Robust Fault-Tolerant Majority-Based Key-Value Store Supporting Multiple Consistency Levels

Ahmad Al-Shishtawy*†, Tareq Jamal Khan*, and Vladimir Vlassov*
*KTH Royal Institute of Technology, Stockholm, Sweden
{ahmadas, tareqjk, vladv}@kth.se
†Swedish Institute of Computer Science, Stockholm, Sweden
ahmad@sics.se

***Abstract*—The wide spread of Web 2.0 applications with rapidly growing amounts of user generated data, such as, wikis, social networks, and media sharing, have posed new challenges on the supporting infrastructure, in particular, on storage systems. In order to meet these challenges, Web 2.0 applications have to tradeoff between the high availability and the consistency of their data. Another important issue is the privacy of user generated data that might be caused by organizations that own and control datacenters where user data are stored. We propose a large-scale, robust and fault-tolerant key-value object store that is based on a peer-to-peer network owned and controlled by a community of users. To meet the demands of Web 2.0 applications, the store supports an API consisting of different read and write operations with various data consistency guarantees from which a wide range of web applications would be able to choose the operations according to their data consistency, performance and availability requirements. For evaluation, simulation has been carried out to test the system availability, scalability and fault-tolerance in a dynamic, Internet wide environment.**

***Keywords*-peer-to-peer; key-value store; consistency models; distributed hash table; majority-based quorum technique.**

I. Introduction

The emergence of Web 2.0 opened the door to new applications by allowing users to do more than just retrieving of information. Web 2.0 applications facilitate information sharing, and collaboration between users. The wide spread of Web 2.0 applications, such as, wikis, social networks, and media sharing, resulted in a huge amount of user generated data that places great demands and new challenges on storage services. An Internet-scale Web 2.0 application serves a large number of users. This number tends to grow as popularity of the application increases. A system running such application requires a scalable data engine that enables the system to accommodate the growing number of users while maintaining a reasonable performance. Low (acceptable) response time is another important requirement of Web 2.0 applications that needs to be fulfilled despite of uneven load on application servers and geographical distribution of users. Furthermore, the system should be highly available as most of the user requests must be handled even when the system experiences partial failures or has a large number of concurrent requests. As traditional database solutions could not keep up with the increasing scale, new solutions, which can scale horizontally, were proposed, such as, PNUTS [1] and Dynamo [2].

However there is a trade-off between availability and performance on one hand and data consistency on the other. As proved in the CAP theorem [3], for distributed systems only two properties out of the three – consistency, availability and partition-tolerance – can be guaranteed at any given time. For large scale systems, that are geographically distributed, network partition is unavoidable [4]; therefore only one of the two properties, either data consistency or availability, can be guaranteed in such systems. Many Web 2.0 applications deal with one record at a time, and employ only key based data access. Complex querying, data management and ACID transactions of relational data model are not required in such systems. Therefore for such applications a NoSQL key-value store would suffice. Also Web 2.0 applications can cope with relaxed consistency as, for example, it is acceptable if one's blog entry is not immediately visible for some of the readers.

Another important aspect associated with Web 2.0 applications is the privacy of user data. Several issues lead to increasing concerns of users, such as, where the data is stored, who owns the storage, and how stored data can be used (e.g. for data mining). Typically a Web 2.0 application provider owns datacenters where user data are stored. User data are governed by a privacy policy. However, the provider may change the policy from time to time, and users are forced to accept this if they want to continue using the application. This resulted in many lawsuits during the past few years and a long debate about how to protect user privacy.

Peer-to-Peer (P2P) networks [5] offers an attractive solution to Web 2.0 storage systems. First, because they are scalable, self-organized, and fault-tolerant; second, because they are typically owned by the community, rather than a single organization, thus allow to solve the issue of privacy.

In this paper, we propose a P2P-based object store with a flexible read/write API allowing the developer of a Web 2.0 application to trade data consistency for availability in order to meet requirements of the application. Our design uses quorum-based voting as a replica control method [6]. Our proposed replication method provides better consistency guaranties than those provided in a classical DHT [7] but yet not as expensive as consistency guaranties of Paxos based replication [8]

Our key-value store is implemented as a Distributed Hash Table (DHT) [7] using Chord algorithms [9]. Our store ben-

1521-9097/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPADS.2011.110

efits from the inherent scalability, fault-tolerance and self-management properties of a DHT. However, classical DHTs lack support for strong data consistency required in many applications. Therefore a majority-based quorum technique is employed in our system to provide strong data consistency guarantees. As mentioned in [10], this technique, when used in P2P systems, is probabilistic and may lead to inconsistencies. Nevertheless, as proved in [10], the probability of getting consistent data using this technique is very high (more than 99%). This guarantee is enough for many Web 2.0 applications that can tolerate relaxed consistency.

To evaluate our approach, we have implemented a prototype of our key-value store and measured its performance by simulating the network using real traces of Internet latencies.

II. Related Work

This section presents the necessary background to our approach and algorithms presented in this paper, namely: Peer-to-peer networks, NoSQL data stores, and consistency models.

