Fault-Scalable Byzantine Fault-Tolerant Services

Michael Abd-El-Malek*, Gregory R. Ganger*, Garth R. Goodson†, Michael K. Reiter*, Jay J. Wylie*

ABSTRACT

A *fault-scalable* service can be configured to tolerate increasing numbers of faults without significant decreases in performance. The Query/Update (Q/U) protocol is a new tool that enables construction of fault-scalable Byzantine fault-tolerant services. The optimistic quorum-based nature of the Q/U protocol allows it to provide better throughput and fault-scalability than replicated state machines using agreement-based protocols. A prototype service built using the Q/U protocol outperforms the same service built using a popular replicated state machine implementation at all system sizes in experiments that permit an optimistic execution. Moreover, the performance of the Q/U protocol decreases by only 36% as the number of Byzantine faults tolerated increases from one to five, whereas the performance of the replicated state machine decreases by 83%.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*

General Terms

Reliability, Security, Design

Keywords

Fault-scalability, Byzantine fault-tolerance, quorums, services, replicated state machines

1. INTRODUCTION

Today's distributed services face unpredictable network delays, arbitrary failures induced by increasing software complexity, and even malicious behavior and attacks from within compromised components. Like many, we believe it is important to design for these eventualities. In fact, it is increasingly important to design for multiple faults and for graceful degradation in the face of faults. Services are being designed for a massive scale in open clusters (e.g., like the Farsite project [3]), over the WAN (e.g., like the OceanStore project [34]), and in administration-free clusters that allow components to fail in place (e.g., like the Collection of Intelligent Bricks project [31]). In such settings, timing faults (such as those due to network delays and transient network partitions) and failures of multiple components, some of which may be Byzantine in nature (i.e., arbitrary or malicious), may arise. Moreover, rates of server and client crashes in such settings will likely be higher than in carefully-administered settings.

There is a compelling need for services (e.g., namespace, key management, metadata, and LDAP) and distributed data structures (e.g., b-trees, queues, and logs) to be efficient and fault-tolerant. Various services and data structures that efficiently tolerate benign failures have been developed (e.g., Paxos [21], Boxwood [25], and Chain Replication [38]). In the context of tolerating Byzantine faults, the focus has been on building services via replicated state machines using an agreement-based approach [23, 36]. Examples of such systems include the BFT system of Castro and Liskov [9] and the SINTRA system of Cachin and Poritz [8].

We define a *fault-scalable* service to be one in which performance degrades gradually, if at all, as more server faults are tolerated. Our experience is that Byzantine fault-tolerant agreement-based approaches are not fault-scalable: their performance drops rapidly as more faults are tolerated because of server-to-server broadcast communication and the requirement that all correct servers process every request.

We have developed the Query/Update (Q/U) protocol as a fault-scalable alternative to these approaches. The Q/U protocol is an efficient, optimistic, quorum-based protocol for building Byzantine fault-tolerant services. Services built with the Q/U protocol are fault-scalable. The Q/U protocol provides an *operations-based interface* that allows services to be built in a similar manner to replicated state machines [23, 36]. Q/U objects export interfaces comprised of deterministic methods: *queries* that do not modify objects and *updates* that do. Like a replicated state machine, this allows objects to export narrow interfaces that limit how faulty clients may modify objects [36]. A client's update of an object is conditioned on the object version last queried by the client. An update succeeds only if the object has not been modified since the client's last query in which case it is retried. Moreover, the Q/U protocol supports *multi-object updates* that atomically update a set of objects conditioned on their respective object versions.

The Q/U protocol operates correctly in an asynchronous model (i.e., no timing assumptions are necessary for safety), tolerates Byzantine faulty clients, and tolerates Byzantine faulty servers. Queries and updates in the Q/U protocol are strictly serializable [6]. In a benign execution (i.e., an execution in which components act according to specification or crash), queries and updates are obstruction-free [14]. The

*Carnegie Mellon University
†Network Appliance, Inc.

SOSP'05, October 23–26, 2005, Brighton, United Kingdom.
Copyright 2005 ACM 1-59593-079-5/05/0010 ...$5.00.

“cost” of achieving these properties with the Q/U protocol, relative to other approaches, is an increase in the number of required servers: the Q/U protocol requires $5b+1$ servers to tolerate b Byzantine faulty servers, whereas most agreement-based approaches require $3b+1$ servers. Given the observed performance of the Q/U protocol, we view this as a good trade-off — the cost of servers continues to decline, while the cost of service failures does not.

The Q/U protocol achieves its performance and fault-scalability through novel integration of a number of techniques. Optimism is enabled by the use of non-destructive updates at versioning servers, which permits operations to efficiently resolve contention and/or failed updates, e.g., by querying earlier object versions and completing partial updates. We leverage this versioning together with a logical timestamping scheme in which each update operation is assigned a timestamp that depends both on the contents of the update and the object state on which it is conditioned. It is thus impossible for a faulty client to submit different updates at the same timestamp—the updates intrinsically have different timestamps—and, so, reaching agreement on the update at a given timestamp is unnecessary. We combine these techniques with quorums and a strategy for accessing them using a *preferred quorum* per object so as to make server-to-server communication an exceptional case. Finally, we employ efficient cryptographic techniques. Our integration of these techniques has enabled, to our knowledge, the first fault-scalable, Byzantine-resilient implementation for arbitrary services.

We implemented a prototype library for the Q/U protocol. We used this library to build two services: a metadata service that exports NFSv3 metadata methods, and a counter object that exports increment (**increment**) and fetch (**fetch**) methods. Measurements of these prototype services support our claims: the prototype Q/U services are efficient and fault-scalable. In contrast, the throughput of BFT [9], a popular agreement-based Byzantine fault-tolerant replicated state machine implementation, drops sharply as the number of faults tolerated increases. The prototype Q/U-based counter outperforms a similar counter object implemented with BFT at all system sizes in contention-free experiments. More importantly, it is more fault-scalable. Whereas the performance of the Q/U-based counter object decreases by 36% as the number of faults tolerated is increased from one to five, the performance of the BFT-based counter object decreases by 83%.

2. EFFICIENCY AND SCALABILITY

In a Byzantine fault-tolerant quorum-based protocol (e.g., [26, 28, 10, 30, 41, 11]), only quorums (subsets) of servers process each request and server-to-server communication is generally avoided. In a Byzantine fault-tolerant agreement-based protocol (e.g., [7, 33, 18, 9, 8, 20]), on the other hand, every server processes each request and performs server-to-server broadcast. As such, these approaches exhibit fundamentally different fault-scalability characteristics.

The expected fault-scalability of quorum- and agreement-based approaches is illustrated in Figure 1. A protocol that is fault-scalable provides throughput that degrades gradually, if at all, as more server faults are tolerated. Because each server must process every request in an agreement-based protocol, increasing the number of servers cannot increase throughput. Indeed, since server-to-server broadcast is required, the useful work each server can do decreases as the number of servers increases. As illustrated, though, agreement-based protocols generally require fewer servers than quorum-based protocols for a given degree of fault-tolerance.

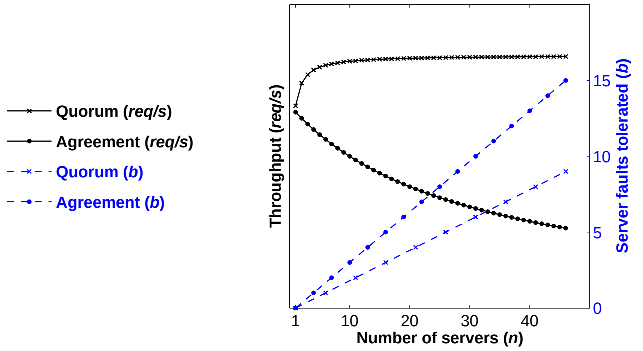

Figure 1: Illustration of fault-scalability. As the number of server faults tolerated increases (right axis, dashed lines), the number of servers required (x-axis) by quorum- and agreement-based protocols increases. In theory, throughput (left axis, solid lines) of quorum-based approaches (e.g., based on a threshold quorom) is sustained as more faults are tolerated, whereas agreement-based approaches do not have this property. In practice, throughput of prototype Q/U-based services degrades somewhat as more faults are tolerated.

With quorum-based protocols, as the number of server faults tolerated increases, so does the quorum size. As such, the work required of a client grows as quorum size increases. Servers do a similar amount of work per operation regardless of quorum size. However, the Q/U protocol does rely on some cryptographic techniques (i.e., *authenticators* which are discussed in §3) whose costs grow with quorum size.

2.1 Efficiency

Much of the efficiency of the Q/U protocol is due to its optimism. During failure- and concurrency-free periods, queries and updates occur in a single phase. To achieve *failure atomicity*, most pessimistic protocols employ at least two phases (e.g., a *prepare* and a *commit* phase). The optimistic approach to failure atomicity does not require a prepare phase; however, it does introduce the need for clients to repair (write-back) inconsistent objects. To achieve *concurrency atomicity*, most pessimistic protocols either rely on a central point of serialization (e.g., a primary) or employ locks (or leases) to suppress other updates to an object while the lock-holder queries and updates the object. The optimistic approach to concurrency atomicity [19] does not require lock acquisition, but does introduce the possibility that updates are rejected (and that clients may livelock).

The Q/U protocol relies on versioning servers for its optimism. Every update method that a versioning server invokes results in a new object version at that server. Queries complete in a single phase so long as all of the servers in the quorum contacted by a client share a common latest object version. Updates complete in a single phase so long as none of the servers in the quorum contacted by a client have updated the object since it was queried by the client. To promote such optimistic execution, clients introduce locality to their quorum accesses and cache object version information. Clients initially send requests to an object's *preferred quorum* and do not issue queries prior to updates for objects whose version information they cache. Finally, versioning servers reduce the cost of protecting against Byzantine faulty clients, since servers need not agree on client requests before processing them.

Many prior protocols, both pessimistic and optimistic, make use of versions and/or logical timestamps. One difference with prior protocols is that server retention of object versions is used to efficiently tolerate Byzantine faulty clients. Most protocols that tolerate Byzantine faulty clients rely on digital signatures or server-to-server broadcasts. Another difference is that there is no concept of a commit phase in the Q/U protocol (not even a lazy commit).

2.2 Throughput-scalability

The primary benefit that the Q/U protocol gains from the quorum-based approach is fault-scalability. Quorum-based protocols can also exhibit *throughput-scalability*: additional servers, beyond those necessary for providing the desired fault-tolerance, can increase throughput [32, 27]. The experience of database practitioners suggests that it may be difficult to take advantage of quorum-based throughput-scalability though. For example, Jiménez-Peris et al. recently concluded that a write-all read-one approach is better for a large range of database applications than a quorum-based approach [16]. But their analysis ignores concurrency control and is based on two phase commit-based data replication with fail-stop failures in a synchronous model. The Q/U protocol provides both concurrency and failure atomicity, provides service replication rather than data replication, relies on no synchrony assumptions, and relies on few failure assumptions. As another example, Gray et al. identify that the use of quorum-based data replication in databases leads to the *scaleup pitfall*: the higher the degree of replication, the higher the rate of deadlocks or reconciliations [12]. But, databases are not designed to scale up gracefully. The ability to update multiple objects atomically with the Q/U protocol allows services to be decomposed into *fine-grained* Q/U objects. Fine-grained objects reduce per-object contention, making optimistic execution more likely, enabling parallel execution of updates to distinct objects, and improving overall service throughput. If a service can be decomposed into fine-grained Q/U objects such that the majority of queries and updates are to individual objects, and the majority of multi-object updates span a small number of objects, then the scaleup pitfall can be avoided. Our experience building a Q/U-NFSv3 metadata service suggests that it is possible to build a substantial service comprised of fine-grained objects. For example, most metadata operations access a single object or two distinct objects. (The only operation that accesses more than two objects is the `RENAME` operation which accesses up to four objects.) In response to some criticisms of quorum-based approaches, Wool argues that quorum-based approaches are well-suited to large scale distributed systems that tolerate malicious faults [40]. Wool's arguments support our rationale for using a quorum-based approach to build fault-scalable, Byzantine fault-tolerant services.

