Dual-Quorum: A Highly Available and Consistent Replication System for Edge Services

Lei Gao, *Member, IEEE,* Mike Dahlin, *Senior Member, IEEE,*
Jiandan Zheng, *Member, IEEE,* Lorenzo Alvisi, *Senior Member, IEEE,*
and Arun Iyengar *Senior Member, IEEE*

Abstract

This article introduces dual-quorum replication, a novel data replication algorithm designed to support Internet edge services. Edge services allow clients to access Internet services via distributed edge servers that operate on a shared collection of underlying data. Although it is generally difficult to share data while providing high availability, good performance, and strong consistency, replication algorithms designed for specific access patterns can offer nearly ideal trade-offs among these metrics. In this article, we focus on the key problem of sharing read/write data objects across a collection of edge servers when the references to each object (a) tend not to exhibit high concurrency across multiple nodes and (b) tend to exhibit bursts of read-dominated or write-dominated behavior. Dual-quorum replication combines volume leases and quorum based techniques to achieve excellent availability, response time, and consistency for such workloads. In particular, through both analytical and experimental evaluation, we show that the dual-quorum protocol can (for the workloads of interest) approach the optimal performance and availability of Read-One/Write-All-Asynchronously (ROWA-A) epidemic algorithms without suffering the weak consistency guarantees and resulting design complexity inherent in ROWA-A systems.

Index Terms

Reliability, availability, and serviceability, performance, distributed system, leases, volume leases, client/server and multitier systems, data replication, quorum system.

L. Gao is with Oracle Inc., M. Dahlin, J. Zheng, and L. Alvisi are with University of Texas Austin, A. Iyengar is with IBM T.J. Watson Research Center.

1 INTRODUCTION

This article introduces dual-quorum replication, a novel data replication algorithm motivated by the desire to support data replication for edge services [1], [2], [3]. As Figure 1 illustrates, the Internet edge service architecture attempts to improve service availability and latency by allowing clients to access the closest available edge server rather than a centralized server or a centralized server cluster. The success of various Content Delivery Networks (CDNs) [4], [5], [6] has shown the promise of this architecture [7], [8]. But as Figure 1 also indicates, to provide a single service from multiple locations, service logic (code) replicated on all edge servers must access a collection of shared data. As a result, the benefits promised by the edge service architecture are limited by the coordination among replicas of shared data. Thus, support for data replication is a key problem in realizing the promise of Internet edge services.

Providing high availability, good performance, and strong consistency for replicated data is fundamentally hard in the general case [9], [10]. On one hand, an edge server ideally would process both reads and writes with local data to offer good service response time and availability; when an edge server has to contact distant servers to process client requests, it loses many of the advantages offered by an edge service architecture. On the other hand, applications using the edge service model desire strong consistency guarantees across their shared data. Distributed applications that assume only weak consistency guarantees must be designed to address subtle consistency issues such as write-write conflicts and staleness bounds [11]. Consequently, the complexity of building, debugging, maintaining, and updating such applications increases dramatically, which is unacceptable for most Internet services. As a result, current edge server deployment often serve only read-only data.

By exploiting object-specific workload characteristics, we seek to design a data replication system for more general edge services by offering optimized trade-offs among availability, consistency, and response time. For example, our previous studies show how to provide nearly optimal replication for *information dissemination* applications such as news [12] and for *e-commerce* applications such as TPC-W [2], an industry standard benchmark that models an online bookstore [13]. In this prior work, we developed customized consistency protocols for three categories of objects: (1) single-writer, multi-reader objects like product descriptions and prices; (2) multi-writer, single-reader objects like lists of orders; and (3) commutative-write, approximate-read objects like the current inventory count of each product.

However, a key limitation of our previous efforts to support edge services was our decision

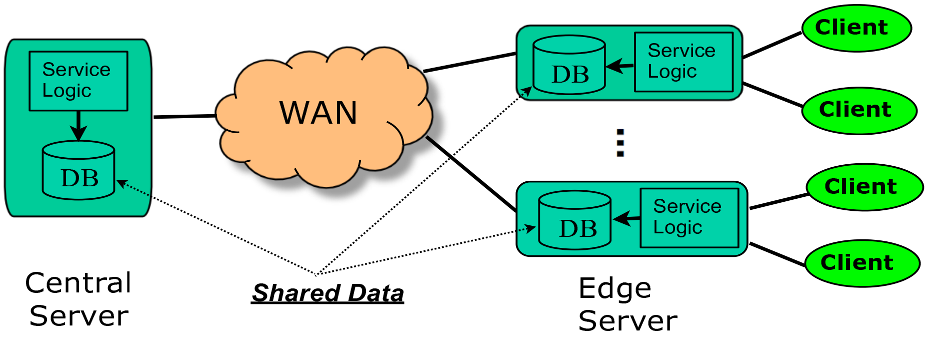


Fig. 1. The edge service architecture

to use weak consistency—and thereby introduce considerable complexity—for the fourth category of objects: multi-writer, multi-reader objects such as TPC-W's per-customer *profile* information. We made use of a Read-One, Write-All-Asynchronously (ROWA-A) protocol [14], [15], [16] that asynchronously propagated writes epidemically and allowed any server to return local copies for read requests. ROWA-A protocols provide excellent read performance and availability but allow applications to observe inconsistencies between reads and writes or among writes. Such inconsistencies introduce considerable complexity into the application design, because all cases must be handled correctly no matter how rare they are and because reasoning about corner cases in consistency protocols is complex. Furthermore, these protocols provide no worst-case bound on staleness, i.e., it is possible for a read to return stale data arbitrarily long after a write, which can be unacceptable for some applications [17].

This paper introduces a new protocol, dual-quorum replication (DQ), to better meet the demands edge services place on such multi-reader multi-writer objects. On one hand, DQ attempts to approach the ideal read performance and availability of ROWA-A protocols. At the same time, the protocol simplifies the application design by greatly strengthening consistency and staleness guarantees compared to ROWA-A.

Achieving strong consistency and staleness guarantees is generally expensive. However, DQ is optimized for workloads that exhibit locality in two dimensions: (1) at any given time access to a given element tends to come from a single server and (2) reads tend to be followed by other reads and writes tend to be followed by other writes. For this type of workloads, DQ approaches the excellent performance and availability of ROWA-A protocols. For other workloads, our algorithm continues to provide the same consistency semantics, but its performance and availability may degrade.

Dual-quorum replication achieves these goals by implementing two key ideas:

- First, we devote two separate quorum systems, an input quorum system (Q_{input}) and an output quorum system (Q_{output}), for write and read requests, respectively, to optimize both write and read's availability and performance. Because traditional quorum systems require each read quorum to intersect each write quorum to provide regular semantics [18], a small read (write) quorum implies a large write (read) quorum; there is thus a tradeoff between read availability and write availability. In dual-quorum, instead of constructing read quorums and write quorums from the same quorum system, clients send their writes to a write quorum formed in Q_{input} and they read from a read quorum in Q_{output}. These two quorums do not need to intersect to enforce regular semantics; instead, regular semantics are enforced by communication between the read quorum in Q_{input} and the write quorum in Q_{output}. By using two separate quorum systems for reads and writes, DQ is able to optimize the construction of Q_{output}'s read quorum to provide low latency and high availability for reads while optimizing the construction of Q_{input}'s write quorum to provide modest overhead and high availability for writes.
- Second, dual-quorum generalizes Yin et al.'s volume lease protocol [19] to reduce the communication overhead between Q_{input} and Q_{output} to enforce consistency and improve write availability. A volume lease is a lease for a group of objects. The Q_{input} servers use volume leases to invalidate cached objects at the Q_{output} servers as objects are updated and to allow writes to continue without invalidating cached objects when leases expire. The protocol uses short-duration volume leases to allow writes to complete despite network partitions, and it aggregates these leases across a large number of objects in a volume to amortize the cost of renewing short leases.

Using our dual-quorum protocol, workloads with a large number of repeated reads (or writes) perform well because reads (or writes) can often be supplied by a read-optimized Q_{output} read quorum (or write-optimized Q_{input} write quorum) without requiring communication with the Q_{input} (or Q_{output}).

Through both analytical and experimental evaluations, we compare the availability, response time, communication overhead, and consistency guarantees of the dual-quorum protocol against other popular replication protocols: the synchronous and asynchronous Read-One/Write-All (ROWA) protocol family [20], a majority quorum system [21], and a grid quorum system [22]. For the important special case of single-server Q_{output} read quorum, average read response time can approach a server's local read time, making the read performance of this approach competitive with ROWA-A epidemic algorithms such as Bayou [23], but the dual quorum approach avoids suffering the weak consistency guarantees and resulting complexity inherent in ROWA-A designs. Additionally, analytical evaluations show that the overall availability of the dual-quorum protocol is competitive with the ideal majority quorum protocol for the targeted workloads. Finally, for the targeted workloads, the communication overheads of this approach are comparable to existing approaches. However, in the worst-case scenario in which the workload consists of only interleaved reads and writes, the dual-quorum protocol requires significantly more message exchanges than traditional quorum protocols to coordinate the separate input and output quorum systems. This communication overhead for low-locality workloads is the cost that the dual-quorum protocol pays to provide the availability, response time, and strong consistency desired for an Internet edge service environment.

The main contribution of this article is to introduce the dual-quorum algorithm, a novel data replication algorithm targeted to a key workload for Internet edge service environments. Note that although our work is motivated by a specific replication scenario, we speculate that it will be more generally useful. In particular, we believe that it may be common in practice for systems that can have any server read or write any item of data to experience sufficient locality to benefit from our approach.

This article is organized as follows. Section 2 presents our system model and a set of assumptions on which our system is built. In Section 3, we present our system's design and correctness proofs. We compare our system with existing ones in Section 4 with both analytical and experimental evaluations. In Section 5, we discuss related work. Concluding remarks are presented in Section 6.