A. Peer-to-Peer Networks

Peer-to-peer (P2P) refers to a class of distributed network architectures that is formed between participants (usually called nodes or peers) on the edges of the Internet. P2P is becoming more popular as edge devices are becoming more powerful in terms of network connectivity, storage, and processing power.

P2P networks are scalable and robust. The fact that each peer plays the role of both client and server allows P2P networks to scale to large number of peers, because adding more peers increases the capacity of the system (such as storage and bandwidth). Another important factor that helps P2P to scale is that peers act as routers. Thus each peer needs only to know about a subset of other peers. The decentralized nature of P2P networks improves their robustness. There is no single point of failure, and P2P networks are designed to tolerate churn (joins, leaves and failures of peers).

Structured P2P network, such as Chord [9], maintains a structure of overlay links. Using this structure allows peers to implement a DHT [7]. Given a key, any peer can efficiently retrieve or store the associated data by routing (in log n hops) a request to the peer responsible for the key. Maintenance of the mapping of keys to peers and of the routing information is distributed among the peers in such a way that churn causes minimal disruption to the lookup service. This maintenance is automatic and does not require human involvement. This feature is known as self-organization.

1) Symmetric Replication: Symmetric replication scheme [11], [12] has been used in our system to replicate data at several nodes. Given a key i, a replication degree f, and the size of the identifier space N, symmetric replication is used to calculate the keys of replicas. The key of the x-th ($1 \leq x \leq f$) replica of the data identified by the key i is computed as follows:

$$r(i, x) = (i + (x - 1)N/f) \bmod N \qquad (1)$$

The advantage of symmetric replication over successor replication [9] is that each join or leave of a node requires only $O(1)$ messages to be exchanged to restore replicas; whereas the successor replication schema requires O(f) messages.

B. Consistency Models

In this context, consistency models define rules that help developers to predict the results of read/write operations performed on data objects stored in a distributed data store. Each particular data store supports a consistency model that heavily affects it performance and guaranties. Most relevant consistency models for our discussions are the following.

- *Sequential consistency* offers strong guaranties. All reads and writes appear as if they were executed in a sequential order; hence, every read returns a latest written value.
- *Eventual consistency* guarantees that after sufficiently long period of the absence of new writes, all read operations will return the latest written value.
- *Timeline consistency* is weaker than sequential consistency because it allows a read operation to return a stale value; however, it is stronger than eventual consistency as it guaranties that the returned (stale) value includes all previous updates.

C. NoSQL Datastores

This section provides some insights into the properties and consistency models of two large-scale data storage systems, Amazon's Dynamo and Yahoo!'s PNUTS.

1) Dynamo: Amazon's Dynamo [2] is a distributed key-value data store designed to provide a large number of services on the Amazon's service oriented platform with an always-on experience despite of certain failure scenarios such as network partitions and massive server outages. These services also have a stringent latency requirement even under high load.

Dynamo is primarily designed for the applications that require high write availability. It provides *eventual consistency* so that it sacrifices data consistency under certain failure scenarios or high write concurrency in order to achieve higher availability, better operational performance, and scalability. Conflicting versions are tolerated in the system during writes. However, the divergent versions must be detected and eventually reconciled. This is done during reads.

Nodes in the system form a ring structured overlay network and use consistent hashing for data partitioning. The system exposes two operations - get(key) and put(key, object, context) where context represents metadata about the object, e.g. version implemented using vector clocks [13]. The context is kept in the store in order to help the system to maintain its consistency guarantee. Dynamo uses successor replication.

2) PNUTS: Yahoo!'s PNUTS [1] is a geographically distributed and replicated large-scale data storage system currently being used by a number of Yahoo! applications. The system offers relaxed data consistency guarantees in order to decrease latency of operations, improve access concurrency and scalability to be able to cope with ever-increasing load.

Although the eventual consistency model adopted by Dynamo is a good fit for many web services, the model is vulnerable to exposing inconsistent data to applications because, even though it guarantees all updates to reach all replicas eventually, it does not guarantee the same order of updates at different replicas. Therefore, for many web applications, this model is a weak and inadequate option for data consistency.

In contrast to Dynamo, PNUTS offers a stronger consistency model, called timeline consistency, to applications that can live with slightly stale but valid data. It has been observed that unlike traditional database applications many web applications typically tend to manipulate only one data record at a time. PNUTS focuses on maintaining consistency for single records and provides a novel *per-record timeline consistency model*, which guarantees that all replicas of a given record apply updates in the exact same order. Developers can control the level of consistency through the following operations: read-any, read-critical, read-latest, write, and test-and-set-write.

The per-record timeline consistency model is implemented by designating one replica of a record as the master replica to which all updates are directed to be serialized as described below. The mastership is assigned on a per-record basis, therefore different records of the same table can have masters at different regions. The mastership can migrate between regions depending on the intensity of updates within a region.

