

Dual-Quorum Replication for Edge Services

Lei Gao, Mike Dahlin, Jiandan Zheng, Lorenzo Alvisi

The University of Texas at Austin

Arun Iyengar

IBM T.J. Watson Research Center

Abstract

This paper introduces dual-quorum replication, a novel data replication algorithm designed to support Internet edge services. Dual-quorum replication combines volume leases and quorum based techniques in order to achieve excellent availability, response time, and consistency for read/write objects 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. Through both analytical and experimental evaluation of a prototype, we show that the dual-quorum protocol can (for the workloads of interest) approach the excellent 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.

1 Introduction

This paper introduces dual-quorum replication, a novel data replication algorithm motivated by the desire to support data replication for edge services [1, 3, 10, 29]. As Figure 1 illustrates, the Internet edge service model attempts to improve service availability and latency by allowing clients to access the closest available edge servers rather than a centralized server (or a centralized server cluster). But as Figure 1 also indicates, in order to provide a single service from multiple locations, service logic (code) replicated on all edge servers must access a collection of shared data. Thus, support for data replication is a key problem in realizing the promise of Internet edge services.

By exploiting object-specific workload characteristics, we seek to design a data replication system for edge services that offers good trade-offs among availability, consistency, and response time. Although it is provably impossible to provide simultaneously optimal consistency, optimal availability, and optimal performance for *general-case* wide-area-network replication [5, 17], we can, perhaps, provide nearly optimal behavior for *specific objects* by taking advantage of a given application's workload characteristics. For example, our previous studies show how to provide nearly optimal replication for *information dissemination* applications such as

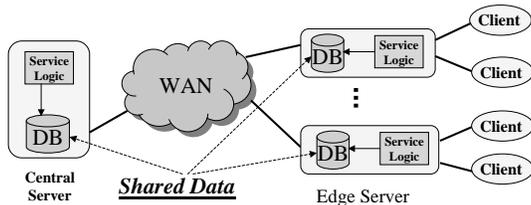


Figure 1: Internet edge service architecture

news [21] and *e-commerce* applications such as TPC-W [10]. In particular, 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 to use weak consistency—and thereby introduce considerable complexity—for a fourth category of objects: multi-writer, multi-reader objects such as per-customer *profile* information (e.g., name, account number, recent orders, credit card number, and address.) We, like several other systems [24, 26, 32], made use of a Read-One, Write-All-Asynchronously (ROWA-A) protocol based on local reads and asynchronous epidemic propagation of writes. ROWA-A protocols provide excellent read performance and availability; and although ROWA-A protocols allow applications to observe inconsistencies between reads and writes, such inconsistencies should be rare because multi-reader, multi-writer shared objects often have workloads with low concurrency to any given object. For example, in our edge-server TPC-W application, reads and writes to a given customer’s profile typically come from just one edge server for some interval of time, until the customer is redirected to a different server. Unfortunately, although inconsistencies are rare for the workloads of interest, these rare cases introduce considerable complexity into the design, because all cases must be handled no matter how rare they are and because reasoning about corner cases in consistency protocols is complex. Furthermore, because reads can always complete locally, 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.

By introducing dual-quorum replication, this paper provides the key missing piece to achieve highly-available, low-latency, and consistent data replication for a range of edge services. In particular, dual-quorum replication optimizes these properties for data elements that can be both read and written from many locations, but whose reads and writes exhibit locality in two dimensions: (1) at any given time access to a given element tends to come from a single node and (2) reads tend to be followed by other reads and writes tend to be followed by other writes. For other workloads, our algorithm continues to provide regular consistency semantics [16], but its performance and availability may degrade.

Our dual-quorum replication protocol combines ideas from volume leases [30] and quorum based techniques [11, 12]. The protocol employs two quorum systems, an input quorum system (*IQS*) and an output quorum system (*OQS*). Clients send their writes to the *IQS* and they read from the *OQS*. The two quorum systems communicate with each other when necessary to synchronize the state of replicated objects. By using two quorum systems, we are able to optimize construction of the *OQS*'s read quorums to provide low latency and high availability for reads while optimizing construction of the *IQS*'s write quorums to provide modest overhead and high availability for writes. In particular, *OQS* nodes cache data from the *IQS* servers using a quorum-based generalization of Yin et al.'s volume lease protocol [30], which invalidates individual cached objects as they are updated. The protocol uses short-duration volume leases to allow writes to complete despite network partitions and aggregates these leases across large numbers of objects in a volume to amortize the cost of renewing short leases. Using our dual-quorum protocol, workloads with large numbers of repeated reads (or writes) perform well because reads (or writes) can often be supplied by a read-optimized *OQS* read quorum (or write-optimized *IQS* write quorum) without requiring communication with the *IQS* (or *OQS*).

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,¹ majority quorums, and grid quorums [7]. For the important special case of single-node *OQS* read quorums, average read response time can approach a node's local read time, making the read performance of this approach competitive with ROWA-A epidemic algorithms such as Bayou [26]. But, the dual quorum approach avoids suffering the weak consistency guarantees and resulting complexity inherent in ROWA-A designs. Additionally, the analytical evaluations show that the overall availability of the dual-quorum protocol is competitive with the majority quorum protocol for the targeted workloads. Finally, for the targeted workloads, the communication overheads of this approach are comparable with 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 internal nodes.

The main contribution of this paper 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 not be uncommon for systems that can, in principle, have any node read or write any item of data to, in practice, experience sufficient locality to benefit from our approach.

Our paper is organized as follows. Section 2 presents our system model and a set of assumptions on which

¹Note that ROWA protocols are, in fact, a special case of quorum protocols, but they are often treated separately in the literature.

our system is built. In Section 3, we present our system’s design. 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

Our edge service environment consists of a collection of edge server nodes that each play one or more of the following three roles: (a) *front end* nodes 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 (OQS)* nodes that process client read requests; and (c) *Input Quorum System (IQS)* nodes that process client write requests. We assume a *request redirection architecture* that directs 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 [15, 31]. Note that clients are unaware of the underlying data storage system and never contact the *OQS* or *IQS* 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 cannot issue incorrect requests or replies. The network may delay, duplicate, or reorder messages. We assume secure communication among nodes and that if the network corrupts a message, this corruption is detected by low-level checksums and the message is silently discarded. Each node can read a local real-time clock and that there exists a maximum drift rate *maxDrift* between any pair of clocks.