2 SYSTEM MODEL AND DEFINITIONS

As Figure 2 illustrates, in order to provide reliable services for multiple-writer multiple-reader objects, our edge service environment removes the central server and constructs the edge servers such that each physical server plays one or more of the following three roles: (a) *front end* servers that handle *service client* requests from across the Internet, execute application-specific processing, and act as *edge server clients* or just *clients* to the dual-quorum storage system; (b) *Output Quorum System* (Q_{output}) servers that process client read requests; and (c) *Input Quorum System* (Q_{input}) servers that process client write requests. We assume a *request redirection architecture* that directs

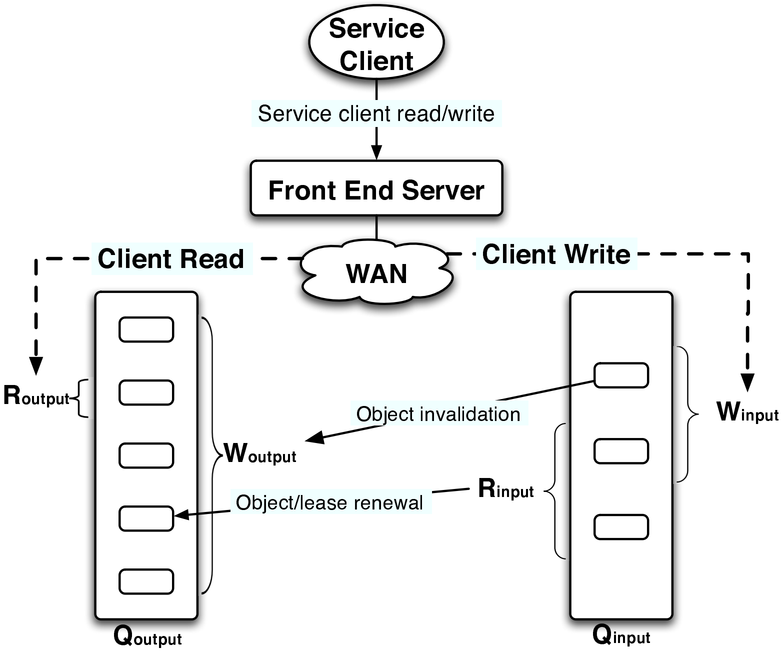


Fig. 2. Edge service system model

clients to a good (e.g., nearby, lightly loaded, or available) front end edge server; a number of suitable redirection systems are discussed in the literature [24], [8]. Note that service clients are unaware of the underlying data storage system and never contact the Q_{output} or Q_{input} interfaces directly.

In an edge service environment, servers typically process sensitive or valuable information, so they must run on trusted machines such as dedicated servers in a hosting center. We therefore assume a fail-stop model in which servers may crash but can not issue incorrect requests or replies. We assume secure communication among servers and that if the network corrupts a message, this corruption is detected by low-level checksums and the message is silently discarded. Each server can read a local real-time clock and there exists a maximum drift rate *maxDrift* between any pair of clocks. The network may delay, duplicate, or reorder messages.

As long as the clock drifts across servers are bounded as described above, our protocol ensures safety regardless of other timing assumptions: servers may operate at arbitrarily different speeds and we require no bound on message delivery delay. However, long processing times or message delays may interfere with liveness for some requests. In particular, if machine A requests a lease at time t_0 and later receives a reply from server B granting a lease of length T, then A conservatively expires the lease at time $t_0 + (1 - maxDrift)T$; this approach ensures that the receiver of a lease

(A) expires the lease no later than the grantor of the lease (B).

We adopt Lamport's register semantic definitions [18]. Two operations are considered concurrent if one starts after the other starts and before it ends. DQ enforces *regular semantics*:

- Property 1: A read of o that is not concurrent with any writes of o can return only the value and logical clock from the completed write of o with the highest logical clock; and
- Property 2: A read of o that is concurrent with one or more writes of o can (a) return the value and logical clock from the completed write of o with the highest logical clock or (b) return the value and logical clock from some concurrent write of o.

Regular semantics guarantee that a read always returns the last completed write or any concurrent partially completed write. We discuss the challenges to adapting the protocol to enforce the stronger *atomic semantics* [18] where reads and writes behave as if they occur instantaneously in some definite order in section 3.3.

In the remaining sections, we describe interactions with a quorum system in terms of a *QRPC operation* [25]. A QRPC operation QRPC($system$, $R|W$, $request$) sends $request$ to a collection of servers in the specified quorum $system$ (e.g., Q_{input} or Q_{output}). The QRPC call then blocks until a set of $replies$ constituting the specified quorum (read quorum if the second parameter is R, or write quorum otherwise) on the specified $system$ have been gathered. The call then returns the set of $replies$ that it received. The QRPC operator abstracts away details of selecting a quorum, retransmissions, and timeouts. In particular, different implementations may choose different ways to select which servers from $system$ to send requests to, and they may select different retransmission strategies: our simple prototype implementation always transmits requests to the local server if it is a member of $system$; it then randomly selects a sufficient number of additional servers to form a read or write quorum and transmits the request to them; retransmissions are each sent to a new randomly selected quorum using an exponentially-increasing retransmission interval. A more aggressive implementation might send to all servers in $system$ and return when the fastest quorum has responded or might track which servers have responded quickly in the past and first try sending to them.

3 Dual-Quorum Protocol Design

This section describes the design of the dual-quorum replication system and the key ideas for achieving our design goals.

We present the protocol in two steps. First, we discuss a simplified asynchronous dual-quorum protocol (ADQ) in Section 3.1. This protocol allows independent optimizations of read and write quorums, but because it assumes an asynchronous system model, a write can block for an arbitrarily long period of time. In Section 3.2 we describe how we introduce volume leases to the protocol to improve write availability while retaining good read performance. Finally, we discuss correctness.

3.1 Asynchronous dual-quorum protocol

The goal of ADQ is to achieve highly-available, low-latency, and consistent data replication for a range of Internet services that exhibit the following characteristics: (1) end clients are widely dispersed and generate read-dominant or write-dominant workloads; (2) a subset of servers may unpredictably fail or be partitioned from the rest of the system; and (3) applications require relatively strong consistency. Therefore we require the protocol to provide regular semantics, optimize read/write performance in normal non-faulty cases, and optimize the read and write availability to survive fail-stop node failures or network partitions.

Quorum-based protocols seem a natural choice for providing the consistency semantics required, but there is a tradeoff between read availability and write availability due to the intersection requirements for read quorums and write quorums. If we use a traditional quorum protocol and make the read quorum large enough to provide good write availability, read performance will be unacceptable because reads will be WAN-distributed rather than local operations.

To address this dilemma, ADQ processes reads and writes in two different quorum systems (Q_{input} and Q_{output}) and uses a cache invalidation strategy to synchronize the state of objects replicated in Q_{input} servers and cached in Q_{output} servers to achieve regular semantics. The key challenge is how to efficiently maintain callbacks in Q_{input} and Q_{output} to reduce the synchronization traffic between them.

In the rest of this subsection, we will describe the basic read/write operations followed by detailed description of the object invalidation and renewal protocol.

Basic read and write operations. From the front end server's perspective, an ADQ read is the same as a standard quorum read [26], [27]. As Figure 2 illustrates, upon receiving a read request from a client, the server contacts a read quorum R_{output} of the output quorum system Q_{output}. A R_{output} server can return a read immediately if it holds a valid copy of the object. We call this case

	Variable	Meaning
Q_{output} (j)	$valid_{o,i}$	Is $true$ if j still has a valid copy from a Q_{input} server i.
	$lastKnown_{o,i}$	The highest version number of o learned from a Q_{input} server i.
	$value_o$	Newest local copy of o including a value and a corresponding version number.
Q_{input} (i)	$lastRead_o$	The last version of o that i has sent to any Q_{output} server.
	$lastAck_{o,j}$	The last invalidation acknowledgement for o from a Q_{output} server j.
	$value_o$	Local known newest value of o and its version number.
	lc	Lamport logical clock to generate version numbers for all objects.

Fig. 3. Data structures on each Q_{output} and Q_{input} server for object o.

a *read hit*. Otherwise, it must renew the object by communicating with a read quorum R_{input} of the input quorum system. We call this case a *read miss*.

Upon receiving a write request from a client, the server contacts every server in a write quorum W_{input} of the input quorum system Q_{input}. Just like in the standard quorum write protocol, the ADQ write has two phases. First, a server i that receives the client's write request retrieves the highest logical clock from every server in a R_{input} via $QRPC$. Then, the server advances the logical clock and assigns it along with its unique id as the write version number. Second, the server sends the write request with the version number to a W_{input} quorum via $QRPC$. The write completes after i receives acknowledgments from every server in a W_{input} quorum. If a Q_{input} server knows that there is no R_{output} quorum that has a valid copy in each server, it can perform the write and send an acknowledgement to i immediately, a case that we call a *write suppress*. Otherwise, the Q_{input} server must first invalidate a W_{output} quorum. We call this case *write through*.

Now the questions are: how does a Q_{output} server know that its local object is valid; how does it renew it if not; when does a Q_{input} server need to send invalidate messages to Q_{output}, and how does it do so? We will answer these questions in the next few paragraphs by first detailing how the system handles a read and then describing how the system handles a write.

Read hit and read miss. In order to ensure that reads always return versions of objects consistent with recent writes, as Figure 3 illustrates, each server maintains a set of per-object and per-server variables. Each Q_{input} server maintains a Lamport logical clock lc for generating version numbers for writes. Both Q_{output} and Q_{input} servers store the newest local copy of an object o in $value_o$ for local reads and writes. $value_o$ includes a value and a version number. To filter redundant or old invalidations or updates, each Q_{output} server j maintains $lastKnown_{o,i}, \forall i,\ i \in Q_{input}$ as the highest version number of o for which an invalidation or an update has been received from a Q_{input}

server i. To track the validity of a local cache, each Q_{output} server j uses $valid_{o,i}, \forall i,\ i \in Q_{input}$ to indicate if j still has a valid local copy from i. $valid_{o,i}$ is true if and only if the newest value received from i is at least as new as $lastKnown_{o,i}$. To track the callback states of Q_{output}, each Q_{input} server maintains a pair of variables: $lastRead_o$ and $lastAck_{o,j}, \forall j,\ j \in Q_{output}$. $lastRead_o$ stores the newest version of o that i has sent to any Q_{output} server; $lastAck_{o,j}$ stores the highest version number contained in the invalidation acknowledgements from a Q_{output} server j for o. The protocol maintains an invariant: if $valid_{o,i} = true$ at j, then $lastRead_o \geq lastAck_{o,j}$ at i.

A Q_{output} server j considers an object o *valid* if its local state satisfies the following condition:

Validity condition 1 (VC1). $\forall i,\ i \in Q_{input},\ value_o.lc \geq max(lastKnown_{o,i})$ and $\exists\ R_{input}$ $s.t. \forall r,\ r \in R_{input},\ valid_{o,r} = true$

If VC1 is true, the cache has the latest version of all learned versions, and j has valid copies from a R_{input} quorum. If j satisfies VC1, j can directly return the current value to a read request, i.e. *read hit*. We will prove in Section 3.3 that it is safe to do so.

