# 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 *

## 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], and GeoQuorums [6] can adjust dynamically both $f$ and

* 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.

$N$, 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.

Unfortunately, space limitation force us to state, rather than prove, the theorems and lemmas that we claim in this paper. The proofs are presented in a technical report [15].

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.

[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

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

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

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

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

whether it was written or not. To capture this diversity, we introduce two properties, *timeliness* and *soundness*. We 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:

$$\begin{array}{ll} \text{(timeliness)} & \forall w \in \mathcal{W}, \forall r \in \mathcal{T}, o(r) \neq \bot \ : \\ & \quad w \rightarrow r \Longrightarrow o(w) \leq o(r) \\ \text{(soundness)} & \forall r \in \mathcal{R}, o(r) \neq \bot : \\ & \quad \exists w \in \mathcal{W} \text{ s.t. } r \not\rightarrow w \ \wedge \ o(w) = o(r) \end{array}$$

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 an extended technical report [15]. Space considerations limit us to consider in this paper only 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 timestamped 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

| 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

largest value in the set $Q$ that has a valid signature using lexicographical ordering: since our values are triplets $(ts, writer\_id, D)$, $\phi$ selects the largest valid timestamp, 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 the table below. 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 $W \rightarrow X \Longrightarrow W < X$ and $X \rightarrow W \Longrightarrow X < W$ 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 are proved in our technical report [15]:

**Lemma 1.** *Our ordering relation “$<$” is a total order on user writes; further, $W \rightarrow X \Longrightarrow W < X$ and $X \rightarrow W \Longrightarrow X < W$ 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 proves 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.

[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.

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 $q(t)$ 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. If we assume that clients have some external, infallible way to know which servers are in an active view 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 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.

[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.

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. 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 $i$ possesses 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:

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

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 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 mes-

```
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 4: Dynamic quorum RPC

sage 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.

We split the implementation of DQ-RPC into three parts. The main DQ-RPC body (Figure 4) 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 6). 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 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.

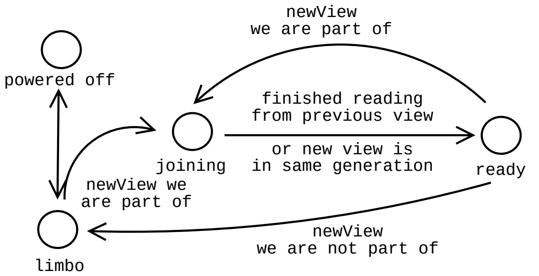

Figure 5: Server transitions for the dissemination protocol

**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 5. 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 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 the extended technical report [15] 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.

**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 proof is presented in our technical report [15]. The main lemmas used in the proof are listed below.

**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 4 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 $\perp$.*

## 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

```
$(meta)$ **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 $msg == \text{CONSULT-ACK}$ 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. $validMessages := \{m \in msgInQuorun : \text{validTag}(m)\}$
  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 6: Definition of the ViewTracker object

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 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.