

Automatic Reconfiguration for Large-Scale Reliable Storage Systems

Rodrigo Rodrigues, Barbara Liskov, Kathryn Chen, Moses Liskov, and David Schultz
MIT Computer Science and Artificial Intelligence Laboratory

Abstract—Byzantine-fault-tolerant replication enhances the availability and reliability of Internet services that store critical state and preserve it despite attacks or software errors. However, existing Byzantine-fault-tolerant storage systems either assume a static set of replicas, or have limitations in how they handle reconfigurations (e.g., in terms of the scalability of the solutions or the consistency levels they provide). This can be problematic in long-lived, large-scale systems where system membership is likely to change during the system lifetime.

In this paper we present a complete solution for dynamically changing system membership in a large-scale Byzantine-fault-tolerant system. We present a service that tracks system membership and periodically notifies other system nodes of membership changes. The membership service runs mostly automatically, to avoid human configuration errors; is itself Byzantine-fault-tolerant and reconfigurable; and provides applications with a sequence of consistent views of the system membership. We demonstrate the utility of this membership service by using it in a novel distributed hash table called dBQS that provides atomic semantics even across changes in replica sets. dBQS is interesting in its own right because its storage algorithms extend existing Byzantine quorum protocols to handle changes in the replica set, and because it differs from previous DHTs by providing Byzantine fault tolerance and offering strong semantics. We implemented the membership service and dBQS. Our results show the approach works well in practice: the membership service is able to manage a large system and the cost to change the system membership is low.



1 INTRODUCTION

Today we are more and more dependent on Internet services, which provide important functionality and store critical state. These services are often implemented on collections of machines residing at multiple geographic locations such as a set of corporate data centers. For example, Dynamo uses tens of thousands of servers located in many data centers around the world to build a storage back-end for Amazon’s S3 storage service and its e-commerce platform [1]. As another example, in Google’s cluster environment each cluster includes an installation of the GFS file system spanning thousands of machines to provide a storage substrate [2].

Additionally, these systems are long-lived and need to continue to function even though the machines they run on break or are decommissioned. Thus there is a need to replace failed nodes with new machines; also it is necessary to add machines to the system for increased storage or throughput. Thus the systems need to be reconfigured regularly so that they can continue to function.

This paper provides a complete solution for reliable, automatic reconfiguration in distributed systems. Our approach is unique because

- It provides the abstraction of a globally consistent view of the system membership. This abstraction simplifies the design of applications that use it, since

it allows different nodes to agree on which servers are responsible for which subset of the service.

- It is designed to work at large scale, e.g., tens or hundreds of thousands of servers. Support for large scale is essential since systems today are already large and we can expect them to scale further.
- It is secure against *Byzantine* (arbitrary) faults. Handling Byzantine faults is important because it captures the kinds of complex failure modes that have been reported for our target deployments. For instance, a recent report about Amazon’s S3 showed that a bit flip in a server’s internal state caused it to send messages with the wrong content [3]. Additionally the Byzantine fault model makes it possible to tolerate malicious intrusions where an attacker gains control over a number of servers.

Earlier proposals for keeping track of a dynamic system membership do not provide all three properties. Many do not tolerate Byzantine failures, (e.g., [4], [5]). Some that handle Byzantine faults provide consistency but work only for a system containing a small number of nodes (e.g., [6]), while others trade off consistency to achieve scalability (e.g., [7]). The one exception is Census [8]; this builds on our techniques for flexible reconfiguration, but uses other mechanisms to track system membership.

Our solution has two parts. The first is a *membership service* (MS) that tracks and responds to membership changes. The MS works mostly automatically, and requires only minimal human intervention; this way we can reduce manual configuration errors, which are a

• Rodrigo Rodrigues is now at the Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern and Saarbrücken, Germany, and Moses Liskov is at the College of William and Mary, Williamsburg, VA.

major cause of disruption in computer systems [9]. Periodically the MS publishes a new system membership; in this way it provides a globally consistent view of the set of available servers. The choice of strong consistency makes it easier to implement applications, since it allows clients and servers to make consistent local decisions about which servers are currently responsible for which parts of the service.

We run the MS on a small group of replicas and use a number of protocols [10], [11], [12], [13] to enable the MS to tolerate malicious attacks; we were able to take advantage of protocols developed by others but combine them in novel ways. Using a small group for the MS is important since these protocols work well only in this case. The design provides scalability to a large number of nodes, most of which are clients of the MS. Additionally it avoids overloading the servers that form the MS by offloading expensive tasks to other nodes.

When there is a reconfiguration, the MS may need to move to a new group of servers. This way we allow the system to continue to operate correctly, even though the failure bound in the original group of MS replicas may subsequently be exceeded. We present a design for reconfiguring the Byzantine-fault-tolerant group.

Tracking membership is only part of what is needed for automatic reconfiguration. In addition, applications need to respond to membership changes appropriately.

Therefore, the second part of our solution addresses the problem of how to reconfigure applications automatically as system membership changes. We present a storage system, dBQS, that provides Byzantine-fault-tolerant replicated storage with strong consistency. dBQS serves as an example application that uses the membership service and takes advantage of its strong consistency guarantees. Additionally, dBQS is important on its own for two reasons. First, to develop dBQS we had to extend existing Byzantine quorum protocols, originally designed for a static replica set, to enable them to be reconfigurable while continuing to provide atomic semantics across changes in the replica set. Second, dBQS implements the popular DHT interface [14], but differs from previous DHTs by handling Byzantine faults and focusing on strong semantics, which can facilitate design of applications that build on a DHT interface. In addition the techniques used to handle membership changes in dBQS could be generalized to other applications.

We have implemented the membership service and dBQS. We present performance results that show that the MS is able to manage a large system and reconfigure in a reasonably short interval, and that the impact of reconfiguration on dBQS performance is small.

2 SYSTEM MODEL AND ASSUMPTIONS

This section defines our model and assumptions.

We assume a system comprised of nodes that can be servers implementing a storage service or clients using that service. We assume without loss of generality that the two sets are disjoint.

We assume nodes are connected by an unreliable asynchronous network like the Internet, where messages may be lost, corrupted, delayed, duplicated, or delivered out of order. While we make no synchrony assumptions for the system to meet its safety guarantees, it is necessary to make partial synchrony assumptions for liveness, which is discussed in Section 5.3.

We assume the existence of the following cryptographic techniques that an adversary cannot subvert: a collision-resistant hash function, a public key cryptography scheme, and forward-secure signing keys [12], [13]. We also assume the existence of a proactive threshold signature protocol [15], [16], [17], [11], [18] that guarantees that threshold signatures are unforgeable without knowing f or more out of n secret shares.

We assume a Byzantine failure model where faulty nodes may behave arbitrarily. We assume a compromised node remains compromised forever. This is a realistic assumption because once a node is Byzantine faulty, its secret information, including its private key, may be known, and therefore it cannot be recovered and then continue to be trusted. Instead it needs a new key, which effectively means its identity has changed.

We assume nodes have clocks whose *rates* should be loosely synchronized to keep time windows during which failure bounds must be met reasonably short. We do not depend on loosely synchronized clocks in order to minimize our assumptions.

3 THE MEMBERSHIP SERVICE

This section describes the membership service (MS), which provides a trusted source of membership information.

The MS describes membership changes by producing a *configuration*, which identifies the set of servers currently in the system, and sending it to all servers. To allow the configuration to be exchanged among nodes without possibility of forgery, the MS authenticates it using a signature that can be verified with a well-known public key.

The MS produces configurations periodically rather than after every membership change. The system moves in a succession of time intervals called *epochs*, and we batch all configuration changes at the end of the epoch. Producing configurations periodically is a key design decision. It allows applications that use the MS to be optimized for long periods of stability (we expect that in storage applications epochs could last for hours, although our evaluation shows we can support short epochs if needed), and it reduces costs associated with propagating membership changes (like signing configurations or transmitting them). It also permits *delayed response to failures*, which is important for several reasons: to avoid unnecessary data movement due to temporary disconnections, to offer additional protection against denial of service attacks (assuming we wait for longer than the duration of such attacks), and to avoid thrashing,

where in trying to recover from a host failure the system overstresses the network, which itself may be mistaken for other host failures, causing a positive feedback cycle.

The notion of epochs provides consistency: all nodes in the same epoch see exactly the same system membership. Each epoch has a sequential *epoch number*. Epoch numbers allow nodes to compare the recency of different configurations. Furthermore, application messages include the epoch number of the sender; this allows nodes to learn quickly about more recent configurations.

