Dual-Quorum Replication for Edge Services

Lei Gao[1], Mike Dahlin[1], Jiandan Zheng[1], Lorenzo Alvisi[1], and Arun Iyengar[2]

[1] University of Texas at Austin, Austin TX 78712, USA
{lgao, dahlin, zjiandan, lorenzo}@cs.utexas.edu
[2] IBM TJ Watson Research Center, Yorktown Heights, NY 10598, USA
aruni@us.ibm.com

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 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-Async (ROWA-A) epidemic algorithms without suffering the weak consistency guarantees and resulting design complexity inherent in ROWA-Async 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 replication system for edge services that offers optimized trade-offs among availability, consistency, and response time. Although it is impossible to simultaneously provide optimal consistency, availability, and 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 news [22] and *e-commerce* applications such as TPC-W [10]. In particular, we developed customized consistency protocols for three categories of objects: (1)

G. Alonso (Ed.): Middleware 2005, LNCS 3790, pp. 184–204, 2005.

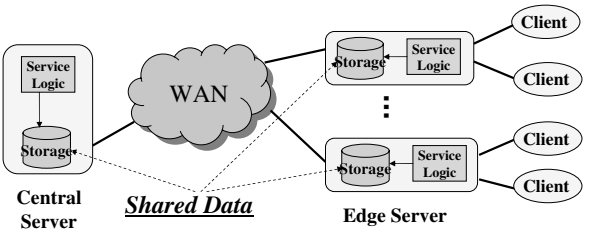


Fig. 1. Internet edge service architecture

single-writer, multi-reader objects like product descriptions and prices; (2) multi-writer, single-reader objects like customer orders; and (3) commutative-write, approximate-read objects like the 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 undesirable complexity—for a fourth category of objects: multi-writer, multi-reader objects such as TPC-W's per-customer *profile* information (e.g., name, account number, recent orders, credit card number, and address.) We, like several other systems [24,26,33], made use of a Read-One, Write-All-Asynchronously (ROWA-Async) protocol based on local reads and asynchronous epidemic propagation of writes. ROWA-Async protocols provide excellent read performance and availability; and although ROWA-Async 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 system 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 quorums [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 synchronize the state of replicated objects among them when necessary. 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 configuration 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-Async epidemic algorithms such as Bayou [26]. But, the dual quorum approach avoids suffering the weak consistency guarantees and resulting complexity inherent in ROWA-Async designs. Additionally, the overall availability of the dual-quorum protocol is competitive with the optimal 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 at 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 our system is built. In Section 3, we present our system's design. We compare our system with existing ones in Section 4 with

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

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 *application client* requests from across the Internet, execute application-specific processing, and act as *service clients* to the dual-quorum storage system; (b) *Output Quorum System* (*OQS*) nodes that process read requests; and (c) *Input Quorum System* (*IQS*) nodes that process write requests. We assume a *request redirection architecture* that directs application 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 application 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 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 (quorum-based remote procedure call) 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, but our protocol does not depend on any specific QRPC implementation. In particular, different implementations may choose different ways to select which nodes from *system* to send requests to, and they may select different retransmission strategies: our simple prototype 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 service clients retrieve objects from a read quorum in OQS and send object updates to a write quorum in IQS. The two quorum systems conditionally synchronize with each other to maintain the consistency of data replicated on them when processing both reads and writes.

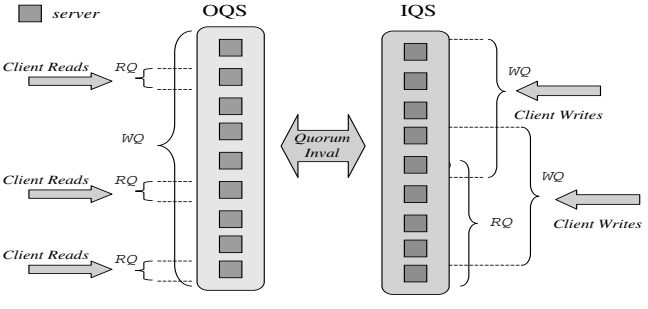


Fig. 2. Dual quorum architecture overview. Note that client reads and writes are issued by the service clients, not the application clients.

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.