For performance, our system assumes that concurrent reads and writes to a given object by different nodes are rare. But, for correctness, we must define the system’s consistency semantics in the presence of concurrent reads and writes to the same object. The dual quorum design provides *regular* semantics [16]: a read r that is not concurrent with any write returns the value of the latest write that completed before r began and a read r that is concurrent with one or more writes returns one of (a) the value of the last write that completed before r began, or (b) the value of one of the writes concurrent with r .

For convenience of exposition, we describe interactions with a quorum system in terms of a QRPC operation [18]. $replies = QRPC(system, READ/WRITE, request)$ sends *request* to a collection of nodes in the specified quorum *system* (e.g., the *IQS* or *OQS*). The QRPC call then blocks until a set of *replies* constituting the specified quorum (*READ* or *WRITE*) 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

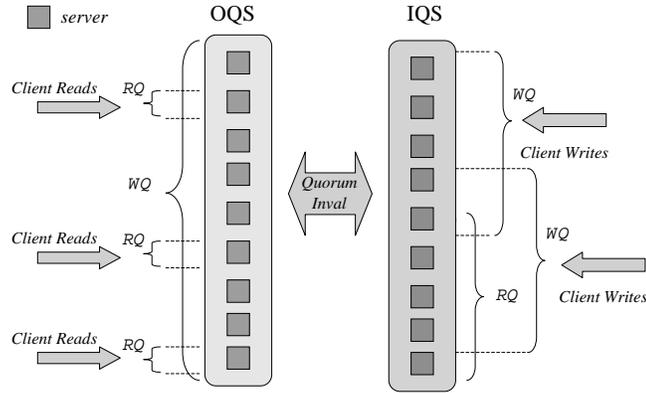


Figure 2: Dual quorum architecture overview. Client reads and writes come from edge-server clients of the dual-quorum system not from untrusted services clients.

to select which nodes from *system* to send requests to, and they may select different retransmission strategies: our simple implementation always transmits requests to the local node if the local node is a member of *system*; it then randomly selects a sufficient number of additional nodes to form a *READ* or *WRITE* quorum and transmits the request to them; retransmissions are each to a new randomly selected quorum using an exponentially-increasing retransmission interval. A more aggressive implementation might send to all nodes in *system* and return when the fastest quorum has responded or might track which nodes 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. The basic idea is to separate the read and write quorum into two quorum systems so that they can be optimized individually to improve response time and availability for read-dominated or write-dominated workloads. The read and write quorums of the *OQS* and *IQS* can be separately configured in any way desired, but we would expect one common configuration to be to optimize read performance by having the *OQS* span all nodes in the system with a read quorum size of 1 and to get good write availability by having the *IQS* span a modest number of nodes with any majority of the *IQS* nodes forming a write quorum. As Figure 2 illustrates, in the dual quorum system clients retrieve objects from *IQS* and send object updates to *OQS*. The two quorum systems conditionally synchronize with each other to maintain the consistency of data replicated on them when processing both reads and writes.

To simplify the discussion, we present the protocol in two steps. First, we will discuss the basic dual-quorum protocol, a simplified asynchronous protocol, in Section 3.1. This protocol allows separate 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. Then, in Section 3.2 we describe how we introduce volume leases to improve write availability while retaining good read performance.

3.1 Dual quorum protocol

In this section, we present the basic dual-quorum protocol by first explaining the behavior of quorums in both *IQS* and *OQS*; then we explain how each node acts in response to events in order to achieve the required quorum-level behavior even in the face of concurrent reads and writes.

OQS servers hold cached copies of objects that are stored on *IQS* servers. A cached object has two states, *valid* or *invalid*. When the read quorum contains a valid object, a read can be processed locally. Otherwise, the read quorum in *OQS* must read the object from a read quorum in *IQS*. To process a write, a write quorum in *IQS* must ensure that no read quorum in *OQS* holds a valid old copy of the object. To this end, the write quorum in *IQS* sends invalidations to and receives acknowledgments from a write quorum in *OQS*.

The key idea is that the *OQS* alone can often handle reads and the *IQS* alone can often handle writes. If the workload consists of consecutive reads on an object with no intervening writes, all subsequent reads to the same read quorum in *OQS* can be processed locally. Similarly, a write quorum in *IQS* does not need to send invalidations to *OQS* as long as no read quorum in *OQS* holds a valid old copy of the object. When the workload consists of consecutive writes to the same object, all subsequent writes to the same write quorum in *IQS* can suppress their invalidations to *OQS*.

In the remainder of this subsection, we detail our protocol by describing the actions taken by individual nodes that ensure read and write quorums behaviors as described above.

Data structures. Each *IQS* node maintains the following state for each object o : $lastWriteLC_o$ stores the logical clock of the last write to o , $lastReadLC_o$ stores the value of $lastWriteLC_o$ from the time of the last read of o , $lastAckLC_{o,n}$ stores the logical clock contained in the highest invalidation reply from node n for o , and $value_o$ stores the value of o . Each node in *IQS* maintains a logical clock $logicalClock$ whose value is always at least as large as the node's largest $lastWriteLC_o$ for any object o . Each node in *OQS* maintains the following per-object o per-node n state: $epoch_{o,n}$ indicates the last epoch for which a valid object lease on o from n was held, $logicalClock_{o,n}$ indicates the highest version number (logical clock) of o for which an invalidation or update has been received from n , and $valid_{o,n}$ is true if $logicalClock_{o,n}$ corresponds to an update (false if it corresponds to an invalidate). Finally $value_o$ stores the update body for the highest logical clock received in any update message for o from any node.