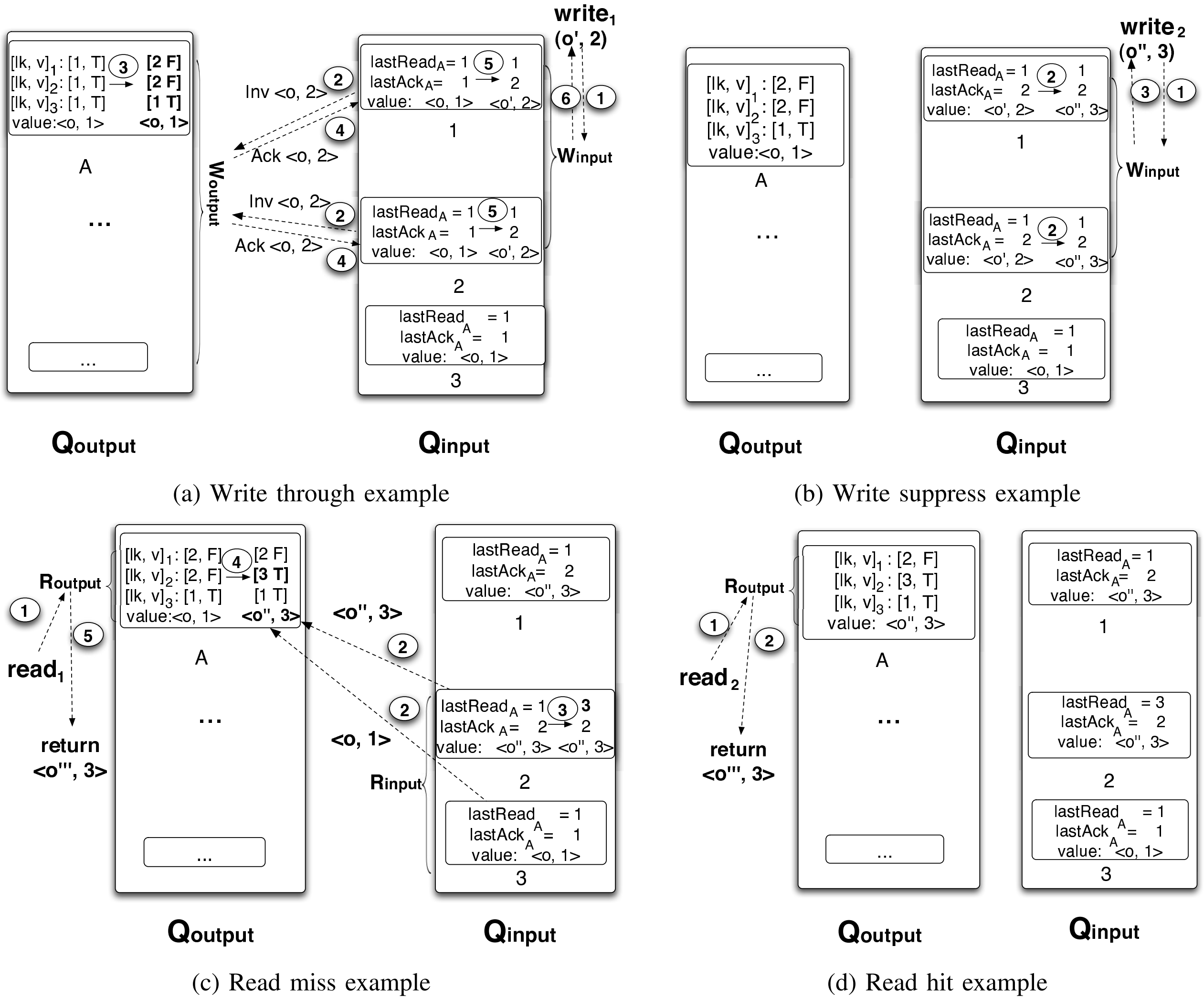
Otherwise, a read on j is a *read miss* and j needs to communicate with Q_{input} servers to get a consistent version. In particular, j sends object renewal messages to a R_{input} quorum via $QRPC$ to renew the object. Each server i in that R_{input} quorum responds to an object renewal request with its local $value_o$ and then updates its local state $lastRead_o$ with $value_o.lc$. Upon receiving an object renewal reply (o', lc) from a Q_{input} server i, if $lc \geq lastKnown_{o,i}$, then j updates $lastKnown_{o,i}$ with lc and sets $valid_{o,i}$ to be true; if $lc > value_o.lc$, then j replaces its $value_o$ with the value in the reply. When VC1 becomes true, j returns its $value_o$ to the client.

Invalidation suppress and write through. A Q_{input} server i processes a write request as a *write suppress* when the following condition is true:

Suppress condition 1 (SC1). $\forall j,\ j \in Q_{output},\ lastRead_o < lastAck_{o,j}$.

As we prove in Section 3.3, if SC1 is true at each server of a write quorum in Q_{input} then VC1 must be false at all read quorums in Q_{output}. Therefore it is safe to suppress the invalidations.

If SC1 is false, it is a *write through*. To ensure that all read quorums in Q_{output} are unable to read an older value, i needs to do some additional tasks before completing the write. i sends invalidations with the version number of the write to Q_{output} using $QRPC$. Upon receiving an invalidation $Inval(o, lc)$ from i, a Q_{output} server j updates its $lastKnown_{o,i}$ to lc and sets $valid_{o,i}$ to $false$ if $lc > lastKnown_{o,i}$. Then j sends an acknowledgement back to i so that i can update its $lastAck_{o,j}$ to lc and completes the write after collecting acknowledgements from a W_{output} quorum.



(a) Write through example

(b) Write suppress example

(c) Read miss example

(d) Read hit example

Fig. 4. Request processing scenarios

Example. Figure 4 illustrates the four read/write scenarios in an edge service system with 3 Q_{input} servers (1, 2, 3) and multiple Q_{output} servers (A, ...). The input quorum system is configured as a majority quorum, i.e. two servers for a read quorum and two servers for a write quorum; the output quorum system is configured as read-one write-all quorum. Initially, all Q_{input} servers replicate the object $\langle value, versionNum \rangle$ of $\langle o, 1 \rangle$ and all Q_{output} servers cache the object from each Q_{input} server (i.e. $lastKnown_i = 1, valid_i = true, i = 1, 2, 3$). Note in the figure, we represents $lastKnown$ by lk, $valid$ by v, $false$ by F, and $true$ by T.

For simplicity, Figure 4 (a) omits the details of retrieving the version number before issuing the write to the quorum. As indicated in Figure 4 (a), when a client issues $write_1$ (o', 2) to a W_{input} quorum composed of server 1 and server 2, it is a *write through* case for both servers since both have $lastRead_A = lastAck_A$, i.e. SC1 is $false$. Therefore, both servers send invalidations to a

W_{output} quorum. Upon receiving an invalidation message from a Q_{input} server i ($i = 1 \; or \; 2$) for o with version number 2, Q_{output} server A updates its $lastKnown_{1,2}$ to 2 and $valid_{1,2}$ to $false$ as indicated in step ③ in the figure. Then A sends an acknowledgement $\langle o, 2\rangle$ back to server 1 and server 2 which update their $lastAck_A$ to 2. Each W_{input} server applies the new version object and returns $write_1$ after receiving acknowledgements from a W_{output}.

Now suppose another write $write_2$ $\langle o'', 3\rangle$ is issued to the same W_{input} quorum, as indicated in Figure 4 (b), SC1 on either server is still $true$ after $write_1$. Therefore both can *write suppress*, i.e. both can update their value to $\langle o'', 3\rangle$ and return immediately.

Figure 4 (c) illustrates a *read miss* scenario. Consider the system in the previous example, $read_1$ on A has to renew object o from a R_{input} quorum because VC1 on A is $false$ after $write_1$. Suppose A selects server 2 and 3 as the R_{input} quorum for its object renewal, then server 2 will send $\langle o'', 3\rangle$ and server 3 will send $\langle o, 1\rangle$ to A as renewal replies. After A applies these two replies, its value becomes $\langle o'', 3\rangle$ and $valid_{o,2}$ becomes true. Therefore VC1 becomes true. A then returns $\langle o'', 3\rangle$ for $read_1$ request. Note that because the W_{input} quorum of $write_1$ intersects the R_{input} quorum of $read_1$, A is able to read the newest completed version.

As indicated in Figure 4 (d), a subsequent $read_2$ right after $read_1$ on A will be a *read hit* since VC1 is still true.

As illustrated from the above examples, for workloads consisting of read bursts, the first read forces all servers in a R_{output} quorum to validate their cached copies to satisfy VC1. Therefore, all subsequent reads to the same read quorum are *read hits*. If we configure the R_{output} quorum to contain only one server, then most reads in a burst are local operations. Therefore, the protocol typically yields nearly optimal read response time and availability for such workloads. Similarly, for workloads consisting of write bursts of the same data, the first write invalidates cached copies in a W_{output} quorum, making all subsequent writes to the same write quorum behave as *write suppresses*. Typically, we configure the Q_{input} as a majority quorum system to provide optimal write availability [28].

3.2 Dual-quorum with volume leases

The ADQ protocol just described allows one to vary read and write quorum sizes independently, therefore our target application would benefit from using a read quorum size of 1 so that reads can be serviced locally in the normal case; any larger read quorum size introduces a network delay to every read and provides qualitatively worse read response time. However, a read quorum size of 1

	Variable	Meaning
Q_{output} (j)	$cTime$	Current local real time.
	$expires_{v,i}$	Expiration time of volume lease for v renewed from a Q_{input} server i.
	$epoch_{o,i}$	Epoch number associated with the current object version from a Q_{input} server i.
Q_{input} (i)	$cTime$	Current local real time.
	$expires_{v,j}$	Expiration time of volume lease for v renewed by a Q_{output} server j.
	$delayed_{v,j}$	Buffered invalidations for updates on volume v for a Q_{output} server j.
	$epoch_{v,j}$	Current epoch number for $delayed_{v,j}$.

Fig. 5. Per Q_{output} and Q_{input} server data structure for object o.

could lead to unacceptable write availability because it requires a write to successfully contact all servers in Q_{output} to invalidate cached data in the *write through* case.

The full DQ protocol therefore adapts Yin et al.'s volume lease protocol [19] to support very small read quorums in Q_{output} while retaining acceptable availability on writes. An object lease represents permission to access an object until specified time [29]. A volume lease is a lease on a group of objects (volume). Under the volume leases protocol, a client may access a cached object if it holds valid leases on both the object and the object's volume, and a server can modify data as soon as either lease expires. The combination of short volume leases and long object leases yields good read response time and high availability for systems with small Q_{output} read quorums; servers in Q_{output} can cache objects locally for a long time to reduce individual object renewal load, and although they must frequently renew volume leases, the cost is amortized across a large number of objects in a volume. At the same time, the combination does not suffer from poor write availability despite large Q_{output} write quorums: a write that can not contact all servers in a Q_{output} write quorum just needs to wait for the (short) volume lease to expire.

To simplify the description of protocol, we assume infinite-length object leases or *callbacks* [30]. The protocol can be generalized to finite-length object leases simply by treating lease expiration like object invalidation in the basic protocol.

