will have noticed that if a single faulty server in the old view has the view certificate for the new view but no correct server in the new view does (which may happen if faulty servers collude and the administrator crashes after sending its first message), the faulty old server can cause our implementation of DQ-RPC to block because the clients will try to get answers from the new servers even though the new servers do not process requests yet. However, the implementation of DQ-RPC that we describe in the optimizations Section (B.2) does not have this problem and will allow reads and writes to continue unhampered because the old view has not ended and DQ-RPC can process its replies.

B.1. DQ-RPC makes U-dissemination dynamic

Theorem 4. *U-dissemination, crash and hybrid-d based on DQ-RPC provide atomic semantics.*

Lemma 12. *The view t chosen by a DQ-RPC operation is concurrent with the DQ-RPC operation.*

Proof. The view t is chosen in ViewTracker's `findConsistentQuorum` method (Figure 9). By inspection we observe that if t is chosen then $q(n(t), f(t), m(t))$ responses from different servers were tagged with a view that was no more recent than t.

Suppose, by contradiction,, that view t has ended. Then, a quorum Q_0 of servers in t have processed the view change message and discarded the view meta-information associated with view t. The forgetting protocol ensures that the servers in Q_0 that were correct during view t will not be able to regain the view meta-information. By quorum intersection[7], the quorum of $q(t)$ replies in t selected by DQ-RPC to vouch for view t intersects the quorum Q_0 in at least one server s that was correct in t. The server s cannot generate a valid tag for view t, contradicting our hypothesis that t had ended. Thus, view t has not ended.

View certificates are signed by the administrator, and such signatures cannot be faked. Since the responses contain a certificate for view t, at least one server received this certificate from the administrator. Therefore, view t has started. □

Lemma 13. *The DQ-RPC protocol in Figure 6 provides the transquorum properties for the ordering function o of Figure 3.*

Lemma 14. *When using DQ-RPC for the U-dissemination, crash or hybrid-d protocol, no $\mathcal{R}$ operation returns $\perp$.*

Lemma 15. *All reads succeed. That is, there is no DQ-RPC$_\mathcal{T}$ or DQ-RPC$_\mathcal{R}$ operation x such that $\phi(x) = \perp$.*

Proof. The $\phi(Q)$ function returns the largest valid element of Q. DQ-RPC replies contain at least $f+1$ responses tagged with the latest generation. Servers only use the most recent tag once they have data, so all DQ-RPC replies contains at least one non-$\perp$ reply from a correct server that can be chosen by ϕ. □

Lemma 16. *All $\mathcal{T}$ operations in the dissemination DQ-RPC are timely.*

Proof. Lemma 15 shows that $o(r) \neq \perp$, so we must prove $\forall w \in \mathcal{W}, \forall r \in \mathcal{T} \ : \ w \to r \implies o(w) \leq o(r)$. Our proof proceeds by case analysis on the views associated with operations r and w.

If w and r picked views that are in the same generation then the two quorums intersect in at least one correct

7 It is safe to use quorum intersection here, since Q_0 and $q(t)$ are quorums of the same view t.

server. Since $w \rightarrow r$ and servers never decrease the timestamp they store, it follows that $o(w) \leq o(r)$.

If w picked a view t that is in the generation that immediately precedes the generation v to which r's view belongs, then we consider the last view u in t's generation. As we have seen in the previous paragraph, reads from a quorum $q(u)$ in u will result in a timestamp that is at least as large as $o(w)$. Such a read occurs when a server transitions to the ready state in v's generation, and all correct servers that enter v's generation therefore have a timestamp at least as recent as $o(w)$. The read r waits until it knows that view v is mature, so at least one correct server answered the DQ-RPC r after installing view v. Since that server is correct, its reply is valid. Since that reply has a timestamp at least as recent as $o(r)$ and dissemination quorums pick the largest valid reply, it follows that $o(w) \leq o(r)$.

If w picked a view that happens several generations before r then we can apply the previous paragraph's reasoning several times to show that $o(w) \leq o(r)$.