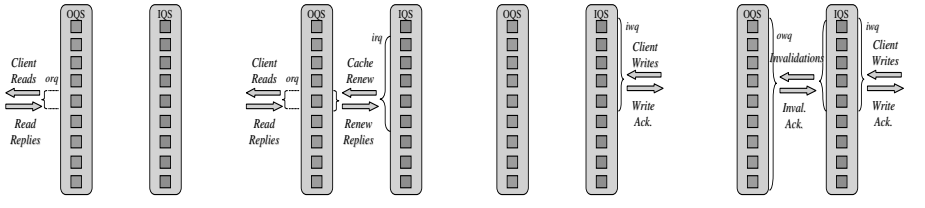
3.1 Dual Quorum Protocol

High level overview. The basic idea of the dual quorum protocol is to process reads and writes in two different quorum systems, *IQS* and *OQS* , and use a cache invalidation strategy to synchronize the state of objects replicated in *IQS* nodes and cached in *OQS* nodes.

Clients perform similar tasks for reading and writing data as in the conventional quorum based protocols. When a client read arrives in *OQS* , two possible scenarios can happen, as illustrated in Figure 3 (a) and (b). In a *read hit* case, the *OQS* read quorum contains a valid cache copy of the requested object, which is immediately sent back to the client. When there is a *read miss*, i.e. the cache copy on the *OQS* read quorum is invalid, the *OQS* read quorum validates the cache copy by querying an *IQS* read quorum for the latest update. Once the cache copy of the *OQS* read quorum is validated, the *OQS* read quorum sends the updated value to the client. There are also two scenarios when processing client writes, as illustrated in Figure 3 (c) and (d). In a *write suppress* case, the cache copy in an *OQS* write quorum is already invalid. The *IQS* write quorum can just apply the write to the local object and send the completion acknowledgment to the client. In the case of a *write through*, an *OQS* write quorum may hold a valid cache copy. Therefore, the *IQS* write quorum that receives the client write has to invalidate the cache copy on one *OQS* write quorum before the write can complete.

For workloads consisting of read bursts, the first read forces all *OQS* nodes of the read quorum to validate their cached copies. Therefore, all subsequent reads via that quorum are *read hits*. If we configure the *OQS* read quorum to contain only one node, reads becomes local, and the protocol can yield near optimal read response time and availability for read-dominated workloads. For workloads consisting of write bursts of the same data, the first write invalidates cached copies in an *OQS* write quorum, making all subsequent writes *write suppresses*. Naturally, we can configure *IQS* as a majority quorum system to provide near optimal write availability for such workloads.

Protocol details. The following paragraphs provide the details of the basic dual-quorum protocol by describing the actions taken at individual nodes.



(a) Read hit (b) Read miss (c) Write suppress (d) Write through

Fig. 3. Request processing scenarios

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, $lastAckLC_{o,j}$ stores the logical clock contained in the highest invalidation reply from *OQS* node j 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-IQS-node i state: $logicalClock_{o,i}$ 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.

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. 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 o is valid. This check is done by first finding the *IQS* nodes i that sends the highest $logicalClock_{o,i}$ to j. Object o is valid if $valid_{o,i} = TRUE$, invalid otherwise. If o is valid, 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 returns 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. 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 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 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 (DQVL)

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.

High level overview. We group objects into collections 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]. Similar to the basic dual quorum protocol described in the previous section, when an OQS read quorum holds both valid leases, all client reads processed by this read quorum are *read hit*. A *read miss* implies that either or both leases are invalid - they can be renewed by querying from an IQS read quorum. Similarly, a *write suppress* occurs when either or both leases are invalid in at least one OQS write quorum. To process a write in the *write through* scenario, the IQS write quorum can (a) invalidate the object lease in an OQS write quorum or (b) wait for the lease to expire on the volume containing the requested object in an OQS write quorum.

The key challenge in introducing volume leases is to manage the callback state when invalidations are suppressed at IQS when the volume lease expires in an OQS write quorum. When an IQS write quorum processes a write to o while the lease expires for the volume v containing o in an OQS write quorum, i.e. a *write suppress* scenario, the IQS write quorum has to enqueue the invalidation of o as a *delayed invalidation* [30]. All delayed invalidations of objects under v must be processed by the OQS write quorum before v's lease can be renewed so that all required callbacks to IQS are installed on OQS . Those callbacks ensure that OQS queries IQS to retrieve possible updates suppressed at IQS.

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

In this case, *OQS* suspects that all objects under this volume are updated at *IQS* and *OQS* needs to query an *IQS* read quorum to validate the cache copy before sending any object to a client.