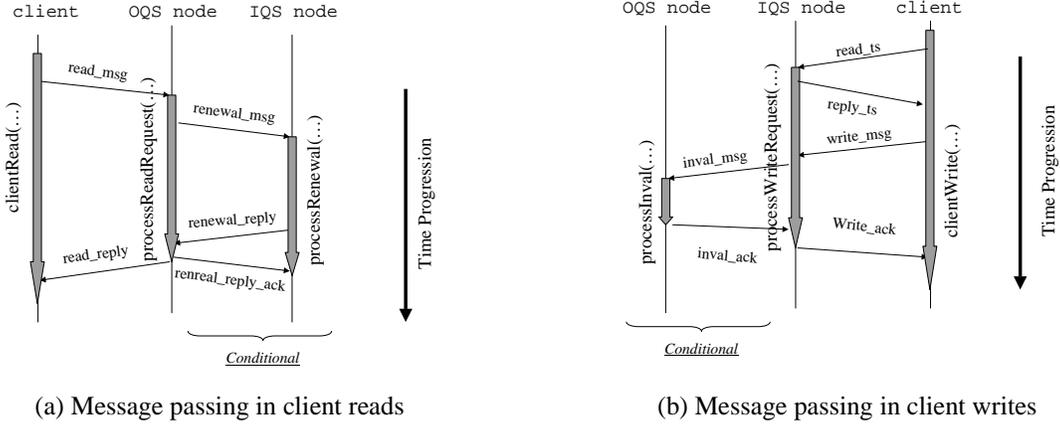


Figure 3: Message passing in the dual-quorum protocol

Object validity. The system maintains the following key invariant: If node j in OQS has from node i in IQS a valid object o ($j.valid_{o,i}$) then node i in IQS knows node j in OQS has a valid object callback ($i.lastReadLC_o > i.lastAckLC_{o,j}$).

Client read. Figure 3(a) shows the message exchange caused by a client read. From the client's point of view, a dual-quorum read is the same as a standard quorum read [11, 12]. *client* sends a read request to the *OQS* via *QRPC*. After receiving replies from a read quorum in *OQS*, *client* selects the value with the highest logical clock

A node j in *OQS* that receives a client read request first checks whether the object is valid. If so, j returns the object's locally-stored logical clock and value. If not, j renews the object by sending object renewal messages to *IQS* using *QRPC*. After receiving replies R from a read quorum in *IQS*, j updates its local state ($\forall i, s.t. i \in R$: if $R.r_{o,i}.lc \geq logicalClock_{o,i}$, then $logicalClock_{o,i} := R.r_{o,i}.lc$ and $valid_{o,i} := true$). Then, j updates $value_o$ with the value in the reply with the highest logical clock and return both the value $value_o$ and the highest logical clock to the client. Each *IQS* server that receives an object renewal message returns to the *OQS* server $value_o$ and $lastWriteLC_o$ and then updates $lastReadLC_o = \max(lastReadLC_o, lastWriteLC_o)$.

Client write. Figure 3(b) shows the message exchange caused by a client write. Just like the standard quorum write protocol [11, 12], *client* first queries *IQS* using *QRPC* to retrieve the highest logical clock from a read quorum in *IQS*. Next, *client* advances the logical clock and embeds it in the write request that is then sent to the *IQS* via *QRPC*. The write completes after *client* receives acknowledgments from a write quorum in *IQS*.

An *IQS* server i that receives a client request for the highest logical clock of the last completed write responds with its logical clock $logicalClock$. When i receives a client write whose logical clock is larger than

```

1 processLCReadRequest(){
2   sendMsg(CLIENT_LC_READ_REPLY, logicalClock);
3 }
4
5 processWriteRequest(Object o, Value v, LogicalClock lc){
6   if (lc > lastWriteLCo){
7     valueo := v;
8     lastWriteLCo := lc;
9     // ensure an OQS write quorum is invalidated
10    while (!isOWQInvalid(o, lc)){
11      invalidateOWQ(o, lc); // see text for descriptions
12    }
13  }
14  sendMsg(CLIENT_WRITE_ACK, o, lc);
15 }
16
17 processInvalAck(Object o, Sender j, LogicalClock lc){
18   // update the last inval ack in the record for the sender
19   lastAckLCo,j := MAX(lastAckLCo,j, lc);
20 }
21
22 processVLRenewal(Volume v, Sender j, RequestorTime tv,0){
23   expiresv,j := L + currentTime;
24   sendMsg(VOLUME_RENEW_REPLY, delayedv,j, L, epochv,j, tv,0);
25 }
26
27 processVLRenewalAck(Volume v, Sender j, LogicalC lc){
28   // remove delayed invals already applied at the sender
29   ∀k, s.t. invalk,j ∈ delayedv,j {
30     if (lc ≥ invalk,j.lc){
31       delete invalk,j;
32     }
33   }
34 }
35
36 processObjRenewal(Object o){
37   // update the last read logical clock
38   lastReadLCo := lastWriteLCo;
39   sendMsg(OBJECT_RENEW_REPLY, valueo, lastWriteLCo);
40 }

```

Figure 4: IQS server operations (pseudocode) - Dual quorum with volume leases

```

1 processVLRenewReply(Volume v, Sender i, Lease L, Epoch e, DI di,
2   RequestorTime tv,0){
3   expiresv,i := MAX(expiresv,i, tv,0 + L * (1 - maxDrift));
4   epochv,i := MAX(epochv,i, e);
5   // apply delayed invals if exist in the reply
6   ∀k, s.t. invalk,i ∈ di {
7     if (invalk,i.lc > logicalClockk,i){
8       logicalClockk,i := invalk,i.lc;
9       validk,i := false;
10    }
11  }
12  sendMsg(VOLUME_RENEW_REPLY_ACK, v, MAX(di.lc));
13 }
14
15 processInval(Object o, Sender i, LogicalClock lc){
16   // update the local logic clock and object status
17   if (logicalClocko,i < lc){
18     logicalClocko,i := lc;
19     valido,i := false
20   }
21   sendMsg(INVAL_ACK, lc);
22 }
23
24 processReadRequest(Object o){
25   // ensure the local object and volume are valid
26   while (!isLocalValid(o)){
27     // if not, renew invalid volume or object or both
28     validateLocal(o);
29   }
30   // once both leases are validated, send reply to client
31   lc := MAX∀i, s.t. valueo,i = true(logicalClocko,i);
32   sendMsg(CLIENT_READ_REPLY, valueo, lc);
33 }
34
35 processRenewReply(Object o, Sender i, Epoch epoch, LogicalClock
36   lc, ObjectValue value){
37   epocho,i := MAX(epocho,i, epoch);
38   if (logicalClocko,i ≤ lc){
39     logicalClocko,i := lc;
40     valido,i := true;
41   }
42   if (valido,i = true &&
43     logicalClocko,i ≥ MAX∀k, s.t. k ∈ IQS(logicalClocko,k)){
44     valueo := value;
45   }
46 }

```

Figure 5: OQS server operations (pseudocode) - Dual quorum with volume leases

that associated with the last completed write of o on i ($lastWriteLC_o$), i updates $lastWriteLC_o$ and $value_o$ with those in the write. Then, to ensure that a write quorum in OQS is unable to read the old version of the data, i performs one of the following tasks: (a) if no OQS server has renewed since the completion of the last write, (e.g. $\forall j, s.t. j \in OQS, lastReadLC_o < lastAckLC_{o,j}$), i suppresses invalidations to OQS; (b) otherwise, i sends invalidations v with the logical clock of the write to OQS using *QRPC*. The write completes after receiving invalidation replies from a write quorum in OQS, at which point i updates $lastAckLC_{o,j}$ for all j in the *QRPC* reply and returns to the client.