Data structures. As Figure 5 illustrates, each server maintains a set of variables in addition to the basic data structures in Figure 3. First, to track the duration of leases, each server maintains a real time clock $cTime$ with a drift rate bounded by $maxDrift$. Each server also maintains an $expires_{v,n}$ indicating when a volume lease for v on server n expires.

The protocol uses *delayed invalidations* and *epoch numbers* to minimize the cost of renewing volume leases. A volume lease can only be renewed by a Q_{output} server if the server can guarantee that it will not allow access to any stale object in that volume. Naive implementation must synchronize the state of each object in a volume, which can yield unacceptable overheads and synchronization delays, especially if volumes span many objects.

Delayed invalidations reduce the cost of short disconnections to $O(\#(missed\, invalidations))$ from $O(\#(objects\, in\, a\, volume))$. When a new write arrives, rather than sending the invalidations immediately to those Q_{output} servers that have valid object leases but expired volume leases, the Q_{input} server can defer the invalidation messages because the Q_{output} can not read the object until it renews the volume lease. It can then send a batch of delayed invalidations when the Q_{output} server renews the volume lease. Therefore, each Q_{input} server also maintains a per-volume invalidation buffer $delayed_{v,j}, \forall j,\, j \in Q_{output}$ to store delayed invalidations of objects in v for server j.

Epoch numbers bound the size of $delayed_{v,j}, \forall j,\, j \in Q_{output}$ and enable fast resynchronization after long disconnections. Each Q_{input} server i maintains an epoch number $epoch_{v,j}, j \in Q_{output}$ and each Q_{output} server j stores the max $epoch_{v,j}$ value associated with each object o received from $\forall i,\, i \in Q_{input}$ as $epoch_{o,i}$. Whenever a server garbage collects $delayed_{v,j}$, it increments $epoch_{v,j}$. Volume lease renewals and object renewals are marked with $epoch_{v,j}$. When $epoch_{v,j}$ on i changes, j conservatively assumes that all object callbacks from i with old epochs have been revoked by i so that any subsequent read will re-validate the cache copy.

The main difference between this protocol and the asynchronous protocol is that the object validity check condition and the write suppress condition are changed because of volume leases. In the rest of this subsection, we will describe how those conditions have changed.

Object validity and renewal. A Q_{output} server j considers an object o under volume v *valid* if its local state satisfies the following condition:

Validity condition 2 (VC2). $\forall i,\, i \in Q_{input},\ value_o.lc \geq max(lastKnown_{o,i})$ and $\exists\ R_{input}$ $s.t. \forall r,\, r \in R_{input},\ \ valid_{o,r} = true \wedge expires_{v,r} > cTime$

Similar to the basic protocol, j uses VC2 to decide whether to process a read as a *read hit* or a *read miss*. In a *read miss*, j needs to send *different* requests to different Q_{input} servers and reply when VC2 becomes true. In particular, for each target server i selected, j sends one of three things: (a) if the volume from i has expired and the object from i is invalid, it sends a combined volume renewal and object renewal request; (b) if just the volume has expired, it sends a volume renewal request; or (c) if just the object is invalid, it sends an object renewal request.

The object renewal process is exactly the same as in the basic dual-quorum protocol we described in subsection 3.1 except that each Q_{input} server i also send its $epoch_{v,j}$ with the object values and

j updates its $epoch_{o,i}$ and $valid_{o,i}$.

The volume lease renewal needs to do a few more things. Upon a volume lease renewal request from a Q_{output} server j, a Q_{input} server i sends the delayed invalidations $delayed_{v,j}$ and a volume renewal message containing a lease length L and the volume epoch number $epoch_{v,j}$. i then records the volume expiration time ($expires_{v,j} = L + cTime$).

When j receives a volume lease renewal reply from j, it first applies the delayed invalidations to affected objects as described in subsection 3.1 and updates $expires_{v,i}$ and $epoch_{o,i}$ for all objects under volume v. To account for worst-case clock drift and any network delays, j conservatively sets $expires_{v,i} = t_o + L * (1 - \mathit{maxDrift})$ where t_o is the time that j *sent* the volume lease renewal request, L is the volume lease length granted in the reply, and *maxDrift* is as defined in Section 2. To allow i to clear its delayed invalidation queue, j sends i a volume lease renewal acknowledgment containing the highest version number among all of the processed invalidations. When i receives a volume lease renewal acknowledgment for volume v and version number lc from j, i clears all delayed invalidations with logical clocks up to lc from $delayed_{v,j}$.

At any time if i wishes to garbage collect delayed invalidations that it has not sent to j or that j has not acknowledged, i advances $epoch_{v,j}$. Note that if j receives from i a volume lease with a new epoch, then $epoch_{v,i} \neq epoch_{o,i}$ for all o in v. As a result, all previously valid objects from i immediately become invalid, i.e. $valid_{o,i} = false$. Therefore, if j misses some object invalidations from i when its volume lease from i has expired, a volume lease renewal from i can resynchronize j's state by either (a) updating $valid_{o,i}$ and $lastKnown_{o,i}$ with the delayed invalidation or (b) advancing $epoch_{v,j}$ by sending a volume renewal with a new epoch number.

Invalidation suppress and write through. A Q_{input} server i processes a write request as a *write suppress*, when the following condition is true:

Suppress condition 2 (SC2). $\forall j,\ j \in Q_{output},\ lastRead_o < lastAck_{o,j}$ or $cTime \geq expires_{v,j}$.

If SC2 is true, i processes the write locally, appends the invalidation for the pending write in $delayed_{v,j}$ for each Q_{output} server j that has expired volume leases (i.e., $expires_{v,j} < cTime$), and acknowledges the write request immediately.

If SC2 is false, it is a *write through*. To ensure that at least a W_{output} is unable to read the old value, i needs to do two things: (1) send an invalidation for the pending write to those Q_{output} servers that have both a valid object lease and a valid volume lease and (2) append the invalidation for the pending write in $delayed_{v,j}$ of each Q_{output} server j that has expired volume lease. As soon as SC2 becomes true, i processes the write locally and acknowledges the client.

Comparing with the basic protocol, the volume lease protocol has better write availability because it can expire volume leases without communicating with any Q_{output} server, but read performance might degrade due to volume lease renewals. Consider the same *write through* scenario in Figure 4 (a). If any of the Q_{output} server (e.g. A) is disconnected, $write_1$ will block until A comes back. With volume leases, $write_1$ only needs to wait at most until $expires_{v,A}$ when the volume lease for A is definitely expired. When $write_1$ waits long enough, eventually SC2 will be true due to volume lease expiration. Therefore, a *write through* scenario can be reduced to a *write suppress* scenario by trading latency for availability. On the other hand, read performance might degrade because of the additional volume lease renewal cost. Consider the same *read hit* scenario in Figure 4 (b). The subsequent read following $read_1$ in the basic protocol is a *read hit*, but it might be a *read miss* due to a volume lease expiration that breaks VC2. Even worse, the volume lease renewal might fail due to network partition of A from any R_{input}. In this case, we assume the underlying request redirection architecture will redirect the read to other available edge-servers.

3.3 Correctness

In this section, we prove that the dual-quorum protocol guarantees *regular semantics*, i.e. satisfies both **property 1** and **property 2** as defined in Section 2. We first prove that the simplified asynchronous dual-quorum protocol (ADQ) satisfies the two properties. Then we give proof of correctness for the full dual-quorum with volume leases protocol (DQ). Finally, we discuss issues with extending the protocol to support stronger semantics such as *atomic semantics* [18].

Asynchronous dual-quorum protocol. We first establish a helpful lemma: once a write completes, no subsequent read at *any* Q_{output} server can return an older value.

Lemma 1. *If a write W for object o completes in the ADQ protocol, then no subsequent read of o returns a value with a timestamp lower than $W.lc$.*

Proof: Consider two cases for W: (1) *write suppresses* at each W_{input} server or (2) a *write through* for at least one server i in a W_{input} quorum. We first prove that any subsequent read in case (1) is a *read miss* and any subsequent read in case (2) is either a *read hit* with a value at least as new as W or a *read miss*. Then we prove that the object renewal invoked by any *read miss* returns a value at least as new as W.

In case (1), each W_{input} server satisfies SC1. Suppose there exists a R_{output} quorum such that each server has a valid copy W' with $W'.lc < W.lc$. Consider any server j in the R_{output} quorum. By VC1, the max version of all invalidations that j receives from all Q_{input} servers is at most $W'.lc$ and there exists a R_{input} quorum such that $valid_{o,i}$ is true for each i in the R_{input} quorum. Therefore each i in the R_{input} quorum has $lastAck_{o,j} \leq lastRead_o$. Since the W_{input} quorum intersects the R_{input}quorum, at least one W_{input} server has $lastAck_{o,j} \leq lastRead_o$ which contradicts SC1. Therefore it is impossible to have such a R_{output} quorum that returns an old value without renewing first; any subsequent read will force at least one Q_{output} server to renew from a R_{input} quorum.

In case (2), i sends invalidation with $W.lc$ to at least a W_{output} quorum before W completes. Since any W_{output} quorum intersects with any R_{output} quorum, any subsequent client read request will be sent to at least one of the W_{output} members j whose $lastKnown_{o,i}$ is at least as new as $W.lc$. Therefore j will return a valid object with a version at least as new as $W.lc$ if VC1 is true. Otherwise, it is a *read miss*.

Finally, we prove that a *read miss* returns a value at least as new as W. Since W has completed, there exists at least a W_{input} quorum whose members have received W. Because any R_{input} quorum intersects any W_{input} quorum, any object renewal from a R_{input} quorum will return a write at least as new as W. □

Theorem 1. *The ADQ protocol provides regular semantics.*

Proof: Two operations $o1$ and $o2$ are considered *concurrent* if $o1$ starts before $o2$ completes and after $o2$ starts or vice versa. Suppose the last completed write is W, by lemma 1, any subsequent read will return a value at least as new as W. Since W is the last completed write, any subsequent read that is not concurrent with any write of o will return W, i.e., **Property 1** holds.

Suppose a write W' is concurrent with a read R and the last completed write is W (Note $W'.lc > W.lc$). By **Property 1**, any reads that precede W' after W completes return W. Therefore before W' or R starts, there are two cases to consider for any Q_{output} read quorum R_{output}: (1) R_{output} has at least one invalid member (lemma 1), or (2) All R_{output} members are valid and at least one valid member holding value of W (renewed by any subsequent read).

When R sends requests to R_{output} of case (1), then we have a situation where both the renewal and the write W' are active in the Q_{input}. Since Q_{input} as traditional quorum systems provides *regular semantics*, the renewal could return the invalid R_{output} member a value of either W or W'. As a result, the read will return either W or W' to the client. Notice that W' might change some

R_{output} quorums from case (2) to case (1), for these R_{output} quorums we have the same result as above. For any R_{output} quorum that remains in case (2) when serving the R request, it will return W.