PNUTS uses Yahoo! Message Broker (YMB), which is a topic-based publish/subscribe system, to implement its asynchronous replication with timeline consistency. When an update reaches the record's master, the master publishes it to the YMB in the region. Once published, the update is considered committed. YMB guarantees that updates published in a particular YMB cluster will be asynchronously propagated to all subscribers (replicas) and delivered in the publish order. Master replica leverages these reliable publish properties of YMB to implement timeline consistency. When the system detects a change in mastership of a particular record, it also publishes identity of the new master to YMB.

III. P2P Majority Based Object Store

In this paper, we propose a distributed key-value object store supporting multiple consistency levels. Our store exposes an API of read and write operations similar to the API of PNUTS [1]. In our store, data are replicated at various nodes in order to achieve fault-tolerance and improve availability, performance, and scalability. Replication causes different versions of an object to co-exist at the same time. In contrast to PNUTS, which uses masters to provide timeline consistency, our system uses a majority-based mechanism to provide multiple consistency guarantees. Our approach to maintaining per-object consistency using a quorum, rather than a master, eliminates a potential performance bottleneck and a single point of failure exposed by the master replica, and allows using our store in a highly dynamic environment such as P2P networks.

Our system is based on a scalable structured P2P overlay network. We use consistent hashing scheme [14] to partition the key-value store and distribute partitions among peers. Each peer is responsible for a range of keys and stores corresponding key-value pairs. The hashing scheme has good scalability in a sense that when a peer leaves or joins the system, only immediate neighbors of the peer are affected as they need to redistribute their partitions.

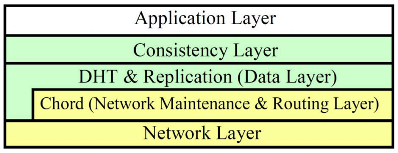
Application Layer
Consistency Layer
DHT & Replication (Data Layer)
Chord (Network Maintenance & Routing Layer)
Network Layer

Fig. 1. Architecture of a peer shown as layers

A. System Architecture

The architecture of a peer is depicted in Fig. 1. It consists of the following layers.

- **Application Layer** is formed of applications that invoke read/write operations of the API exposed by the Consistency Layer to access the underlying key-value store in the Data Layer with various consistency levels.
- **Consistency Layer** implements the following read and write operations of the key-value store API. The operations follow timeline consistency and have semantics similar to semantics of operations provided by PNUTS [1].
  - *Read Any (key)* can return an older version of the data even after a successful write. This operation has lowest latency and can be used by applications that prefer fast data access over data consistency.
  - *Read Critical (key, version)* returns data with a version, which is the same or newer than the requested version. Using this operation, an application can enforce read-your-writes consistency, i.e., the application can read the version that reflects previous updates made by the application.
  - *Read Latest (key)* returns the latest version of the data associated with the given key, and is expected to have the highest latency compared to other reads. This operation is useful for applications to which consistency matters more than performance.
  - *Write (key, data)* stores the given data associated with the given key. The operation overwrites existing data, if any, associated with the given key.
  - *Test and Set Write (key, data, version)* writes the data associated with the key only if the given version is the same as the current version in the store, otherwise the update is aborted. This operation can be used to implement transactions.
- **DHT Layer** of a peer hosts a part of the DHT (the key-value object store) for which the peer is responsible according to consistent hashing. It also stores replicas of data of other peers. The read and write operations described above are translated into Get (key, attachment) and Put (key, value, attachment) operations implemented in the DHT Layer. The attachment argument contains metadata for the operations, e.g., a version number. A

Algorithm 1 Replica Location and Data Access

```
procedure GetNodes($key$)    ▷ Locates nodes responsible for replicas
    for $x \leftarrow 1, f$ do    ▷ $f$ is the replication degree
        $id \leftarrow \mathbf{r}(key, x)$    ▷ Calculate replica id using equation 1
        $nodes[x] \leftarrow$ Lookup($ids$)    ▷ Lookup node responsible for replica id
    return $nodes[\,]$    ▷ Returns references to all nodes responsible for replicas

receipt of ReadReq(key, rank) from m at n
    (val, ver) ← LocalStore,Read(key, rank)
    sendto m : ReadResp(key, val, ver)

receipt of VerReq(key, rank) from m at n
    (val, ver) ← LocalStore.Read(key, rank)
    sendto m : VerResp(key, ver)

receipt of WriteReq(key, rank, value, ver) from m at n
    LocalStore.Write(key, rank, value, ver)    ▷ Fails if data is locked
    sendto m : WriteAck(key)

receipt of LockReq(key, rank) from m at n
    ver ← LocalStore.Lock(key, rank)    ▷ Fails if data is locked
    sendto m : LockAck(key, ver)

receipt of WriteUnlockReq(key, rank, value, ver) from m at n
    if LocalStore.IsLocked then    ▷ Fails if data is unlocked
        LocalStore.Write(key, rank, value, ver)
        LocalStore,Unlock(key, rank)
        sendto m : WriteUnlockAck(key)
```