An OQS server j that receives from node i in IQS an invalidation with a logical clock $lc_{o,i}$ compares $lc_{o,i}$ with $logicalClock_{o,i}$. If the invalidation has the higher logical clock, j updates the local state ($logicalClock_{o,i} = lc_{o,i}$ and $valid_{o,i} = false$). Finally, j sends an invalidation acknowledgment back to i .

3.2 Dual quorum with volume leases

The basic protocol just described allows one to vary read and write quorum sizes independently. However, our application would benefit from using a read quorum size of 1 so that reads can be serviced locally; 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 could lead to unacceptable write availability because it could require a write to contact all nodes in the OQS to invalidate cached data. We therefore adapt Yin et al.’s volume lease protocol [30] to support very small read quorums in OQS while retaining acceptable availability on writes.

We first describe the quorum-level interactions of the Dual Quorum with Volume Lease (DQVL) protocol. We then detail how each node in IQS and OQS acts in order to affect these quorum-level behaviors.

We group objects into collections of objects called volumes. To process a read, a read quorum in OQS must hold both a valid *volume lease* and a valid *object lease* for some read quorum in IQS. A lease represents permission to access some object that expires at some specified time [13]. When a read quorum already holds these leases, a read can be processed locally. Otherwise, the read quorum in OQS must acquire volume and/or object leases from a read quorum in IQS.

To process a write of object o in volume v , a write quorum in IQS must ensure that no read quorum in OQS holds the leases necessary to read an old version of o . It does so by (a) sending invalidations of o to nodes in OQS (and receiving acknowledgments from them) or (b) waiting for the volume lease on v to expire at one or more nodes in OQS and then enqueueing invalidations of o as *delayed invalidations* [30] that must be processed before the volume lease is renewed.

The key idea to making this approach work well is that volume leases are of short duration while object leases are of long duration.² This combination yields good read response time; nodes in OQS can cache objects locally for a long time, and although they must frequently renew volume leases, this cost is amortized across many objects in a volume. This combination also yields good write responsiveness and availability: a write can complete by invalidating nodes caching data *or* waiting for a (short) volume lease to expire.

A final implementation detail we take from Yin et al. [30] is to bound the size of the list of delayed invalidations for unreachable nodes using *epochs*. Volume lease renewals are marked with an epoch number, and when this epoch number changes, nodes conservatively assume all object callbacks have been revoked as described below.

We now explain details of this protocol by describing the actions each individual node takes in order to assure that the read and write quorums behave as just described. Pseudo-code for an IQS and OQS node is in

²For simplicity, we will assume infinite-length object leases or *callbacks* [14] Generalizing to finite-length object leases is straightforward and can help optimize space and network costs [9].

Figure 4 and Figure 5. Note that client behavior is as before—use QRPC to issue read and writes to IQS and OQS, respectively.

Data structures. Each node in IQS maintains a real time clock $currentTime$ (with bounded drift with respect to the other clocks as described in Section 2) and a logical clock $logicalClock$. Each IQS node also maintains the following per-volume v , per-OQS-node j state: $expires_{v,j}$ which indicates when v expires at j , $delayed_{v,j}$ which contains a list of delayed invalidations that must be delivered to j before v is renewed, and $epoch_{v,j}$ which indicates j 's current epoch number for v . Finally, each IQS node maintains the following per-object o state: $lastWriteLC_o$ stores the logical clock of the last write to o , $lastReadLC_o$ stores the value of $lastWriteLC_o$ from the time of the last read of o , $lastAckLC_{o,j}$ stores the logical clock contained in the highest invalidation reply from node j for o , and $value_o$ stores the value of o .

Each node in OQS maintains a bounded-drift real time clock $currentTime$. In addition, it maintains the following per-volume v per-IQS-node i state: $epoch_{v,i}$ is the highest epoch number for which a valid volume lease from i was held on v and $expires_{v,i}$ is the time when the lease on v from i will expire. And, it maintains the following per-object o per-IQS-node i state: $epoch_{o,i}$ indicates the last epoch for which a valid object lease on o from i was held, $logicalClock_{o,i}$ indicates the highest version number (logical clock) of o for which an invalidation or update has been received from i , and $valid_{o,i}$ is true if $logicalClock_{o,i}$ corresponds to an update (false if it corresponds to an invalidate). Finally $value_o$ stores the update body for the highest logical clock received in any update message for o from any node.

Volume and object validity. The system maintains the following key invariant: If node j in OQS has from node i in IQS both a valid volume v ($expires_{v,i} > currentTime$) and a valid object o ($epoch_{v,i} = epoch_{o,i}$ && $valid_{o,i}$) then node i in IQS knows node j in OQS has a valid volume lease ($expires_{v,j} > currentTime$) and valid object callback ($lastReadLC_o > lastAckLC_{o,j}$).

Client read. As detailed by **processReadRequest** in the pseudo-code, in A node j in OQS processes a client read of object o as follows. j must ensure Condition C : there exists a read quorum irq in IQS such that j holds both a valid volume lease and valid object lease from irq . If C is already true, then j can immediately return the value $value_o$ and the associated logical clock $MAX_{\forall i, s.t. i \in IQS}(logicalClock_{o,i})$.