Finally, it is not possible for w to pick a view in a later generation than what r picks if $w \rightarrow r$. DQ-RPCs decide on a view whose generation is vouched for by at least $f+1$ servers with data. That means that one of these servers is correct. Correct servers only install a generation after all previous generation have ended, so r cannot pick an earlier generation.

This covers all the possible ordering for generations, so $\forall w \in \mathcal{W}, \forall r \in \mathcal{T} \; : \; w \rightarrow r \Longrightarrow o(w) \leq o(r)$. □

Lemma 17. *All $\mathcal{R}$ operations in the dissemination DQ-RPC are sound.*

Proof. Since no dissemination operation returns $\perp$ we show $\forall r \in \mathcal{R} : \exists w \in \mathcal{W}$ s.t. $r \not\rightarrow w \wedge o(w) = o(r)$. Since $o(r)$ picks the largest valid value and the cryptographic primitives hold, $o(r)$ can only return a value that was written previously, so one for which a w operation exists. Furthermore that value must have been forwarded to the reader so the write must happen before the read or concurrently with it. □

B.2. Optimizations

B.2.1. Single-Roundtrip Reads Both the U-dissemination and U-masking protocols (and their hybrid counterparts) can be sped up by skipping the write-back in the case of *unanimous reads*, i.e. reads in which responses in the quorum agree. This idea is not new but it is interesting since it leads to single-roundtrip reads in the common case where no operation is parallel with the read.

For single-roundtrip reads the TRANS-$Q_{\mathcal{R}}$ operations must have the property that $\forall r_1, r_2 \in \mathcal{R}$: unanimous$(r_1) \Rightarrow o(r_1) \leq o(r_2)$. This property holds for quorum intersection (since an unanimous read means that the service state is similar to what it would be after writing that value), and it also holds for both our dissemination and masking DQ-RPC implementations.

B.2.2. Reducing Transmissions The protocol, as described in this paper, piggybacks view information onto each message sent by the servers. Also, clients verify all of these messages. Since in most cases the view will be the same as it was in the previous exchange, several optimizations can be used to decrease both the amount of data that needs to be transmitted and the computations necessary to verify that information.

Servers send the view information along with the administrator certificate and signed nonce. To optimize for the common case while retaining the forgetting property described in Section 5.2.1, servers could omit the view information and instead just send the view number t. If the client knows about that view then it has all the necessary data to verify the signature. If it does not, then the client sends a request to the server to retrieve the complete view information.

Another opportunity arises in the choice of quorums. Instead of using the same quorums for the views meta-information as for the data, we can use asymmetric quorums. This is beneficial because view meta-data is read more often than it is written. These asymmetric quorums use the smallest possible read quorums ($2f+1$ since the view meta-information is self-verifying) and the largest possible write quorums ($n-f$). The current approach in ViewTracker is to verify that (1) there is a dissemination (resp. masking) quorum of responses such that no member claims that the current view has ended and (2) enough responses are in the same generation as the current view. The first point can be changed to (1) there is a read quorum of responses such that no member claims that the current view has ended. Reads naturally still need to gather a dissemination (resp. masking) quorum of responses in order to read the variable, but with this change it is not necessary anymore that all the responses be tagged with view information. The client can then indicate in its queries whether the server should include a view certificate in its reply.

Another natural optimization is that in the few cases were servers still need to send the view meta-information to the clients, the servers can send the difference between their information and the one the client knows instead of sending the whole thing.

B.2.3. Proactive Recovery Proactive recovery is a technique in which machines are periodically refreshed to a known good state. This brings down the number of faulty machines and thus reduces the risk that the system will fail because more than f servers are faulty.

Proactive recovery requires us to remove a server, refresh it (for example by rebooting it), and then bring it back in the system under a different name. Our dynamic quorums are particularly well suited for proactive recovery because we can add and remove servers with little overhead, and client operations can complete even if they span several different views.

B.2.4. Tolerating More Faults We can replace the administrator machine with a replicated state machine to reduce the likelihood that it fails.