Similarly, we can prove that for multiple concurrent writes and read, we still have the same result. Therefore, ADQ provides both **Property 1** and **Property 2**.

□

Dual-quorum protocol with volume leases. The proof for the full DQ protocol that makes use of volume leases is similar to the proof for ADQ. First, a property similar to lemma 1 is still true for dual-quorum protocol with volume leases.

Lemma 2. *If a write W for object o completes in the dual-quorum with volume leases protocol, then no subsequent read of o returns a value with a timestamp lower than $W.lc$.*

Proof: Consider the same two cases in the proof of lemma 1. By replacing VC1 with VC2 and SC1 with SC2 in the proof of lemma 1, we can easily derive the same conclusion about case (1): any subsequent read in case (1) is a *read miss*.

Now we prove that any subsequent read in case (2) is either a *read hit* with a value at least as new as W or a *read miss*. In case (2), after W completes, SC2 is true on each server of a W_{input} quorum. Therefore any output server j either (1) receives an invalidation with $W.lc$ or (2) $j.expires_{v,i} < j.cTime$ for all i in the W_{input} quorum. If j receives an invalidation with $W.lc$ during W *write through*, then VC2 makes sure that it returns a value at least as new as W. If j does not receive any invalidation with $W.lc$, then its volume leases must have expired from at least a W_{input} quorum. Because the W_{input} quorum intersects with any R_{input} quorum, j can not have valid volume leases from a R_{input} quorum. Therefore VC2 on j is false, i.e. *read miss*.

Finally we prove that any *read miss* returns a value at least as new as W. In a *read miss*, if any of the R_{output} server renews the object, from proof of lemma 1, it will get a value at least as new as W. Otherwise, it needs to renew some volume leases to make sure that it has valid volume leases from a R_{input} quorum. According to the volume lease renewal protocol, at least one of the R_{input} quorum that intersects any W_{input} quorum has a delayed invalidation with $W.lc$ for j or a newer epoch number than j's current object epoch number. Therefore, the renewal of volume leases makes j's local stale object invalid if it is older than W and invoke an object renewal which brings a version at least as new as W. □

Theorem 2. *The dual-quorum protocol with volume leases provides regular semantics.*

Proof: Similar to the proof for the basic dual-quorum protocol, by lemma 2, we can easily derive that Dual-quorum protocol with volume leases provides regular semantics. □

Atomic semantics. Though in principle the dual-quorum protocol can be extended to support atomic semantics [18], doing so would likely give up most of the benefits of the approach. In general, there are two approaches to support atomic semantics for quorum systems: *writeback* [31] and *majority matching* [32]. The writeback mechanism implements atomic semantics by requiring each read operation to write back the read value to a write quorum. The majority matching approach blocks a read until it collects matching replies from at least a majority quorum. Either approach is problematic for our efforts to optimize read performance by supporting small read quorums. In the case of a writeback, reads must access both a read quorum and a write quorum. In the case of majority matching, each read must contact at least a majority of servers.

4 Evaluation

Through both analytical and experimental evaluations, we compare the availability, performance, and communication overhead of dual-quorum with volume leases protocol against other popular replication protocols. We show that DQ yields read performance competitive with ROWA-A epidemic algorithms and that overall availability is competitive with the majority quorum protocol.

4.1 Response time

Analytical evaluation. First, we analyze the response time of DQ and make comparisons with other popular protocols in the context of the edge service environment where every service client connects to a nearby edge server via a fast connection, e.g. a LAN-like connection, lan, with 6 ms RTT. All edge servers connect to each other through an overlay network, $overlay$, with RTT delays of 80 ms. For a client to connect to servers other than its nearby edge server, it has to go through a WAN-like connection, wan, with 86 ms RTT.

To preserve the optimal availability, the Q_{input} is configured as a majority quorum system. But the read quorum in Q_{output} can be configured to consist of one server so that a client needs to read only from its nearby server. Therefore, the response time of a *read hit* will only involve lan delays, but the response time of a *read miss* is $lan + overlay$ because this closest server needs to renew from other edge servers. The response time of *write suppress* is $2wan$, one round trip to retrieve the highest timestamp and another round trip to perform the actual write. The response time of *write through* is $2wan + overlay$ because the write has to send invalidations and wait for acknowledgments to come back from a write quorum Q_{output} in addition to retrieving the highest timestamps and sending the write to be performed. If we assume the workload consists of groups of consecutive reads followed by consecutive writes, most reads are *read hit* (except for the first one in each group) and most writes are *write suppress* (except for the first one in each group). Suppose the write percentage is w, then the read percentage is $1 - w$ and we have the best case average response time for DQ:

$$resp_{DQ-Best} = w \times 2wan + (1 - w) \times lan$$

When the workload consists of interleaved reads and writes, most reads are *read miss* and most writes are *write through*. The average response time for these workload is potentially poor. Depending on the write ratio, there are two cases for this scenario:

- When $w \geq 0.5$, the worst workload pattern is a set of interleaved writes and reads followed by a set of consecutive writes. Therefore the response time is :

$$resp^{w \geq 0.5}_{DQ-Worst} = (1 - w) \times (2wan + overlay) + (1 - w) \times (lan + overlay) + (2w - 1) \times 2wan$$

- When $w \leq 0.5$, the worst workload pattern is a set of interleaved writes and reads followed by a set of consecutive reads. For the consecutive reads, the worst scenario is that different reads touch different R_{output}in Q_{output} which still requires renewal from a R_{input} quorum. Therefore the response time is

$$resp^{w \leq 0.5}_{DQ-Worst} = w \times (2wan + overlay) + (1 - w) \times (lan + overlay)$$

Protocol comparison. Given the above formulation of response time, we can compare DQ with a range of algorithms.

In comparing with DQ, the read-one-write-all (ROWA) protocols read from only one server and write to all replicas. Although ROWA protocols are often treated separately in the literature [33], [20], they are, in fact, a special case of quorum protocols in which the read quorum is composed of any one server in the system and the write quorum is the entire set of servers. In the context of the edge service environment, the ROWA protocols read from a nearby edge server via a fast

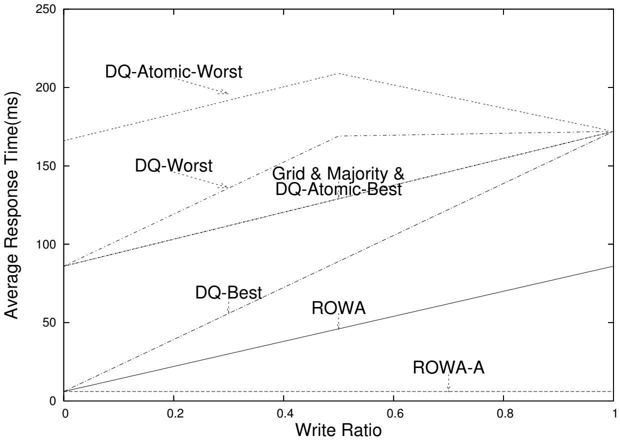


Fig. 6. Average response time (number of replicas = 15).

connection, and they block a write until all the edge servers have received the write. Therefore, the average response time for ROWA protocols is:

$$resp_{ROWA} = w \times wan + (1 - w) \times lan$$

ROWA-A protocols [14], [15], [16] are variations of ROWA protocols that allow the write to be propagated asynchronously to other servers. Therefore the response time is:

$$resp_{ROWAA} = lan$$

Other traditional quorum systems such as majority quorums [21] or grid quorums [22] need two round trips for a write (get timestamp and write) and need to contact more than one server for read. Therefore their response times are:

$$resp_{Majority} = resp_{Grid} = w \times 2wan + (1 - w) \times wan$$

Average response times of various protocols are illustrated in Figure 6 where we plot the average response times while varying the write ratio and fix the number of replicas to 15. DQ provides its best case response time when workloads consist of only *read hits* and *write suppresses*. As Figure 6 shows, DQ is an order of magnitude better for read dominated workloads (i.e. w close to 0) than traditional quorum systems and yields comparable response time for write dominated workloads. As indicated by the third line from the bottom, DQ *read hits* yield performance competitive with ROWA-A epidemic algorithms against read-dominated workloads because they only need to communicate with the closest server.

However, when the workloads are composed of interleaved reads and writes, DQ response time can be 40ms longer than the traditional quorum systems. DQ has the worst case response time against workloads consisting of a large number of *read misses* and *write throughs*. DQ *read misses* and *write throughs* require communication with distant servers similar to the behaviors of both majority and grid quorum operations. Therefore, they all experience the *wan* delays. Furthermore, because writes in quorum systems (including DQ) require one *wan* trip to retrieve the highest timestamp and another to perform the actual write, the response time of write-dominant workloads is twice that of ROWA. *Write throughs* require an additional *wan* trip to invalidate a write quorum in Q_{output}. At a 50% write ratio, when DQ has the maximum amount of *write throughs*, the overall response time of DQ reaches its worst case relative to the other protocols as indicated by the top most curve.

Atomic semantics. Although the studied DQ only supports regular semantics, for completeness, Figure 6 also shows the average response time of a DQ variation that supports atomic semantics [18]. As we described in section 3.3, DQ can not achieve the above performance improvement if it supports atomic semantics by either *writeback* or *majority matching*. For simplicity, here we only show the results of the *majority matching* approach and assume that there always exists a read quorum with matching values when a read happens.

Since *majority matching* requires majority quorums in both input quorums and output quorums, the R_{output} size can not be optimized to be one. As a result, the read must contact multiple nodes and the read response time involves wan delay instead of lan delay. Therefore, the best case average response time for DQ-Atomic is the same as majority quorums with atomic semantics:

$$resp_{DQ-Atomic-Best} = w \times 2wan + (1-w) \times wan$$

Similarly, when reads and writes interleave,

- If $w \geq 0.5$, the response time is :

$$resp^{w \geq 0.5}_{DQ-Atomic-Worst} = (1-w) \times (2wan + overlay) + (1-w) \times (wan + overlay) + (2w-1) \times 2wan$$

- If $w \leq 0.5$, the response time is :

$$resp^{w \leq 0.5}_{DQ-Atomic-Worst} = w \times (2wan + overlay) + (1-w) \times (wan + overlay)$$

Note the actual read response time is longer than what we show here because the read might be blocked for a majority of nodes to get the same value which is not necessary for regular semantics. If there are always concurrent updates, the read might be blocked for a long time.

As indicated in Figure 6, DQ-Atomic performance is at best the same as the performance of majority quorums. In the worst case, it has an additional 80ms latency to coordinate Q_{input} and Q_{output}. Compared to DQ with regular semantics, the average response time for DQ-Atomic is at least 40ms longer in both the best and worst cases because DQ-Atomic cannot take advantage of smaller read quorums.