We begin by specifying the functionality of the MS in Section 3.1. We discuss how to implement that specification in a way that tolerates Byzantine faults in Section 3.2. We discuss some refinements to the design to achieve better scalability in Section 3.3. Section 3.4 discusses the impact of faulty servers on our system.

3.1 MS Functionality

3.1.1 Membership Change Requests

The MS responds to requests to add and remove servers.

We envision a managed environment with admission control since otherwise the system would be vulnerable to a Sybil attack [19] where an adversary floods the system with malicious servers. Thus we assume servers are added by a trusted authority that signs certificates used as parameters to these requests. The certificate for an ADD request contains the network address and port number of the new server, as well as its public key, whereas the certificate to REMOVE a node identifies the node whose membership is revoked using its public key.

The MS assigns each server a unique node ID uniformly distributed in a large, circular ID space, which enables the use of consistent hashing [20] to assign responsibility for work in some of our MS protocols; applications can also use these IDs if desired. The MS chooses the server's node ID as a SHA-1 hash of the values in the add certificate. To prevent an attacker from adding a group of servers that are all nearby in the ID space, we require that the node's public key be chosen by the trusted authority.

We need to prevent replay of add requests; otherwise a server that was removed could be re-added by a malicious party. We do this by putting an interval of epoch numbers in the add request; the request is only good for those epochs. This approach allows us to limit how long the MS needs to remember revocations.

3.1.2 Probing

The MS detects unreachable servers and marks them as inactive. To do this, the MS probes servers periodically, normally using unauthenticated ping messages, which we expect to be sufficient to detect most unreachable servers. Infrequently, probes contain nonces that must be signed in the reply, to avoid an attack that spoofs ping replies to maintain unavailable servers in the system. Signed pings are used sparingly since they require additional processing on the MS to verify signatures.

However, once a server fails to reply to a signed ping, subsequent pings to that server request signatures until a correctly signed response arrives.

If a server fails to reply to a threshold n_{evict} number of probes, it is declared to be *inactive* and will be removed from the active system membership in the next epoch. The probe frequency and eviction threshold are system parameters that control how quickly the MS responds to failures.

If an inactive server contacts the MS, it will be marked as *reconnected*, but first the MS sends it a challenge that is signed in the reply to avoid a replay attack. Servers are not allowed to remain inactive indefinitely; instead such servers are evicted automatically after some number of epochs has passed.

Probes allow us to remove crashed servers. For Byzantine-faulty servers, we rely on manual intervention: an administrator obtains a revocation certificate from the trusted authority and uses it to remove the compromised server.

3.1.3 Ending Epochs

Epochs can terminate after a fixed duration or some number of membership changes or a combination of the two. The termination condition is a system parameter that must be set by a system administrator based on deployment characteristics, e.g., expected churn.

To determine when the epoch ends, the MS tracks the termination condition. When the termination threshold is reached the MS stops probing, and produces an *epoch certificate* signed by the MS's private key. The signature in the certificate covers a digest of the membership (list of servers and their reachability status) and the epoch number of the new epoch.

Then the MS sends a NEWEPOCH message to the other servers describing the next configuration. This message contains the certificate and new epoch number, and describes the configuration changes using deltas: it contains lists of added, removed, inactive, and reconnected servers. The message is authenticated by the MS so that verifying it is easy. Transmitting deltas is important for scalability, as discussed further in Section 3.3.

3.1.4 Freshness

Clients of the application using the MS need to verify the freshness of their configuration information to ensure they are communicating with the group that currently stores an item of interest, and not an old group (which may have exceeded the failure threshold).

We provide freshness by means of *freshness certificates*. The mechanism works as follows. To use the replicated service the client requires an unexpired freshness certificate. It obtains a certificate by issuing a challenge to the MS. The challenge contains a random nonce; the MS responds by signing the nonce and current epoch number. The response gives the client a period of time T_{fc} during which it may execute requests; the value of T_{fc} is another system parameter. The client determines

Operations supported by the MS	Requests initiated by the MS
ADD(<i>cert</i>) – takes a certificate signed by the trusted authority describing the node; adds the node to the set of system members.	PROBE(<i>nonce</i>) – the MS sends probes to servers periodically; servers respond with a simple ack, or, when a nonce is sent, by repeating the nonce and signing the response.
REMOVE(<i>cert</i>) – also takes a certificate signed by the trusted authority that identifies the node to be removed; removes this node from the current set of members.	NEWÉPOCH(<i>cert,e,changes</i>) – informs nodes of a new epoch. Here, <i>cert</i> is a certificate vouching for the configuration, and <i>changes</i> represents the delta in the membership.
FRESHNESS(<i>nonce</i>) – receives a freshness challenge; the reply contains the nonce and current epoch number signed by the MS.	

TABLE 1
Membership service interface.

when the period has expired by using its clock: it reads the time when it sends the challenge, and treats the corresponding freshness certificate as valid until that time plus the duration T_{fc} . If the certificate expires, the client halts application work until it obtains a new one.

When a client joins the system or reconnects after an absence, it can contact any system participant to obtain a system configuration; if the client knows of no current system members it can learn of some through an out-of-band mechanism. The contacted node sends it both the epoch certificate it received in the most recent NEWÉPOCH message, and also the configuration it computed as a result of processing that message. The client can use the certificate to verify that the configuration is authentic. In addition the client must obtain a freshness certificate to ensure the configuration is current.

Freshness certificates do not constrain the MS: it moves to the next epoch when thresholds are reached, without regard for freshness certificates. They also do not prevent clients from moving to a new epoch, and a client need not refresh its certificate when it does so. Rather these certificates ensure that clients do not use an old configuration for “too long”. As explained in Section 5, the correctness conditions for applications that use the MS require that old groups meet their failure thresholds until the last client freshness certificate expires.

Table 1 summarizes the specification of the service. Next we describe the techniques we use to implement it in a reliable and scalable way.

3.2 Byzantine Fault Tolerance

To provide Byzantine fault tolerance for the MS, we implement it with a group of $3f_{MS}+1$ replicas executing the PBFT state machine replication protocol [10]. These MS replicas can run on server nodes, but the size of the MS group is small and independent of the system size.

We describe how the MS operations are translated to request invocations on the PBFT group in Section 3.2.1, and how to reconfigure the MS (e.g., to handle failures of the nodes that compose it) in Section 3.2.2.

3.2.1 PBFT Operations

PBFT provides a way to execute operations correctly even though up to f replicas out of $3f+1$ are faulty. Therefore we can implement ADD and REMOVE as PBFT

operations, which take as arguments the respective certificate, and whose effect is to update the current set of members (which is the PBFT service state maintained by the MS). Freshness challenges can also be implemented as PBFT operations.

However the MS does more than perform operations: it probes servers, decides when they are faulty, decides when to end an epoch, and propagates information about the new configuration to all the servers. This additional work must be done in a way that prevents faulty members of the MS from causing a malfunction, while at the same time ensuring progress. We avoid problems due to faulty nodes by requiring that $f_{MS}+1$ MS replicas vouch for any action that is based on non-deterministic inputs.

Replicas probe independently, and a replica proposes an eviction for a server node that has missed $n_{propose}$ probe responses. It does this by sending eviction messages to other MS replicas and then waiting for signed statements from at least $f_{MS}+1$ MS replicas (including itself) that agree to evict that node. Other MS replicas accept the eviction (and sign a statement saying so) if their last n_{evict} pings for that node have failed, where $n_{evict} < n_{propose}$. Because the initiation of the eviction waited a bit longer than necessary, most eviction proposals will succeed if the node is really down.

Once the replica has collected the signatures, it invokes the EVICT operation, which runs as a normal PBFT operation. This operation has two parameters: the identifier of the node being evicted and a vector containing $f_{MS}+1$ signatures from MS replicas agreeing to evict the node. The operation will fail if there are not enough signatures or they do not verify.

We use a similar scheme for reconnecting servers and ending epochs: the proposer waits until other nodes are likely to agree, collects $f_{MS}+1$ signatures, and invokes the RECONNECT or MOVEÉPOCH operation, respectively.

After the MOVEÉPOCH operation is executed, all MS replicas agree on the membership in the next epoch: server nodes for which REMOVE operations have been executed are removed and those for which ADD operations have been executed are added. Also EVICT and RECONNECT operations mark server nodes as inactive or active. Then the MS replicas can produce a certificate describing the membership changes for the new epoch.