Many recent fault-tolerant systems achieve throughput-scalability by partitioning the services and data structures they provide (e.g., [24, 13, 3, 34, 25, 38]). By partitioning different objects into different server groups, throughput scales with the addition of servers. However, to partition, these systems either forego the ability to perform operations that span objects (e.g., [24, 13, 34, 38]) or make use of a special protocol/service for "transactions" that span objects (e.g., [3, 25]). Ensuring the correctness and atomicity of operations that span partitions is complex and potentially quite expensive, especially in an asynchronous, Byzantine fault-tolerant manner. To clarify the problem with partitioning, consider a `RENAME` operation that moves files between directories in a metadata service. If different server groups are responsible for different directories, an inter-server group protocol is needed to atomically perform the `RENAME` operation (e.g., as is done in Farsite [3]). Partitioning a service across server groups introduces dependencies among server groups. Such dependencies necessarily reduce the reliability of the service, since many distinct server groups must be available simultaneously.

3. THE QUERY/UPDATE PROTOCOL

In this section, we begin with the system model and an overview of the Q/U protocol for individual objects. We discuss constraints on the quorum system needed to ensure the correctness of the Q/U protocol. We present detailed pseudo-code and discuss implementation details and design trade-offs. To make the Q/U protocol's operation more concrete, we describe an example system execution. We describe the extension of the Q/U protocol for individual objects to multiple objects. Finally, we discuss the correctness of the Q/U protocol.

3.1 System model

To ensure correctness under the broadest possible conditions, we make few assumptions about the operating environment. An asynchronous timing model is used; no assumptions are made about the duration of message transmission delays or the execution rates of clients and servers (except that they are non-zero and finite). Clients and servers may be Byzantine faulty [22]: they may exhibit arbitrary, potentially malicious, behavior. Clients and servers are assumed to be computationally bounded so that cryptographic primitives are effective. Servers have persistent storage that is durable through a crash and subsequent recovery.

The server failure model is a hybrid failure model [37] that combines Byzantine failures with crash-recovery failures (as defined by Aguilera et al. [4]). A server is *benign* if it is correct or if it follows its specification except for crashing and (potentially) recovering; otherwise, the server is *malevolent*. Since the Byzantine failure model is a strict generalization of the crash-recovery failure model, another term — malevolent — is used to categorize those servers that in fact exhibit out-of-specification, non-crash behavior. A server is *faulty* if it crashes and does not recover, crashes and recovers *ad infinitum*, or is malevolent.

We extend the definition of a fail prone system given in Malkhi and Reiter [26] to accommodate this hybrid failure model. We assume a universe U of servers such that $|U| = n$. The system is characterized by two sets: $\mathcal{T} \subseteq 2^U$ and $\mathcal{B} \subseteq 2^U$ (the notation 2^{set} denotes the power set of *set*). In any execution, all faulty servers are included in some $T \in \mathcal{T}$ and all malevolent servers are included in some $B \in \mathcal{B}$. It follows from the definitions of faulty and malevolent that $B \subseteq T$.

The Q/U protocol is a quorum-based protocol. A *quorum system* $\mathcal{Q} \subseteq 2^U$ is a non-empty set of subsets of U, every pair of which intersect; each $Q \in \mathcal{Q}$ is called a *quorum*. Constraints on the quorum system are described in §3.3. For simplicity, in the pseudo-code (§3.4) and the example execution (§3.6) presented in this paper, we focus on threshold quorums. With a threshold quorum system, a fail prone system can simply be described by bounds on the total number of faulty servers: there are no more than t faulty servers of which no more than $b \leq t$ are malevolent.

Point-to-point authenticated channels exist among all of the servers and between all of the clients and servers. An infrastructure for deploying shared symmetric keys among pairs of servers is assumed. Finally, channels are assumed to be unreliable, with the same properties as those used by Aguilera et al. in the crash-recovery model (i.e., channels do not create messages, channels may duplicate messages a finite number of times, and channels may drop messages a finite number of times) [4]. Such channels can be made reliable by repeated resends of requests.

3.2 Overview

This section overviews the the Q/U protocol for an individual object.

Terminology. Q/U objects are replicated at each of the n servers in the system. Q/U objects expose an operations-based interface of deterministic methods. Read-only methods are called *queries*, and methods that modify an object's state are called *updates*. Methods exported by Q/U objects take *arguments* and return *answers*. The words "operation" and "request" are used as follows: clients *perform operations* on an object by issuing *requests* to a *quorum* (subset) of servers. A server *receives* requests; if it *accepts* a request, it *invokes* a *method* on its local object replica.

Each time a server invokes an update method on its copy of the object, a new *object version* results. The server retains the object version as well as its associated *logical timestamp* in a version history called the *replica history*. Servers return replica histories to clients in response to requests.

A client stores replica histories returned by servers in its *object history set* (*OHS*), which is an array of replica histories indexed by server. In an asynchronous system, clients only receive responses from a subset of servers. As such, the OHS represents the client's partial observation of global system state at some point in time. Timestamps in the replica histories in the OHS are referred to as *candidates*. Candidates are *classified* to determine which object version a client method should be invoked on at the servers. Note that there is a corresponding object version for every candidate. Figure 2 illustrates candidates, replica histories, and the object history set.

Client side. Because of the optimistic nature of the Q/U protocol, some operations require more client steps to complete than others. In an optimistic execution, a client completes queries and updates in a single phase of communication with servers. In a non-optimistic execution, clients perform additional *repair* phases that deal with failures and contention.

The operations-based interface allows clients to send operations and receive answers from the service; this provides light-weight client-server communication relative to reading and writing entire objects. To perform an operation, a client first retrieves the object history set. A client's operation is said to *condition on* its object history set (the *conditioned-on OHS*). The client places the object history set, the method to invoke, and the arguments for the method in a request it sends to servers. By sending the object history set, the client communicates information to the servers about global system state.

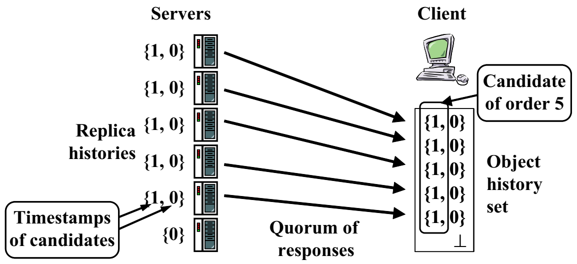

Figure 2: Example of client and server state.

Both clients and servers classify the conditioned-on OHS to determine which object version a client operation should be performed on at the servers. Classification of a candidate is based on the subset of server replica histories in the object history set in which it appears. (For threshold quorums, classification of a candidate is based on its *order*, the number of replica histories in the object history set in which it appears.) If all of the replica histories in the conditioned-on OHS have the same latest candidate, this candidate is classified as complete. This is illustrated in Figure 2.

The optimistic nature of the Q/U protocol allows a client to complete queries, and updates conditioned on a cached object history set, in a single phase of client-server communication during failure- and concurrency-free access. Clients cache object history sets. This allows clients to avoid retrieving an object history set before every update operation. So long as no other clients update the object, the cached object history set remains current. Only the quorum of servers that accepted the last update operation have the latest object version. As such, it is most efficient for these servers to process the subsequent update operation. To promote locality of access, and thus efficiency, each object has a *preferred quorum* at which clients try to access it first. In the case of server failures, clients may have to access a non-preferred quorum. Single phase operations are premised on the client's cached object history set being current (concurrency-free access) and on accessing an object via its preferred quorum (failure-free access).

Server side. Upon receipt of a request, a server first validates the integrity of the conditioned-on OHS. Each server pairs an *authenticator* with its replica history and clients include the authenticators in the conditioned-on OHS. Authenticators are lists of HMACs that prevent malevolent clients and servers from fabricating replica histories for benign servers. Servers cull replica histories from the conditioned-on OHS for which they cannot validate the authenticator. Authenticators are necessary because servers do not directly exchange replica histories with one another; replica histories are communicated between servers via the client.

If the conditioned-on OHS passes integrity validation, the server performs classification. Next, a server validates that the conditioned-on OHS is current. The server does this by comparing the timestamps of candidates in its replica history with the current timestamp classification identifies for the conditioned-on OHS. If the server has a higher timestamp in its replica history then the current timestamp identified by classification, the conditioned-on OHS is not current. Validating currentness ensures that servers invoke methods on the latest complete object version, rather than one of its predecessors.

If all validation passes, a server accepts the request and invokes the requested method. The method is invoked on the object version corresponding to the latest candidate in the conditioned-on OHS. If the method is an update, a new version of the object results. The timestamp for the resulting object version is a deterministic function of the conditioned-on OHS and the operation performed. As such, all servers that receive the same operation and object history set create the same object version and construct the same timestamp for that object version. Timestamps are constructed so that they always increase in value. The server updates its replica history with the resulting timestamp and stores the resulting object version indexed by that timestamp. The server sends a response to the client, indicating success, including the answer to the method, its replica history, and its authenticator. The client updates its object history set with the returned replica histories and authenticators. Once the client receives a quorum of responses that return success, the operation returns successfully.

Tolerating failures. If a server crashes, some quorums may become unavailable. This may lead to clients *probing* additional servers to collect a quorum of responses (probing is the term used to describe finding a live quorum). To protect against malevolent components, timestamps contain the client ID, the operation (method and arguments), and the conditioned-on OHS. Because the operation is tied to the timestamp, an operation can only complete successfully if a client sends the same operation to all servers in a quorum. This mechanism makes it impossible for malevolent clients to force object versions at different benign servers with the same timestamp to differ. The authenticators in the object history set make it impossible for malevolent components to forge replica histories of benign servers; in conjunction with classification rules (see §3.3), this ensures that benign servers only accept requests conditioned on the latest complete object version.

Concurrency and repair. Only one update of a specific object version can complete successfully. As such, concurrent accesses of an object may fail and result in replica histories at different servers that have a different latest candidate. In such a situation, a client must bring the servers into a consistent state. To do so, a client performs repair.