Experimental evaluation. We have also developed prototypes for DQ, primary/backup, majority quorum, ROWA-A, and ROWA protocols. All the prototypes are built in Java and run on eight Emulab [34] servers. To simulate the edge service architecture as described in Figure 2, we set the "lan" delay between an application client and its closest edge server to 8 ms; the "overlay" delay among the edge servers is 80 ms; the "wan" delay between an application client and other edge servers is 86 ms.

In the rest of this section, we compare the response time of five protocols under our target workload, the subset of the TPC-W workload that operates on the user profile. We show that DQ yields better response time than protocols providing strong consistency guarantees and competitive response time to protocols with relaxed consistency guarantees.

Write ratio. We use the TPC-W workload [13] for our prototype experiments. TPC-W specifies an ecommerce workload that simulates the activities of a retail book store website. There are three scenarios: browsing, shopping, and ordering. We are interested in the most popular browsing scenario which consists of a mix of 95% browsing interactions, such as searches and product detail displays, and 5% ordering interactions. In particular, we are interested in the workload on the multi-writer multi-reader profile object in this scenario.

We first evaluate the response time by fixing the write rate to 5%, which is the update rate for TPC-W profile object, i.e. a workload with a low update rate and strong access locality. Accesses to the profile object consist of 95% reads on a customer's purchase history, credit information, and addresses and 5% writes on a customer's shipping address when processing an online purchase. When the profile is replicated on edge servers, a customer is routed to the closest edge server to access its profile information.

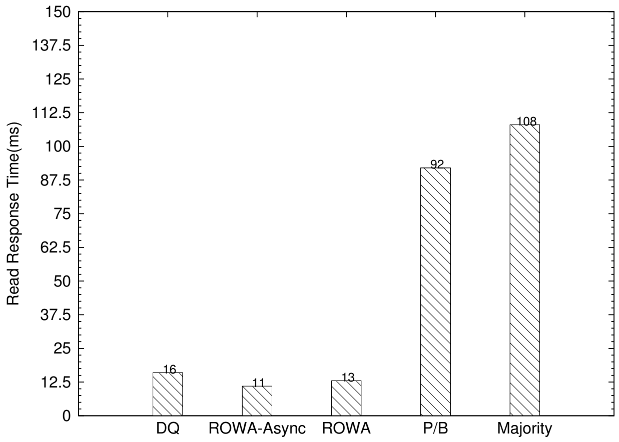


Fig. 7. Response time vs. 5% write ratio.

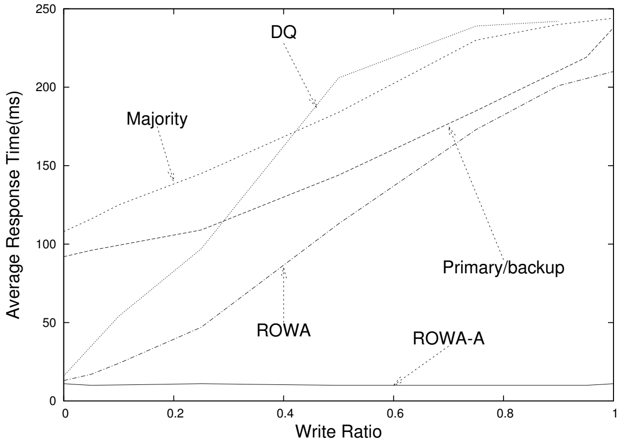


Fig. 8. Response time vs. write ratio (number of replicas = 15).

As illustrated in Figure 7, DQ provides at least six times better read response time than primary/backup and majority quorum protocols that are used to provide strong consistency guarantees. DQ yields almost the same read response time as ROWA and ROWA-A protocols because it allows most client reads to be processed only at the client's closest replicas with only 8 milliseconds RTT while maintaining the same level of consistency guarantees as both primary/backup and majority quorum protocols by running the dual-quorum invalidation protocol between the closest replica and the rest of replicas in the system. Note that response times of all prototypes are higher than the underlying minimum network delays due to experimental variation and un-tuned code.

Figure 8 is the sensitivity graph illustrating how the overall read and write response time changes as we vary the write rate. The response time is the average read and write response time over a two-hour period. As writes dominate the workload, DQ's response time approximates that of the

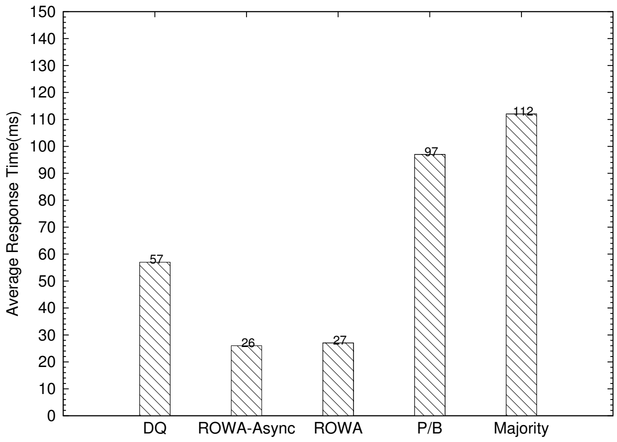


Fig. 9. Average response time vs. access locality(5% write ratio & 90% access locality).

majority quorum protocol and becomes higher than those of primary/backup and ROWA. The main reason is that DQ clients, following the same procedure as the majority quorum protocol, need to obtain the latest timestamp from a read quorum before sending the write to a write quorum in Q_{input}. Two round trips are required for both the majority quorum protocol and DQ while only one round trip is needed for primary/backup and ROWA protocols. The additional trip to obtain the timestamp prior to performing the actual write increases the average response times of both DQ and the majority quorum protocol compared with ROWA protocol.

Access locality. In this subsection, we evaluate response time when some portion of client requests are routed to replicas other than the client's default closest one. Under normal circumstances, requests are routed to the client's closest server. But the unavailability of the closest replica or the geographical movement of the client can sometimes result in the requests being routed to distant replicas.

Figure 9 illustrates protocols' response times at our target 5% write rate and 90% access locality (i.e. 10% of client requests are sent to distant replicas and 90% of client requests are sent to the client's closest replica). The 90% access locality is a pessimistic measure for Internet edge servers given typical network failure rate is well below 10% and the majority of end users do not travel frequently. DQ outperforms both primary/backup and majority quorum protocols for this workload while preserving the same consistency level even in cases where client requests are directed to distant replicas. Note that ROWA-A protocol yields the optimal response time at the cost of serving reads with potentially inconsistent data when requests are directed to the distant replicas.

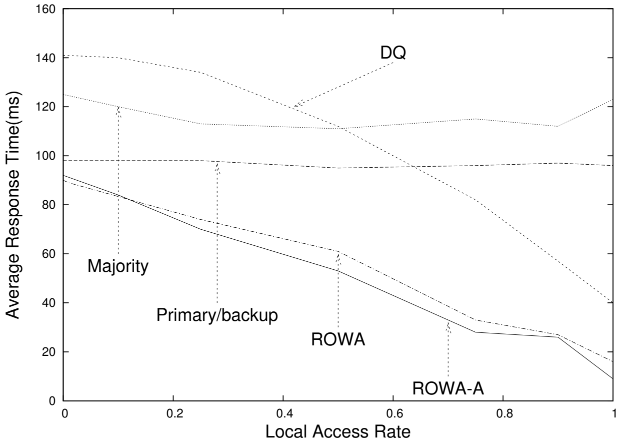


Fig. 10. Average response time vs. access locality(5% write ratio & varying access locality).

In the DQ protocol, the response time of reads at distant replicas is higher than the normal response time experienced when reading from the closest one. As the access locality varies, the overall response time changes accordingly. Figure 10 indicates the relationship between the access locality and the overall read and write response time of five protocols. The response time is the average read and write response time over a two-hour period. DQ suffers when access locality is low because both reads and writes need to contact replicas in both input and output quorum systems. But DQ's response time keeps improving as the access locality becomes higher. The majority quorum and primary/backup protocols are not affected by the access locality because neither protocol is designed to take advantage of the access locality in the edge service environment. This graph suggests that when the access locality is 70% or higher, DQ should be preferred over primary/backup or majority quorum protocols for replication systems that require low response time and strong consistency guarantees.

4.2 Availability

In this section, we provide analytical models to evaluate the availability of the dual quorum protocol in comparison with other popular replication protocols.

We define the availability (av) as the number of client requests successfully processed by the system over the total number of requests submitted to the system during a given time period. A request is rejected by the system when target consistency semantics can not be satisfied [35] or if insufficient servers are available to process requests. In the context of this discussion, systems are required to provide regular semantics [18]. For example, if more than half of the servers are unavailable in Q_{input} of a dual quorum system or in a majority quorum system, a client write will be rejected because the system can no longer guarantee that a later read can always retrieve the value of this write.

The availability of *read hit* is the availability of a read quorum in Q_{output} $av(R_{output})$. *read miss* not only needs to contact a read quorum in Q_{output}, but also needs to renew from a read quorum in Q_{input}. Suppose each server participates both in Q_{input} and Q_{output}, then the availability of *read miss* is the minimum of the availability of a read quorum in Q_{output} $av(R_{output})$ and the availability of a read quorum in Q_{input} $av(R_{input})$. Since the volume leases are normally short, we conservatively assume that availability of read is dominated by *read miss*, i.e. $\min(av(R_{output}), av(R_{input}))$. The write availability has similar results. The availability of *write suppress* is the availability of a write quorum in Q_{input} $av(W_{input})$. The *write through* needs to contact a write quorum W_{output} in Q_{output} besides the write quorum in Q_{input} W_{input}. Similarly, we conservatively assume the availability of write is dominated by *write through*, i.e., $\min(av(W_{output}), av(W_{input}))$.