If C is not true, then j performs a variation on QRPC. QRPC as defined in Section 2 sends and resends a request to different nodes until it receives a quorum of replies. This variation sends *different* requests to different nodes and processes replies until condition C becomes true. In particular, for each target node 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 read; (b) if just the volume has expired, it sends a volume renewal; or (c) if just the

object is invalid, it sends an object read. As detailed in the pseudo-code **processVLRenewReply**, j processes replies to volume renewal requests from IQS node i by applying the delayed invalidations included in the reply (in the same way as applying normal invalidations as described below) and updating $expires_{v,i}$ as well as $epoch_{v,i}$. To account for worst-case clock drift, j conservatively sets $expires_{v,i} = t_o + L * (1 - maxDrift)$ 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. Finally, j sends i a volume lease renewal acknowledgment (which i uses to clear its delayed invalidation queue.) As detailed in the pseudo-code **processRenewReply**, j processes object renewal replies from i by updating $epoch_{o,i}$, $logicalClock_{o,i}$, and $valid_{o,i}$; furthermore, if $valid_{o,i}$ is true and $logicalClock_{o,i}$ exceeds the logical clock of any other *valid* logical clock for this object, j updates $value_o$. The repeated sends and the processing of replies in this QRPC variation ensure that C eventually becomes true, at which point j returns $value_o$ and the associated logical clock ($logicalClock_{o,i_{max}}$) as the result of the read.

On the IQS side, node i in IQS processes volume renewal messages for volume v from node j as described in the pseudo-code **processVLRenewal**: i sends the delayed invalidations $delayed_{v,j}$ and the volume renewal, containing the epoch number $epoch_{v,n}$ and lease length L . i then records the volume expiration time ($expires_{v,j} = L + currentTime$). When i receives a volume lease renewal acknowledgment for volume v and logical clock lc from j , as detailed in the pseudo-code **processVLRenewalAck**, i clears all delayed invalidations with logical clocks up to lc from $delayed_{v,j}$. As **processObjRenewal** indicates, when i in IQS processes a read of object o from OQS node j , it replies with $value_o$ and $lastWriteLC_o$ and updates $lastReadLC_o = lastWriteLC_o$. Note that $lastReadLC_o$, $lastAckLC_{o,j}$, and $lastWriteLC_o$ allow i in IQS to track which nodes j in OQS may hold valid object callbacks. Finally, if an IQS server i wishes to garbage collect delayed invalidation state for j , i advances $epoch_{v,j}$ and deletes the delayed invalidations $delayed_{v,j}$. 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 . So all previously valid object leases from i immediately become invalid. Thus, 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 $valid_{o,i}$ with the missing delayed invalidations or (b) advancing $epoch_{v,i}$ by sending a volume renewal with a new epoch number.

Client write. A client first determines the highest logical clock of any completed write by calling IQS's **processLCReadRequest**. A node i in IQS responds to such a call for object o by returning the node's *global* logical clock $logicalClock$. A client then issues the actual write of object o . As detailed in **processWriteRequest** in the pseudo-code, if the write's logical clock exceeds that of the highest write seen so far ($lastWriteLC_o$), node i stores the write's logical clock and value. i then ensures that a write quorum in OQS is unable to read the old

version of the data by performing a variation on QRPC that “sends” differently to different nodes depending on whether their volume and object leases are valid. There are three cases for i to consider for node j , object o , and volume v : (a) if i knows o is invalid at j (e.g., $lastReadLC_o < lastAckLC_{o,j}$) then i need take no action for j ; (b) otherwise if o is valid at j but v is invalid at j (e.g., $expires_{v,j} < currentTime$) then i enqueues an invalidation in $delayed_{v,j}$ which will be processed at j when it renews its volume; or (c) both the object and volume are valid (e.g., $lastReadLC_o > lastInvalLC_{o,j}$) then j sends an object invalidation containing the write’s logical clock ($lastWriteLC_o$) to j . In this last case, if j receives an invalidation from i for object o with logical clock lc , then as the pseudo-code in **processInval** describes, j applies the invalidation: if the invalidation is the newest information about o from i (e.g., $lc > logicalClock_{o,i}$) then update the logical clock and validity information ($\{logicalClock_{o,i} = lc; valid_i = false\}$). Finally, if i receives an invalidation-acknowledgment from j for logical clock lc , then as the pseudo-code in **processClientInvalAck** describes, i updates $lastAckLC_{o,j} = max(lastAckLC_{o,j}, lc)$.

Correctness Due to space constraints, we omit the proof of correctness,³ which proves that the system has regular semantics [16]. In particular, it shows (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 (2) a read of o that is concurrent with one or more writes of o can return only (a) the value and logical clock from the completed write of o with the highest logical clock or (b) the value and logical clock from some concurrent write of o .

To give intuition for why DQVL provides *regular semantics*, consider the invariant: If node j in OQS has from node i in IQS both a valid volume v ($expires_{v,i} > currentTime$) and a valid object o ($epoch_{v,i} = epoch_{o,i} \ \&\& \ valid_{o,i}$) then node i in IQS knows node j in OQS has a valid volume lease ($expires_{v,j} > currentTime$) and valid object callback ($lastReadLC_o > lastAckLC_{o,j}$). For a read that is not concurrent with any writes: This invariant is established by having j renew its volume v and (or) object o from i . Therefore, j contains the last completed write $value_o$ on node i when j has both a valid volume v and a valid object o from node i . Furthermore, j will contain the last completed write $value_o$ on a write quorum iwq in IQS when j has both a valid volume v and a valid object o from a read quorum irq in IQS (because orq and owq intersects by at lease one node). Because a client write is performed on an iwq , $value_o$ held on j is actually the last completed client write in the system. Because j can not process any client read unless it holds both a valid volume v and a valid object o from a read quorum irq , j guarantees to always return the value $value_o$ of the last completed write in the system. For a read that is concurrent with some writes: Assume that the last completed write has logical clock lc_0 and a read r that is concurrent with some writes with logical clock $lc_1 \dots lc_n$ ($lc_t > lc_0$) is sent

³The detailed proof is presented in the extended technical report available at www.cs.utexas.edu/users/lqao/papers/dualQuorumTR.pdf.

to an *orq*. If the invariant is established in the *orq*, *r* returns the value associated with lc_0 . Otherwise, the *orq* will try to establish the invariant by querying an *irq*. Because some writes are being processed in *IQS*, the *irq* may return to the *orq* the value associated with any of the logical clock $o.lc_0 \dots o.lc_n$. Meanwhile, some *iwq* may send invalidations with logical clock $inval.lc_0 \dots inval.lc_n$ to the *orq* as the result of the concurrent writes. When the maximum logical clock received in the renew replies is less than that of any invalidations on any server *j* of the *orq*, this server keeps renewing from some *irq*. As long as those concurrent writes terminate, *j* will eventually receive $o.lc_n$ (the highest logical clock among all concurrent writes) from some *irq*. Therefore, *r* may return the value associated with any of the logical clock $lc_0 \dots lc_n$.