Repair consists of a sequence of two special operations: first a *barrier* is performed, and then a *copy* is performed. Barrier candidates have no data associated with them and so are safe for servers to accept during periods of contention. Indeed, if contention is indicated by the conditioned-on OHS sent to a server, the server can only accept a barrier candidate. Barrier operations allow clients to safely advance logical time in the face of contention; barriers prevent operations with earlier timestamps from completing. Once sufficient servers have accepted barriers, all contending operations have been suppressed. At this point the client can perform a *copy* operation. The copy operation copies the latest object version prior to the barrier forward in logical time. Such an object version is either the result of one of the contending operations that triggered repair or the object version these operations conditioned on. In the former case, none of the contending operations was successful, and in the latter case, one of the contending operations was successful. Once a copy operation successfully completes, then it is possible for another update or query to be performed.

Classification of the OHS dictates whether or not repair is necessary and, if so, whether a barrier or copy must be performed. Since both clients and servers base their actions on the OHS, both perform the action dictated by classification.

In the face of contention, clients may have to repeatedly perform barrier and copy operations: different clients could be copying different repairable candidates forward, never completing the copy operation. Client backoff is relied upon to promote progress when there is such contention. Since servers always return their latest replica history to the client, the client can, after updating its object history set, observe the effect of its operation.

Non-preferred quorum access. To be efficient, clients normally access an object via the object's preferred quorum. If a client accesses a non-preferred quorum, some servers that receive requests may not have the conditioned-on object version. In such a scenario, the server must *object sync* to retrieve the conditioned-on object version. The *sync-server* requests the needed object data from *host-servers*. The replica histories in the conditioned-on OHS provides the sync server with information about which other servers are host-servers. Responses from at least $b + 1$ host-servers are required for the sync-server to validate that the data it receives is correct.

3.3 Classification and constraints

In an asynchronous system with failures, clients can only wait for a subset of servers to reply. As such, object history sets are a partial observation of the global system state. Based on these partial observations, the latest candidate is classified as *complete*, *repairable*, or *incomplete*. With perfect global information, it would be possible to directly observe whether or not the latest candidate is *established*. An established candidate is one that is accepted at all of the benign servers in some quorum. Constraints on the quorum system, in conjunction with *classification rules*, ensure that established candidates are classified as repairable or complete. In turn, this ensures that updates are invoked on the latest object version.

Repairing a candidate, i.e., performing a barrier and then copying the candidate, is a fundamental aspect of the Q/U protocol. To allow us to state the classification rules and quorum intersection properties regarding repair in the Q/U protocol, we define *repairable sets*. Each quorum $Q \in \mathcal{Q}$ defines a set of repairable sets $\mathcal{R}(Q) \subseteq 2^Q$.

Given a set of server responses S that share the same candidate, the classification rules for that candidate are as follows:

$$\textbf{classify}(S) = \begin{cases} \textit{complete} & \text{if } \exists Q \in \mathcal{Q} : Q \subseteq S, \\ \textit{repairable} & \text{if } (\forall Q \in \mathcal{Q} : Q \not\subseteq S) \wedge \\ & (\exists Q \in \mathcal{Q}, R \in \mathcal{R}(Q) : \\ & R \subseteq S), \\ \textit{incomplete} & \text{otherwise.} \end{cases} \qquad (1)$$

Constraints on the quorum system that we require are as follows:

$$\forall Q \in \mathcal{Q}, \forall T \in \mathcal{T} : Q \cup T \subseteq U; \quad (2)$$

$$\forall Q_i, Q_j \in \mathcal{Q}, \forall B \in \mathcal{B}, \exists R \in \mathcal{R}(Q_i) : R \subseteq Q_i \cap Q_j \setminus B; \quad (3)$$

$$\forall Q_i, Q_j \in \mathcal{Q}, \forall B \in \mathcal{B}, \forall R \in \mathcal{R}(Q_j) : Q_i \cap R \not\subseteq B. \quad (4)$$

Constraint (2) ensures that operations at a quorum may complete in an asynchronous system. Constraint (3) ensures that some repairable set of an established candidate is contained in every other quorum. This constraint ensures that an established candidate is always classified as repairable or complete. For example, if a candidate is established at quorum Q_i, then a subsequent quorum access to Q_j is guaranteed to observe a repairable set R of Q_i despite any malevolent servers.

A candidate that, in some execution, could be classified as repairable is a *potential* candidate. Specifically, a potential candidate is one that is accepted at all of the benign servers in some repairable set. Constraint (4) ensures that an established candidate intersects a potential candidate at at least one benign servers. This constraint ensures that at most one of any concurrent updates in the Q/U protocol establishes a candidate and that, if a candidate is established, no other concurrent updates yield a potential candidate.

The logic that dictates which type of operation (method, barrier, or copy) must be performed is intimately connected with the classification rules and with the quorum system constraints. The constraints allow for multiple potential candidates. As such, there could be multiple distinct potential candidates conditioned on the same object version. However, if there are any potential candidates, constraint (4) precludes there from being any established candidates. Since none of these potential candidates can ever be established, the Q/U protocol requires barriers to safely make progress. Since barriers do not modify object state, they can be accepted at servers if the latest candidate is classified as incomplete or repairable. Once a barrier is established, it is safe to copy the latest object version forward (whether it is a potential candidate or an established candidate). And, if the copy establishes a candidate, then the corresponding object version can be conditioned on.

Classification and threshold quorums. Many quorum constructions can be used in conjunction with the Q/U protocol (e.g., those in [26, 28]). We have implemented threshold (majority) quorum constructions and recursive threshold quorum constructions from [28]. However, since the focus of this paper is on fault-scalability, we focus on the smallest possible threshold quorum construction for a given server fault model.

For the remainder of this paper, we consider a threshold quorum system in which all quorums $Q \in \mathcal{Q}$ are of size q, all repairable sets $R \in \mathcal{R}(Q)$ are of size r, all faulty server sets $T \in \mathcal{T}$ are of size t, all malevolent server sets $B \in \mathcal{B}$ are of size b, and the universe of servers is of size n. In such a system, (2) implies that $q + t \leq n$, (3) implies that $2q - b - n \geq r$, and (4) implies that $q + r - n > b$. Rearranging these inequalities allows us to identify the threshold quorum system of minimal size given some t and b:

$$n = 3t + 2b + 1;$$
$$q = 2t + 2b + 1;$$
$$r = t + b + 1.$$

As such, if $b = t$, then $n = 5b + 1$, $q = 4b + 1$, and $r = 2b + 1$. For example, if $b = 1$, then $n = 6$, $q = 5$, and $r = 3$. If $b = 4$, then $n = 21$, $q = 17$, and $r = 9$. For such a threshold quorum system, classification is based on the order of the candidate. The order is simply the number of servers that reply with the candidate. (This is illustrated in Figure 2.) The classification rules (1) for threshold quorums are as follows:

$$\textbf{classify}(\mathit{Order}) = \begin{cases} \mathit{complete} & \text{if } q \leq \mathit{Order}, \\ \mathit{repairable} & \text{if } r \leq \mathit{Order} < q, \\ \mathit{incomplete} & \text{otherwise.} \end{cases} \quad (5)$$

3.4 Q/U protocol pseudo-code

Pseudo-code for the Q/U protocol for a single object is given in Figure 3. To simplify the pseudo-code, it is written especially for threshold quorums. The symbols used in the pseudo-code are summarized in the caption. Structures, enumerations and types used in the pseudo-code are given on lines 100–107. The definition of a *Candidate* includes the conditioned-on timestamp LT_{CO}, even though it can be generated from $LT.\mathit{OHS}$. It is included because it clearly indicates the conditioned-on object version. The equality (=) and less than (<) operators are well-defined for timestamps; less than is based on comparing *Time*, *BarrierFlag* (with FALSE < TRUE), *ClientID*, *Operation* (lexigraphic comparison), and then *OHS* (lexigraphic comparison).

Clients initialize their object history set to a well-known value (line 200). Two example methods for a simple counter object are given in the pseudo-code: a query **c_fetch** and an update **c_increment**. The implementation of these two functions are similar, the only difference being the definition of the operation in each function (lines 300 and 400). First, requests based on the operation are issued to a quorum of servers (line 301). The OHS passed to **c_quorum_rpc** is the clients cached OHS. If this OHS is not current, servers reject the operation and return their latest replica history, in an effort to make the client's OHS current. If the operation is accepted at the quorum of servers (line 302), the function returns. Otherwise, the client goes into a repair and retry cycle until the operation is accepted at a quorum of servers (lines 302–306).

The details of the function **c_quorum_rpc** are elided; it issues requests to the preferred quorum of servers and, if necessary, probes additional servers until a quorum of responses is received. Because of the crash-recovery failure model and unreliable channels, **c_quorum_rpc** repeatedly sends requests until a quorum of responses is received. Server responses include a status (success or failure), an answer, and a replica history (see lines 1008, 1021, and 1030 of **s_request**). From the quorum of responses, those that indicate success are counted to determine the order of the candidate. The answer corresponding to the candidate is also identified, and the object history set is updated with the returned replica histories.