3.2.2 Reconfiguring the MS

There are two plausible ways to run the MS. The first is to use special, separate nodes that are located in particularly secure locations. The second is to use an “open” approach in which the MS runs on regular system members: servers occasionally act as MS replicas, in addition to running the application. Our system can accommodate either view, by considering the first one as a special case of the open approach: we can mark servers (when they are added) to indicate the roles they are allowed to assume.

At the end of the epoch the system may decide to move the MS to a different set of servers. This can happen because one of the MS replicas fails; in the open approach it may also happen proactively (every k epochs) since the nodes running the MS are attractive targets for attack and this way we can limit the time during which such an attack can be launched. The steps that are needed to move the MS occur after the old MS executes the MOVEEPOCH operation, and are summarized in Figure 1.

Choosing MS replicas. When the MS is reconfigured proactively, we must prevent an attacker from predicting where it will run next since otherwise an attack could be launched over many epochs. This is accomplished by choosing the MS replicas based on a random number, r_{e+1} . We choose this number by running a PBFT operation that contains as an argument a hash of a random number chosen by $2f_{MS} + 1$ replicas. After this step a second operation discloses the corresponding random value. The value of r_{e+1} is produced by the hash of the concatenation of the first $f_{MS} + 1$ random values. A proof that this scheme works even when f_{MS} replicas including the primary are faulty is contained in [21].

Signing. The MS needs to sign certificates with the private key that corresponds to its well known public key. But no MS replica can know this key, since if it were faulty it could expose it. Therefore each MS replica holds a share of the associated private key and the signature is produced using a proactive threshold signature scheme [15], [16], [17], [11], [18]. This scheme will only generate a correct signature if $f_{MS} + 1$ replicas agree on signing a statement.

When the MS moves, the new replicas obtain new shares from the old replicas, allowing the next MS to sign. Non-faulty replicas discard old shares after the epoch transition completes. Different shares must be used by the new MS replicas because otherwise these shares could be learned by the attacker once more than f_{MS} failures occurred in a collection of MS groups. Ensuring that data about shares is erased completely is non-trivial; methods to achieve this are discussed in [22].

Freshness Challenges. Threshold signatures are expensive and we would like to avoid their use for signing freshness challenges. Instead we would rather use normal keys to sign the challenge responses: a challenge response is legitimate if it contains individual signatures from $f_{MS} + 1$ MS replicas. This approach is feasible if MS

- | |
|---|
| <ol style="list-style-type: none"> 1) Choose the random number. 2) Sign the configuration using the old shares. 3) Carry out a resharing of the MS keys with the new MS members. 4) Discard the old shares. |
|---|

Fig. 1. Summary of steps for reconfiguring the MS.

replicas have forward secure signing keys [12], [13]. The approach works as follows. Clients accept a freshness response for an epoch e only from a replica that is a member of the MS in e and furthermore the response is signed using that replica’s key for e . When a server moves to the next epoch, it advances its signing key to that epoch, and thus becomes unable any longer to sign for earlier epochs. This ensures that it will be impossible to obtain $f_{MS} + 1$ freshness responses for an old epoch, even if more than f_{MS} members of the MS in that epoch are now faulty, since at least $2f_{MS} + 1$ of them have forgotten their keys for that epoch.

3.3 Scalability

We want to support large numbers of servers and clients. To achieve this goal we must consider storage size, communication requirements, and load at the MS replicas.

Storage size. In our scheme each node stores the configuration in memory, but this is not a concern: if we assume that node identifiers are 160 bits (based on a SHA-1 cryptographic hash function), and we use 1024 bit RSA public keys, the entire configuration for a system of 100,000 servers will fit in approximately 14.7 megabytes, which is small compared to current memory sizes.

Communication requirements. Although storing configurations isn’t a problem, it would not be desirable to communicate this much information at each new epoch. Instead we communicate only the membership events that occurred in the previous epoch. Even though this set of events grows linearly with system size, in our envisioned deployments we expect low churn and therefore the constant will be small enough to make this communication feasible. For instance, if a large-scale failure causes 10% of nodes in a 100,000 node system to fail, the amount of information transmitted would be 10,000 node ids, which takes only 200 KB.

A related issue is the time it takes for a new client to download the configuration. (This is not a problem for new servers since they won’t be active until the next epoch). Here we can use Merkle trees [23] so that clients can download specific parts of the configuration, e.g., to learn about nodes in a particular ID interval first. Merkle trees can also be used by a reconnecting node to identify the minimum information that needs to be transmitted.

Load on the MS. Running the MS on a small subset of system members is crucial for scalability since the agreement protocol is quadratic in the number of nodes in the group. However, we must also ensure that server nodes acting as MS replicas are not overloaded. There are three activities of concern. First is communication at

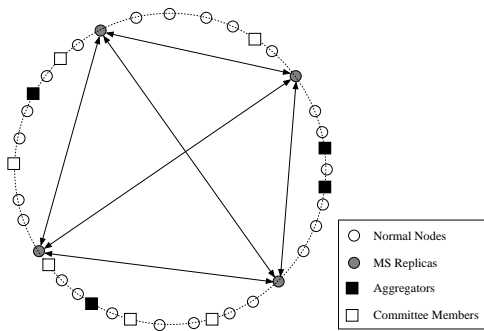


Fig. 2. System architecture, showing a subset of nodes running the MS, two committees, and four aggregators.

the end of an epoch – if the MS had to inform every node about the new configuration. We can avoid this expense by using distribution trees [21], [8]. Even if some node does not receive the message containing the delta (e.g., due to a failure of an ancestor in the tree), it will detect the failure when it receives a message with a larger epoch number (recall that application messages contain the epoch number of the sender); then it can retrieve the configuration from the sender of that message.

The second potential source of overload is probing. To avoid this expense, we use *committees*. A committee is a group $2f_{MS} + 1$ servers who are not members of the MS. Each committee is responsible for probing part of the server space; the committee members do these probes independently. The MS interrogates committees periodically and infrequently. If $f_{MS} + 1$ members of a committee report that a server is unresponsive, that node can be moved to the replica’s inactive list (since at least one honest server has found it to be unresponsive). We require only $2f_{MS} + 1$ members in committees since they need not carry out agreement. Committee reports can be used as the argument of the `EVICT` or `RECONNECT` operation to prove that the request is valid.

The third potential source of overload is freshness challenges. Although freshness certificates can be valid for a long time, there could be so many clients that the MS could not keep up with the challenges. We solve this problem using aggregation. Clients send challenges to particular (non-MS) servers; different groups of clients use different servers. A server collects challenges for some time period, or until it has enough of them (e.g., a hundred), hashes the nonces, sends the challenge to the MS, and forwards the signed response to the clients, together with the list of nonces that were hashed.

Figure 2 illustrates the final system architecture containing committees and aggregators.

3.4 Faulty Servers

Faulty servers cannot cause our system to behave incorrectly (under the correctness conditions we define in Section 5), but they can degrade performance of our protocols. In the case of PBFT, this can be problematic due to the fact the primary replica plays a special role in

the protocol, but recent work explains how to minimize this performance degradation [24].

In our remaining protocols the roles of different replicas are symmetric, so they are less affected by Byzantine replicas. The aggregation protocol is an exception: a single Byzantine replica can prevent freshness responses, but if this happens clients will switch to a different aggregator after a timeout.

4 DYNAMIC REPLICATION

This section describes how storage applications (or other services) can be extended to handle reconfigurations using the membership service. In particular, we present dBQS, a read/write block storage system based on Byzantine quorums [25]. dBQS uses input from the MS to determine when to reconfigure.

dBQS is a strongly consistent distributed hash table (DHT) that provides two types of objects. *Public-key* objects are mutable and can be written by multiple clients, whereas *content-hash* objects are immutable: once created, a content-hash object cannot change. In this paper we describe only public-key objects. In a separate document [21] we describe the simpler protocols for content-hash objects, and also dBFT, a state machine replication system based on the PBFT algorithm, and a general methodology for how to transform static replication algorithms into algorithms that handle membership changes. A complete, formal description of the main protocols used in dBQS and a proof of their correctness can be found in a technical report [26].

Data objects in dBQS have *version numbers* that are used to determine data freshness, and identifiers that are chosen in a way that allows the data to be self-verifying (similarly to previous DHTs such as DHash [14]). The ID of a public-key object is a hash of the public key used to verify the integrity of the data. Each object is stored along with a signature that covers both the data and the version number. When a client fetches an object its integrity can be checked by verifying the signature using a public key that is also validated by the ID of the object. Version numbers are assigned by the writer (in such a way that distinct writers always pick distinct version numbers, e.g., by appending client IDs).