Algorithm 2 ReadAny

```
boolean firstResponse ← true
procedure ReadAny(key)    ▷ Called by application
    nodes[] ← GetNodes(key)    ▷ Nodes hosting replicas
    for $i \leftarrow 1, f$ do
        sendto nodes[$i$] : ReadReq(key, $i$)    ▷ Request replica $i$ of key key

receipt of ReadResp(key, val, ver) from m at n
    if firstResponse then    ▷ Note that version ver is not used
        firstResponse ← false
        return (key, val)    ▷ Return (key, val) pair to application
    else DoNothing( )
```

Algorithm 3 ReadCritical

```
integer version ← 0, boolean done ← false

procedure ReadCritical(key, ver)    ▷ Called by application
    version ← ver
    nodes[] ← GetNodes(key)    ▷ Nodes hosting replicas
    for $i \leftarrow 1, f$ do
        sendto nodes[$i$] : ReadReq(key, $i$)

receipt of ReadResp(key, val, ver) from m at n
    if not done and ver $\geq$ version then
        done ← true
        return (key, val, ver)    ▷ Return (key, val, ver) to application
    else DoNothing( )
```

Get/Put operation looks up the node responsible for the given key using the underlying Chord Layer. After the key is resolved, the responsible node is contacted directly through the Network Layer to perform the requested operation. The DHT Layer also manages data handover and recovery caused by churn.

- **Chord Layer** performs lookup operations requested by the upper DHT Layer efficiently using the Chord lookup algorithm [9] and returns the address the node responsible for the given key. The layer enables nodes to join or leave the ring, also carries out periodic stabilization to keep network pointers correct in the presence of churn.
- **Network Layer** provides simple interfaces for sending/receiving messages to/from peers.

B. Algorithms

In this section, we describe the read/write operations introduced in Section III-A and present corresponding algorithms. Algorithm 1 includes common procedures used by other algorithms. The algorithms are simplified and some practical issues such as timeouts and error handling are not presented.

1) Read-Any: The *Read-Any* operation (Algorithm 2) sends the read request to all nodes hosting replicas (*replicas* thereafter) and, as soon as it receives the first successful response, it returns the received data. If the data is found locally, Read-Any returns immediately. If no successful response is received within a timeout, the operation fails, e.g., raises an exception. Read-Any also fails if it receives failure responses from all replicas. A failure response is issued when data is not found at the expected node or lookup fails at the Chord layer because of churn. Although the default is to send requests to all replicas, an alternative design choice is to send requests to two random replicas [15] with a view to reducing the number of messages.

2) Read-Critical: The Read-Critical operation (Algorithm 3) sends read requests to all nodes hosting replicas (*replicas* thereafter) and, as soon as it receives data with a version not less than the required version, it returns the data it has received. Read-Critical fails in the case of timeout. It also fails if it receives failure responses or old versions from all replicas. An alternative design choice is to send requests to a majority of replicas to reduce the number of messages. If all nodes in the majority are alive during the operation, it is guaranteed that the requested version will be found (if it exists) because write operations also use majorities.

3) Read-Latest: The Read-Latest operation (Algorithm 4) sends requests to all replicas and, as soon as it receives successful responses from a majority of replicas, it returns the latest version of the received data. Reading from a majority of replicas R guaranties to return the latest version because R always overlaps with the Write majority W, as $|R|+|W| > n$ (n is the number of replicas). Read-Latest fails in the case of timeout or when it receives failures from a majority of replicas.

4) Write: First, the Write operation (Algorithm 5) sends version requests to all nodes hosting replicas (*replicas* thereafter) and waits for responses from a majority of replicas. Requesting from a majority ensures that the latest version number is obtained. Note that for new inserts, the version number 0 is returned as nodes responsible for replicas do not have data. Next, the operation increments the latest version number and sends a write request with the new version of data to all replicas. When a majority of replicas has acknowledged the write requests, the Write operation successfully returns. If two or more distinct nodes try to write data with the same version number, the node with the highest identifier wins. The Write operation can fail for a number of reasons such as timeout, lookup failure, a replica is being locked by a Test-and-Set-Write operation, or collision with another write.

Algorithm 4 ReadLatest

```
integer version ← -1, count ← 0
object value ← null, boolean done ← false

procedure READLATEST(key)                              ▷ Called by application
    nodes[] ← GETNODES(key)                            ▷ Nodes hosting replicas
    for $i ← 1, f$ do
        sendto nodes[$i$] : READREQ(key, $i$)

receipt of READRESP(key, val, ver) from m at n
    if not done then
        count ← count + 1
        if ver > version then                          ▷ Find the latest version and value
            version ← ver
            value ← val
        if count $= f/2 + 1$ then                      ▷ Reached majority?
            done ← true
            return (key, value, version)               ▷ Return to application
    else DONOTHING( )
```