If an operation does not successfully complete, then repair is performed. Repair involves performing barrier and copy operations until classification identifies a complete object version that can be conditioned on. The function **c_repair** performs repair. To promote progress, **backoff** is called (line 502, pseudo-code not shown). Note that an operation that does not complete because its conditioned-on OHS is not current may not need to perform repair: after a client updates its OHS with server responses, classification may

```
structures, types, & enumerations:
100: $Class \in \{\textsc{query}, \textsc{update}\}$                    /* Enumeration. */
101: $Type \in \{\textsc{method}, \textsc{copy}, \textsc{barrier}\}$     /* Enumeration. */
102: $Operation \equiv \langle Method, Class, Argument\rangle$
103: $LT \equiv \langle Time, BarrierFlag, ClientID, Operation, OHS\rangle$
104: $Candidate \equiv \langle LT, LT_{\mathrm{CO}}\rangle$              /* Pair of timestamps. */
105: $RH \equiv \{Candidate\}$                   /* Ordered set of candidates. */
106: $\alpha \equiv \mathrm{HMAC}[U]$                    /* Array indexed by server. */
107: $OHS \equiv \langle RH, \alpha\rangle[U]$                /* Array indexed by server. */

c_initialize() :                                  /* Client initialization. */
200: $\forall s \in U, OHS[s].RH := \langle \mathbf{0}, \mathbf{0}\rangle, OHS[s].\alpha := \perp$

c_increment($Argument$):                  /* Example update operation. */
300: $Operation := \langle$increment$, \textsc{update}, Argument\rangle$
301: $\langle Answer, Order, OHS\rangle :=$ c_quorum_rpc($Operation, OHS$)
302: while ($Order < q$) do
303:    /* Repair and retry update until candidate established. */
304:    c_repair($OHS$)
305:    $\langle Answer, Order, OHS\rangle :=$ c_quorum_rpc($Operation, OHS$)
306: end while
307: return ($\langle Answer\rangle$)

c_fetch():                                  /* Example query operation. */
400: $Operation := \langle$fetch$, \textsc{query}, \perp\rangle$
401: $\langle Answer, Order, OHS\rangle :=$ c_quorum_rpc($Operation, OHS$)
402: while ($Order < q$) do
403:    c_repair($OHS$)
404:    $\langle Answer, Order, OHS\rangle :=$ c_quorum_rpc($Operation, OHS$)
405: end while
406: return ($\langle Answer\rangle$)

c_repair($OHS$):                  /* Deal with failures and contention. */
500: $\langle Type, \perp, \perp\rangle :=$ classify($OHS$)
501: while ($Type \neq \textsc{method}$) do
502:    backoff()                             /* Backoff to avoid livelock. */
503:    /* Perform a barrier or copy (depends on $OHS$). */
504:    $\langle \perp, \perp, OHS\rangle :=$ c_quorum_rpc($\perp, OHS$)
505:    $\langle Type, \perp, \perp\rangle :=$ classify($OHS$)
506: end while
507: return

c_quorum_rpc($Operation, OHS$) :                         /* Quorum RPC. */
600: /* Eliding details of sending to/probing for a quorum. */
601: /* Get quorum of s_request($Operation, OHS$) responses. */
602: /* For each response, update $OHS[s]$ based on $s.RH$. */
603: /* $Answer$ and $Order$ come from successful responses. */
604: return ($\langle Answer, Order, OHS\rangle$)

classify($OHS$):          /* Classify candidate in object history set. */
700: /* Determine latest object version and barrier version. */
701: $ObjCand :=$ latest_candidate($OHS, \textsc{false}$)
702: $BarCand :=$ latest_candidate($OHS, \textsc{true}$)
703: $LT_{\mathrm{lat}} :=$ latest_time($OHS$)
704: /* Determine which type of operation to perform. */
705: $Type := \textsc{barrier}$                            /* Default operation. */
706: /* If an established barrier is latest, perform copy. */
707: if ($LT_{\mathrm{lat}} = BarCand.LT$) $\wedge$ (order($BarCand, OHS$) $\geq q$)
708:       then $Type := \textsc{copy}$
709: /* If an established object is latest, perform method. */
710: if ($LT_{\mathrm{lat}} = ObjCand.LT$) $\wedge$ (order($ObjCand, OHS$) $\geq q$)
711:       then $Type := \textsc{method}$
712: return ($\langle Type, ObjCand, BarCand\rangle$)

order($Candidate, OHS$) :         /* Determine order of candidate. */
800: return ($|\{s \in U : Candidate \in OHS[s].RH\}|$)

s_initialize() :                                  /* Initialize server s. */
900: $s.RH := \{\langle \mathbf{0}, \mathbf{0}\rangle\}$
901: $\forall s' \in U, s.\alpha[s'] :=$ hmac($s, s', s.RH$)

s_request($Operation, OHS$) :         /* Handle request at server s. */
1000: $Answer := \perp$                    /* Initialize answer to return. */
1001: $\forall s' \in U$, if (hmac($s, s', OHS[s'].RH$) $\neq OHS[s'].\alpha[s]$)
1002:       then $OHS[s'].RH := \{\langle \mathbf{0}, \mathbf{0}\rangle\}$     /* Cull invalid RHs. */
1003: /* Setup candidate based on $Operation$ and $OHS$. */
1004: $\langle Type, \langle LT, LT_{\mathrm{CO}}\rangle, LT_{\mathrm{current}}\rangle :=$ s_setup($Operation, OHS$)
1005: /* Eliding details of receiving same request multiple times. */
1006: /* Determine if $OHS$ is current (return if not). */
1007: if (latest_time($s.RH$) $> LT_{\mathrm{current}}$)
1008:       then reply($s, \textsc{fail}, \perp, s.\langle RH, \alpha\rangle$)
1009: /* Retrieve conditioned-on object version. */
1010: if ($Type \in \{\textsc{method}, \textsc{copy}\}$) then
1011:    $Object :=$ retrieve($LT_{\mathrm{CO}}$)
1012:    /* Object sync if object version not stored locally. */
1013:    if (($Object = \perp$) $\wedge$ ($LT_{\mathrm{CO}} > \mathbf{0}$))
1014:       then $Object :=$ object_sync($LT_{\mathrm{CO}}$)
1015: end if
1016: /* Perform operation on conditioned-on object version. */
1017: if ($Type = \textsc{method}$) then
1018:    $\langle Object, Answer\rangle :=$                         /* Invoke method. */
1019:          $Operation.Method(Object, Operation.Argument)$
1020:    if ($Operation.Class = \textsc{query}$)
1021:       then reply($s, \textsc{success}, Answer, s.\langle RH, \alpha\rangle$)
1022: end if
1023: /* Update server state to include new object version. */
1024: atomic
1025:    $s.RH := s.RH \cup \{\langle LT, LT_{\mathrm{CO}}\rangle\}$
1026:    $\forall s' \in U, s.\alpha[s'] :=$ hmac($s, s', s.RH$)
1027:    store($LT, Object$)
1028:    /* Could prune replica history. */
1029: end atomic
1030: reply($s, \textsc{success}, Answer, s.\langle RH, \alpha\rangle$)

s_setup($Operation, OHS$) :                           /* Setup candidate. */
1100: $\langle Type, ObjCand, BarCand\rangle :=$ classify($OHS$)
1101: $LT_{\mathrm{CO}} = ObjCand.LT$
1102: $LT.Time :=$ latest_time($OHS$)$.Time + 1$
1103: $LT.ClientID := ClientID$
1104: $LT.OHS := OHS$
1105: if ($Type = \textsc{method}$) then
1106:    $LT.BarrierFlag := \textsc{false}$
1107:    $LT.Operation := Operation$
1108:    $LT_{\mathrm{current}} := ObjCand.LT$      /* ($= LT_{\mathrm{CO}}$) cf. line 1101 */
1109: else if ($Type = \textsc{barrier}$) then
1110:    $LT.BarrierFlag := \textsc{true}$
1111:    $LT.Operation := \perp$
1112:    $LT_{\mathrm{current}} := LT$
1113: else
1114:    $LT.BarrierFlag := \textsc{false}$                    /* $Type = \textsc{copy}$ */
1115:    $LT.Operation := LT_{\mathrm{CO}}.Operation$
1116:    $LT_{\mathrm{current}} := BarCand.LT$
1117: end if
1118: return ($Type, \langle LT, LT_{\mathrm{CO}}\rangle, LT_{\mathrm{current}}\rangle$)

latest_candidate($OHS, BarrierFlag$)
1200: /* Set of all repairable/complete objects (barriers). */
1201: $CandidateSet := \{Candidate :$ (order($Candidate, OHS$) $\geq r$)
1202:       $\wedge(Candidate.LT.BarrierFlag = BarrierFlag)\}$
1203: /* Repairable/complete $Candidate$ with max timestamp. */
1204: $Candidate := (Candidate : (Candidate \in CandidateSet)$
1205:       $\wedge(Candidate.LT = \mathbf{max}(CandidateSet.LT)))$
1206: return ($Candidate$)
```

Figure 3: Query/Update pseudo-code. Client functions are prefixed with c_ and server functions with s_. The following symbols are used in the pseudo-code: s (server), U (the universe of servers), OHS (object history set), RH (replica history), α (authenticator), LT (timestamp), LT_{CO} (conditioned-on timestamp), $\mathbf{0}$ (initial logical time), $\perp$ (null value), q (the threshold for classifying a candidate complete), and r (the threshold for classifying a candidate repairable).

indicate that the client perform a method. This happens if a client's cached OHS is out of date and the object is accessed contention-free. Classification of the OHS dictates whether a barrier or a copy is performed. In both cases, the client issues a **c_quorum_rpc** with the OHS. Once a copy successfully completes, a method is allowed to be performed and repair returns.

The function **classify** classifies an object history set: it identifies what type of operation (method, barrier, or copy) must be performed and the timestamps of the latest object and barrier versions. Clients and servers both call **classify**: a client calls it to determine the action dictated by its cached OHS and a server calls it to determine the action dictated by the conditioned-on OHS sent from a client. The function **latest_candidate** determines the latest candidate that is classified as either repairable or complete in the conditioned-on OHS is latest. It is called to identify the latest object version (line 701) and latest barrier version (line 702). The function **latest_time** (pseudo-code not shown) is called to determine the latest timestamp in the conditioned-on OHS. If the latest timestamp in the conditioned-on OHS is greater than the timestamps of the latest object and barrier versions, then the latest candidate is incomplete and a barrier must be performed (line 705). If the latest timestamp matches either the latest object or barrier version, but it is not classified as complete, then a barrier is performed. If the latest candidate is a complete barrier version, then a copy is performed (line 708). If the latest candidate is a complete object version, then a query or update is performed, conditioned on it (line 711).

Server initialization illustrates how authenticators are constructed via the **hmac** function (line 901). Each entry in an authenticator is an HMAC (keyed hash) over the server's replica history [5]. The two servers passed into **hmac** identify the shared key for taking the HMAC.

The function **s_request** processes requests at a server. First, the server validates the integrity of the conditioned-on OHS. Any replica histories with invalid authenticators are set to null. Next, the server calls **s_setup** to setup the candidate. It calls **classify** (line 1100) and sets various parts of the logical timestamp accordingly. In performing classification on the OHS, the server determines whether to perform a barrier, a copy, or a method operation. Given the setup candidate, the server determines if the object history set is current. If it is, the server attempts to fetch (via **retrieve** on line 1011) the conditioned-on object version from its local store. If it is not stored locally, the conditioned-on object version is retrieved from other servers (via **object_sync**, pseudo-code not shown, on line 1014). For queries and updates, the server invokes a method on the conditioned-on object version. If the method invoked is a query, then the server returns immediately, because the server state is not modified. If the method invoked is an update, then a new object version results. For update methods and for copy methods, the server adds the new candidate to its replica history, updates its authenticator, and locally stores (via **store**) the new object version indexed by timestamp. These three actions (lines 1025–1027) are performed atomically to be correct in the crash-recovery failure model. Finally, the server returns the answer and its updated replica history and authenticator.

3.5 Implementation details

This section discusses interesting aspects of system design, as well as implementation details and optimizations omitted from the pseudo-code.

Cached object history set. Clients cache object history sets of objects they access. After a client's first access, the client performs operations without first requesting replica histories. If the cached object history set is current, then the operation completes in a single phase. If not, then a server rejects the operation and returns its current replica history so that the client can make its object history set more current.

Optimistic query execution. If a client has not accessed an object recently (or ever), its cached OHS may not be current. It is still possible for a query to complete in a single phase. Servers, noting that the conditioned-on OHS is not current, invoke the query method on the latest object version they store. The server, in its response, indicates that the OHS is not current, but also includes the answer for the query invoked on the latest object version and its replica history. After the client has received a quorum of server responses, it performs classification on its object history set. Classification allows the client to determine if the query was invoked on the latest complete object version; if so the client returns the answer.

This optimization requires additional code in the function **s_request**, if the condition on line 1007 is true. To optimistically execute the query, the server retrieves the latest object version in its replica history and invokes the query on that object version (as is done on lines 1018 and 1019).

Quorum access strategy and probing. In traditional quorum-based protocols, clients access quorums randomly to minimize per-server load. In contrast, in the Q/U protocol, clients access an object's preferred quorum. Clients initially send requests to the preferred quorum. If responses from the preferred quorum are not received in a timely fashion, the client sends requests to additional servers. Accessing a non-preferred quorum requires that the client probe to collect a quorum of server responses. Servers contacted that are not in the preferred quorum will likely need to object sync.