We partition objects among system nodes using consistent hashing [20]: servers are assigned random IDs in the same ID space as objects, and the replica group responsible for ID i consists of the first $3f + 1$ servers in the current system membership (as dictated by the MS) whose identifiers are equal to or follow i in the identifier space. Figure 3 illustrates the assignment of replica groups to objects.

dBQS ensures that concurrent accesses to the entire set of public-key objects are atomic [27]: all system operations appear to execute in a sequential order that is consistent with the real-time order in which the operations actually execute. We avoided designing the protocols for public-key objects from scratch, but instead

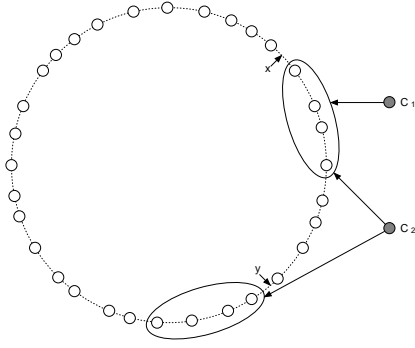


Fig. 3. dBQS replica groups. The figure illustrates the membership of two groups, one responsible for object x , accessed by clients c_1 and c_2 , and the other responsible for object y , accessed only by c_2 .

extended existing Byzantine quorum protocols [25] in novel ways to support reconfigurations, and provided some optimizations for them.

4.1 Storage Algorithms in the Static Case

Read and write operations are performed using *quorums*. Each object is stored at $n = 3f + 1$ nodes and quorums consist of any subset containing $2f + 1$ nodes. We begin by describing the normal case when all communicating nodes are in the same epoch; the epoch can be checked because all protocol messages contain the sender's epoch number. A high-level description of the client-side read and write protocols for this case is shown in Figure 4.

The write operation for a public-key object normally has two phases. In the *read phase*, a quorum of $2f + 1$ replicas is contacted to obtain a set of version numbers for the object. Then the client picks a (unique) version number greater than the highest number it read, signs the object contents and the new version number, and performs the *write phase* where it sends the new signed object to all replicas and waits until it hears replies from a quorum. The client sends a random nonce with both requests, and the reply contains a signature that covers this nonce together with the current version number, to prevent replay attacks; clients know the public keys of all nodes since this is part of the configuration information. Clients repeat unanswered requests after a timeout to account for message loss. Clients that access the same objects repeatedly can use MAC-based authentication instead of signatures.

Replicas receiving a write verify that the content of the object, along with the version number, match the signature. If this verification succeeds, the replica replies to the client, but it only overwrites the object if the new version number is greater than the one currently stored.

The read phase can be omitted if there is a single writer (as in some existing applications, e.g., Ivy [28]); in this case, the writer can increment the last version number it knows and use it in the write phase.

To perform a read operation, the client requests the object from all replicas in the *read phase*. Normally there

WRITE($x, data$)

- 1) Send $\langle \text{READVERSION}, x, nonce, e \rangle$ messages to the replicas in the group that stores x and wait for $2f + 1$ valid responses, all for epoch e .
- 2) Choose the largest version number returned by these responses and increment it to obtain a unique larger version number v . Then send $\langle \text{WRITE}, x, data, v, e \rangle$ messages to all the replicas and wait for $2f + 1$ valid responses.

$data = \text{READ}(x)$

- 1) Send $\langle \text{READ}, x, nonce, e \rangle$ messages to replicas of the group that stores x and wait for $2f + 1$ valid responses, all for epoch e .
- 2) If all responses contain the same version number, return that data. Otherwise select the highest version number among those returned, send $\langle \text{WRITE}, x, data, v, e \rangle$ messages to the replicas, and wait for $2f + 1$ valid responses. Then return $data$ to the user.

Fig. 4. Simplified protocols for read and write operations.

will be $2f + 1$ valid replies that provide the same version number; in this case the result is the correct response and the operation completes. However, if the read occurs concurrently with a write, the version numbers may not agree. In this case, there is a *write-back* phase in which the client picks the response with the highest version number, writes it to all replicas, and waits for a reply from a quorum. Again, nonces are employed to avoid replays, and the reply contains a signature that covers the nonce, and in the first phase also the current version number. (We optimize the read phase of the read protocol by sending a small request for the signed version number to all replicas; the entire object is downloaded from the first replier with the highest version number.)

This scheme offers atomic semantics for crash-faulty clients [25]. It does not prevent Byzantine-faulty clients from causing atomicity violations, for instance, by writing different values to different replicas; a way to handle Byzantine-faulty clients is described in a separate publication [29].

4.2 Membership Changes

Our protocols for reading and writing need to be modified to handle reconfigurations. We first explain how to handle the situation of nodes being in different epochs, then we explain how to perform state transfer when object responsibility shifts.

4.2.1 Executing Requests Across Epochs

When a client or a server receives a valid, more recent configuration (either through the multicast of NEW-POCH messages or as a reply to contacting a node that is more up-to-date), it immediately switches to that epoch. Of course this switch will happen at different times at different nodes. Here we explain what happens when a client knows a different epoch than a server; Figure 5 shows the messages exchanged by the server and client.

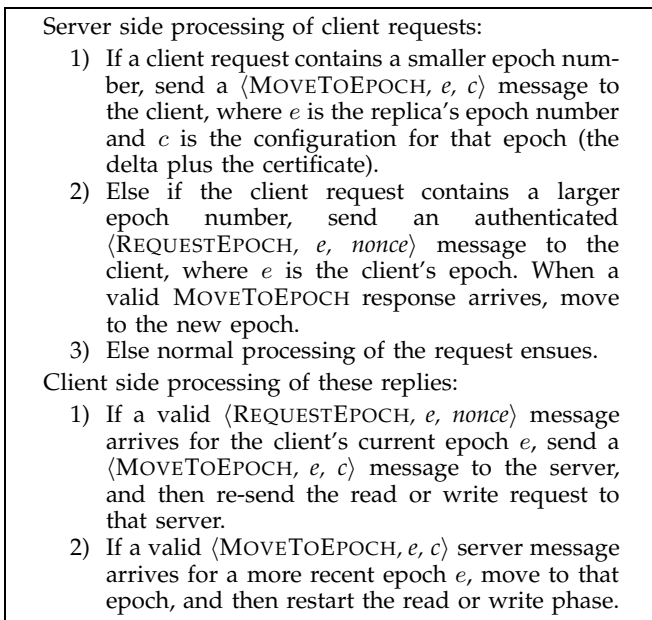


Fig. 5. Processing of messages during epoch changes.

As we explained earlier, a client request contains its epoch number. If the client's epoch number is smaller than the replica's, the replica rejects the request and instead sends the current configuration information. The client will then retry the request in the new epoch if necessary (if the request has not yet completed). Each individual phase of an operation can complete only when the client receives $2f+1$ replies from replicas in the same epoch. This constraint is needed for correctness: If an individual phase used results from different epochs, we could not ensure the intersection properties that make the operations atomic. Note that the read and write phases can complete in distinct epochs, however.

If the client's epoch number is larger than the replica's, the replica requests the new configuration, and the client pushes it to the replica before retrying (if necessary). Also, the replica won't reply to a request if it has not completed state transfer for that object (discussed next).

Clients need not refresh their freshness certificate when they move to a later epoch. Rather these are used to avoid communicating with an old replica group that now contains more than f faulty nodes.

4.2.2 State Transfer

When the configuration changes, the set of servers storing certain objects may change. Each server must identify objects that it is no longer responsible for and refuse subsequent requests for them. Each node also does *state transfer*; it identifies objects it is newly responsible for and fetches them from their previous replicas (in the earlier epoch). To do state transfer nodes must know both the old and new configuration. Therefore when a node receives information about the next epoch, it retains the earlier configuration until it completes state transfer. (Also, a node can fetch information about earlier

configurations from the MS if necessary; this raises an issue of when to garbage collect old configurations at the MS, which is addressed elsewhere [21].)

The node sends state transfer requests to the old replicas requesting all objects in a particular interval in the ID space (these replicas enter the new epoch at that point if they have not done so already). In practice, this is done in two steps, with a first message requesting the IDs of all objects in the interval, and a second step to fetch the objects using the read protocols described above, except that, unlike normal reads, state transfer does not need to write back the highest value it read because the first client operation in the new epoch will take care of the write-back. A further point is that our implementation prioritizes state transfer requests for objects that have outstanding client requests.

State transfer ensures atomicity even if responsibility for an object moves to a completely disjoint group of servers in the new epoch. The new replicas will be unable to answer client requests for that object until they receive its state from $2f+1$ old replicas. However the old replicas switch to the new epoch before replying to state transfer requests and once a replica switches it won't reply to client requests for that object. This means that by the time the new replicas respond to client requests for that object, the old replicas will no longer do so.