Given that the size of a quorum is qs, the total replication size is n, and the per-server independent failure probability is p, the availability of the quorum is

$$av_{quroum} = \sum_{i=0}^{n-qs} \frac{n}{qs}(1-p)^{qs+i}p^{n-qs-i}$$

The availability of the dual-quorum system can be expressed as

$$av_{DQ} = (1-w) \times min(av(R_{output}), av(R_{input})) + w \times min(av(W_{input}), av(W_{output}))$$

Similarly, we derive the availability models of other quorum systems as the following:

$$av_{ROWA} = (1-w) \times (1-p^n) + w \times (1-p)^n$$

$$av_{ROWAA} = 1 - p^n$$

$$av_{Majority} = \sum_{i=1}^{\frac{n-1}{2}+1} \frac{n}{\frac{n-1}{2}+i}(1-p)^{\frac{n-1}{2}+i} \times p^{\frac{n-1}{2}+1-i}$$

$$av_{Grid} = (1-p^{\sqrt{n}})^{\sqrt{n}} - w \times (1-(1-p)^{\sqrt{n}} - p^{\sqrt{n}})^{\sqrt{n}}$$

Note that the ROWA-A protocol does not provide regular semantics, because it allows servers without the latest update to return stale data. Therefore, in order to compare the availability of

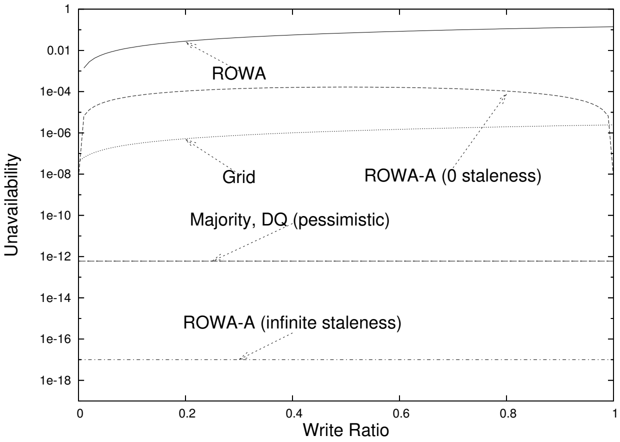


Fig. 11. Unavailability vs. write ratio (number of replicas = 15).

ROWA-A with the other protocols to satisfy the *same* consistency requirements, we model the availability of ROWA-A protocol by altering the ROWA-A protocol to avoid returning stale data. In particular, we assume there is an oracle in each server who always knows if an object is stale or not. When a server that only has stale data receives a read request, it will reject the request. The client will retry the read request by contacting other servers. Only when all available servers are stale, will we consider the request a failure. Therefore the availability of the ROWA-A without staleness is:

$$av_{ROWAA}(0\, staleness) = 1 - p^n - (1-w) \times (1/n) \times w \times p \times (1-p^{n-1})$$

Figure 11 and 12 illustrate the unavailability of DQ in comparison with other protocols in log scale. The unavailability is computed as $1 - av$. An unavailability of 10^{-i} corresponds to the availability of i 9's. Our simple model assumes a per-server failure probability $p = 0.01$ and that failures (including server crashes and network failures) are independent. Read and write rates are defined as $1 - w$ and w. This simple model is intended to illustrate the properties of the systems, not to model any realistic environment.

Figure 11 illustrates the systems' unavailability as we vary the write ratio and fix the number of replicas to 15 (in both Q_{input} and Q_{output}). Therefore, for dual-quorum input quorum systems and ROWA protocols, the read quorum size is 1 and the write quorum is 15; for output quorum systems and other majority quorums, the read quorum size is 7 and the write quorum is 8. The key result is that DQ's availability tracks that of the majority quorum. Note that the DQ's availability measurement is pessimistic because a read can proceed without contacting any read

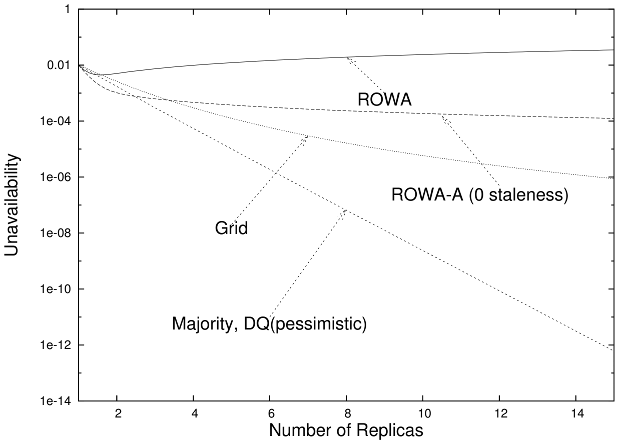


Fig. 12. Unavailability vs. number of replicas when fixing write ratio to 25%.

quorum in Q_{input} if the read quorum in Q_{output} holds valid volume and object leases; this effect may mask some failures that are shorter than the volume lease duration. Note the ROWA-A protocol provides excellent availability by allowing reads to return arbitrarily stale data to clients. When our experiments allow no stale reads in ROWA-A protocol, it yields poor availability that is several orders of magnitude worse than other quorum based protocols and our DQ protocol.

Figure 12 illustrates systems' unavailability as we vary the number of replicas and fix the write ratio at 25%. It shows that the unavailability of DQ has similar behavior as the majority quorum system. The availability of quorum based protocols, including DQ, improves as the total number of servers increases. The availability of ROWA and ROWA-A with no stale reads is insensitive to the number of servers in the system.

4.3 Communication Overhead

This section analyzes DQ's communication overhead in terms of the number of message exchanges required to process a client request. To simplify the model, the study assumes the costs of all message types are equal. In addition to notation used in the previous section, we introduce $|R_{input}|$ to represents the size of a read quorum in Q_{input}. When a Q_{output} server sends an object or renews an volume lease from a read quorum in Q_{input}, we use $|R_{input}|$ to indicate the number of messages sent by the Q_{output} server (one message to each server of the Q_{input} read quorum). msg_r and msg_w denote numbers of message exchanges when processing a read and a write. Our model targets the average number of message exchanges which is calculated as $msg_r \times (1-w) + msg_w \times w$.

A *read hit* requires $msg_{readHit} = 2|R_{output}|$ messages because a client sends to and receives from each server of a Q_{output} read quorum one message. But for a *read miss*, each participating

Q_{output} server that needs to renew the volume lease or the object sends a renewal request, receives a renewal reply, and responds with an renewal acknowledgment to a read quorum in Q_{input}, which requires $3|R_{input}|$ messages in addition to the $2|R_{output}|$ messages. When all servers of the Q_{output} read quorum need to renew their local volume leases or the object, the total message cost is $msg_{readMiss} = 2|R_{output}| + 3|R_{output}| \times |R_{input}|$. A *write suppress* requires $msg_{writeSuppress} = 2(|R_{input} + W_{input}|)$ messages because it retrieves the highest timestamp from a Q_{input} read quorum and performs the write on a Q_{input} write quorum. But a *write through* requires additional $2|W_{input}| \times |W_{output}|$ messages because of invalidations and acknowledgments between a Q_{input} write quorum and a Q_{output} write quorum. The total number of messages required for a *write through* is $msg_{writeThrough} = 2(|R_{input} + W_{input}| + |W_{input}| \times |W_{output}|)$.

Therefore, the average number of message exchanges for DQ when workload consists of only consecutive reads followed by consecutive writes (or vice versa) is:

$$msg_{DQ-best} = w \times msg_{writeSuppress} + (1-w) \times msg_{readHit}$$

When the workload consists of only interleaving reads and writes, the average number of messages required is:

$$msg^{w<0.5}_{DQ-worst} = w \times msg_{writeThrough} + w \times msg_{readMiss} + (1-2w) \times msg_{readHit}$$

and

$$msg^{w \geq 0.5}_{DQ-worst} = (1-w) \times msg_{writeThrough} + (2w-1) \times msg_{writeSuppress} + (1-w) \times msg_{readMiss}$$

The average number of messages required in other protocols are as follows:

$$msg_{ROWA} = 2w \times n + 2(1-w)$$

$$msg_{Majority} = msg_{Grid} = 2w \times (|rq| + |wq|) + 2(1-w) \times |rq|$$

We first examine the case where both Q_{input} and Q_{output} systems of DQ are configured the same as in the previous study, i.e. read and write quorums of Q_{input} include a majority of servers and the read quorum size of Q_{output} is one.

Figure 13 and 14 show the average number of messages required to process a client request in log scale. As illustrated in Figure 13, in the worst case where the write ratio is at 50%, DQ

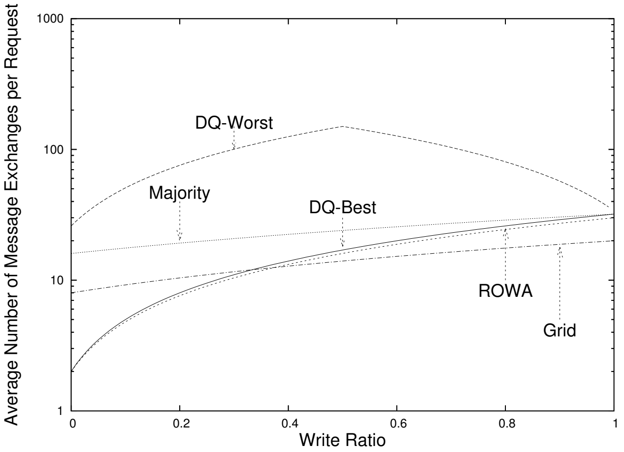


Fig. 13. Communication overhead vs write ratio (number of replicas = 15).

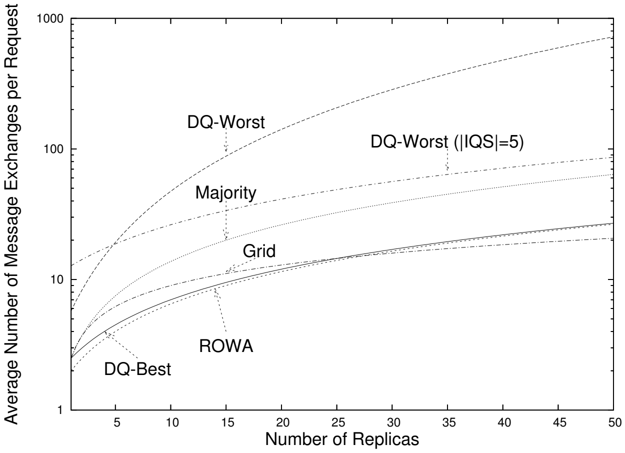


Fig. 14. Communication overhead vs the number of replicas.

can have high communication overhead as reads and writes interleave with each other. In this case, most reads are *read misses* and most writes are *write throughs* which involve both Q_{input} and Q_{output} in processing requests. However, DQ's overhead should be comparable to other approaches in practice. First, workloads that DQ is designed to support are dominated by reads. Consecutive reads are likely to benefit from having objects cached on Q_{output} servers, i.e. the target workloads have a large number of *read hits*. Second, the design of DQ allows us to vary the Q_{output} size to meet read performance goals while varying the Q_{input} size to balance overhead vs. availability goals. As shown in Figure 14, once we fix Q_{input} at a moderate size while letting the Q_{output} size grow, the communication overhead yielded by DQ is at the same level as the majority quorum without requiring many *read hits* in the workload.

Although the dual quorum protocol is described in terms of two quorum systems, Q_{input} and Q_{output}, a Q_{input} server can physically be on the same server as a Q_{output} server. Therefore, the overall communication overhead could be less than shown here because some messages become local.

5 Related Work

This article is an extended version of [36] in which we introduced the dual-quorum algorithms. In this version, we provide detailed proofs of the correctness of the asynchronous dual-quorum algorithm and the full protocol with volume leases. We also present more evaluation results including analytical evaluation of the response time and availability of DQ by comparing with other popular protocols.