In our system, all objects have an ID associated with them. In the current implementation, a deterministic function is used to map an object's ID to its preferred quorum. A simple policy of assigning preferred quorums based on the object ID modulo n is employed. Assuming objects are uniformly loaded (and object IDs are uniformly distributed), this approach uniformly loads servers.

The probing strategy implemented in the prototype is parameterized by the object ID and the set of servers which have not yet responded. Such a probing strategy maintains locality of quorum access if servers crash and disperses load among non-crashed servers.

Quorum-based techniques are often advocated for their ability to disperse load among servers and their throughput-scalability (e.g., [32, 40, 27]). Preferred quorums, because of the locality of access they induce, do not share in these properties if all operations are performed on a set of objects with a common preferred quorum. To address such a concern, the mapping between objects and preferred quorums could be made dynamic. This would allow traditional load balancing techniques to be employed that "migrate" loaded objects to distinct preferred quorums over time.

Inline repair. Repair requires that a barrier and copy operation be performed. *Inline repair* does not require a barrier or a copy; it repairs a candidate "in place" at its timestamp by completing the operation that yielded the candidate. Inline repair operations can thus complete in a single round trip. Inline repair is only possible if there is no *contention* for the object. Any timestamp in the object history set greater than that of the repairable candidate indicates contention. A server processes an inline repair request in the same manner as it would have processed the original update for the candidate (from the original client performing the update). Inline repair can also be performed on barrier candidates. Inline repair is useful in the face of server failures that lead to non-preferred quorum accesses. The first time an object is accessed after a server in its preferred quorum has failed, the client can likely perform inline repair at a non-preferred quorum.

Handling repeated requests at the server. A server may receive the same request multiple times. Two different situations lead to repeated requests. The crash-recovery failure model requires clients to repeatedly send requests to ensure a response is received. As such, servers may receive the same request from the same client multiple times. Additionally, inline repair may lead to different clients (each performing inline repair of the same candidate) sending the same request to the server. Regardless of how many times a server receives a requests, it must only invoke the method on the object once. Moreover, the server should respond to a repeated request in a similar manner each time. As such, it is necessary for the server to store the answer to update operations with the corresponding object version. This allows a server to reply with the same answer each time it receives a repeated request. Before checking if the conditioned-on OHS is current, the server checks to determine if a candidate is already in its replica history with the timestamp it setup (cf. line 1005). If so, the server retrieves the corresponding answer and returns immediately.

Retry and backoff policies. Update-update concurrency among clients leads to contention for an object. Clients must backoff to avoid livelock. A random exponential backoff policy is implemented (cf. [10]).

Update-query concurrency does not necessarily lead to contention. A query concurrent to a single update may observe a repairable or incomplete candidate. In the case of the former, the client performs an inline repair operation; such an operation does not contend with the update it is repairing (since servers can accept the same request multiple times). In the case of the latter, additional client logic is required, but it is possible to invoke a query on the latest established candidate, in spite of there being later incomplete candidates. On the client-side, after a query operation is deemed to have failed, additional classification is performed. If classification identifies that all of the candidates with timestamps later then the latest established candidate are incomplete, then the answer from invoking the query on the established candidate is returned. This situation can arise if an update and query are performed concurrently. Because of inline repair and this optimization, an object that is updated by a single client and queried by many other clients does not ever require barriers, copies, or backoff.

Object syncing. To perform an object sync, a server uses the conditioned-on object history set to identify $b+1$ servers from which to solicit the object state. Like accessing a preferred quorum, if $b+1$ responses are not forthcoming (or do not all match) additional servers are probed.

It is possible to trade-off network bandwidth for server computation: only a single correct server need send the entire object version state and other servers can send a collision-resistant hash of the object version state. A different approach is to include the hash of the object state in the timestamp. This requires servers to take the hash of each object version they create. However, it allows for servers to contact a single other server to complete object syncing or for the object state to be communicated via the client.

Authenticators. Authenticators consist of n HMACs, one for each server in the system. Authenticators are taken over the hash of a replica history, rather than over the entire replica history. The use of n HMACs in the authenticator, rather than a digital signature, is based on the scale of systems currently being evaluated. The size and computation cost of digital signatures do not grow with n. As such, if n were large enough, the computation and network costs of employing digital signatures would be less than that of HMACs.

Compact timestamps. The timestamps in the pseudo-code are quite large, since they include the operation and the object history set. In the implementation, a single hash over the operation and the object history set replaces these elements of logical timestamps. The operation and object history set are necessary to uniquely place a specific operation at a single unique point in logical time. Replacing these elements with a collision resistant hash serves this purpose and improves space-efficiency.

Compact replica histories. Servers need only return the portion of their replica histories with timestamps greater than the conditioned-on timestamp of the most recent update accepted by the server. As such, a server *prunes* its replica history based on the conditioned-on timestamp after it accepts an update request. (Although pruning pseudo-code is not shown, pruning would occur on line 1028.) In the common case, the replica history contains two candidates: one for the latest object version and one for the conditioned-on object version. Failures and concurrency may lead to additional barrier and incomplete candidates in the replica history.

Object versions corresponding to candidates pruned from a server's replica history could be deleted. Doing so would reclaim storage space on servers. However, deleting past object versions could make it impossible for a server to respond to slow repeated requests from clients and slow object sync requests from other servers. As such, there is a practical trade-off between reclaiming storage space on servers and ensuring that clients do not need to abort operations.

Malevolent components. As presented, the pseudo-code for the Q/U protocol ensures safety, but not progress, in the face of malevolent components. Malevolent components can affect the ability of correct clients to make progress. However, the Q/U protocol can be modified to ensure that isolated correct clients can make progress.

Malevolent clients may not follow the specified backoff policy. For contended objects, this is a form of denial-of-service attack. However, additional server-side code could rate limit clients.

Malevolent clients may issue updates only to subsets of a quorum. This results in latent work in the system, requiring correct clients to perform repair. Techniques such as *lazy verification* [1], in which the correctness of client operations is verified in the background and in which limits are placed on the amount of unverified work a client may inject in the system, can bound the impact that such malevolent clients have on performance.

A malevolent server can return an invalid authenticator to a client. Note that a client cannot validate any HMACs in an authenticator and each server can only validate a single HMAC in an authenticator. On line 1002, servers cull replica histories with invalid authenticators from the conditioned-on OHS. Servers return the list of replica histories (i.e., the list of invalid authenticators) culled from the conditioned-on OHS to the client. A server cannot tell whether a malevolent client or server corrupted the authenticator. A malevolent server can also return lists of valid authenticators to the client, indicating they are invalid. If a malevolent server corrupts more than b HMACs in its authenticator, then a client can conclude, from the responses of other servers, that the server is malevolent and exclude it from the set of servers it contacts. If a malevolent server corrupts fewer than b HMACs in its authenticator, then a client can conclude only that some servers in the quorum contacted are malevolent. This is sufficient for the client to solicit more responses from additional servers and make progress.

A malevolent server can reject a client update for not being current by forging a candidate with a higher timestamp in its replica history. Due to the constraints on the quorum system and the classification rules, malevolent servers cannot forge a potential or established candidate: any forged candidate is classifiable as incomplete. As such, the client must perform repair (i.e., a barrier then a copy). Consider a client acting in isolation. (Recall that contending clients can lead to justifiably rejected updates.) If an isolated client has its update accepted at all benign servers in a quorum, it by definition establishes a candidate (even though the client may not be able to classify the candidate as complete). Given that a server rejected its update, the client initiates repair with a barrier operation. Malevolent servers can only reject the isolated client's barrier operation by forging a barrier candidate with a higher timestamp. Any other action by the malevolent server is detectable as incorrect by the client. The Q/U protocol, as described in the pseudo-code, allows a malevolent server to keep generating barriers with higher timestamps. To prevent malevolent servers from undetectably repeatedly forging barrier candidates, classification is changed so that any barrier candidate not classified as complete is inline repaired. This bounds the number of barriers that can be accepted before a copy must be accepted to be the number of possible distinct incomplete candidates in the system (i.e., n). With such a modification, to remain undetected, a malevolent server must either accept the copied candidate or not respond to the client; either action allows the client to make progress. Since only potential candidates can be copied forward and potential candidates must intersect established candidates, malevolent servers cannot undetectably forge a copied candidate (given an established candidate prior to the barriers).

In summary, with the modifications outlined here, the Q/U protocol can ensure that isolated clients make progress in spite of malevolent components.

Pseudo-code and quorums. Extended pseudo-code is available in a companion technical report [2]. The extended pseudo-code includes optimistic query execution, additional client-side classification for queries to reduce query-update contention, inline repair of value candidates and barrier candidates, handling repeated requests at the server, pruning replica histories, deleting obsolete object versions, and object syncing.

The pseudo-code is tailored for threshold quorums to simplify its presentation. For general quorums that meet the constraints given in §3.3, changes to the pseudo-code are localized. The functions **c_quorum_rpc** and **order** must change to handle general quorums rather than threshold quorums. Any tests of whether a candidate is repairable or complete must also change to handle general quorums (e.g., lines 302, 402, 707, 710, and 1201).

3.6 Example Q/U protocol execution

An example execution of the Q/U protocol is given in Table 1. The caption explains the structure of, and notation used in, the table. The example is for an object that exports two methods: **get** (a query) and **set** (an update). The object can take on four distinct values (♣, ♢, ♡, ♠) and is initialized to ♡. The server configuration is based on the smallest quorum system for $b = 1$. Clients perform optimistic queries for the **get** method; the conditioned-on OHS sent with the **set** method is not shown in the table. The sequence of client operations is divided into four sequences of interest by horizontal double lines. For illustrative purposes, clients X and Z interact with the object's preferred quorum (the first five servers) and Y with a non-preferred quorum (the last five servers).

The first sequence demonstrates failure- and concurrency-free execution: client X performs a **get** and **set** that each complete in a single phase. Client Y performs a **get** in the second sequence that requires repair. Since there is no contention, Y performs inline repair. Server s_5 performs object syncing to process the inline repair request.

In the third sequence, concurrent updates by X and Y are attempted. Contention prevents either update from completing successfully. At server s_4, the **set** from Y arrives before the **set** from X. As such, it returns its replica history $\{\langle 3,1\rangle, \langle 1,\mathbf{0}\rangle\}$ with FAIL. The candidate $\langle 3,1\rangle$ in this replica history dictates that the timestamp of the barrier be $4\mathbf{b}$. For illustrative purposes, Y backs off. Client X subsequently completes barrier and copy operations.

In the fourth sequence, X crashes during a **set** operation and yields a potential candidate. The remaining clients Y and Z perform concurrent **get** operations at different quorums; this illustrates how a potential candidate can be classified as either repairable or incomplete. Y and Z concurrently attempt a barrier and inline repair respectively. In this example, Y establishes a barrier before Z's inline repair requests arrive at servers s_3 and s_4. Client Z aborts its inline repair operation. Subsequently, Y completes a copy operation and establishes a candidate $\langle 8,5\rangle$ that copies forward the established candidate $\langle 5,2\rangle$. Notice that the replica histories returned by the servers in response to the **get** requests are pruned. For example, server s_0 returns $\{\langle 6,5\rangle, \langle 5,2\rangle\}$, rather than $\{\langle 6,5\rangle, \langle 5,2\rangle, \langle 4\mathbf{b},2\rangle, \langle 2,1\rangle, \langle 1,\mathbf{0}\rangle, \langle \mathbf{0},\mathbf{0}\rangle\}$, because the object history set sent by X with **set** operation $\langle 6,5\rangle$ proved that $\langle 5,2\rangle$ was established.