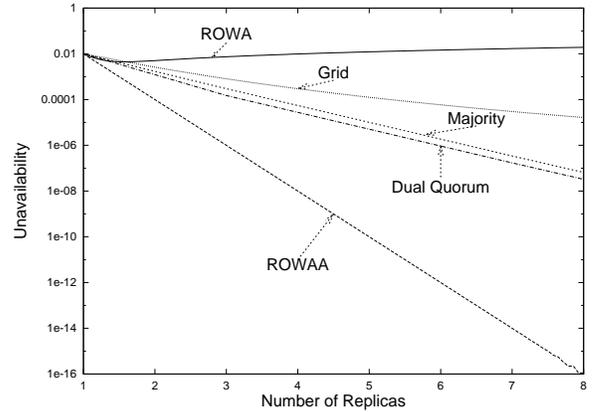
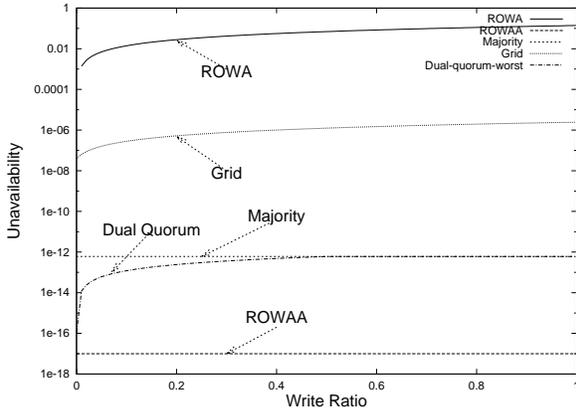
4 Evaluation

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

4.1 Analytical models

To help understand our analytical models, we first introduce some notation. For a read to complete in DQVL, a read quorum in *OQS* must be available and must hold a volume lease; additionally, if a volume lease must be acquired, a read quorum in *IQS* must also be available. When the volume lease is already held, we have a *read hit*. When a lease renewal is required, we have a *read miss*. Similarly, for a write to complete in DQVL, a write quorum in *IQS* must be available and a write quorum in *OQS* must be invalidated. So when there exists a write quorum in *OQS* with all its servers holding expired volume leases or invalid copies of the object, we have a *write suppress*. When invalidations must be sent to a write quorum of *OQS*, we have a *write thru*. In all of the analyses, we configure *IQS* as a majority quorum system to provide optimal availability [23] for writes and set the size of read quorum in *OQS* to one to optimize read performance for edge-server-like workloads. Future work will examine a broader range of quorum configurations.

Availability Figure 6 (a) and (b) illustrate the unavailability of DQVL in comparison with other protocols in log scale. The unavailability is computed as $1 - \text{availability}(av)$. An unavailability of 10^{-i} corresponds to the availability of *i* 9’s. Our simple model assumes the failure probability $p = 0.01$ and that failures (including server crashes and network failures) are independent.



(a) System unavailability with respect to the write ratio (b) System unavailability with respect to the number of replicas

Figure 6: System unavailability

The availability of both *read hit* and *read miss* are $\min(av_{orq}, av_{irq})$. The availability of both *write thru* and *write suppress* are $\min(av_{irq}, av_{iwq})$. Therefore, the availability of DQVL is $av_{DQVL} = (1 - w) * \min(av_{orq}, av_{irq}) + w * \min(av_{iwq}, av_{irq})$.⁴

Figure 6 (a) illustrates the systems' unavailability as we vary the write ratio and fix the number of replicas to 15. The key result is that DQVL's availability tracks that of the majority quorum. Note that the DQVL's availability measurement is pessimistic because a read can proceed without contacting any read quorum in IQS if the read quorum in OQS holds valid volume and object leases; this effect may mask some failures that are shorter than the volume lease duration. ROWAA protocol shows the best availability which increases with the number of replicas in the system. But, the cost of ROWAA's high availability is that the system has to manage conflict detection and resolution and it may not be able to provide worst case staleness guarantees during network partitions.

Figure 6 (b) illustrates systems' unavailability as we vary the number of replicas and fix the write ratio at 25%. It shows that the unavailability of DQVL has similar behavior as the majority quorum system and is lower by a factor which is determined by the write ratio, as shown in (a). ROWAA protocol has the lowest unavailability which decreases as the number of replicas increases.

Response time In this subsection we analyze the response time of DQVL and make comparisons with other popular protocols in the context of the edge service environment where every client connects to a closest 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 closest edge server, it has to go through a WAN-like connection, *wan*, with 86 ms RTT. Our model

⁴The detailed analysis is presented in the extended technical report available at www.cs.utexas.edu/users/lgao/papers/dualQuorumTR.pdf.

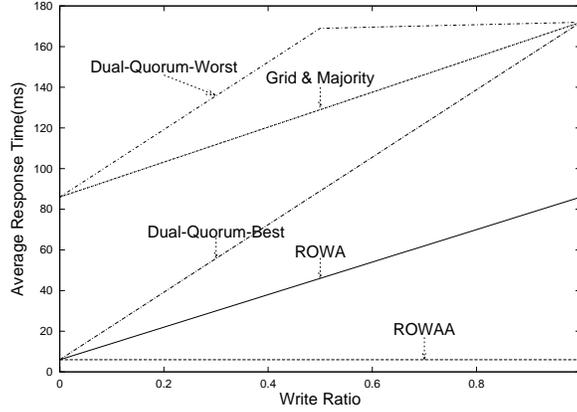


Figure 7: Average response time

calculates the average response time as $resp_r * (1 - w) + resp_w * w$. Because of space constraints, we omit the detailed derivation of $resp_r$ and $resp_w$ for DQVL.⁴

Average response times of various protocols are illustrated in Figure 7 where we plot the average response times while varying the write ratio and fix the number of replicas to 15. DQVL provides the best case response time when workloads consist of only *read hits* and *write suppresses*. As indicated by the third curve from the bottom, DQVL *read hits* yield performance competitive with ROWA-A epidemic algorithms against read-dominated workloads because they only need to communicate with the closest server. DQVL has the worst case response time against workloads consisting of a large number of *read misses* and *write thrus*. DQVL *read misses* and *write thrus* 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 DQVL) require one *wan* trip to retrieve the highest timestamp and another to perform the actual write, their response time is twice as much as that of ROWAA. *write thrus* require an additional *wan* trip to invalidate a write quorum in *OQS*. At 50% write ratio, when DQVL has the maximum amount of *write thrus*, the overall response time of DQVL reaches its worst case as indicated by the top most curve.