In the read-one/write-all (ROWA) protocol family the “read-one” property yields excellent read availability and response time. But this class of protocols has limited write availability and response time because writes can not complete if any of the replicas are unavailable. Read-one/write-all-async protocols (ROWA-A) [14], [15], [16] yield better write availability and response time by allowing writes to be propagated to other replicas asynchronously. But they are only suitable for weakly consistent replication because they can not guarantee that reads will always return the data modified by the latest completed write. A variation of ROWA [20] performs writes synchronously on the available replicas to provide better consistency, but it requires membership protocols to maintain the consistent view of active members.

Quorum-based protocols [26], [27], [37], [21] can tolerate network partitions as long as connected replicas can form a quorum to process reads/writes. However, most quorum systems’ read response time and availability are worse than those of ROWA-A or primary-backup based protocols because reads usually need to query a larger set of servers. Therefore they are not desirable to handle a read-dominated workload, e.g. a workload from interactive online applications.

Some quorum based techniques use light-weight nodes, such as ghosts [38], to help form quorums for processing requests. When propagating a write, a replica only sends to these nodes the timestamp and object ID of the write. Our dual-quorum invalidation protocol shares the idea in terms of replacing writes with invalidations when propagating to some replicas. However our use of invalidations also allows us to reduce the future message propagation to other replicas.

As another approach for highly-available consistent data replication, state machine replication [39], [40] relies on various agreement protocols to achieve linearizability [41] while tolerating benign or Byzantine faults in different system models. In essence, as Li et al. illustrate in [42], agreement protocols such as Paxos [43] and PBFT [44] are actually elaborations on majority quorum systems.

A variation of the state machine replication approach such as [45] leverages a ring reliable multicast protocol instead of Paxos-like protocols to provide certain consistency guarantees for replication systems built upon it. To provide linearizability under network partitions, the replication system built on such a group communication protocol needs to block reads and writes until the node becomes a member of the primary partition. This approach introduces at least approximately half the token rotation time delay on average to deliver a message, which is not desirable for edge services where edge servers communicate in a WAN. Although the read liveness can be improved by allowing reads in non-primary partitions, doing so only provides serializability and does not provide regular semantics or any staleness guarantees. In addition, this class of techniques may have degraded performance in a WAN because it must run the membership protocol to include/exclude certain replicas when they are mistakenly considered as crashed/recovered due to slow WAN links.

Traditional cache invalidation protocols [29], [19] are primarily used in the client-server model where the single server hosts the objects and clients keep cached copies. Those protocols assume that an object always has a home location that can grant leases to cached copies, but this single centralized server may hurt availability.

6 Conclusions

This article presents dual-quorum replication, a novel data replication algorithm designed to support Internet edge services. Through both analytical and experimental evaluations, we demonstrate that this replication protocol offers nearly ideal trade-offs among high availability, good performance, and strong consistency under the target workloads.

Several important issues will be addressed in our future work. It will be interesting to configure both Q_{input} and Q_{output} to optimize other metrics. For example, we can configure the read quorum size in Q_{output} to be larger than one to avoid timeouts on invalidations. We can also configure Q_{input} as a grid quorum system [46] to reduce the overall system load.

Acknowlegement

The authors would like to thank Amitanand Aiyer, Navendu Jain, Edmund L. Wong and anonymous reviewers for their detailed feedback on earlier drafts of this paper.

References

[1] A. Awadallah and M. Rosenblum, “The vMatrix: A Network of Virtual Machine Monitors for Dynamic Content Distribution,” in *7th International Workshop on Web Content Caching and Distribution*, Aug. 2002.

[2] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar, “Improving Availability and Performance with Application-Specific Data Replication,” *IEEE Transactions on Knowledge and Data Engineering*, Mar. 2005.

[3] A. Whitaker, M. Shaw, and S. Gribble, “Scale and Performance in the Denali Isolation Kernel,” in *OSDI02*, Dec. 2002.

[4] I. Akamai Technologies, “Akamai-The Business Internet - A Predictable Platform for Profitable E-Business,” http://www.akamai.com/BusinessInternet/whitepaper_business_internet.pdf, 2004.

[5] I. Limelight Networks, “Limelight Networks CDN,” http://www.limelightnetworks.com.

[6] I. SAVVIS, “SAVVIS,” http://www.savvis.net.

[7] L. Bent, M. Rabinovich, G. Voelker, and Z. Xiao, “Characterization of a Large Web Site Population with Implications for Content Delivery,” in *WWW13*, May 2004.

[8] A. Su, D. Choffnes, A. Kuzmanovic, and F. E. Bustamante, “Drafting Behind Akamai (Travelocity-Based Detouring),” in *SIGCOMM06*, Sep. 2006.

[9] S. Gilbert and N. Lynch, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services,” in *SigAct News*, Jun 2002.

[10] R. Lipton and J. Sandberg, “PRAM: A Scalable Shared Memory,” Princeton, Tech. Rep. CS-TR-180-88, 1988.

[11] M. Frigo, “The Weakest Reasonable Memory Model,” Master’s thesis, MIT, 1988.

[12] A. Nayate, M. Dahlin, and A. Iyengar, “Transparent Information Dissemination,” in *ACM/IFIP/USENIX 5th International Middleware Conference*, Oct. 2004.

[13] T. P. P. Council, “TPC BENCHMARK W,” http://www.tpc.org/tpcw/spec/-tpcw_V1.8.pdf, 2002.

[14] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen, “Ivy: a Read/write Peer-to-peer File System,” in *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[15] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers, “Flexible Update Propagation for Weakly Consistent Replication,” in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, Oct. 1997.

[16] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam, “Taming aggressive replication in the pangaea wide-area file system,” in *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[17] A. Sherman, P. Liesiecki, A. Berkheimer, and J. Wein, “ACMS: Akamai Configuration Management System,” in *NSDI05*, May 2005.

[18] L. Lamport, “On Interprocess Communications,” *Distributed Computing*, pp. 77–101, 1986.

[19] J. Yin, L. Alvisi, M. Dahlin, and C. Lin, “Volume Leases to Support Consistency in Large-Scale Systems,” *IEEE Transactions on Knowledge and Data Engineering*, Feb. 1999.

[20] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control adn Receivery in Database Systems*. Addison Wesley, 1987.

[21] R. Thomas, “A Majority Consensus Approach To Concurrency Control for Multiple Copy Database,” in *ACM Transactions on Database Systems*, Jun. 1979, pp. 180–209.

[22] S. Cheung, M. Ahamad, and M. H. Ammar, “Optimizing Vote and Quorum Assignments for Reading and Writing Replicated Data,” *IEEE Transactions on Knowlegde and Data Engineering*, vol. 1, no. 3, pp. 387–397, Sep. 1989.

[23] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser, “Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System,” in *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, Dec. 1995, pp. 172–183.

[24] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler, “Using Smart Clients to Build Scalable Services,” in *Proceedings of the 1997 USENIX Technical Conference*, Jan. 1997.

[25] D. Malkhi and M. Reiter, “An Architecture for Survivable Coordination in Large Distributed Systems,” *IEEE Transactions on Knowledge and Data Engineering*, pp. 187–202, Mar. 2000.

[26] H. Garcia-Molina and D. Barbara, “How to Assign Votes in a Distributed System,” in *Journal of the ACM 32 (4)*, 1985.

[27] D. Gifford, “Weighted Voting for Replicated Data,” in *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, Dec. 1979.

[28] D. Barbara and H. Garcia-Molina, “The Reliability of Vote Mechanisms,” *IEEE Transactions on Computers*, vol. 36, no. 10, pp. pp 1197–1208, Oct. 1987.

[29] C. Gray and D. Cheriton, “Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency,” in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 1989, pp. 202–210.

[30] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, “Scale and Performance in a Distributed File System,” *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 51–81, Feb. 1988.

[31] E. Pierce and L. Alvisi, “A Recipe for Atomic Semantics for Byzantine Quorum Systems,” 2000. [Online]. Available: citeseer.ist.psu.edu/pierce00recipe.html

[32] J.-P. Martin, L. Alvisi, and M. Dahlin, “Minimal Byzantine storage,” in *Distributed Computing, 16th international Conference, DISC 2002*, Oct. 2002, pp. 311–325. [Online]. Available: http://link.springer.de/link/service/series/0558/tocs/t2508.htm

[33] P. Bernstein and N. Goodman, “The Failure and Recovery Problem for Replicated Distributed Databases,” *ACM Trans. Database Syst.*, vol. 14, no. 2, pp. 264–290, 1984.

[34] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, “An Integrated Experimental Environment for Distributed Systems and Networks,” in *OSDI02*, Dec. 2002. [Online]. Available: citeseer.ist.psu.edu/white02integrated.html

[35] H. Yu and A. Vahdat, “Design and Evaluation of a Conit-based Continuous Consistency Model for Replicated Services,” *ACM Transactions on Computer Systems*, pp. 239–282, Aug. 2002.

[36] L. Gao, M. Dahlin, J. Zheng, L. Alvisi, and A. Iyengar, “Dual-quorum replication for edge services,” in *Middleware.*, Nov. 2005.

[37] J. Paris and D. Long, “Efficient Dynamic Voting Algorithms,” in *Int’l Conference on Data Engineering*, 1988.

[38] R. van Renesse and A. Tanenbaum, “Voting with Ghosts,” in *Proceedings of the Eighth International Conference on Distributed Computing Systems*, 1988, pp. 456–462.

[39] F. Schneider, “Implementing Fault-tolerant Services Using the State Machine Approach: A tutorial,” *Computing Surveys*, vol. 22, no. 3, pp. 299–319, Sep. 1990.

[40] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, “Zyzzyva: Speculative Byzantine Fault Tolerance,” in *Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[41] M. Herlihy and J. Wing, “Linearizability: A Correctness Condition for Concurrent Objects,” *ACM Trans. Prog. Lang. Sys.*, vol. 12, no. 3, 1990.

[42] H. C. Li, A. Clement, A. Aiyer, and L. Alvisi, “The Paxos Register,” in *Proceedings of the 26th IEEE International Symposium Reliable Distributed Systems*, Oct. 2007.

[43] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Communications of the ACM*, vol. 21, no. 7, July 1978.

[44] M. Castro and B. Liskov, “Practical Byzantine Fault Tolerance and Proactive Recovery,” *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[45] Y. Amir, “Replication using group communication over a partitioned network,” Ph.D. dissertation, 1995. [Online]. Available: citeseer.ist.psu.edu/amir95replication.html

[46] S. Cheung, M. Ahamad, and M. Ammar, “The Grid Protocol: a High Performance Scheme for Maintaining Replicated Data,” in *Proceedings of the Sixth International Conference on Data Engineering*, 1990, pp. 438–445.