3.7 Multi-object updates

We promote the decomposition of services into objects to promote concurrent non-contending accesses. The Q/U protocol allows some updates to span multiple objects. A multi-object update atomically updates a set of objects. To perform such an update, a client includes a conditioned-on OHS for each object being updated. The set of objects and each object's corresponding conditioned-on OHS, are referred to as the multi-object history set (*multi-OHS*). Each

Operation	s_0	s_1	s_2	s_3	s_4	s_5	**Result**
Initial system	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	$\langle \mathbf{0}, \mathbf{0} \rangle, \heartsuit$	
X completes **get**()	$\{\mathbf{0}\}, \heartsuit$	$\{\mathbf{0}\}, \heartsuit$	$\{\mathbf{0}\}, \heartsuit$	$\{\mathbf{0}\}, \heartsuit$	$\{\mathbf{0}\}, \heartsuit$		$\langle \mathbf{0}, \mathbf{0} \rangle$ complete, return $\heartsuit$
X completes **set**($\clubsuit$)	$\langle 1, \mathbf{0} \rangle, \clubsuit$	$\langle 1, \mathbf{0} \rangle, \clubsuit$	$\langle 1, \mathbf{0} \rangle, \clubsuit$	$\langle 1, \mathbf{0} \rangle, \clubsuit$	$\langle 1, \mathbf{0} \rangle, \clubsuit$		$\langle 1, \mathbf{0} \rangle$ established
Y begins **get**()...		$\{1, \mathbf{0}\}, \clubsuit$	$\{1, \mathbf{0}\}, \clubsuit$	$\{1, \mathbf{0}\}, \clubsuit$	$\{1, \mathbf{0}\}, \clubsuit$	$\{\mathbf{0}\}, \heartsuit$	$\langle 1, \mathbf{0} \rangle$ repairable
...Y performs **inline**						$\{1, \mathbf{0}\}, \clubsuit$	$\langle 1, \mathbf{0} \rangle$ complete, return $\clubsuit$
X attempts **set**($\diamondsuit$)...	$\langle 2, 1 \rangle, \diamondsuit$	$\langle 2, 1 \rangle, \diamondsuit$	$\langle 2, 1 \rangle, \diamondsuit$	$\langle 2, 1 \rangle, \diamondsuit$	FAIL		$\langle 2, 1 \rangle$ potential
Y attempts **set**($\heartsuit$)		FAIL	FAIL	FAIL	$\langle 3, 1 \rangle, \heartsuit$	$\langle 3, 1 \rangle, \heartsuit$	Y backs off
...X completes **barrier**	$\langle 4\mathbf{b}, 2 \rangle, \perp$	$\langle 4\mathbf{b}, 2 \rangle, \perp$	$\langle 4\mathbf{b}, 2 \rangle, \perp$	$\langle 4\mathbf{b}, 2 \rangle, \perp$	$\langle 4\mathbf{b}, 2 \rangle, \perp$		$\langle 4\mathbf{b}, 2 \rangle$ established
...X completes **copy**	$\langle 5, 2 \rangle, \diamondsuit$	$\langle 5, 2 \rangle, \diamondsuit$	$\langle 5, 2 \rangle, \diamondsuit$	$\langle 5, 2 \rangle, \diamondsuit$	$\langle 5, 2 \rangle, \diamondsuit$		$\langle 5, 2 \rangle$ established
X crashes in **set**($\spadesuit$)	$\langle 6, 5 \rangle, \spadesuit$	$\langle 6, 5 \rangle, \spadesuit$	$\langle 6, 5 \rangle, \spadesuit$				$\langle 6, 5 \rangle$ potential
Y begins **get**()...		$\{6, 5\}, \spadesuit$	$\{6, 5\}, \spadesuit$	$\{5, 4\mathbf{b}, 2, 1\}, \diamondsuit$	$\{5, 4\mathbf{b}, 2, 1\}, \diamondsuit$	$\{3, 1\}, \heartsuit$	$\langle 6, 5 \rangle$ inc., $\langle 5, 2 \rangle$ rep.
Z begins **get**()...	$\{6, 5\}, \spadesuit$	$\{6, 5\}, \spadesuit$	$\{6, 5\}, \spadesuit$	$\{5, 4\mathbf{b}, 2, 1\}, \diamondsuit$	$\{5, 4\mathbf{b}, 2, 1\}, \diamondsuit$		$\langle 6, 5 \rangle$ repairable
...Y completes **barrier**		$\langle 7\mathbf{b}, 5 \rangle, \perp$	$\langle 7\mathbf{b}, 5 \rangle, \perp$	$\langle 7\mathbf{b}, 5 \rangle, \perp$	$\langle 7\mathbf{b}, 5 \rangle, \perp$	$\langle 7\mathbf{b}, 5 \rangle, \perp$	$\langle 7\mathbf{b}, 5 \rangle$ established
...Z attempts **inline**				FAIL	FAIL		Z backs off
...Y completes **copy**		$\langle 8, 5 \rangle, \diamondsuit$	$\langle 8, 5 \rangle, \diamondsuit$	$\langle 8, 5 \rangle, \diamondsuit$	$\langle 8, 5 \rangle, \diamondsuit$	$\langle 8, 5 \rangle, \diamondsuit$	$\langle 8, 5 \rangle$ est., return $\diamondsuit$

Table 1: Example Q/U protocol execution. Operations performed by three clients (X, Y, and Z) are listed in the left column. The middle columns list candidates stored by and replies (replica histories or status codes) returned by six benign servers ($s_0, ..., s_5$). The right column lists if updates yield an established or potential candidate (assuming all servers are benign) and the results of classification for queries. Time "flows" from the top row to the bottom row. Candidates are denoted $\langle LT, LT_{\text{CO}} \rangle$. Only $LT.Time$ with "b" appended for barriers is shown for timestamps (i.e., client ID, operation, and object history set are not shown). Replica histories are denoted $\{LT_3, LT_2, ...\}$. Only the candidate's LT, not the LT_{CO}, is listed in the replica history.

server locks its local version of each object in the multi-OHS and validates that each conditioned-on OHS is current. The local locks are acquired in object ID order to avoid the possibility of local deadlocks. Assuming validation passes for all of the objects in the multi-OHS, the server accepts the update for all the objects in the multi-OHS simultaneously.

So long as a candidate is classified as complete or incomplete, no additional logic is required. However, to repair multi-object updates, clients must determine which objects are in the multi-OHS. As such, the multi-OHS is included in the timestamp. Note that all objects updated by a multi-object update have the same multi-OHS in their timestamp.

To illustrate multi-object repair, consider a multi-object update that updates two objects, o_a and o_b. The multi-object update results in a candidate for each object, c_a and c_b respectively. Now, consider a client that queries o_a and classifies c_a as repairable. To repair c_a, the client must fetch a current object history for o_b because o_b is in the multi-OHS of c_a. If c_a is in fact established, then c_b is also established and there could exist a subsequent established candidate at o_b that conditions on c_b. If c_a is not established, then c_b also is not established and subsequent operations at o_b may preclude c_b from ever being established (e.g., a barrier and a copy that establishes another candidate at o_b with a higher timestamp than c_b). The former requires that c_a be reclassified as complete. The latter requires that c_a be reclassified as incomplete.

Such reclassification of a repairable candidate, based on the objects in its multi-OHS, is called *classification by deduction*. If the repairable candidate lists other objects in its multi-OHS, then classification by deduction must be performed. If classification by deduction does not result in reclassification, then repair is performed. Repair, like in the case of individual objects, consists of a barrier and copy operation. The multi-OHS for multi-object barriers and multi-object copies have the same set of objects in them as the multi-OHS of the repairable candidate. Because multi-object operations are atomic, classification by deduction cannot yield conflicting reclassifications: either all of the objects in the multi-OHS are classified as repairable, some are classified complete (implying that all are complete), or some are classified incomplete (implying that all are incomplete).

Multi-object updates may span objects with different preferred quorums. The preferred quorum for one of the objects involved is selected for the multi-object update. In our implementation, we use the preferred quorum of the highest object ID in the multi-OHS. As such, some servers will likely have to object sync for some of the objects. Moreover, some of the candidates a multi-object write establishes may be outside of their preferred quorums. If there is no contention, such candidates are repairable at their preferred quorum via inline repair.

3.8 Correctness

This section discusses, at a high level, the safety and liveness properties of the Q/U protocol. In the companion technical report [2], the safety and liveness of the Q/U protocol is discussed in more detail, extended pseudo-code is provided, and a proof sketch of safety is given for a variant of the Q/U protocol, the Read/Conditional-Write protocol.

In the Q/U protocol, operations completed by correct clients are *strictly serializable* [6]. Operations occur atomically, including those that span multiple objects, and appear to occur at some point in time between when the operation begins and some client observes its effect. If the Q/U protocol is used only to implement individual objects, then correct clients' operations that complete are linearizable [15].

To understand how the Q/U protocol ensures strict serializability, consider the *conditioned-on chain*: the set of object versions that are found by traversing back in logical time from the latest established candidate via the conditioned-on timestamp. Every established object version is in the conditioned-on chain and the chain goes back to the initial candidate $\langle \mathbf{0}, \mathbf{0} \rangle$. The conditioned-on chain induces a total order on update operations that establish candidates, which ensures that all operations are strictly serializable. The conditioned-on chain for the example system execution in Table 1 is $\langle 8, 5\rangle \rightarrow \langle 5, 2\rangle \rightarrow \langle 2, 1\rangle \rightarrow \langle 1, \mathbf{0}\rangle \rightarrow \langle \mathbf{0}, \mathbf{0}\rangle$.

Given the crash-recovery server fault model for benign servers, progress cannot be made unless sufficient servers are up. This means that services implemented with the Q/U protocol are not available during network partitions, but become available and are correct once the network merges.

The liveness property of the Q/U protocol is fairly weak: it is possible for clients to make progress (complete operations). Under contention, operations (queries or updates) may abort so it is possible for clients to experience livelock. In a benign execution — an execution without malevolent clients or servers — the Q/U protocol is *obstruction-free* [14]: an isolated client can complete queries and updates in a finite number of steps. The extensions sketched in §3.5 also allow an isolated client to complete queries and updates in a finite number of steps in an execution with malevolent servers.

4. EVALUATION

This section evaluates the Q/U protocol as implemented in our prototype library. First, it compares the fault-scalability of a counter implemented with the Q/U protocol and one implemented with the publicly available[1] Byzantine fault-tolerant, agreement-based BFT library [9]. Second, it quantifies the costs associated with the authenticator mechanism and with non-optimal object accesses in the Q/U protocol. Third, it discusses the design and performance of an NFSv3 metadata service built with the Q/U protocol.

