Robust, fault-tolerant majority based key-value data store supporting multiple data consistency

TAREQ JAMAL KHAN

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2011

TRITA-ICT-EX-2011:178

KTH Information and Communication Technology

Robust, fault-tolerant majority based key-value data store supporting multiple data consistency

By

Tareq Jamal Khan

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering of Distributed Systems

Supervisor: Ahmad Al-Shishtawy

Examiner: Prof. Vladimir Vlassov

Unit of Software and Computer Systems

School of Information and Communication Technology

Royal Institute of Technology (KTH)

Stockholm, Sweden.

July 2011

Abstract

Web 2.0 has significantly transformed the way how modern society works now-a-days. In today's Web, information not only flows top down from the web sites to the readers; but also flows bottom up contributed by mass user. Hugely popular Web 2.0 applications like Wikis, social applications (e.g. Facebook, MySpace), media sharing applications (e.g. YouTube, Flickr), blogging and numerous others generate lots of user generated contents and make heavy use of the underlying storage. Data storage system is the heart of these applications as all user activities are translated to read and write requests and directed to the database for further action. Hence focus is on the storage that serves data to support the applications and its reliable and efficient design is instrumental for applications to perform in line with expectations.

Large scale storage systems are being used by popular social networking services like Facebook, MySpace where millions of users' data have been stored and fully accessed by these companies. However from users' point of view there has been justified concern about user data ownership and lack of control over personal data. For example, on more than one occasions Facebook have exercised its control over users' data without respecting users' rights to ownership of their own content and manipulated data for its own business interest without users' knowledge or consent.

The thesis proposes, designs and implements a large scale, robust and fault-tolerant key-value data storage prototype that is peer-to-peer based and intends to back away from the client-server paradigm with a view to relieving the companies from data storage and management responsibilities and letting users control their own personal data. Several read and write APIs (similar to Yahoo!'s PNUTS but different in terms of underlying design and the environment they are targeted for) with various data consistency guarantees are provided from which a wide range of web applications would be able to choose the APIs according to their data consistency, performance and availability requirements. An analytical comparison is also made against the PNUTS system that targets a more stable environment. For evaluation, simulation has been carried out to test the system availability, scalability and fault-tolerance in a dynamic environment. The results are then analyzed and conclusion is drawn that the system is scalable, available and shows acceptable performance.

Keywords: *Web 2.0 applications, peer-to-peer (P2P) system, key-value data store, relaxed consistency, Distributed Hash Table (DHT), majority based quorum technique*.

Acknowledgements

I would like to thank my supervisor Ahmad Al-Shishtawy and examiner Prof. Vladimir Vlassov for their technical guidance and continued support. I am also grateful for my family's support that kept in the track.

Table of Contents

List of Figures

List of Tables

1. Introduction

Web 2.0 has significantly transformed the way how modern society works now-a-days [1]. In today's Web, information not only flows top down from the web sites to the readers; but also flows bottom up contributed by mass user. This opens the door for a lot of applications in areas like social interaction, information and media sharing, business marketing and policy making, disaster management [2], e-learning and many more. Hugely popular Web 2.0 applications like Wikis, social applications (e.g. Facebook, MySpace), media sharing applications (e.g. YouTube, Flickr), blogging and numerous others generate lots of user generated contents and make heavy use of the underlying storage. Data storage system is the heart of these applications as all user activities are translated to read and write requests and directed to the database for further action. Hence focus is on the storage that serves data to support the applications and its reliable and efficient design is instrumental for applications to perform in line with expectations.

Internet-scale Web 2.0 applications potentially serve huge number of users and this number tends to grow as popularity of system increases. In such a context applications require scalable data engine that enables the system to accommodate growing users while still maintaining a reasonable performance. Lower response time or latency is another important requirement of applications despite uneven load conditions on web platform and users being geographically scattered. System should also be highly available as most of the user requests must be met even when system experiences partial failure or has large number of concurrent requests.

However there is a trade-off between degree of availability and performance on one hand and data consistency on the other. As proved in the CAP theorem [3], for distributed shared memory 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 distributed geographically, network partition is given [4]. Therefore only one property between strong data consistency and high availability can be guaranteed in such systems. Traditional database management systems favor data consistency over

availability during concurrent operations and implement *serializable* model [5] while carrying out transactions. It has also been observed that scaling out serializable transactions over globally distributed and replicated data systems is very costly [6]. Many web applications deal with one record at a time, require flexible schema and employ only key based data access. Complex querying, data management and ACID transactions of relational data model aren't required in such systems. Also Web 2.0 applications can cope with relaxed consistency as it is considered alright if one's blog entry is not immediately visible or friends can not see one's latest status in social networking services. Therefore for such applications key-value data store (NoSQL model) would suffice given the applications' availability and performance requirements and the large scale they operate on.

1.1 Motivation

Several large scale key-value data stores [7, 8, 21] are currently in production serving Web 2.0 applications with relaxed data consistency. Yahoo!'s PNUTS [7] offers a geographically distributed and replicated massive key-value based data storage that provides web applications with required availability and performance guarantees against relaxed consistency of its hosted data. PNUTS data store is currently supporting social web applications in production. Similar storage systems [8] (in terms of relaxed consistency) are being used by hugely popular social networking services like Facebook, MySpace where millions of users' data have been stored and are fully accessed by these companies.