4.2.3 Deletion of Old Data

Old replicas must delete data objects they are no longer responsible for to avoid old information accumulating forever. A new replica sends acks to the old replicas when it completes state transfer for some object. An old replica counts acks and deletes the object once it has $2f+1$ of them. It explicitly requests acks after a timeout to deal with message loss.

After it deletes this state, the old replica can still receive state transfer requests from a correct but slow new replica (one it had not heard from previously). In this case it replies with a special null value and lowest version number. If a new replica receives $2f+1$ such replies it uses this value and version number for the initial state of the object; the quorum intersection properties ensure that the values stored at the replicas that got the actual content and version number for the object will be seen in any quorum that is subsequently used to carry out operations.

5 CORRECTNESS

This section presents the correctness conditions and the semantics provided by the system when such conditions are met.

5.1 Safety Properties for the MS

The protocols used by the MS have different correctness conditions, all of which must be met. This constraint leads to the following correctness condition:

Correctness condition for the MS: For each epoch e , the MS replica group for e must contain no more than f_{MS} faulty replicas up until the moment when the last non-faulty MS replica finishes that epoch, discards its secret threshold signature share, and advances its forward-secure signing key.

Given this assumption, and assuming that the various protocols are able to terminate, we are able to offer very strong guarantees: at the end of an epoch the MS produces a certificate that describes the new system membership, which can be used by clients and servers to agree on the membership in that epoch. Additionally, freshness certificates for epoch e cannot be produced after the last non-faulty MS replica for epoch e finishes that epoch because the forward-secure signing keys were advanced (and at most f_{MS} of these replicas could have been compromised up to that moment).

A further point is that the MS will not falsely exclude correct, reachable nodes because $f_{MS} + 1$ nodes must agree to remove a node and by assumption at least one of them must be non-faulty. If we use committees, we must also assume that no more than f_{MS} of the $2f_{MS} + 1$ committee members are faulty during the period of time covered by the above condition.

In essence, the correctness condition defines a window of vulnerability (a time interval when the MS group cannot contain more than f_{MS} faults). The interval isn't bounded from below since it is irrelevant whether a bad MS replica became Byzantine faulty before or during the epoch. The interval is bounded from above by the occurrence of certain events (like the conclusion of the epoch transition protocols). If an adversary can delay such an occurrence it might be able to extend the window long enough to allow it to break the correctness conditions of the system by corrupting more than f_{MS} nodes.

We cannot guarantee that the system will be able to run forever. For instance, an adversary can break the system if it is able to corrupt more than f_{MS} replicas in the MS for epoch e before the end of e . However, our system is useful because it provides a much longer lifetime than one that uses a static group of replicas. A static system is likely to run into problems as the nodes that compose the MS start to fail. In contrast, in a dynamic system, nodes that fail can be removed upon detection, and new (or recovered) nodes are constantly being added, which substantially increases the period during which the correctness conditions are met.

5.2 Safety Properties for dBQS

To operate correctly, dBQS requires correctness of the MS. But it also depends on a correctness condition that is similar, but not identical, to that for the MS:

Correctness condition for dBQS. For any replica group g_e for epoch e that is produced during the execution of the system, g_e contains no more than f faulty replicas up until the later of the following two events: (1) every non-faulty node in epoch $e + 1$ that needs state from g_e has completed state transfer, or (2) the last client freshness certificate for epoch e

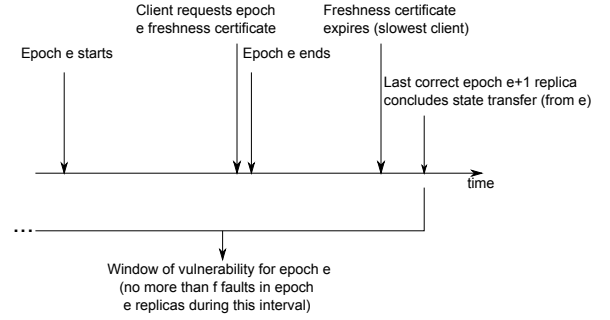


Fig. 6. Window of vulnerability for epoch e .

or any earlier epoch expires at any non-faulty client c that accesses data stored by g_e .

Given this condition, dBQS provides atomic semantics; a proof is given in a separate technical report [26].

As was the case for the MS, the condition defines a time interval during which the number of simultaneous node failures is constrained. The ending of the interval depends on the occurrence of certain events, namely ending state transfer and expiration of the last client freshness certificate; this is illustrated in Figure 6. The figure shows the time for state transfer being greater than the freshness expiration time, which can happen if state transfer takes a long time, e.g., when the state is large. However the reverse may also be true, e.g., if we use a long freshness period.

5.3 Liveness

Liveness of the MS depends on two factors. First, the various protocols must be able to complete. Not only this would be impossible in an asynchronous system [30], but just assuming eventual delivery isn't sufficient, e.g., for PBFT to make progress [10]. Instead we need the stronger partial synchrony assumption of eventual time bounds [27]. This means that the system runs asynchronously for an unknown amount of time, but eventually starts to satisfy timing constraints with respect to message delivery and processing times.

Under this condition the MS can transition between epochs, but if epochs are too short there may not be enough time for the MS to process membership events. Therefore in addition we must assume epochs are long enough so that probing can detect unreachable nodes and requests to add and remove nodes can be executed.

In dBQS, ensuring that client operations eventually terminate requires only a weak guarantee on message delivery, such as asynchronous fair channels (i.e., a message sent infinitely often would be received infinitely often). Additionally, epochs must be long enough that clients are able to execute their requests.

Liveness of the MS influences correctness of dBQS: in practice it is important for the MS to make progress so that the failure threshold imposed by the dBQS safety condition is met. But as long as the failure threshold holds, and the liveness conditions for dBQS are

met, dBQS is live regardless of whether the MS makes progress.

6 EVALUATION

We implemented the membership service and dBQS in C++. The MS is implemented as a BFT service using the publicly available PBFT/BASE code [31]. For asynchronous proactive threshold signatures we implemented the APSS protocol [11], but we adapted it to support resharing to a different set of replicas [32]. (Wong et al. [17] were the first to propose this kind of resharing.) dBQS is based on the code for the DHash peer-to-peer DHT built on top of Chord [5]. Inter-node communication is done over UDP with a C++ RPC package provided by the SFS toolkit [33]. Our implementation uses the 160-bit SHA-1 cryptographic hash function and the 1024-bit Rabin-Williams public key cryptosystem implemented in the SFS toolkit for authenticating communication and signing data objects. The MS and dBQS run as separate user-level processes and communicate using a Unix domain socket.

This section presents our experimental evaluation. Section 6.1 evaluates performance during an epoch, when no reconfiguration is happening; most of the time the system is in this state, since the reconfiguration period is typically long (e.g., on the order of hours). Section 6.2 evaluates the cost of reconfiguring the system.

The experiments we describe in this section used machines from the PlanetLab and RON infrastructures located in approximately 200 sites on four continents (and, where noted, we used additional nodes in our LAN).

6.1 Performance During an Epoch

This section evaluates the performance of the MS during an epoch, and the impact of superimposing the service on dBQS servers. A more detailed evaluation of the base performance of dBQS can be found in [21].

Three types of membership activities happen during an epoch: processing of membership events, such as node additions and deletions; handling of freshness certificates; and probing of system members. The first two are not a major performance concern. We assume a deployment in which membership events happen only occasionally; processing of these events requires the use of PBFT, but previous work [34] shows that this cost is modest for reasonable values of f . Renewals of freshness certificates are not a problem because the certificates are refreshed infrequently and can be aggregated.

However probing is potentially a problem since probes need to be sent regularly to all system members. Here we examine the load on the probers (committee members or members of the MS if there are no committees); this analysis in turn allows us to determine how many committees are needed, given a target probe frequency and a system of size N nodes.

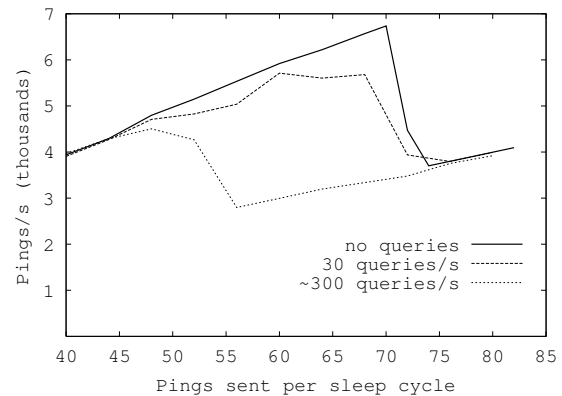


Fig. 7. Ping throughput.