Algorithm 5 Write

```
integer maxVer ← -1, count ← 0, object value ← null
boolean done1 ← false, done2 ← false

procedure WRITE(key, val)                              ▷ Called by application
    nodes[] ← GETNODES(key)                            ▷ Nodes hosting replicas
    value ← val
    for $i ← 1, f$ do
        sendto nodes[$i$] : VERREQ(key, $i$)

receipt of VERRESP(key, ver) from m at n
    if not done1 then
        count ← count + 1
        if ver > maxVer then                           ▷ Find the latest version
            maxVer ← ver
        if count $= f/2 + 1$ then                      ▷ Reached majority?
            done1 ← true
            maxVer ← maxVer + 1
            WRITEVER(key, maxVer)
    else DONOTHING( )

procedure WRITEVER(key,ver)
    nodes[] ← GETNODES(key)
    for $i ← 1, f$ do
        sendto nodes[$i$] : WRITEREQ(key, $i$, value, ver)

receipt of WRITEACK(key) from m at n
    if not done2 then
        count ← count + 1
        if count $= f/2 + 1$ then                      ▷ Majority
            done2 ← true
            return (key, SUCCESS)                      ▷ Return to application
    else DONOTHING( )
```

Algorithm 6 Test-and-Set-Write

```
integer version ← 0, maxVer ← -1, count ← 0
object value ← null, boolean done1 ← false, done2 ← false

procedure TSWRITE(key, val, ver)                       ▷ Called by application
    nodes[] ← GETNODES(key)                            ▷ Nodes hosting replicas
    value ← val, version ← ver
    for $i ← 1, f$ do
        sendto nodes[$i$] : LOCKREQ(key, $i$)

receipt of LOCKACK(key, ver) from m at n
    if not done1 then
        count ← count + 1
        if ver > maxVer then                           ▷ Find the latest version
            maxVer ← ver
        if count $= f/2 + 1$ then                      ▷ Reached majority?
            done1 ← true
            if maxVer equals version then              ▷ Test version then set value
                maxVer ← maxVer + 1
                WRITEUNLOCKVER(key, version)
            else
                ABORT( )                               ▷ Unlocks the replicas
                return (key, ABORTED)                  ▷ Return to application
    else DONOTHING( )

procedure WRITEUNLOCKVER(key,ver)
    nodes[] ← GETNODES(key)
    for $i ← 1, f$ do
        sendto nodes[$i$] : WRITEUNLOCKREQ(key, $i$, value, ver)

receipt of WRITEUNLOCKACK(key) from m at n
    if not done2 then
        count ← count + 1
        if count $= f/2 + 1$ then                      ▷ Majority
            done2 ← true
            return (key, SUCCESS)                      ▷ Return to application
    else DONOTHING( )
```

5) Test-and-Set-Write: The Test-and-Set-Write operation (Algorithm 6) starts with sending a lock request to all nodes hosting replicas (*replicas* thereafter). Each replica, if the data is unlocked, locks the data and sends a successful lock response together with the current version number to the requesting node. After receiving lock responses from a majority of replicas, the operation tests if the latest version number obtained from the majority, matches the required version number. If they do not match, the operation aborts (sends unlock requests to all replicas and returns). If they do match, the given data is sent to all replicas to be written with a new version number. Each of the replicas, which have locked data, writes the received new version, unlocks the data, and sends a write acknowledgement to the requesting node. As soon as acknowledgements are received from a majority of replicas, the operation successfully completes. Note that in order to ensure high read availability, read operations are allowed to read the locked data.

A Test-and-Set-Write operation can fail for a number of reasons such as timeout, lookup failure, a replica is being locked by another Test-and-Set-Write operation, or collision with another write. When the operation fails, the replicas locked by it have to be unlocked, and this is requested by the node which has initiated the operation. After the operation has completed, a late lock request issued by that operation might arrive at a replica which was not a part of the majority. In this case, according to the algorithm, the replica locks the data and sends a lock response to the requesting node, which, upon receiving the response, requests to unlock the data because the operation has been already completed.

IV. Discussion

A. Majority Versus Master

Even though the API of PNUTS is preserved in our key-value store and semantics of read/write operations are kept largely unchanged; our majority-based approach to implement the operations is different from the master-based approach adopted in PNUTS. In PNUTS, all writes to a given record are forwarded to the master replica, which ensures that updates are applied to all replicas in the same order. Serialization of updates through a single master works efficiently for PNUTS due to the high write locality (the master is placed in the geographical region with the highest intensity of updates). The master-based consistency mechanism can generally hurt the scalability of a system. It leads to uneven load distribution and

Operation	P2P Object Store		PNUTS	
	Hops	messages	Hops	messages
Read-Any	2	4	2	2
Read-Critical	2	5	3	3
Read-Latest	2	5	3	3
Write	4	10	5	5
Test-and-Set-Write	4	10	5	5

TABLE I
ANALYTICAL COMPARISON OF THE COST OF EACH OPERATION

makes the master replica a potential performance bottleneck. Furthermore, the master represents a single point of failure and may cause a performance penalty by delaying read/write operations until the failed master is restored.