The key benefit of volume leases is that they can be 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.

Protocol details. The protocol details at the node level are similar to the basic dual quorum protocol except that each *IQS* node tracks the volume lease and callback state on all *OQS* nodes. The pseudo-code describing actions at an IQS and an OQS node is shown in Figures 4 and 5.

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 ($lastReadLC_o > lastAckLC_{o,j}$).

[2] For 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].

```
1  processLCReadRequest(){
2    sendMsg(CLIENT_LC_READ_REPLY, logicalClock);
3  }
4
5  processWriteRequest(Object o, Value v,
6                      LogicalClock lc){
7    if(lc > lastWriteLC_o){
8      value_o := v;
9      lastWriteLC_o := lc;
10     //ensure an invalid OQS write quorum
11     while(!isOWQInvalid(o, lc)){
12       invalidateOWQ(o, lc);
13       //see text for descriptions
14     }
15   }
16   sendMsg(CLIENT_WRITE_ACK, o, lc);
17 }
18
19 processInvalAck(Object o, Sender j,
20                 LogicalClock lc){
21   //update the last inval ack in
22   //the record for the sender
23   lastAckLC_{o,j} := MAX(lastAckLC_{o,j}, lc);
24 }
```

```
24 processVLRenewal(Volume v, Sender j,
25                   RequestorTime t_{v,0}){
26   expires_{v,j} := L + currentTime;
27   sendMsg(VOLUME_RENEW_REPLY, delayed_{v,j},
28           L, epoch_{v,j}, t_{v,0});
29 }
30
31 processVLRenewalAck(Volume v, Sender j,
32                     LogicalC lc){
33   //remove delayed invals already
34   //applied at the sender
35   ∀k, s.t. inval_{k,j} ∈ delayed_{v,j}{
36     if(lc ≥ inval_{k,j}.lc){
37       delete inval_{k,j};
38     }
39   }
40 }
41
42 processObjRenewal(Object o){
43   //update last-read logical clock
44   lastReadLC_o := lastWriteLCo;
45   sendMsg(OBJECT_RENEW_REPLY, value_o,
46           lastWriteLC_o);
47 }
```

Fig. 4. IQS server operations (pseudocode) – Dual quorum with volume leases

Client read. As detailed by **processReadRequest** in the pseudo-code, a node j in OQS processes a client read of object o by ensuring 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

```
1  processVLRenewReply(Volume v, Sender i,
2                      Lease L, Epoch e, DI di,
3                      RequestorTime t_{v,0}){
4    expires_{v,i} := MAX(expires_{v,i}, t_{v,0} + L * (1 - maxDrift))
5  ;
6    epoch_{v,i} := MAX(epoch_{v,i}, e);
7    //apply delayed invals in the reply
8    ∀k, s.t. inval_{k,i} ∈ di {
9      if(inval_{k,i}.lc > logicalClock_{k,i}){
10       logicalClock_{k,i} := inval_{k,i}.lc;
11       valid_{k,i} := false;
12     }
13   }
14   sendMsg(VOLUME_RENEW_REPLY_ACK,
15           v, MAX(di.lc));
16 }
17
18 processInval(Object o, Sender i,
19              LogicalClock lc){
20   //update the local logic clock
21   //and object status
22   if(logicalClock_{o,i} < lc){
23     logicalClock_{o,i} := lc;
24     valid_{o,i} := false
25   }
26   sendMsg(INVAL_ACK, lc);
27 }

27 processReadRequest(Object o){
28   //ensure valid local object and volume
29   while(!isLocalValid(o)){
30     //renew invalid volume and object
31     validateLocal(o);
32   }
33   //send reply to cilent
34   lc := MAX_{∀i, s.t. value_{o,i}=true}(logicalClock_{o,i});
35   sendMsg(CLIENT_READ_REPLY, value_o, lc);
36 }
37
38 processRenewReply(Object o, Sender i,
39                   Epoch epoch, LogicalClock lc,
40                   ObjectValue value){
41   epoch_{o,i} := MAX(epoch_{o,i}, epoch);
42   if(logicalClock_{o,i} ≤ lc){
43     logicalClock_{o,i} := lc;
44     valid_{o,i} := true;
45   }
46   if(valid_{o,i} = true &&
       logicClock_{o,i} ≥ MAX_{∀k,k∈IQS}(logicalClock_{o,k})
47 ){
48     value_o := value;
49   }
50 }
```

Fig. 5. OQS server operations (pseudocode) – Dual quorum with volume leases

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 j updates 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 $lastAckLC_{o,j} = max(lastAckLC_{o,j}, lc)$.