B.2.5. Faster Reads and Writes It is possible to speed up quorum operations by slightly weakening the conditions under which the DQ-RPC function returns. The correctness of the protocol only requires that DQ-RPC selects as its current view t one that is concurrent with the DQ-RPC. The DQ-RPC we show in this paper always picks the most recent concurrent view that is knows of. This causes DQ-RPC to sometimes wait for messages when that is not necessary. Consider the case where $q+1$ responses are received. The first response is in view $t+1$ and all the others are in view t. In that situation it would be perfectly reasonable to pick the last q responses as the result of the DQ-RPC operation, but our simplified operation will wait until it gets a q responses in view $t+1$.

The change impacts ViewTracker (Section 9). Instead of keeping track of the most recent view certificate it sees (*m_maxS*), ViewTracker must now inspect each set of responses to see if there exists some view that can be considered current. `findQuorum` is replaced with the following code:

```
1. find $qrm \subseteq messageTriples$ such that
2.    let $mt$ := largest-timestamped element of $qrm$
3.    $\forall m \in qrm : m.sender \in mt.meta.N$, and
4.    $|qrm| = q(|mt.meta.N|, mt.meta.f, mt.meta.m)$,
5.    and $|\{m \in qrm :$ validTag$(m) \wedge m.meta.g == mt.meta.g\}| \geq mt.meta.f + 1$
6. **if** no such $qrm$ exists **then** return $(\emptyset, \perp)$
7. return $(qrm, mt.meta)$
```

This code is slightly harder to read than the original, but it still picks a current view that is concurrent with the DQ-RPC and it allows DQ-RPC to complete sooner in the case outlined above. The termination condition here is strictly weaker than before, so there is no situation where this DQ-RPC would be slower than the original one.

The `get` method must also be modified to include more servers than just the last view, for example the union of the two most recent views.

B.2.6. Faster Generation Changes Our protocols' ability to add servers when necessary relies on the fact that the data will be copied to the new servers. The DQ-RPC and view change protocols make sure that the protocol semantics are maintained despite the copying. However, copying data takes time. There are some cases where we can speed up generation changes.

Consider first the case where some servers of the new view are also part of the old view. It would be unwise for them to just keep whatever data they have, for the data they are storing could be untimely and the new view may require them to hold timely data (for example if the quorum size changes). In most cases, however, the data on the server is timely and we can avoid the copy by using *conditional reads*. In a conditional read, the server issuing the read indicates the timestamp of the data that it has. If the respondent does not have data that is newer than the indicated timestamp then it sends a response with empty data (but the timestamp and view certificates are still included when appropriate). If the respondent has newer data then it sends it as usual. As a result, servers that are already timely do not need to transfer the data across the network and they can join the new view much more quickly.

Conditional reads yield their full power when used in combination with our second optimization. Recall that the spread parameter allows new servers to join the system without having to receive a copy of the data first (intra-generation view changes). These servers normally participate in the protocol and refresh their data in the next generation change. We can choose, instead, to have the new (blank) servers read the current value of the data in the background (perhaps using TCP Nice [21]). When it comes time for the generation change, the servers can use conditional reads: if they already have the right data then they can move to the new view instantly.

C. Generic Data

C.1. Masking Protocols with Transquorums

In this section we show that the U-masking protocol provides partial-atomic semantics despite up to b Byzantine faulty servers. This protocol assumes that the network links are asynchronous authenticated and fair. Clients are assumed to be correct and the administrator machine may crash.

The U-masking protocol is shown in Figure 11. The only change from its original form [18] is that we have substituted TRANS-Q for Q-RPC operations.

Partial-atomic semantics: All reads R either return $\perp$, or return a value that satisfies atomic semantics.

Theorem 5. *The U-masking protocol provides partial-atomic semantics if the Q-RPC operation it uses has the transquorums properties for the function o defined below.*

Proof. We define o and O in the exact same way as we did for the dissemination protocol in Section 4.2.1. Then we show the following three properties:

1. $X \rightarrow W \implies O(X) < O(W)$ and $W \rightarrow X \implies O(W) < O(X)$
2. $O(W_1) = O(W_2) \implies W_1 = W_2$
3. Read R returns either $\perp$ or the value written by some W such that
   (a) $R \not\rightarrow W$, and
   (b) $\nexists W' : O(W) < O(W') < O(R)$

The first two points show that O defines a total order on the writes and that the ordering is consistent with "happens before". The third point shows that reads return the value of the most recent preceding write.