In order to eliminate the aforementioned potential drawbacks of using the master-based consistency mechanism in a distributed key-value store deployed in a dynamic environment, we propose to use a majority-based quorum technique to maintain consistency of replicas when performing read/write operations. Using a majority rather than a single master removes a single point of failure and allows the system to withstand a high level of churn in a dynamic environment that was not considered for the stable and reliable environment (data centers) of PNUTS. Our mechanism is decentralized so that any node in the system receiving a client request can coordinate read/write operations among replicas. This improves load balancing. However, delegating the coordination role to any node in the system while maintaining the required data consistency, incurs additional complexity due to distribution and concurrency.

B. Performance Model

The number of network hops and the corresponding number of messages for each API operation of both PNUTS and our system, are compared in Table I. Worst case scenarios are considered for both systems and the replication degree is assumed to be 3. For simplicity, we abstract low-level details of communication protocols and count only the number of communication steps (as hops) and a corresponding number of messages. For example, in the case of PNUTS, we assume 2 hops and 2 messages for a roundtrip (request/response) interaction of a requesting node with a master in a local region; Interaction with a master in a remote region adds one more hop and entails one extra message. The asynchronous message propagation to replicas in YMB is not included (as, to our best knowledge, details of YMB were not published at the time this paper was written). For our store, we assume that it takes 2 hops to send a request from a requesting node to all replicas and receive responses from a majority, whereas the number of messages depends on the replication degree. It is worth noting that, taking into account the asynchronous messages propagated by YMB, both systems use about the same amount of messages, which is proportional to the number of replicas.

C. Other Approaches

Other approaches could have been used to implement our P2P based object store. For example, a P2P based self-healing replicated state machine [16] could have been used to implement the read/write operations. However, we believe that Paxos based replicated state machine is too powerful for implementing timeline consistency and should only be used if stronger guaranties and/or more powerful operations (e.g., transactions) are required.

V. EVALUATION

We present a simulation-based performance evaluation of our majority based key-value store in various scenarios. We are mainly interested in measuring the *operation latency* and the *operation success ratio* under different conditions by varying churn rate, request rate, network size, and replication degree. To evaluate the performance and to show the practicality of our algorithms, we built a prototype implementation of our object store using the *Kompics* [17] which is a framework for building and evaluating distributed systems in simulation, local execution, and distributed deployments. In order to make network simulation more realistic, we used the King latency dataset, available at [18], that contains measurements of the latencies between DNS servers obtained using the King [19] technique. To evaluate the performance of our algorithms in various churn scenarios, we have used the *lifetime-based node failure* model [20], [21] with the shifted Pareto lifetime Distribution. Note that the smaller the mean life time, the higher the level of churn. In our experiments, the mean lifetime of 10 and 30 minutes considered to be very low and used to stress test the system in order to find the breaking point.

When evaluating our approach, we did not make a quantitative comparison of our approach with PNUTS [1] mainly because we did not have access to details of PNUTS and YMB algorithms, which, to our best knowledge, were not published at the time this paper was written. Therefore we could not accurately implement PNUTS algorithms in our simulator in order to compare PNUTS with our system. Furthermore, PNUTS was designed to run on geographically distributed but fairly stable datacenters, whereas our system targets an Internet-scale dynamic P2P environment with less reliable peers and high churn. The reason for us to choose such unreliable environment over datacenters is mainly to reduce costs and improve data privacy. We expect that master-based read/write algorithms used in PNUTS will perform better in a stable environment whereas our quorum-based algorithms will win in a highly dynamic environment as discussed in Section IV-A.

A. Varying Churn Rate

In this experiment, the system is run with various churn rate represented as the mean node lifetime. Fig. 2(a) shows the impact of churn on latency. The latency for each operation does not vary too much for different levels of churn.

As expected, Read-Any and Read-Critical perform much faster than Read-Latest. Read-Latest shows higher latency because it requires responses from a majority of replicas, whereas other reads do not require a majority in order to complete. Although the Read-Critical latency is expected to be higher than the Read-Any latency (as in the case for PNUTS)

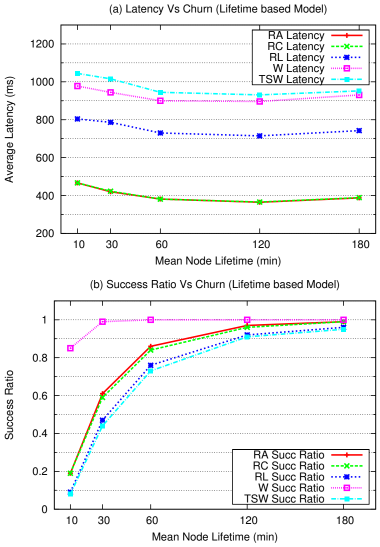

Fig. 2. The effect of churn on operations (lower mean lifetime = higher level of churn)

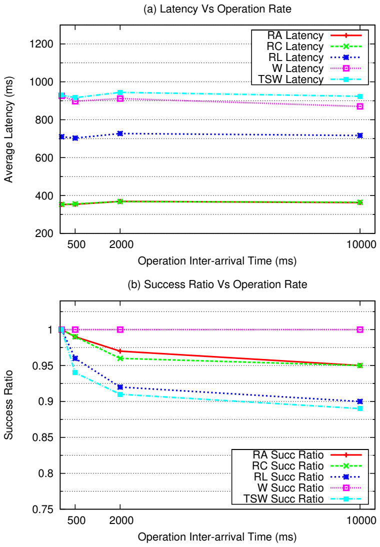

Fig. 3. The effect of operation rate operations (lower inter-arrival time = higher op rate)