3.3 DQVL Correctness

Because of space constraints, we omit the proof [3] that the system has regular semantics [16]. In particular, the proof 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 in IQS (iwq) when j has both a valid volume v and a valid object o from a read quorum in IQS (irq) (because an OQS read quorum (orq) and an OQS write quorum (owq) intersect 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 $lc_1 ... lc_n$ ($lc_t > lc_0$) is sent to an orq. If the invariant is

[3] The details are presented in Chapter 4 of Lei Gao's dissertation available at www.cs.utexas.edu/users/lgao/papers/dissertation.pdf.

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 $o.lc_0...o.lc_n$. Meanwhile, some iwq may send invalidations with logical clock $inval.lc_0...inval.lc_n$ 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 $lc_0...lc_n$.

4 Evaluation

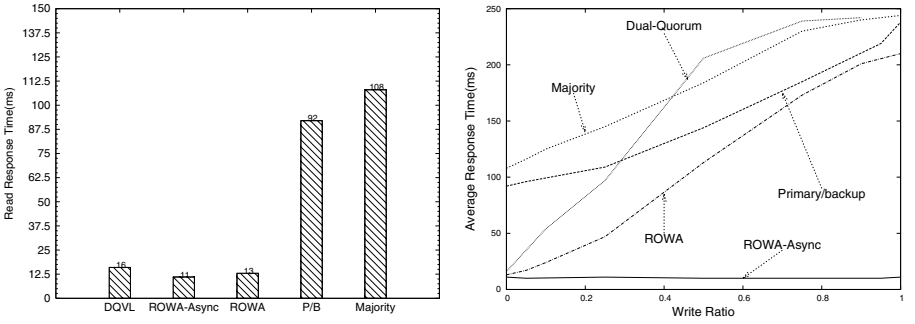
Through both analytical and experimental evaluations, we compare the availability, performance, and communication overhead of DQVL against other popular replication protocols. We show that DQVL yields a read performance competitive with ROWA epidemic algorithms and overall availability competitive with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

4.1 Response Time

A prototype has been implemented by using DQVL and other popular replication protocols, such as primary/backup, majority quorum, ROWA-Async and ROWA, to compare their response times. The prototype is similar to a read/write register in that it allows clients to read and write the value of a *single* object. But our prototype supports reads and writes on *multiple* objects and ensures a consistent view of all objects on every server.

All the prototypes are built in Java. In our prototype experiment, we set the "LAN" delay between an application client and its closest edge server to 8 ms. The "WAN" delay between the application client and other edge servers is 86 ms. And the network delay among edge servers is 80 ms. Because the experiments focus on how various protocols can minimize WAN delays by taking advantage of having an edge server near every application client, we assume a constant processing delay on every edge server for both reads and writes. An application client sends requests to the system with a specified write ratio. The application client sends the next request only after it receives the response of the current request. We run up to nine edge servers and three application clients in the experiment.

This section compares the response time of five protocols under our target workloads. We show that DQVL yields better response time than protocols providing strong consistency guarantees and competitive response time to protocols with relaxed consistency guarantees.



(a) 5% write rate (b) varying write rate

Fig. 6. Response time vs. write rate

Write ratio. We first evaluate the response time by fixing the write rate to 5%, which is the update rate for TPC-W[4] 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.

As illustrated in Figure 6 (a), DQVL provides at least a six times read response time improvement over primary/backup and majority quorum protocols that are used to provide strong consistency guarantees. DQVL yields comparable read response time to ROWA and ROWA-Async protocols because it allows most client reads to be processed locally at the client's closest edge server while maintaining the same level of consistency guarantees as both primary/backup and majority quorum protocols by running the dual-quorum protocol between the closest replica and the rest of the replicas in the system.

Figure 6 (b) is the sensitivity graph illustrating the response time as we vary the write rate. As writes dominate the workload, DQVL's response time approximates that of the majority quorum protocol and becomes higher than those of primary/backup and ROWA. The main reason is that DQVL 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 *IQS*. Two round trips are required for both the majority quorum protocol and DQVL while only one round trip is needed for primary/backup and ROWA protocols. For this reason, the average response times of both DQVL and the majority quorum protocol are worse than that of ROWA although both protocols do not require every write to be processed by all nodes.