We prove that the protocols satisfy partial-atomic semantics by building an ordering function O for the read and write operations that satisfies the requirements for partial-atomic semantics.

Both read and write end with a $\mathcal{W}$ quorum operation w. The first quorum operation in writes never returns $\perp$. By the first property of transquorums, that operation therefore has a timestamp that is at least as large as that of w. The write operation then increases that timestamp further, ensuring that $X \rightarrow W \implies O(X) < O(W)$. Our construction of the mapping O ensures that if a read happens after a write, then that read gets ordered after the write. These two facts imply property (1).

The o value includes the *writer_id*, which is different for each writer - and if the same writer performs two writes then (1) implies that they'll have different values. Therefore property (2) holds: $O(W_1) = O(W_2) \implies W_1 = W_2$. These two properties together show that writes are totally ordered in a way that is compatible with the happens before relation.

Next we show that non-aborted reads return the value of the preceding write (property (3)). Soundness tells us that this value does not come from an operation that happened after R (3a). We know that the value returned by reads must come from a write operation since only writes can introduce new values that are reported by $b+1$ servers: so the value returned by a read R comes from some write W. Note that $O(R)$ and $O(W)$ have the same ts, $writer\_id$ and D; they only differ in the last element (and so $O(W) < O(R)$). Thus, any write $W' > W$ will necessarily also be ordered after R since $O(W') > O(R)$ (3b). □

If the Q-RPC operations have the *non-triviality property* that $\mathcal{R}$ quorum operations that are concurrent with no other quorum operation never return $\perp$, then U-masking has the property that reads that are not concurrent with any operation never return $\perp$ either. This follows directly from the fact that if no operation is concurrent with a read R then no quorum operation is concurrent with any of R's quorum operations. Our implementation of DQ-RPC has the non-triviality property.

C.2. DQ-RPC for Masking Quorums

In this section we show how to build the DQ-RPC and view change protocol for masking quorums, when data is not signed. Only one line needs to change: line 5 of View-Tracker's `findQuorum` (Figure 9), shown below.

```
if $|recentMessages| < 2 * m\_maxMeta.f + 1$ then
    return $(\emptyset, \perp)$
```

Thus read operations now wait until they get $2f + 1$ servers vouching for the current generation instead of $f + 1$. It follows that $f + 1$ correct servers have entered the new generation, so they will be able to countermand any old value proposed by servers that have not finished the view change.

The view change protocol must be modified however, because as described in Section 5.3.2 it relies on the fact that the servers' read of the current value never fail. This is not true in the case of masking quorums, where reads may fail if some write is concurrent with it.

Figure 13 gives the view change protocol for the administrator. If clients are correct then the function is guaranteed to eventually terminate. If no write is concurrent with the view change then the administrator only goes through the loop once. Once the newView operation returns, it is safe to turn off the machines in the old view that are not part of the new view – we say that the new view is *weaned*.

In order to provide atomic semantics, we must ensure that reads reflect the values written previously, and thus we must propagate data from the old view to the new one. The view change protocol allows clients to query the new servers right away, before the administrator copies any data. How can this work?

The key is that (as shown in Figure 15) the new servers will get their data from the old ones to service client requests. Once a new server has read some value from the old servers it never needs to contact the old servers again since writes are directed to the new ones (we say that the server is *weaned*). Once enough new servers have data stored locally, it is possible to shut down the old servers – we must just be careful that nothing bad happens to new servers that were in the middle of reading from the old ones.