The number of nodes a prober can monitor depends on two factors: the maximum rate of pings each prober is sending and the desired inter-ping arrival rate at each system member. To determine how system load limits the maximum probe rate we ran a simple load test where we monitored an MS replica in our LAN while varying the rate at which it probed other nodes in a large scale system; the replica verified a signed nonce for 10% of the ping replies. During the experiment the replica didn't process adds and removes and therefore the experiment also shows the load on a committee member. The replica ran on a local machine with a 2 GHz Pentium 4 processor and 1 GB of memory running Linux 2.4.20. We populated the system with many additional nodes located in our LAN to avoid saturating wide-area links.

Throughout our experiments, we tried to determine how system components interfere with each other. For this experiment, the replica doing the pings was associated with an instance of dBQS (since we intend to run committees on system nodes). We repeated the experiment under three different degrees of activity of the dBQS server: when it is not serving any data (which will be the case when the MS does the pings and runs on special nodes that don't also handle the application), when it is handling 30 queries per second, and when clients saturate the server with constant requests, which leads to the maximal number of about 300 queries per second. Each query requested a download of a 512 byte block.

Figure 7 shows how many pings the replica could handle as we varied the number of probes sent each time the replica resumed execution after a short sleep cycle (of about 10 ms). The experimental methodology was to run the system for a long time (until the replica handled a total 400,000 pings), determine the length of time required to do this, and compute the average throughput for handling the pings. The figure shows three lines, corresponding to the different levels of activity in terms of serving data. It shows that the number of pings the node can handle increased linearly with the number of pings sent per sleep interval up to almost 7000 pings per second if the node is not serving data, or up to about

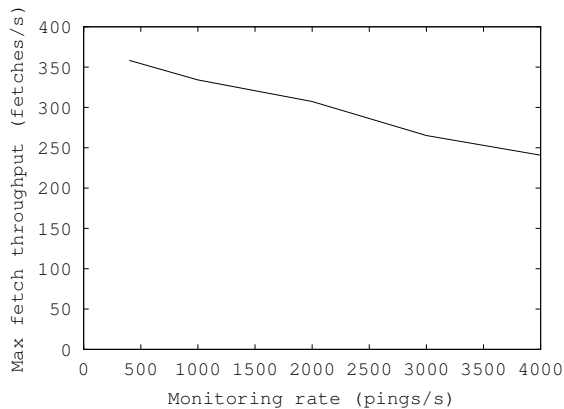


Fig. 8. Fetch throughput, while varying the ping rate.

5000 pings per second if the node is also serving content. After this point, the node goes into a state of receive livelock, in which it spends most of its time processing interrupts, and fails to perform other tasks. This leads to decreasing ping throughput as we try to send more pings.

We also evaluated the impact of sending probes on dBQS performance. We ran an experiment to determine how many fetch requests a dBQS node can handle, depending on how fast the committee member running on that node is sending probes. The results in Figure 8 show that fetch throughput decreases from 350 fetches/s to 250 fetches/s as we increase ping load to near maximal.

A final factor to consider is bandwidth consumption. In our system, outgoing packets contain 8 bytes for the nonce and control information, plus the 28 byte UDP header (ignoring Ethernet headers). Thus, for example, a probing node would spend 21 KB/s for a probe rate of 600 pings per second.

The main conclusion is that the architecture can scale well without interfering with dBQS performance. The exact parameters have to be set by an administrator when deploying the system; however, with a target ping interval of 1 minute and committee members that perform 500 pings/second, a single committee can monitor 30K nodes. This means that few committees are needed for the data center services of today [1], [2], but more will be needed in the future as system size increases.

A final point is that the MS can adjust the number of committees dynamically, based on system size and probe rate. In the future it would be interesting to add this extension to our system.

6.2 Moving to a New Epoch

The second part of the evaluation concerns the cost of moving from one epoch to the next. Here we have two concerns: the cost at the MS, and the impact on the performance of the storage protocols.

6.2.1 Cost at the MS

This part of the evaluation addresses the cost of reconfiguring the membership service at the end of an

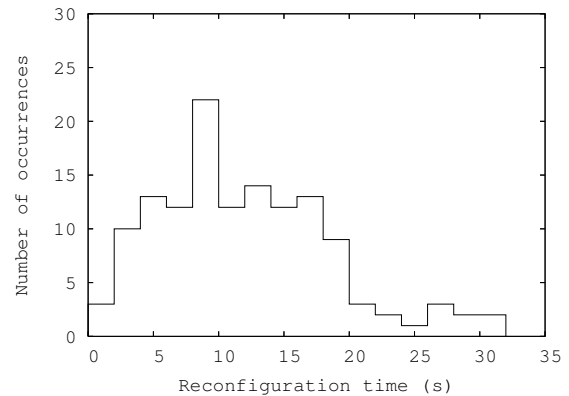


Fig. 9. Variation in time to reconfigure the MS.

epoch. We deployed our system in PlanetLab, which represents a challenging environment with frequently overloaded nodes separated by a wide area network, and measured the time it takes to move the MS during epoch transitions. The goal of the experiment was to provide a conservative estimate of running the sequence of steps for changing epochs (PBFT operations, threshold signatures, and resharing). For analysis and evaluation of the individual steps we refer the reader to [34], [11], [32].

We ran the MS for several days and measured, for each reconfiguration, the amount of time that elapsed between the beginning of the reconfiguration and its end. The MS was running on a group of 4 replicas (i.e., $f_{MS} = 1$) and moved randomly among the system nodes. Even though we were limited by the size of the testbed (in this case, hundreds of nodes), the main costs are proportional to the number of changes and the MS size, not the number of nodes, and therefore we expect our results to apply to a larger deployment.

Figure 9 presents the results. The figure shows that some reconfigurations were fast, but there is a large variation in the time to reconfigure; this is explained by the fact that nodes in the PlanetLab testbed are running many other applications with varying load, and this concurrent activity can affect the performance of the machines significantly. The main conclusion is that, even in a heterogeneous, often overloaded environment, most reconfigurations take under 20 seconds to complete. This indicates that the time to reconfigure is not a serious factor in deciding on epoch duration.

6.2.2 dBQS

When the system moves to a new epoch, any application that uses the MS must adapt to the membership changes. In this section we evaluate this cost using dBQS. The experiment measures the cost of a reconfiguration by considering servers that run dBQS but are not implementing MS functions. In this case the cost of reconfiguration is minor: for a particular client, there is the possibility that a single operation is delayed because of the need for either the client or the servers to upgrade to the

Reconfiguration Scenario	Write Latency	Read Latency
No reconfiguration	227.0	102.8
Client is behind	249.5	126.1
Servers are behind	298.6	174.9

TABLE 2

Performance of reads and writes under different reconfiguration scenarios (values in milliseconds).

new epoch, but all other operations complete normally. Note that this result assumes that state transfer is not required, which would be the case if the replica group did not move at the end of the epoch. State transfer can delay replies for objects that have not been transferred yet. By changing the rate at which we transfer objects, we can reduce delaying a reply at the cost of more communication to do state transfer. We evaluate the impact of state transfer on performance elsewhere [21].

To determine the cost of operations when clients and servers do not agree on the current epoch we used a simple micro-benchmark that conducted a single read and write operation on a 1 KB object. The object was held at four replicas (i.e., we used $f = 1$), located at MIT, UCSD, Cornell, and University of Utah. The client that measured performance was located at CMU. The experiments ran at night when network traffic was low and machines were unloaded. We repeated each operation (reading and writing a public key object) under three different scenarios: when there are no reconfigurations (client and replicas are in the same epoch); when the client is one epoch behind the servers; and when the client is one epoch ahead of the servers.

Table 2 summarizes the results; the data shown reflect the average of five trials with a standard deviation of less than 2 ms. The figure shows that performance of individual operations when the system reconfigures is close to performance when there are no reconfigurations. Performance is better when the client upgrades than when servers upgrade since the client upgrades and retries the operation after talking to the closest replica, whereas in the other case all replicas in a quorum need to upgrade.

7 RELATED WORK

We begin by discussing prior work on systems like our MS that provide membership control. Then we discuss work on replicated systems, like dBQS, that support a dynamic set of replicas. At the end of the section we discuss other examples of large-scale Byzantine-fault-tolerant storage systems.