4.1 Experimental setup

The Q/U protocol prototype is implemented in C/C++ and runs on both Linux and Mac OS X. Client-server communication is implemented via RPCs over TCP/IP sockets. MD5 cryptographic hashes are employed [35]; the authors recognize that the MD5 hash is showing its age [39], however its use facilitates comparisons with other published results for BFT. All experiments are performed on a rack of 76 Intel Pentium 4 2.80 GHz computers, each with 1 GB of memory, and an Intel PRO/1000 NIC. The computers are connected via an HP ProCurve Switch 4140gl with a specified internal maximum bandwidth of 18.3 Gbps (or 35.7 mpps). The computers run Linux kernel 2.6.11.5 (Debian 1:3.3.4-3). Experiments are run for 30 seconds and measurements are taken during the middle 10 seconds; experiments are run 5 times and the mean is reported. The standard deviation for all results reported for the Q/U prototype is less than 3% of the mean (except, as noted, for the contention experiment). The working sets of objects for all experiments fit in memory, and no experiments incur any disk accesses.

4.2 Fault-scalability

We built a prototype counter object with the Q/U protocol and with the BFT library. The **increment** method, an update (read-write operation in BFT terminology), increments the counter and returns its new value. The **fetch** method, a query, simply returns the current value of the counter. A counter object, although it seems simple, requires the semantics these two protocols provide to correctly increment the current value. Moreover, the simplicity of the object allows us to focus on measuring the inherent network and computation costs of these Byzantine fault-tolerant protocols.

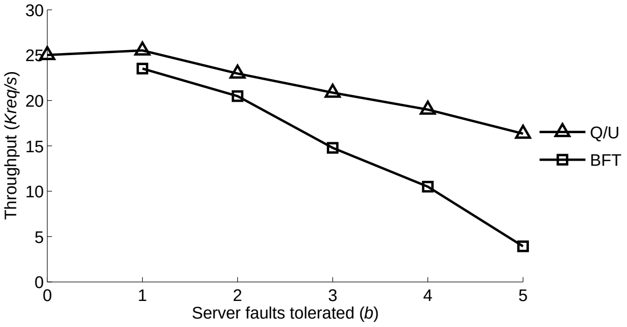

Figure 4: Fault-scalability.

We measure the throughput (in requests per second) of counter objects as the number of malevolent servers tolerated increases. Since both the Q/U protocol and BFT implement efficient, optimistic queries (read-only in BFT terminology), we focus on the cost of updates. These experiments are failure-free. For the Q/U-based counter, clients access counter objects via their preferred quorum and in isolation (each client accesses a different set of counter objects). We ran a single instance of the BFT-based counter so that there is a single primary replica that can make effective use of batching (an optimization discussed below). As such, this experiment compares best case performance for both protocols.

Figure 4 shows the fault-scalability of the Q/U-based and BFT-based counters. The throughput of **increment** operations per second, as the number of server faults tolerated increases from $b = 0$ to $b = 5$, is plotted. No data point is shown for the BFT-based counter at $b = 0$, a single server that tolerates no failures, because we had difficulties initializing the BFT library in this configuration. The increase in throughput from $b = 0$ to $b = 1$ for the Q/U-based counter is due to quorum throughput-scalability: at $b = 1$, each server processes only five out of every six update requests.

The data points shown in Figure 4 correspond to the *peak* throughput observed. To determine the peak throughput, we ran experiments with one physical client, then three physical clients, and so on, until we ran with thirty-three physical clients. Two client processes ran on each physical client; in all cases, this was sufficient to load both the Q/U-based and BFT-based counters. The throughput plotted for each value of b corresponds to that for the number of clients that exhibited the best throughput averaged over five runs. The BFT-based counter is less well-behaved under load than the Q/U-based counter: for each value of b, the performance of the BFT-based counter depends significantly on load. Throughput increases as load is added until peak throughput is achieved, then additional load reduces the observed throughput dramatically. For the Q/U-based counter, in contrast, peak throughput is maintained as additional load is applied. Due to the behavior of BFT under load, we report the peak throughput rather than the throughput observed for some static number of clients.

[1] `http://www.pmg.lcs.mit.edu/bft/#sw`

The Q/U-based counter provides higher throughput than the BFT-based counter in all cases. Both counters provide similar throughput when tolerating one or two faults. However, because of its fault-scalability, the Q/U-based counter, provides significantly better throughput than the BFT-based counter as the number of faults tolerated increases. When compared to the BFT-based counter, the Q/U-based counter provides 1990 more requests per second at $b = 1$ and 12400 more requests per second at $b = 5$. As b is increased from 1 to 5, the performance of the Q/U counter object degrades by 36%, whereas the performance of the BFT-based counter object degrades by 83%.

In BFT, server-to-server broadcast communication is required to reach agreement on each *batch* of requests. In batching requests, BFT amortizes the cost of generating and validating authenticators over several requests. For low values of b, batching is quite effective (fifteen or more requests per batch). As the number of faults tolerated increases, batching becomes less effective (just a few requests per batch at $b = 5$). Moreover, the cost of constructing and validating authenticators grows as the number of faults tolerated increases. The cost of authenticators grows slower for BFT than for the Q/U protocol, since BFT requires fewer servers to tolerate a given number of faults (as is illustrated in Figure 1, $3b + 1$ rather than $5b + 1$ servers).

BFT uses multicast to reduce the cost of server-to-server broadcast communication. However, multicast only reduces the number of messages servers must send, not how many they must receive (and authenticate). Since BFT employs multicast, all communication is based on UDP. We believe that the backoff and retry policies BFT uses with UDP multicast are not well-suited to high-throughput services.

4.3 Fault-scalability details

Authenticators and fault-scalability. Figure 1 in §2 illustrates the analytic expected throughput for the threshold quorums employed in the Q/U protocol prototype: $\frac{5b+1}{4b+1}\times$ the throughput provided by a single server. Our measurements for the Q/U-counter object in Figure 4 do not match that expectation. Throughput of the Q/U-counter declines (slowly) as the number of faults tolerated increases beyond one. The decline is due to the cost of authenticators.

As the number of faults tolerated increases, authenticators require more server computation to construct and validate, as well as more server bandwidth to send and receive. At $b = 1$, it takes $5.6\,\mu$s to construct or validate an authenticator, whereas at $b = 5$, it takes $17\,\mu$s. In Figure 5, we show the fault-scalability of the Q/U protocol for two hypothetical settings: one in which authenticators are not used at all (the "No α" line) and one in which HMACs require no computation to construct or validate (the "0-compute α" line). Compare these results with the "Q/U" line (the line also shown in Figure 4). The results for No α demonstrate the best fault-scalability that the Q/U protocol can achieve on our experimental infrastructure. Dedicated cryptographic processors or additional processing cores could someday realize the 0-compute α results. If the server computation required to construct and validate authenticators could be offloaded, the Q/U protocol would exhibit near ideal fault-scalability.

Cached object history sets. Figure 5 also shows the cost of accessing objects for which a client has a stale OHS or none at all. The difference between the "Q/U" and "Uncached OHS" lines is the additional round trip required to retrieve a current OHS. Even if the OHS must be read before performing an update, the Q/U protocol exhibits fault-scalability. For a given service implemented with the Q/U protocol, the usage pattern of objects will dictate whether most updates are based on a current cached OHS or not.

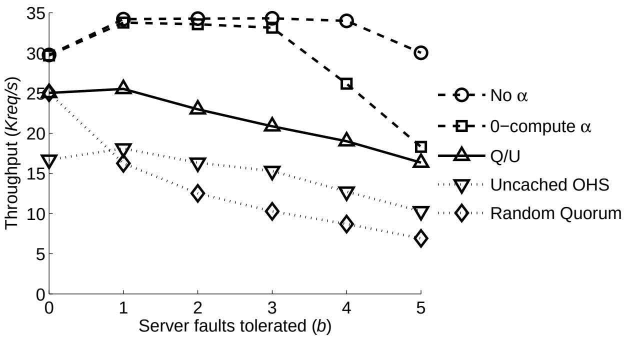

Figure 5: Details of Q/U protocol fault-scalability.

Non-preferred quorums. Accessing an object at a non-preferred quorum requires some servers to object sync. To quantify the benefit that preferred quorums provide, we implemented a random quorum access policy against which to compare. We measured the average response time for a single client process performing **increment** operations on a single counter object. For the case of $b = 1$ with $n = 6$, the response time using the preferred quorum access policy is $402\,\mu$s. Using the random quorum access policy, the average client response time is $598\,\mu$s. Because of the small quorum system size, the random policy issues updates to the quorum with the latest object version one in six times. For the other five in six, it takes a server $235\,\mu$s to perform an object sync. In the prototype implementation, to perform such an object sync if $b = 1$, a server retrieves the desired object version from two $(b + 1)$ other servers.

We also measured the peak throughput provided as the number of faults tolerated increases. The Random Quorum line in Figure 5 shows the results. Comparing the Random Quorum and Q/U lines, it is clear that the locality of access afforded by the preferred quorum access policy provides much efficiency to the Q/U protocol. As the number of faults tolerated increases, the cost of object syncing, given a random quorum access policy, increases for two reasons. First, the average number of servers that object sync increases. Second, the number of servers that must be contacted to object sync increases.

Contention. Clients accessing an object concurrently may lead to contention in the Q/U protocol. We ran a series of experiments to measure the impact of contention on client response time. In each of these experiments, $b = 1$ and five clients perform operations. In all cases, one client performs **increment** update operations to a shared counter object. In one set of experiments the other four clients perform **fetch** operations, and in another set, the other four clients also perform **increment** operations. Clients retrieve current replica histories before performing an **increment** operation; this emulates many distinct clients without a cached OHS contending for the counter object. Within a set of experiments, the number of contending clients that perform operations on the shared counter is varied from zero to four.

		Contending clients				
		0	**1**	**2**	**3**	**4**
fetch	**Isolated**	320	331	329	326	-
	Contending	-	348	336	339	361
increment	**Isolated**	693	709	700	692	-
	Contending	-	1210	2690	4930	11400

Table 2: Average response time in μs.

The number of isolated clients that perform operations on unshared counters is four minus the number of contending clients. Five clients are used in each experiment so that the load on servers is kept reasonably constant. The results of these experiments are reported in Table 2.

The **fetch** results demonstrate the effectiveness of the inline repair and optimistic query execution optimizations: queries do not generate contention with a single concurrent update. The results of the **increment** experiments show the impact of the backoff policy on response time. Our backoff policy starts with a $1000\,\mu$s backoff period and it doubles this period after each failed retry. As contention increases, the amount clients backoff, on average, increases. Not shown in Table 2, is that as update contention increases, the variance in per client response time also increases. For example, the standard deviation in client response time is $268\,\mu$s with one contending **increment** client and $4130\,\mu$s with four contending **increment** clients.

4.4 Q/U-NFSv3 metadata service

To explore a more complete service, we built a metadata service with the Q/U protocol. The Q/U-NFSv3 metadata service exports all namespace/directory operations of NFSv3. Two types of Q/U objects are implemented: *directory* objects and *attribute* objects. Attribute objects contain the per-file information expected by NFS clients. Directory attributes are stored in the directory object itself rather than in a separate attribute object, since they are needed by almost all exported methods on directories. Directory objects store the names of files and other directories. For each file in a directory, the directory object lists the object ID of the attribute object for the file. The metadata service is intended to only service namespace operations; this model of a metadata service separate from bulk storage is increasingly popular (e.g., both Farsite [3] and Pond/OceanStore [34] use this approach).