Communication overhead This section analyzes DQVL’s communication overhead in terms of the number of message exchanges required in processing a client request. To simplify the model, the study assumes weights of all message types are equal. Due to space constraints, we omit the detailed discussion on the communication overhead model for DQVL.⁴

Figure 8 shows the average number of messages required to process a client request in log scale as we vary the write ratio and fix the number of replicas at 15 in (a) and vary the number of replicas and fix the write ratio at 25% in (b).

In the worst case, DQVL can have high communication overhead as illustrated by the top most curve in

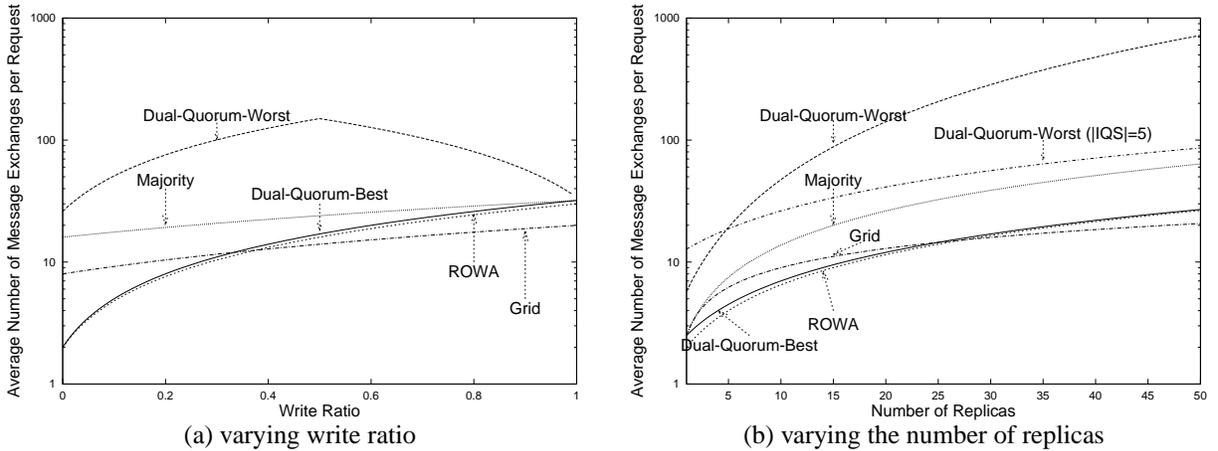


Figure 8: Communication overhead

(a). Because in the worst case, most reads are *read misses* and most writes are *write thrus*. Such situations occur when the workload consists of interleaved reads and writes on the same object. A read following a write always requires *OQS* to renew from a read quorum in *IQS* and a write following a read always requires *IQS* to invalidate a write quorum in *OQS*. When the write ratio is at 50% in this case, every read interleaves with a write and DQVL yields the worst communication overhead.

However, DQVL’s overhead should be comparable to other approaches in practice. First, workloads that DQVL is designed to face are dominated by reads. Consecutive reads are likely to benefit from having objects cached on OQS servers, i.e. the target workloads have a large number of *read hits*. As indicated by the second curve from bottom in (b), DQVL’s overhead is close to ROWA when there is a large number of *read hits*. Second, the design of DQVL allows us to vary the OQS size to meet read performance goals while varying the IQS size to balance overhead vs. availability goals. The second curve from the top in (b) shows that once we fix IQS at a moderate size (5 in this case) while letting the OQS size grow, the communication overhead yield by DQVL is at the same level as the majority quorum without requiring many *read hits* in the workload.

4.2 Experimental evaluation

We have developed replication prototypes for DQVL, majority quorum, and ROWA protocols. All the prototypes are built in Java and run on eight Emulab nodes. In our prototype experiment, we set the “lan” delay between a client and its closest edge server to 8 ms. The “wan” delay between the client and other edge servers is 86 ms. And the network delay among edge servers is 80 ms.

Through our prototype experiment, we demonstrate that the dual-quorum protocol is highly suitable for managing replicated data with the workload characteristics of the TPC-W⁵ profile object, i.e. a workload

⁵TPC-W is a transaction processing benchmark for the web [8].

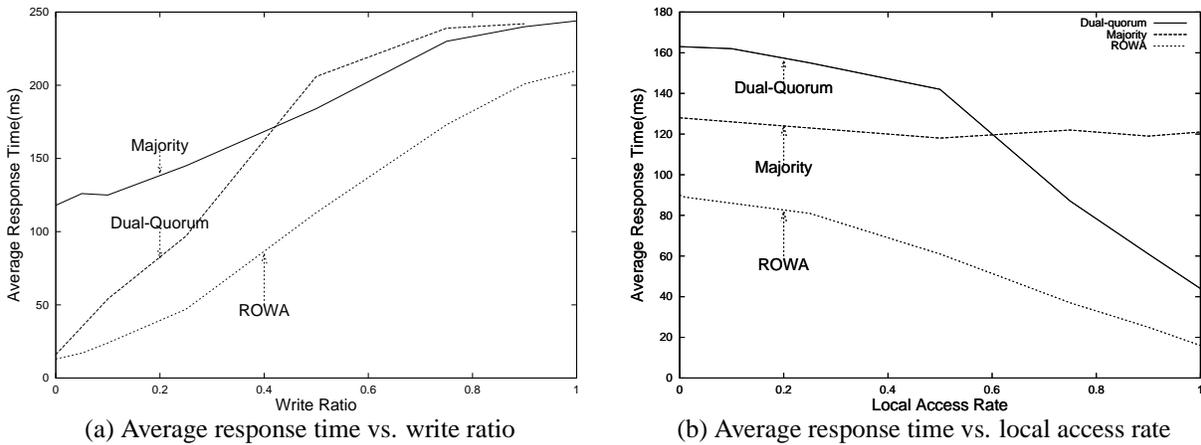


Figure 9: Experimental results - average response time

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.

Write ratio The dual-quorum protocol provides excellent performance for workloads with low write ratios. Figure 9 (a) illustrates the average response time of the dual-quorum system with eight replicas. Similar to our analytical result, the response time of the dual-quorum system is competitive with that of the ROWA protocol family when the write ratio is below 20%. Furthermore, at 5% write rate, the write rate of the TPC-W profile object, the average response time of the dual-quorum protocol is 41 ms,⁶ which is about twice as much as that of the ROWA and one sixth of that of the majority quorum system. The low response times of both the dual-quorum and ROWA are due to the fact that reads are performed only on the closest edge server which has 8 ms network delay.