So the new servers, when they are asked for data that they don't have, first check whether the old servers are still available by checking whether a peer server has a wean certificate (using the READ_LOCAL call). If the server receives a wean certificate, it knows that there is no point

```
READ
  1. $Q$ := TRANS-Q$_{\mathcal{R}}$("READ")        // reply is of the form $(ts, writer\_id, data)$
  2. reply $r$ := $\phi(Q)$     // $\phi$ : the only non-countermanded value vouched by $b+1$ servers, or $\perp$
  3. if $r$==$\perp$ then return $\perp$
  4. $Q$ := TRANS-Q$_{\mathcal{W}}$("WRITE",$r$)
  5. return $r.data$
WRITE($D$)
  1. $Q$ := TRANS-Q$_{\mathcal{T}}$("GET_TS")
  2. $ts := max\{Q.ts\} + 1$
  3. $Q$ := TRANS-Q$_{\mathcal{W}}$("WRITE",$(ts, writer\_id, D)$)
```

Figure 11: U-masking protocol for correct clients

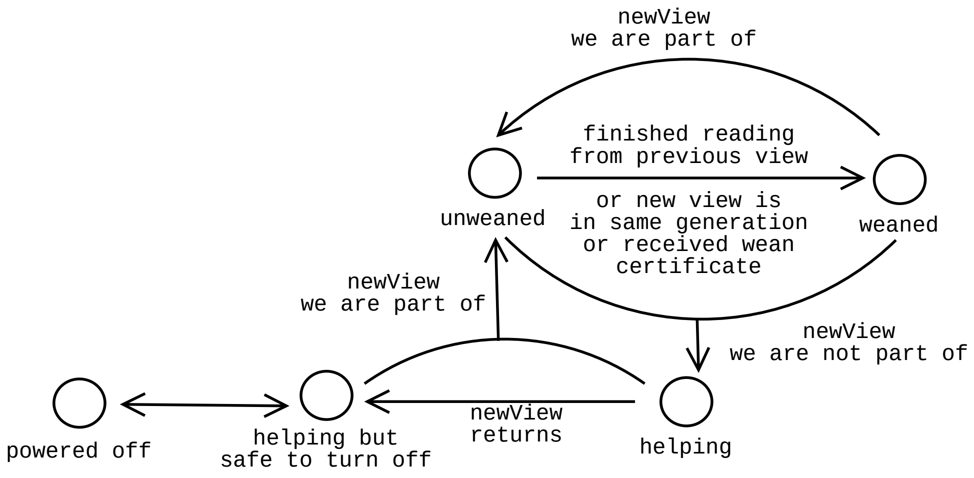


Figure 12: Server transitions for the masking protocol

in trying to contact the old servers: the server then returns whatever local data it has, possibly $\perp$. If there is no wean certificate then the server forwards the request to the old servers. If the old servers have been shut down in the mean time then this request may take forever; that's OK because the old servers are only turned off if the administrator completed successfully, and in that case the `waitForWean` function will eventually stop any read thread that is stuck in this manner.

The `waitForWean` function periodically queries the peers to see if they have a wean certificate. This ensures that if the new view is weaned then eventually all servers will know about it (or move on to an even more recent view).

When new unweaned servers receive a write request, they make sure that the old view has ended, then store the data and acknowledge. But servers do not consider themselves weaned as a result of a write. So when someone tries to read that data, the servers will still try to contact the servers in the old view to make sure the local data is recent enough.

The servers go through different states, as described in Figure 12. A server that is not part of the current view is in the *helping* state. In that state it responds to queries but tags them with the most recent view certificate, thus directing clients to more recent servers. When a server receives a new view certificate (and the server is part of the new view), it moves on to the *unweaned* state. It accepts requests from clients right away and starts `waitForWean` in a parallel thread to detect when the system becomes weaned. Read requests are forwarded to the old servers; if a non-$\perp$ reply can be determined then that reply is stored locally before being forwarded to the client and the server moves on to the *weaned* state. Servers will also move to *weaned* when they receive a wean certificate from their peers.

newView

```
1. Give their view certificate to a quorum in the new view
2. Give info about the new view to a quorum in the old view
3. Repeat
4.         $a$ := read on old view
5.         $b$ := read on new view
6. Until ($a \neq \perp \vee b \neq \perp$)
7. Generate wean certificate ("old view is gone now")
8. Write $max(a, b)$ to a quorum in the new view
9. Write the wean certificate to a quorum in the new view
```