Table 3 lists the average response times for the operations that the Q/U-NFSv3 metadata service exports. These results are for a configuration of servers that tolerates a single malevolent server (i.e., $b = 1$ and $n = 6$). The operation type (query or update) and the types of objects accessed are listed in the table. The cost of a single-object query, relative to a single-object update, is illustrated by the difference in cost between **getattr** and **setattr**. The cost of a multi-object update, relative to a single object update, is illustrated by the difference in cost between **create** and **setattr**. Multi-object updates, such as **create** operations, may span objects with different preferred quorums. For the **create** operation, the multi-object update is sent to the preferred quorum of the directory object. As such, the first access of the created attribute object incurs the cost of an inline repair (if it does not share the directory object's preferred quorum).

Operation	**Type**	**Objects**	**Response time (μs)**
getattr	query	attribute	570
lookup	query	directory	586
readlink	query	directory	563
readdir	query	directory	586
setattr	update	attribute	624
create	update	attr. & dir.	718
link	update	attr. & dir.	690
unlink	update	attr. & dir.	686
rename	update	2 attr. & 2 dir.	780

Table 3: Q/U-NFSv3 metadata service operations.

Faults tolerated (b)	0	1	2	3	4	5
Transactions/second	129	114	102	93	84	76

Table 4: Q/U-NFSv3 PostMark benchmark.

Table 4 lists results of running the metadata-intensive PostMark benchmark [17] on a single client for different fault-tolerances (from $b = 0$ to $b = 5$). PostMark is configured for 5000 files and 20000 transactions. Since Q/U-NFSv3 only services namespace operations, file contents are stored locally at the client (i.e., the read and append phases of PostMark transactions are serviced locally). The decrease from $b = 0$ to $b = 1$ occurs because PostMark is single-threaded. As such, Q/U-NFSv3 incurs the cost of contacting more servers (and of larger authenticators) but does not benefit from quorum throughput-scalability. As with the counter object micro-benchmarks however, performance decreases gradually as the number of malevolent servers tolerated increases.

5. RELATED WORK

Much related work is discussed in the course of this paper. This section reviews other Byzantine fault-tolerant protocols that achieve comparable semantics to the Q/U protocol. As such, we do not focus on the extensive prior work implementing Byzantine fault-tolerant read/write registers.

Efficient cryptography. Castro and Liskov, in BFT [9], pioneered the use of lists of HMACs appended to messages as authenticators to implement a replicated state machine. This approach to integrity protection is often achieved via more expensive asymmetric cryptography (i.e., digital signatures). As mentioned in §3.5, the merit of using lists of HMACs for authenticators rather than digital signatures is based on the size of system being evaluated.

Byzantine faulty clients. In agreement-based protocols, servers reach agreement on the request sent by a client. Such protocols rely on clients either digitally signing requests or broadcasting their requests to all servers. Previous quorum-based protocols either require an *echo* phase to ensure that clients prepare the same update at each server in a quorum (e.g., [28]) or require all servers in a quorum to broadcast messages to all other servers (e.g., [30]). The Q/U protocol relies on servers retaining object versions ordered by timestamps that include (the hash of) operations and object history sets to protect against Byzantine faulty clients.

Agreement-based protocols. There is a long tradition of improving the efficiency of agreement-based protocols (e.g., [7, 33, 18, 9, 8, 20]). Cachin and Poritz, in the SINTRA project, use a randomized protocol to reach agreement that can be used to build services for WAN environments [8]. The BFT protocol of Castro and Liskov [9] provides comparable semantics and guarantees to the Q/U protocol and is quite responsive (more so than SINTRA). Services implemented with BFT and with the Q/U protocol rely on synchrony to guarantee progress (bounded message delay and backoff respectively), whereas those implemented with SINTRA do not. The public availability, responsiveness, and similarity in guarantees is our rationale for comparing the BFT prototype to the Q/U prototype. Many optimizations make BFT responsive: there is a fast path for read-only operations (like optimistic query execution); BFT reaches agreement on batches of requests to amortize the cost of agreement; and checkpoints, which require digital signatures, are further amortized over multiple batches of requests. Amortizing the cost of digital signatures and agreement over batches of messages was also by Reiter in Rampart [33] and Kihlstrom et al. in SecureRing [18]. Kursawe proposed an optimistic approach that requires one fewer phase of server-to-server broadcast communication, relative to other agreement-based protocols, during failure-free periods [20]. The recent FaB Paxos protocol of Martin and Alvisi also requires one fewer phase of communication, relative to other agreement-based protocols, during *stable periods* in which correct processes agree upon which correct process is currently the leader (stable periods are more general than failure-free periods) [29].

Quorum-based protocols. Byzantine quorum systems were introduced by Malkhi and Reiter [26], and several protocols have been proposed to use them for implementing arbitrary Byzantine-resilient services (e.g., [28, 10, 11]). To our knowledge, all such protocols are "pessimistic", utilizing multiple phases per update in the common case, and have designs substantially different than that of the Q/U protocol. Our work is the first to demonstrate, in empirical evaluations, a quorum-based protocol for implementing arbitrary Byzantine-resilient services that is competitive with modern implementations of state machine replication at small system sizes, and convincingly outperforms them as the system scales.

6. CONCLUSIONS

The Q/U protocol supports the implementation of arbitrary deterministic services that tolerate the Byzantine failures of clients and servers. The Q/U protocol achieves efficiency by a novel integration of techniques including versioning, quorums, optimism, and efficient use of cryptography. Measurements for a prototype service built using our protocol shows significantly better fault-scalability (performance as the number of faults tolerated increases) in comparison to the same service built using a popular replicated state machine implementation. In fact, in contention-free experiments, its performance is better at every number of faults tolerated: it provides 8% greater throughput when tolerating one Byzantine faulty server, and over four times greater throughput when tolerating five Byzantine faulty servers. Our experience using the Q/U protocol to build and experiment with a Byzantine fault-tolerant NFSv3 metadata service confirms that it is useful for creating substantial services.

Acknowledgements

This paper improved significantly from the detailed comments of the anonymous reviewers and the guidance of our shepherd, Miguel Castro. We thank Miguel Castro and Rodrigo Rodrigues for making the implementation of BFT publicly available, Gregg Economou for technical assistance, Charles Fry for invaluable feedback on our pseudo-code, and Dushyanth Narayanan for flexible internship work hours. We thank the CyLab Corporate Partners for their support and participation. This work is supported in part by Army Research Office grant number DAAD19-02-1-0389 and by Air Force Research Laboratory grant number FA8750-04-01-0238. We thank the members and companies of the PDL Consortium (including APC, EMC, Equallogic, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, and Sun) for their interest, insights, feedback, and support.

7. REFERENCES

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Lazy verification in fault-tolerant distributed storage systems. Symposium on Reliable Distributed Systems, 2005.

[2] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. *The Read/Conditional-Write and Query/Update protocols*. Technical report CMU–PDL–05–107. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, September 2005.

[3] 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. Symposium on Operating Systems Design and Implementation, pages 1–15. USENIX Association, 2002.

[4] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, **13**(2):99–125. Springer-Verlag, 2000.

[5] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. Advances in Cryptology - CRYPTO, pages 1–15. Springer-Verlag, 1996.

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

[7] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, **32**(4):824–840. ACM, October 1985.

[8] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the Internet. International Conference on Dependable Systems and Networks, pages 167–176. IEEE, 2002.

[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, **20**(4):398–461, November 2002.

[10] G. Chockler, D. Malkhi, and M. Reiter. Backoff protocols for distributed mutual exclusion and ordering. International Conference on Distributed Computing Systems, pages 11–20. IEEE, 2001.

[11] C. P. Fry and M. K. Reiter. Nested objects in a Byzantine quorum-replicated system. Symposium on Reliable Distributed Systems. IEEE, 2004.

[12] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. ACM SIGMOD International Conference on Management of Data. Published as *SIGMOD Record*, **25**(2):173–182. ACM, June 1996.

[13] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. Symposium on Operating Systems Design and Implementation, 2000.

[14] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: double-ended queues as an example. International Conference on Distributed Computing Systems, pages 522–529. IEEE, 2003.

[15] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, **12**(3):463–492. ACM, July 1990.

[16] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM Transactions on Database Systems (TODS)*, **28**(3):257–294. ACM, September 2003.

[17] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[18] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing group communication system. *ACM Transactions on Information and Systems Security*, **1**(4):371–406. IEEE, November 2001.

[19] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, **6**(2):213–226, June 1981.

[20] K. Kursawe. Optimistic Byzantine agreement. Symposium on Reliable Distributed Systems, pages 262–267. IEEE, 2002.

[21] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, **16**(2):133–169. ACM Press, May 1998.

[22] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, **4**(3):382–401. ACM, July 1982.

[23] L. L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, **2**:95–114, 1978.

[24] W. Litwin and T. Schwarz. LH*RS: a high-availability scalable distributed data structure using Reed Solomon Codes. ACM SIGMOD International Conference on Management of Data, pages 237–248. ACM, 2000.

[25] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: abstractions as the foundation for storage infrastructure. Symposium on Operating Systems Design and Implementation, pages 105–120. USENIX Association, 2004.

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

[27] D. Malkhi, M. Reiter, and A. Wool. The load and availability of Byzantine quorum systems. *SIAM Journal of Computing*, **29**(6):1889–1906. Society for Industrial and Applied Mathematics, April 2000.

[28] D. Malkhi and M. K. Reiter. An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, **12**(2):187–202. IEEE, April 2000.

[29] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. International Conference on Dependable Systems and Networks. IEEE, 2005.

[30] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. International Symposium on Distributed Computing, 2002.

[31] R. Morris. Storage: from atoms to people. Keynote address at Conference on File and Storage Technologies, January 2002.

[32] M. Naor and A. Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, **27**(2):423–447. SIAM, April 1998.

[33] M. K. Reiter. The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems* (Lecture Notes in Computer Science 938), pages 99–110, 1995.

[34] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. Conference on File and Storage Technologies. USENIX Association, 2003.

[35] R. L. Rivest. *The MD5 message-digest algorithm*, RFC–1321. Network Working Group, IETF, April 1992.

[36] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, **22**(4):299–319, December 1990.

[37] P. Thambidurai and Y.-K. Park. Interactive consistency with multiple failure modes. Symposium on Reliable Distributed Systems, pages 93–100. IEEE, 1988.

[38] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. Symposium on Operating Systems Design and Implementation, pages 91–104. USENIX Association, 2004.

[39] X. Wang, D. Feng, X. Lai, and H. Yu. *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*. Report 2004/199. Cryptology ePrint Archive, August 2004. `http://eprint.iacr.org/`.

[40] A. Wool. Quorum systems in replicated databases: science or fiction. *Bull. IEEE Technical Committee on Data Engineering*, **21**(4):3–11. IEEE, December 1998.

[41] L. Zhou, F. B. Schneider, and R. V. Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, **20**(4):329–368. ACM, November 2002.