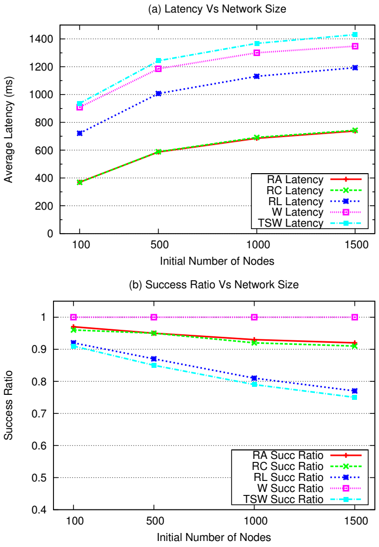

Fig. 4. The effect of network size on operations

because the former requests a specific version, latencies of both operations in our case are almost identical. This is because a write operation sends updates to all replicas and completes as soon as it receives acknowledgements from a majority of replicas, and thus, the first (fastest) reply to a Read-Critical request with a high probability will come from a node which has the required version. Compare to reads, both Write and Test-and-Set-Write have higher latency because they involve more communication steps than reads. Furthermore, Test-and-Set-Write has slightly higher latency than Write due to possible contention because of locking.

Fig. 2(b) shows operation success ratio versus churn rate (represented as the mean node lifetime). As expected, the success ratio degrades for increasing churn rates. As mentioned in Section III, there are several reasons for failures of operations due to churn. After analyzing the logs, the primary cause of failures has been identified as the unavailability of data at the responsible node. Another major reason is lookup failure.

B. *Varying Operation Rate*

Several experiments have been conducted to observe how the system performs under different load scenarios in a dynamic environment. Read/write operations are generated using an exponential distribution of inter-arrival time of operations. The operation rate (load) is higher for lower mean inter-arrival time. Fig. 3(a) shows the impact of the operation rate on latency. The latency for each operation does not vary too much for different operation rates. This is due to the simulation that assumes unlimited computing resources (no hardware bottlenecks). Nevertheless, these experiments show that our system can serve the increasing number of requests despite of churn (Fig. 3(b)), as it quickly heals after failures. One interesting observation is that under the highest load in our experiments, operations have the highest success ratio. This is due to the fact that, when the intensity of writes in a dynamic environment increases, the effect of churn on the success ratio diminishes as the data are updated more often and, as a consequence, the success ration of operations improves.

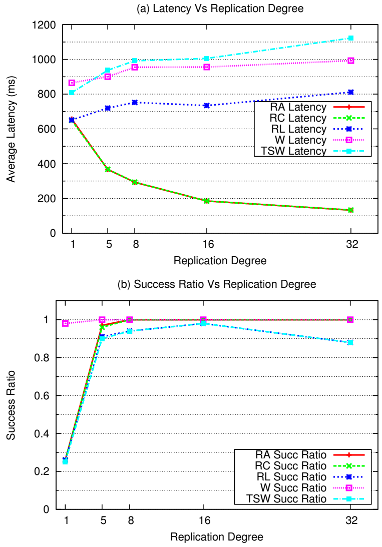

Fig. 5. The effect of replication degree on operations

C. *Varying P2P Network Size*

In this experiment, we evaluate the scalability by running the system with various network sizes (the number of nodes). Fig. 4(a) shows the impact of the network size on latency. For all operations, the latency grows when the network size increases. However the increase is logarithmic because of the logarithmic latency in the Chord Layer.

D. Varying Replication Degree

In this experiment, the system is run with various replication degrees. Fig. 5(a) shows the impact of the replication degree on the operation latency. The latency of Read-Any and Read-Critical is highest when there is no replication, but it noticeably decreases as more replicas are added. This is because both operations complete after receiving the first successful response, and having more replicas increase the probability to get the response from a fast (close) node and hence reduce the latency. For Read-Latest, Write, and Test-and-Set-Write operations the latency gets slower with increasing replication degree. This is because in these operations, a requesting node has to wait for a majority of responses, and as the number of replicas grows, the majority increases causing longer waiting time.

VI. Conclusions and Future Work

We have presented a majority-based key-value store (architecture, algorithms, and evaluation) intended to be deployed in a large-scale dynamic P2P environment. The reason for us to choose such unreliable environment over datacenters is mainly to reduce costs and improve data privacy. Our store provides a number of read/write operations with multiple consistency levels and with semantics similar to PNUTS.