Figure 13: View change protocol for masking quorums

Server i's variables

m_D	the current data
m_ts	the data's timestamp (initially -1)
m_meta	current view meta-information: (N,f,m,t,g,$pubKey$)
$m\_oldMeta$	meta-information for the previous view: (N,f,m,t,g,$pubKey$)
m_cert	admin certificate for (m_meta)
m_priv	private key matching certificate
$m\_weanCert$	certificate that the view in m_meta is weaned
$m\_serverWeaned$	true if the server is weaned (initially false)
$m\_oldEnded$	true if the server knows that the old view ended (initially false)

Figure 14: Server variables for masking quorums

C.3. DQ-RPC Satisfies Transquorums for Masking Quorums

We now show that DQ-RPC also satisfies transquorums when we use the masking quorum's ϕ operation. Recall that that ϕ returns the value that is vouched for by $f+1$ servers and that is not countermanded, or $\perp$ if there is no such value.

Lemma 18. *The masking DQ-RPC operations are timely.*

Proof. Recall that timeliness means $\forall w \in \mathcal{W}, \forall r \in \mathcal{T}, o(r) \neq \perp : w \rightarrow r \implies o(w) \leq o(r)$. The proof is similar to that for the dissemination case. If w and r picked views in the same generation then the two quorums intersect in at least $f+1$ correct servers. Since w happened before r and servers never decrease the timestamp they store, it follows that $o(r) \neq \perp \Rightarrow o(w) \leq o(r)$.

If w picked a view t that is in the previous generation from r's view (say v), then we consider the last view u in t's generation. As we have seen in the previous paragraph, non-aborted reads from a quorum $q(u)$ in u will result in a timestamp that is at least as large as $o(w)$. Since r picked a view that is in a more recent generation than u, it follows that r received $2f(v)+1$ replies in v's generation (so at least one correct). Correct servers in the new view only respond to a read request until they know that either they or their view has weaned. It follows that the replies in r contained at least $f(v)+1$ replies C that are at least as recent as the highest-timestamped value whose write completed in view u, which in turn is at least $o(w)$. So if r were to pick any value such that $o(r) < o(w)$ then that value would be countermanded by C. Therefore $o(r) \neq \perp \Rightarrow o(w) \leq o(r)$.

It is not possible for w to pick a view in a later generation than what r picks if $w \rightarrow r$ since $\mathcal{R}$ masking DQ-RPCs wait until any previous generation has ended. This concludes our proof that $\forall w \in \mathcal{W}, \forall r \in \mathcal{R}, o(r) \neq \perp : w \rightarrow r \implies o(w) \leq o(r)$. □

Lemma 19. *The masking DQ-RPC operations are sound.*

Proof. Soundness requires that $\forall r \in \mathcal{R}, o(r) \neq \perp : \exists w \in \mathcal{W}$ s.t. $r \not\rightarrow w \wedge o(w) = o(r)$.

Correct servers only respond to read queries with data that was previously written – either directly to them or to the previous quorum. The ϕ function ensures this property by only accepting values that are vouched for by $f+1$ servers. □

```
write($ts$,$D$)
  1. if ($m\_ts$<$ts$) then ($m\_ts$,$m\_D$) := ($ts$,$D$)
  2. if not $m\_oldEnded$ then askOldView()
     // $m\_oldEnded$ holds at this point
  3. return "OK"

read()
  1. if ($m\_serverWeaned \vee m\_weanCert \neq \perp$) then return ($m\_ts$,$m\_D$)
  2. if askPeers() then return ($m\_ts$,$m\_D$)
  3. if askOldView() then return ($m\_ts$,$m\_D$)
     // not $m\_serverWeaned$ and $m\_weanCert == \perp$, and the read from the old servers failed
  4. return $(-1, \perp)$

readLocal()
  1. return $m\_weanCert$

private askOldView()
  1. $Q'$:=Q-RPC("READ+HELP",$m\_cert$) to a quorum of servers in $m\_oldMeta.N$
  2. m_oldEnded := true
  3. if $\phi(Q') \neq \perp$ then
  4.         $m\_serverWeaned$ := true
  5.         if $(m\_ts, m\_D) < \phi(Q')$ then ($m\_ts$,$m\_D$) := $\phi(Q')$
  6. if $|\{m \in Q' : m\_ts < m.ts\}| < m\_oldMeta.f + 1$ then
  7.         $m\_serverWeaned$ := true
  8. return $m\_serverWeaned \vee m\_weanCert \neq \perp$

private askPeers()
  1. $Q'$:=Q-RPC("READ_LOCAL") to a quorum of servers in $m\_meta.N$
  2. if any response includes a valid wean certificate $cert$ for this view then
  3.         $m\_oldEnded$ := true
  4.         $m\_weanCert$ := $cert$
  5. return $m\_serverWeaned \vee m\_weanCert \neq \perp$

private waitForWean()          // started on its own thread when the server hears of the new view
  1. while (not $m\_serverWeaned$) $\wedge$ ($m\_weanCert == \perp$) do
  2.         askPeers()
  3.         wait for some time
  4. kill any read() or write() thread that is waiting for the old servers
```