7.1 Membership Control

The membership service has the same goals as the group membership modules present in group communication systems and our concepts of configurations and epochs

are equivalent to the notions of process groups and views introduced in the virtual synchrony model [4]. The initial work on group communication systems only tolerated crash failures. Byzantine failures are handled by the Rampart [35] and SecureRing [36] systems. Adding and removing processes in these systems is a heavyweight operation: all nodes in the system execute a three-phase Byzantine agreement protocol that is introduced by these systems [6], which scales poorly with system size. We get around this limitation by treating most nodes in the system as clients, and using only a small subset of system nodes to carry out the protocol. Thus our solution is scalable with the number system nodes, which are only clients of the protocols.

Guerraoui and Schiper [37] define a generic consensus service in a client-server, crash-failure setting, where servers run a consensus protocol, and clients use this service as a building block. The paper mentions as an example that the servers could be used to track membership for the clients; it also mentions the possibility of the service being implemented by a subset of clients. However the paper does not provide any details of how the membership service would work. We show how to implement a membership service that tolerates Byzantine faults, and discuss important details such as how to reconfigure the service itself.

Peer-to-peer routing overlays (e.g., Chord [5]) can be seen as a loosely-consistent group membership scheme: by looking up a certain identifier, we can determine the system membership in a neighborhood of the ID space near that identifier. Castro et al. proposed extensions to the Pastry peer-to-peer lookup protocol to make it robust against malicious attacks [38]. Peer-to-peer lookups are more scalable and resilient to churn than our system, but unlike our membership service, do not provide a consistent view of system membership. As a result concurrent lookups may produce different “correct” results.

Fireflies [7] is a Byzantine-fault-tolerant, one-hop (full membership) overlay. Fireflies uses similar techniques to ours (such as assigning committees that monitor nodes and sign eviction certificates). However, it does not provide a consistent view of membership, but rather ensures probabilistic agreement, making it more challenging to build applications that provide strong semantics.

Census [8] includes a membership service and provides consistent views based on epochs; it builds on the techniques described in this paper for the MS, both to end the epoch and to allow the MS to move in the next epoch. It is designed to work for very large systems, and divides the membership into “regions” based on coordinates. Each region tracks its own membership changes and reports to the MS toward the end of the epoch; the MS then combines these reports to determine the membership during the next epoch and disseminates the changes using multicast.

7.2 Dynamic Replication Protocols

Earlier work extended replication protocols (either based on read/write quorums or on state machine replication) to handle some forms of reconfiguration. Such proposals assume an administrator who determines when to move to the next epoch and what the membership will be. Our proposal for the MS is therefore complementary since it could be used in those systems to automate the role of the administrator in a way that tolerates Byzantine faults. We now discuss how the existing dynamic replication protocols relate to the ones used by dBQS.

There have been proposals for dynamic replication protocols that tolerate crash failures, such as Rambo [39] and SMART [40]. Our algorithms build on that body of work, but extend it to the more challenging Byzantine fault assumption.

Alvisi et al. [41] presented a first proposal for reconfiguring a Byzantine quorum system. However, they assume a fixed set of replicas, and only allow the fault threshold f to change between f_{min} and f_{max} . To support changing the threshold, the system uses more replicas and a larger quorum size than optimal, which enables enough intersection to ensure that values are not lost across threshold changes. (Both the state that is written by clients and the current threshold value are maintained using quorum protocols.)

Kong et al. [42] improve on this result by allowing nodes to be removed from the set of servers; adding nodes is not supported. Nodes are removed when suspected to be faulty; this decision is made by a special, fault-free monitoring node that tracks results returned by different servers in response to read operations. The techniques to ensure quorum intersection across node removals are similar to the previous work.

The work of Martin and Alvisi [43] allows the configuration to change; this work was concurrent with our dynamic read/write algorithm [44]. Their proposal assumes a trusted administrator (which the paper mentions can be implemented as an algorithm running in a BFT manner) that is responsible for issuing a view certificate at the beginning of each epoch. This certificate contains a secret epoch key for each replica; replicas use these keys to sign replies to client requests and clients accept replies only when signed for the current epoch. When replicas leave the epoch they erase their previous epoch key; this way the system guarantees that clients cannot get results from an old group. This approach solves the freshness problem without synchrony assumptions. However, our approach avoids the need to generate the epoch keys, which is problematic if done by the MS: generating the keys in a way that avoids exposure by Byzantine-faulty MS replicas is expensive, and does not scale well. Additionally, we present an implementation of the system and our design addresses the problem of automatically determining membership changes in a large scale system.

Antiquity [45] uses a secure log construction for wide-

area BFT storage. In this design, the head of the log is the only mutable state of the system, which is maintained using an evolution of Martin and Alvisi's protocol [43], and thus shares the same design choice of having an administrator that takes an active part in the epoch change protocol for each replica group. The administrator is also tasked to select the sets of storage servers that host the logs, and the deployment of Antiquity uses a DHT to trigger reconfigurations.

7.3 Large-Scale Byzantine Storage

There are also proposals for Byzantine-fault-tolerant storage systems that are related to our work. We highlight how our research could be useful in two such systems: OceanStore and Farsite.

OceanStore [46], [47] is a two-tiered system BFT storage system. The primary tier of replicas offers strong consistency for mutable data using the PBFT protocol, and the secondary tier serves static data and thus has simpler semantics. The only follow-up work that addresses reconfiguration is Antiquity, described above. We believe our membership service would be an interesting addition to this system as a means to determine the current membership for the primary tier.

Farsite [48] is a BFT file system that uses spare resources from desktop PCs to logically function as a centralized file system. The paper mentions as future work the design of a mechanism to determine which machines to place file replicas on, but, to our knowledge, no subsequent publications address this issue. Again, we believe that the membership service would be a possible mechanism to implement these features so our research would also be of use in Farsite. More recently, the replication protocols in Farsite use SMART [40] to handle a dynamic set of replicas, but this represents a change from the Byzantine failure model to only handling crash faults, and it assumes an external source that triggers reconfigurations.

8 CONCLUSION

This paper presents a complete solution for building large-scale, long lived systems that must preserve critical state in spite of malicious attacks and Byzantine failures. We present a storage service with these characteristics called dBQS, and a membership service that is part of the overall system design, but can be reused by any Byzantine-fault-tolerant large-scale system.

The membership service tracks the current system membership in a way that is mostly automatic, to avoid human configuration errors. It is resilient to arbitrary faults of the nodes that implement it, and is reconfigurable, allowing us to change the set of nodes that implement the MS when old nodes fail, or periodically to avoid a targeted attack.

When membership changes happen, the replicated service has work to do: responsibility must shift to the new replica group, and state transfer must take place

from old replicas to new ones, yet the system must still provide the same semantics as in a static system. We show how this is accomplished in dBQS.

We implemented the membership service and dBQS. Our experiments show that our approach is practical and could be used in a real deployment: the MS can manage a very large number of servers, and reconfigurations have little impact on the performance of the replicated service.