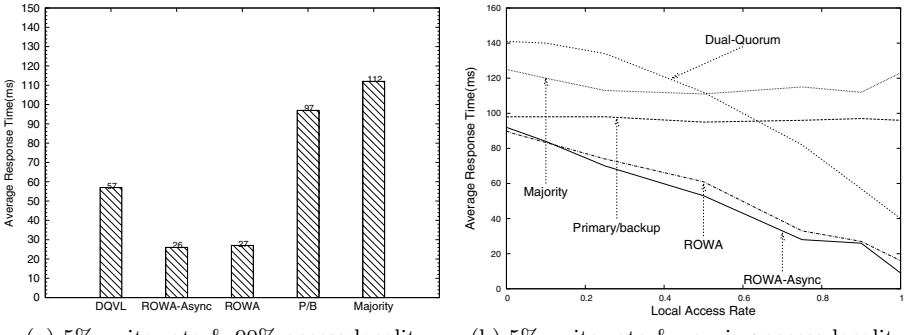
Access locality. In this subsection, we evaluate response time when some portion of client requests are routed to replicas other than the client's closest one. Under normal circumstances, requests are routed to the client's closest server.

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

But the unavailability of the closest replica or the geographical movement of the client can sometimes result in a request being routed to a distant replica.

Figure 7 (a) illustrates the protocols' response times at our target 5% write rate and at 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 rates below 10% and infrequent mobility by most end users. DQVL outperforms both primary/backup and majority quorum protocols for the workload while preserving the same consistency level in cases where client requests are directed to distant replicas. Note that that ROWA-Async protocol yields the optimal response time at the cost of serving reads with potentially inconsistent data when requests are directed to the distant replicas.

In the DQVL 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 7 (b) indicates the relationship between the access locality and the overall response time of five protocols. DQVL suffers when access locality is low because both reads and writes need to contact replicas in both input and output quorum systems. But DQVL'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, DQVL should be preferred over primary/backup or majority quorum protocols for replication systems requiring low response time and strong consistency guarantees.



(a) 5% write rate & 90% access locality (b) 5% write rate & varying access locality

Fig. 7. Average response time vs. access locality

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.

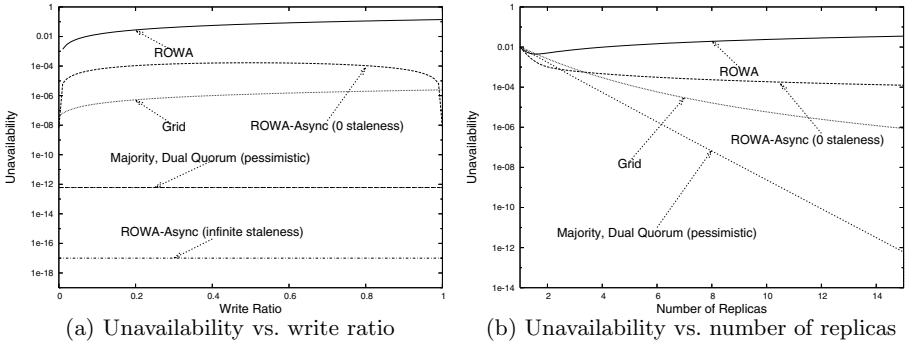
We define the availability (av) as the number of client requests successfully processed by the system over the total number of requests submitted to the

system during a given time period. A request is rejected by the system when target consistency semantics can not be satisfied. In the context of this paper, systems are required to provide regular semantics [16]. For example, if more than half of the nodes are unavailable in the *IQS* 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. Because the ROWA-Async protocol allows reads to return stale data from nodes without the latest update, it does not provide regular semantics. Therefore, to make the comparison fair [32], our analysis of the system implementing ROWA-Async protocol assumes that the system rejects client reads that would return stale data.

Figure 8 illustrates the unavailability of DQVL 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 node unavailability $p = 0.01$ and that node failures (including server crashes and network failures) are independent. Read and write rates are defined as $1 - w$ and w.