The store uses the majority-based quorum technique to maintain consistency of replicated data. Our majority-based store provides stronger consistency guarantees than guarantees provided in a classical DHT but less expensive than guaranties of Paxos-based replication. Using majority allows avoiding potential drawbacks of a master-based consistency control, namely, a single-point of failure and a potential performance bottleneck. Furthermore, using a majority rather than a single master allows the system to achieve robustness and withstand churn in a dynamic environment. Our mechanism is decentralized and thus allows improving load balancing and scalability.

Evaluation by simulation has shown that the system performs rather well in terms of latency and operation success ratio in the presence of churn.

In our future work, we intend to evaluate our approach on larger scales and extreme values of load and churn rate, and to optimize the algorithms in order to reduce the amount of messages and improve performance. As the proposed key-value store is to be used in a P2P environment, there is a need to ensure security and protect to personal information by using cryptographic means. This is also to be considered in our future work.

Acknowledgments

This research is supported by the E2E Clouds project funded by the Swedish Foundation for Strategic Research (SSF), and the Complex Service Systems (CS2) focus project, a part of the ICT-The Next Generation (TNG) Strategic Research Area (SRA) initiative at the KTH Royal Institute of Technology.

References

[1] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, “Pnuts: Yahoo!’s hosted data serving platform,” *Proc. VLDB Endow.*, vol. 1, pp. 1277–1288, August 2008. [Online]. Available: http://dx.doi.org/10.1145/1454159.1454167

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: amazon’s highly available key-value store,” *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, October 2007. [Online]. Available: http://doi.acm.org/10.1145/1323293.1294281

[3] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT News*, vol. 33, pp. 51–59, June 2002. [Online]. Available: http://doi.acm.org/10.1145/564585.564601

[4] W. Vogels, “Eventually consistent,” *Queue*, vol. 6, pp. 14–19, October 2008. [Online]. Available: http://doi.acm.org/10.1145/1466443.1466448

[5] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, “A survey and comparison of peer-to-peer overlay network schemes,” *Communications Surveys & Tutorials, IEEE*, vol. 7, no. 2, pp. 72–93, 2005.

[6] D. K. Gifford, “Weighted voting for replicated data,” in *Proceedings of the seventh ACM symposium on Operating systems principles*, ser. SOSP ’79. New York, NY, USA: ACM, 1979, pp. 150–162. [Online]. Available: http://doi.acm.org/10.1145/800215.806583

[7] F. Dabek, “A distributed hash table,” Ph.D. dissertation, Massachusetts Institute of Technology, November 2005.

[8] L. Lamport, “Paxos made simple,” *SIGACT News*, vol. 32, no. 4, pp. 51–58, December 2001.

[9] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” in *ACM SIGCOMM’01*, Aug. 2001, pp. 149–160.

[10] T. M. Shafaat, M. Moser, A. Ghodsi, T. Schütt, S. Haridi, and A. Reinefeld, “On consistency of data in structured overlay networks,” in *Proceedings of the 3rd CoreGRID Integration Workshop*, April 2008.

[11] A. Ghodsi, “Distributed k-ary system: Algorithms for distributed hash tables,” Ph.D. dissertation, Royal Institute of Technology (KTH), 2006.

[12] A. Ghodsi, L. O. Alima, and S. Haridi, “Symmetric replication for structured peer-to-peer systems,” in *Proceedings of The 3^{rd} Int. Workshop on Databases, Information Systems and P2P Computing*, Trondheim, Norway, 2005.

[13] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, pp. 558–565, July 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[14] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, “Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web,” in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, ser. STOC ’97. New York, NY, USA: ACM, 1997, pp. 654–663. [Online]. Available: http://doi.acm.org/10.1145/258533.258660

[15] M. Mitzenmacher, “The power of two choices in randomized load balancing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, pp. 1094–1104, October 2001. [Online]. Available: http://dx.doi.org/10.1109/71.963420

[16] A. Al-Shishtawy, M. A. Fayyaz, K. Popov, and V. Vlassov, “Achieving robust self-management for large-scale distributed applications,” in *Self-Adaptive and Self-Organizing Systems (SASO), 2010 4th IEEE International Conference on*, October 2010, pp. 31 –40.

[17] C. Arad, J. Dowling, and S. Haridi, “Building and evaluating P2P systems using the Kompics component framework,” in *Peer-to-Peer Computing (P2P’09)*. IEEE, Sep. 2009, pp. 93–94.

[18] “Meridian: A lighweight approach to network positioning,” http://www.cs.cornell.edu/People/egs/meridian.

[19] K. P. Gummadi, S. Saroiu, and S. D. Gribble, “King: estimating latency between arbitrary internet end hosts,” in *IMW’02: 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM, 2002, pp. 5–18.

[20] J. S. Kong, J. S. Bridgewater, and V. P. Roychowdhury, “Resilience of structured P2P systems under churn: The reachable component method,” *Computer Communications*, vol. 31, no. 10, pp. 2109–2123, June 2008.

[21] D. Leonard, Z. Yao, V. Rai, and D. Loguinov, “On lifetime-based node failure and stochastic resilience of decentralized peer-to-peer networks,” *IEEE/ACM Trans. Networking*, vol. 15, no. 3, pp. 644–656, Jun. 2007.