REFERENCES

- [1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007, pp. 205–220.
- [2] J. Dean, "Designs, lessons and advice from building large distributed systems," Keynote talk at LADIS 2009.
- [3] "Amazon S3 Availability Event: July 20, 2008," <http://status.aws.amazon.com/s3-20080720.html>, 2008.
- [4] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, Nov. 1987, pp. 123–138.
- [5] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. SIGCOMM 2001*.
- [6] M. Reiter, "A secure group membership protocol," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 31–42, Jan. 1996.
- [7] H. D. Johansen, A. Allavena, and R. van Renesse, "Fireflies: scalable support for intrusion-tolerant network overlays," in *Proceedings of the 2006 EuroSys Conference*, Leuven, Belgium, pp. 3–13.
- [8] J. Cowling, D. R. K. Ports, B. Liskov, R. A. Popa, and A. Gaikwad, "Census: Location-aware membership management for large-scale distributed systems," in *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, USA, Jun. 2009.
- [9] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *USITS '03, 4th USENIX Symposium on Internet Technologies and Systems*, Seattle, Washington, Mar. 2003.
- [10] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, LA, Feb. 1999.
- [11] L. Zhou, F. B. Schneider, and R. van Renesse, "Coca: A secure distributed on-line certification authority," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 329–368, Nov. 2002.
- [12] M. Bellare and S. Miner, "A forward-secure digital signature scheme," in *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pp. 431–448.
- [13] R. Canetti, S. Halevi, and J. Katz, "A forward-secure public-key encryption scheme," in *Advances in Cryptology-Eurocrypt'03*, LNCS 2656, pp. 255–271.
- [14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.
- [15] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung, "Proactive public key and signature systems," in *Proceedings of the 4th ACM conference on Computer and communications security (CCS)*, 1997, pp. 100–110.
- [16] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strohli, "Asynchronous verifiable secret sharing and proactive cryptosystems," in *Proc. 9th ACM conf. on Computer and Communications Security*, 2002, pp. 88–97.
- [17] T. M. Wong, C. Wang, and J. M. Wing, "Verifiable secret redistribution for archive systems," in *Proceedings of the 1st International IEEE Security in Storage Workshop*, 2002, pp. 94–105.
- [18] D. Schultz, B. Liskov, and M. Liskov, "Brief announcement: Mobile proactive secret sharing," in *Proc. 27th Annual Symposium on Principles of Distributed Computing (PODC)*, Aug. 2008.
- [19] J. Douceur, "The sybil attack," in *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002.
- [20] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the WWW," in *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, El Paso, Texas, May 1997, pp. 654–663.
- [21] R. Rodrigues, "Robust services in dynamic systems," Ph.D. dissertation, Massachusetts Institute of Technology, Feb. 2005.
- [22] G. Di Crescenzo, N. Ferguson, R. Impagliazzo, and M. Jakobsson, "How to forget a secret," in *STACS 99: 16th Annual Symposium on Theoretical Aspects of Computer Science*, Mar. 1999, pp. 500–509.
- [23] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *Advances in Cryptology*, 1987.
- [24] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin, "Making byzantine fault tolerant systems tolerate byzantine faults," in *Proc. 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2009.
- [25] D. Malkhi and M. Reiter, "Secure and scalable replication in phalanx," in *Proc. of the 17th Symposium on Reliable Distributed Systems*, Oct. 1998.
- [26] R. Rodrigues and B. Liskov, "A correctness proof for a byzantine-fault-tolerant read/write atomic memory with dynamic replica membership," MIT LCS TR/920, Sep. 2003.
- [27] N. Lynch, *Distributed Algorithms*. Morgan Kaufmann Pub., 1996.
- [28] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen, "Ivy: A read/write peer-to-peer file system," in *Proc. 5th symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.
- [29] B. Liskov and R. Rodrigues, "Tolerating byzantine faulty clients in a quorum system," in *Proc. of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006)*, Lisbon, Portugal.
- [30] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the impossibility of group membership," in *PODC '96: Proc. of the 15th symposium on Principles of distributed computing*, pp. 322–330.
- [31] R. Rodrigues, M. Castro, and B. Liskov, "BASE: Using abstraction to improve fault tolerance," in *Proceedings of the 18th Symposium on Operating System Principles (SOSP'01)*.
- [32] K. Chen, "Authentication in a reconfigurable byzantine fault tolerant system," Master's thesis, MIT, Jul. 2004.
- [33] D. Mazieres, "A toolkit for user-level file systems," in *Proc. Usenix Technical Conference*, Jun. 2001, pp. 261–274.
- [34] M. Castro, "Practical byzantine fault tolerance," Ph.D. dissertation, Massachusetts Institute of Technology, 2001.
- [35] M. Reiter, "The Rampart toolkit for building high-integrity services," *Theory and Practice in Distributed Systems*, pp. 99–110, 1995.
- [36] K. Kihlstrom, L. Moser, and P. Melliar-Smith, "The SecureRing Protocols for Securing Group Communication," in *Proc. of the Hawaii International Conference on System Sciences*, Jan. 1998.
- [37] R. Guerraoui and A. Schiper, "The generic consensus service," *IEEE Trans. Software Eng.*, vol. 27, no. 1, pp. 29–41, 2001.
- [38] M. Castro, P. Druschell, A. Ganesh, A. Rowstron, and D. Walach, "Security for structured peer-to-peer overlay networks," in *Proceedings of the 5th symposium on Operating systems design and implementation (OSDI'02)*, Boston, Massachusetts, Dec. 2002.
- [39] N. Lynch and A. A. Shvartsman, "Rambo: A reconfigurable atomic memory service," in *Proceedings of the 16th International Symposium on Distributed Computing (DISC'02)*, 2002.
- [40] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, "The smart way to migrate replicated stateful services," in *Proc. of 2006 EuroSys Conference*, Leuven, Belgium, pp. 103–115.
- [41] L. Alvisi, D. Malkhi, E. Pierce, M. Reiter, and R. Wright, "Dynamic Byzantine Quorum Systems," in *Proceedings of the International Conference on Dependable Systems and Networks, DSN 2000*, New York, New York, Jun. 2000, pp. 283–292.
- [42] L. Kong, A. Subbiah, M. Ahamad, and D. M. Blough, "A reconfigurable byzantine quorum approach for the agile store," in *Proc. 22nd IEEE Symposium on Reliable Distributed Systems*, Oct. 2003.
- [43] J.-P. Martin and L. Alvisi, "A framework for dynamic byzantine storage," in *Proc. Intl. Conf. on Dependable Systems and Networks (DSN)*, Jun. 2004.
- [44] R. Rodrigues and B. Liskov, "Reconfigurable byzantine-fault-tolerant atomic memory," in *Proc. of the 23rd Annual ACM SIGACT-SIGOPS Symposium on Principles Of Distributed Computing (PODC)*, St. John's, Newfoundland, Canada, Jul. 2004, pp. 386–386.
- [45] H. Weatherspoon, P. R. Eaton, B.-G. Chun, and J. Kubiatowicz, "Antiquity: exploiting a secure log for wide-area distributed storage," in *Proceedings of the 2007 EuroSys Conference*, Lisbon, Portugal, 2007, pp. 371–384.

- [46] J. Kubiawicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *ASPLOS-IX: Proc. 9th int. conf. on Architectural support for programming languages and operating systems*, 2000, pp. 190–201.
- [47] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiawicz, "Pond: the OceanStore prototype," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, San Francisco, California, Mar. 2003.
- [48] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment," in *Proc. of the 5th symposium on Operating systems design and implementation (OSDI'02)*, Dec. 2002.

PLACE
PHOTO
HERE

Moses Liskov is a Lead Infosec Scientist/Engineer at the MITRE corporation and a research assistant professor at the College of William and Mary. His interests include computer security, cryptography, and theory. This work was performed while Liskov was an assistant professor at the College of William and Mary.

PLACE
PHOTO
HERE

Rodrigo Rodrigues is a tenure-track faculty at the Max Planck Institute for Software Systems (MPI-SWS) where he leads the Dependable Systems Group. Previously, he was an assistant professor at the Technical University of Lisbon / INESC-ID. He graduated from the Massachusetts Institute of Technology with a doctoral degree in 2005. During his PhD, he was a researcher at MIT's Computer Science and Artificial Intelligence Laboratory, under the supervision of Prof. Barbara Liskov. He received

his Master's degree from MIT in 2001, and an undergraduate degree from the Technical University of Lisbon in 1998. He has won several fellowships and awards, including a best paper award at the 18th ACM Symposium on Operating Systems Principles (SOSP), and a special recognition award from MIT's Department of Electrical Engineering and Computer Science. His primary technical interest is in distributed systems, with a particular focus on system dependability.

PLACE
PHOTO
HERE

Barbara Liskov is an MIT Institute Professor of Computer Science and Engineering and head of the Programming Methodology Group. Liskov's research interests lie in programming methodology, programming languages and systems, and distributed computing. Major projects include: the design and implementation of CLU, the first language to support data abstraction; the design and implementation of Argus, the first high-level language to support implementation of distributed programs; and the Thor object-oriented

database system, which provides transactional access to persistent, highly-available objects in wide-scale distributed environments. Her current research interests include Byzantine-fault-tolerant storage systems and efficient and secure online storage. Prof. Liskov is a member of the National Academy of Engineering, and a fellow of the American Academy of Arts and Sciences, and the Association for Computer Machinery. She received The Society of Women Engineers' Achievement Award in 1996, the IEEE von Neumann medal in 2004, the ACM SIGPLAN Programming Languages Achievement Award in 2008, and the ACM A. M. Turing Award in 2008.

PLACE
PHOTO
HERE

David Schultz received a BA in computer science (with distinction) from the University of California at Berkeley and a MS in computer science from the Massachusetts Institute of Technology (MIT). He is currently working toward a PhD degree at MIT, with a focus in information flow control. His interests include distributed systems, security, and cryptography.

PLACE
PHOTO
HERE

Kathryn Chen graduated from MIT with a Master of Engineering degree in 2004 and from Wharton with a Master of Business Administration in 2009.