Figure 15: Server protocol for masking quorums

Theorem 6. *DQ-RPC satisfies transquorums even if the old servers are taken offline after the newView call returns.*

The masking DQ-RPC also tolerates crashes from the administrator: all operations still eventually complete as long as the servers from the old view are *not* taken offline.

Lemma 20. *If some view t never ends then all quorum operations to that view eventually complete.*

Proof. An individual server s responds to read quorum operations once either it receives a reply from the old servers or it knows that its view is weaned (in that latter case the server only responds after the client resends its query). If the administrator failed then the old servers are not taken offline and thus they will eventually respond. If the administrator did not fail then eventually s will know that its view is weaned.

The wait quorum operation waits on the same conditions, thus individual servers will eventually respond to write quorum operations. The help and read_local operations do not block on anything, thus they will complete trivially. □

It follows from the above lemma that as long as some view stays around long enough, all DQ-RPC operations will complete.

Theorem 7. *DQ-RPC satisfies transquorums even if the administrator crashes during a view change, as long as both the old and the new servers are kept online.*

C.4. Masking Protocols for Byzantine Faulty Clients

We now turn our attention to a variant of the U-masking protocol that can handle Byzantine failures from the client. We use Phalanx [14] with an improved ϕ function that provides partial-atomic semantics (the original Phalanx only provides safe semantics). The client code is shown in Figure 16. Phalanx does not require the clients to have public-private key pairs, but the servers do. In step 4 of the WRITE operation, the clients collect signatures from the servers (the *echoes*). Servers only accept writes if they are accompanied by a quorum of valid signatures. For write-backs, servers require $f+1$ of a different type of signature. Notice that the signature step is neither timely nor sound: the signatures' only purpose is to make the write call succeed.

It is natural to ask why we consider Byzantine faulty clients. After all, nothing prevents faulty clients from continuously writing incorrect values. However, in many practical situations such faulty clients would eventually be identified and removed from the system. More to the point, our goal here is to show that the DQ-RPC operation can make many protocols dynamic. We have included the protocol of Figure 16 for completeness.

The protocol guarantees partial-atomic semantics, meaning that all reads that return a value satisfy atomic semantics and reads that are not concurrent with any other operation always return a value.

READ

```
1. $Q$ := TRANS-Q$_{\mathcal{R}}$("READ")
2. reply $r$ := $\phi(Q)$        // largest non-countermanded triple vouched for by at least $f+1$ servers
3. if $r$==$\perp$ then return $\perp$
4. let $V$ be $f+1$ valid signatures for $r$ taken from $Q$
5. $Q$ := TRANS-Q$_{\mathcal{W}}$("WRITE-BACK",$r$, $V$)
6. return $r.data$
```

WRITE(D)

```
1. $Q$ := TRANS-Q$_{\mathcal{T}}$("READ_TS")
2. $ts$ := $max\{Q.ts\}+1$
3. let $m$ := $(ts, writer\_id, D)$
4. $Q'$ := TRANS-Q("SIGN",$m$)
5. let $V$ be a quorum of valid signatures for $m$ taken from $Q'$
6. $Q''$ := TRANS-Q$_{\mathcal{W}}$("WRITE",$m$,$V$)
```

Figure 16: Masking quorum or hybrid-m for Byzantine faulty clients