For DQVL, the availability of both *read hit* and *read miss* are $\min(av_{orq}, av_{irq})$. The availability of both *write through* and *write suppress* are $\min(av_{irq}, av_{iwq})$. Therefore, the availability of DQVL is $av_{DQVL} = (1 - w) * min(av_{orq}, av_{irq}) + w * min(av_{iwq}, av_{irq})$.[5]

Figure 8 (a) illustrates the unavailability of our target protocols as we vary the write ratio and fix the number of replicas to 15 (in both *IQS* and *OQS*). 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. Note that ROWA-Async protocol provides excellent availability by allowing reads to return arbitrary stale data to clients. But if we allow no stale reads by the ROWA-Async protocol, its availability decreases to several orders of magnitude worse than other quorum based protocols and our DQVL protocol.



(a) Unavailability vs. write ratio (b) Unavailability vs. number of replicas

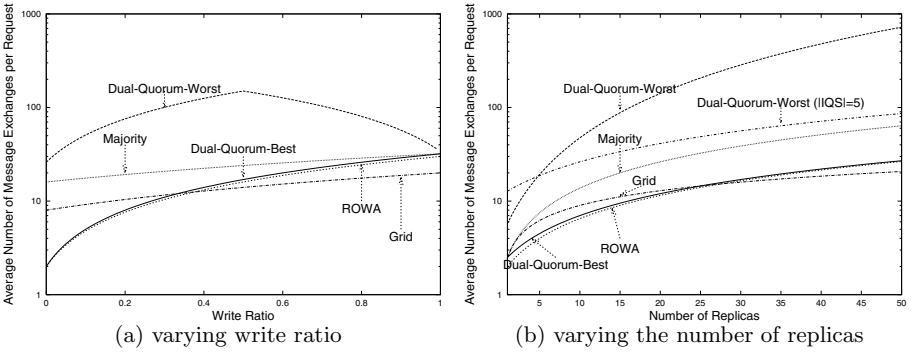
Fig. 8. System unavailability

Figure 8 (b) illustrates unavailability as we vary the number of replicas and fix the write ratio at 25%. The unavailability of DQVL is similar to that of the majority quorum system. The availability of quorum based protocols, including DQVL, improves as the total number of nodes increases. The availability of ROWA and ROWA-Async with no stale reads is insensitive to the number of nodes in the system.

4.3 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 the weights of all message types are equal. Because of space constraints, we omit a detailed discussion of the communication overhead model.[5] Figure 9 shows the average number of messages required to process a client request in log scale. As illustrated in Figure 9 (a), in the worst case where the write ratio is 50%, DQVL 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 *IQS* and *OQS* in processing requests. 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*. 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. As shown in Figure 9 (b), once we fix IQS at a moderate size while letting the OQS size grow, the communication overhead yielded by DQVL is comparable to that of the majority quorum protocol without requiring many *read hits* in the workload.

Note that although the dual quorum protocol is described in terms of two quorum systems, *IQS* and *OQS* , an *IQS* server could physically be on the same node as an *OQS* server, reducing the overall communication overhead.



(a) varying write ratio (b) varying the number of replicas

Fig. 9. Communication overhead

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-Async) [21,24,25] 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 [4] performs writes synchronously on the available replicas to provide better consistency, but it requires membership protocols to maintain a consistent view of active members.

The primary-backup (or primary-copy) model [2] tolerates 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 [19,20], 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 [11,12,23,27] tolerate network partitions as long as connected replicas can form a quorum to process requests. 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 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 server hosts the objects and clients keep cached copies. Those protocols assume that an object 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 replication algorithm designed to support Internet edge services. Through both analytical and experimental evaluations, we demonstrate that the protocol offers nearly ideal tradeoffs among high availability, good performance, and strong consistency for some workloads of interest.

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. AkamaiThe Business Internet A Predictable Platform for Profitable EBusiness. 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 adn Receivery 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 Knowlegde 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. D L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended virtual synchrony. In *Proceedings of the Fourteenth International Conference on Distributed Computing Systems*, June 1994.
21. 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.
22. A. Nayate, M. Dahlin, and A. Iyengar. Transparent Information Dissemination. In *ACM/IFIP/USENIX 5th International Middleware Conference*, October 2004.
23. J. Paris and D. Long. Efficient Dynamic Voting Algorithms. In *Int'l Conference on Data Engineering*, 1988.
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 ACMSymposium 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. The Costs and Limits of Availability for Replicated Services. In *Proceedings of the Eightteenth ACM Symposium on Operating Systems Principles*, 2001.
33. 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.