Access locality In this experiment, we fix the write ratio to 5% as specified for the TPC-W profile object and the total number of replicas to eight. The system has 10 volumes and 10 objects under each volume. The client has a 95% probability of accessing the objects in the same volume and a 5% chance of accessing objects in other volumes. Under normal circumstances, a client always gets connected to the same closest server. But the unavailability of the closest server or the geographical movement of the client can sometimes result in a client being routed to other edge servers. The response time for those reads will be higher than the normal response time experienced when reading from the closest server because the probability of a read hit is lower.

Figure 9 (b) illustrate the average response time of DQVL as we vary the percentage of client requests sent

⁶Response times of all prototypes are higher than the underlying network delays due to experimental errors and un-tuned code.

to its closest edge server. When the access locality is low, reads sent to DQVL are often routed to distant servers resulting in high response time. As the access locality increases, reads are more likely to land on the client's closest server where target objects are usually valid so performance is good. The majority quorum system is not affected by the access locality since clients always send their requests to a quorum of servers.

5 Related Work

In read-one/write-all (ROWA) protocol the “read-one” property yields excellent read availability and response time. But this protocol has limited write availability and response time because writes can not complete if any of the replicas are unavailable. Protocols with the read-one/write-all-async property (ROWA-A) [20, 24, 25] 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-A [4] performs writes synchronously on the available replicas to provide better consistency, but its write latency is worse.

The primary-backup (or primary-copy) model [2] can tolerate network partitions by only allowing the partition with the primary server to perform writes. However the primary server becomes the bottleneck when it can not meet required levels of availability and performance. Group-communication based techniques, such as extended virtual synchrony [19], enable the election of a new primary by actively propagating updates to all group members and constantly running membership protocols to maintain the correct memberships. The new primary can be selected from active members and the change of the primary is also broadcast to all active members as well. This class of techniques has degraded performance in WANs because the membership protocol may always need to run to constantly include/exclude certain replicas when they are mistakenly considered as crashed/recovered due to slow WAN links. In addition, all primary-server based protocols are inflexibly in favor of reads' availability and performance.

Quorum based protocols [11, 12, 22, 27] can tolerate network partitions as long as connected replicas can form a quorum to process reads/writes. However, the reads' response time and availability of most quorum systems are worse than those of ROWAA or primary-backup based protocols because reads usually need to query a larger set of servers. Quorum based protocols may not be 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 [28] 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. But our use of invalidations also allows us to reduce the future message propagation to other replicas.

The traditional cache invalidation protocols [13, 30] 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 Conclusion

This paper 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.

Several important issues will be addressed in our future work. It will be interesting to configure both IQS and OQS to optimize other metrics. For example, we can configure the read quorum size in OQS to be larger than one to avoid timeouts on invalidations. We can also configure IQS as a grid quorum system [6] to reduce the overall system load. We are also interested in modifying DQVL to provide different consistency semantics (e.g. atomic semantics [16]) and comparing the cost difference.

References

- [1] Inc. Akamai Technologies. Akamai-The Business Internet - A Predictable Platform for Profitable E-Business. http://www.akamai.com/BusinessInternet/whitepaper_business.internet.pdf, 2004.
- [2] P. Alsberg and J. Day. A Principle for Resilient Sharing of Distributed Resources. In *the 2nd Intl. Conference on Software Engineering*, 1976.
- [3] 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*, August 2002.
- [4] P Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.
- [5] E. Brewer. Lessons from giant-scale services. In *IEEE Internet Computing*, July/August 2001.
- [6] 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*, pages 438–445, 1990.
- [7] S. Cheung, M. Ahamad, and M. H. Ammar. Optimizing Vote and Quorum Assignments for Reading and Writing Replicated Data. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):387–397, September 1989.
- [8] Transaction Processing Performance Council. TPC BENCHMARK W. http://www.tpc.org/tpcw/spec/-tpcw_V1.8.pdf, 2002.
- [9] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive Lease: A Strong Consistency Mechanism for the World Wide Web. In *Proceedings of IEEE Infocom*, March 2000.
- [10] 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*, March 2005.
- [11] H. Garcia-Molina and D. Barbara. How to Assign Votes in a Distributed System. In *Journal of the ACM* 32 (4), 1985.
- [12] D. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, December 1979.
- [13] 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*, pages 202–210, 1989.

- [14] 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*, 6(1):51–81, February 1988.
- [15] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-ninth ACM Symposium on Theory of Computing*, 1997.
- [16] L. Lamport. On interprocess communications. *Distributed Computing*, pages 77–101, 1986.
- [17] R. Lipton and J. Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Princeton, 1988.
- [18] D. Malkhi and M. Reiter. An Architecture for Survivable Coordination in Large Distributed Systems. *IEEE Transactions on Knowledge and Data Engineering*, pages 187–202, March 2000.
- [19] D. Malki, K. Birman, A. Schiper, and A. Ricciardi. Uniform Actions in Asynchronous Distributed Systems. In *ACM SIGOPS-SIGACT*, August 1994.
- [20] 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*, December 2002.
- [21] A. Nayate, M. Dahlin, and A. Iyengar. Transparent Information Dissemination. In *ACM/IFIP/USENIX 5th International Middleware Conference*, October 2004.
- [22] J. Paris and D. Long. Efficient Dynamic Voting Algorithms. In *Int'l Conference on Data Engineering*, 1988.
- [23] D. Peleg and A. Wool. The Availability of Quorum Systems. In *Information and Computation*, pages 210–223, 1995.
- [24] 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*, October 1997.
- [25] 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*, December 2002.
- [26] 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 ACM Symposium on Operating Systems Principles*, pages 172–183, December 1995.
- [27] R. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Database. In *ACM Transactions on Database Systems*, pages 180–209, June 1979.
- [28] R. van Renesse and A. Tanenbaum. Voting with Ghosts. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 456–462, 1988.
- [29] A. Whitaker, M. Shaw, and S. Gribble. Scale and Performance in the Denali Isolation Kernel. In *OSDI02*, December 2002.
- [30] 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*, February 1999.
- [31] 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*, January 1997.
- [32] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, pages 239–282, August 2002.