However from users' point of view there has been justified concern about user data ownership and lack of control over personal data. For example, on more than one occasions [9, 10] Facebook have exercised its control over users' data without respecting users' rights to ownership of their own content and manipulated data for its own business interest without users' knowledge or consent. With the wealth of user data these companies are sitting on, there is no guarantee something similar or worse won't happen again. So there is a motivation to relieve these companies from data storage and management responsibilities and let users control their own personal data. The solution

could be a paradigm shift from company-centric client-server system to peer-to-peer system where users can have full control and ownership of their own data.

The motivation to focus PNUTS storage system in the thesis project is because PNUTS has a convenient set of read and write APIs (see Section 3.1) supporting various data consistency guarantees. The system is therefore very useful for wide range of Web 2.0 applications that can tolerate relaxed consistency for high availability and performance, as well as for applications that would require stronger data consistency accepting lesser availability in certain failure scenarios. However PNUTS targets a stable and controlled environment comprising of handful of data centers in different geographic regions. The system employs a master based approach (see Section 4.1) to maintain data consistency which is architecturally not sound and can become a performance bottleneck. Its availability also suffers in case an entire region goes down. Also hosted user data can potentially become prone to company manipulation and privacy violation. The proposed storage system intends to address these limitations and would offer the same set of PNUTS APIs from which applications can choose the ones that would give the intended degree of consistency, availability and performance.

1.2 Goal

- The goal of the thesis is to design and implement a key-value data store that is robust, fault-tolerant and can work in Internet-scale.
- The storage system would be deployed on peer-to-peer infrastructure backing away from hierarchical client-server model where single host company could store and monopolize all of users' data.
- The intended property of the system is that it should be able to withstand some degree of churn expected in a dynamic environment and would still provide applications with required availability and performance guarantees. It would also be completely distributed and wouldn't depend on any master replica.
- A set of APIs would offer various degrees of read and write data consistency for a wide range of applications to exploit.

1.3 Contribution

- A fully functional memory based key-value data store has been designed and presented. The data store is implemented as Distributed Hash Table (DHT) [12] and uses Structured Overlay Network (SON) Chord [11] that can provide Internet-scale lookup services. The prototype system benefits from the inherent characteristics of scalability, fault-tolerance and self-management that are associated with peer-to-peer systems.
- Similar read and write APIs as those of PNUTS have been implemented keeping the functionalities and semantics largely unchanged. However the underlying mechanism that is used in the implementation is completely different and devised for peer-to-peer environment. The system is also designed to avoid master-based approach adopted in PNUTS that is generally not scalable and architecturally not sound (see Section 4.1).
- Majority based quorum technique is employed to get stronger data consistency guarantee than what can be achieved in DHTs [12]. Still the consistency is weaker than atomic consistency (see Section 2.1.1) guaranteed in Paxos systems [13]. However as shown in [14], majority based approach provides data consistency guarantee with a very high probability (fraction more than 99%) which should be acceptable for applications that can tolerate relaxed consistency. For Web 2.0 applications, Paxos [15] might not be a good choice because it limits scalability, availability and performance when system is intended to be globally distributed and replicated.
- The storage system is designed to be deployed in a dynamic environment and can tolerate high degree of churn unlike PNUTS which operates in a controlled and stable environment using few data centers located in different geographic regions.
- The system is simulated under various settings (see Section 6.2) and the performance results have been presented in sufficient details.

The thesis work is structured as follows. Chapter 2 gives different consistency models as background. Chapter 3 discusses related work. Chapter 4 provides the design of the prototype. Chapter 5 describes the implementation aspects. Chapter 6 presents evaluation

results with extensive analysis. Finally chapter 7 concludes the thesis with a direction on future work.

2. Consistency Models

As applications' data usage scenario varies depending on their requirements, distributed shared memory systems offers a spectrum of consistency models that differs in the consistency guarantee they provide to the applications. This chapter gives a brief description of the more common data consistency models that are in use today in shared memory systems. Basic understanding of their respective specifications will help shed some more light on the consistency model that is implemented in the thesis project.

2.1 Existing Models

Data consistency models introduced in the distributed shared memory systems can be broadly classified as either strong consistency model or weak consistency model. Strong consistency model guarantees any read operation will return the value of the last successful write; whereas weak consistency models don't guarantee subsequent read will return the last updated value even after a successful write and a few conditions is required to be met before the value will be returned. The consistency models described in this section differ in how they handle concurrency and failure issues.

2.1.1 Atomic Consistency

Atomic Consistency [16], also known as *Linearizability* [17], offers the strongest level of data consistency in shared memory systems. In this model, any read of the atomic register (shared memory can be viewed as array of shared registers) returns the value of the last preceding write. If the read is concurrent with another write or doesn't precede a failed write (failed write is also considered concurrent with read that doesn't precede it), it can return either the value concurrently written or the value of the last write that preceded it. However once a new value has been read by any process, no other process can return the old value. It is to be mentioned that a read/write operation precedes another operation only if the former completes before the invocation of the latter in real time, and two

operations become concurrent if they overlap in real time. This consistency model is strong since it takes into consideration the real time occurrence-order of events, and orders the operations accordingly as if the system has a 'global clock' thereby giving the illusion of a single storage system. The 'last' preceding operation refers to the most recent completed operation in real time order.

Atomic register has two types [18] of specifications – (1, N) atomic register and (N, N) atomic register. (1, N) refers to a single writer and multiple readers, whereas (N, N) register allows all processes to read and write. When there are multiple writers potentially writing concurrently along with other reads, concurrency issue in that case is modeled in the following way. Every complete operation appears to have been executed at all participating nodes at some instant between its invocation and the corresponding response, whereas every failed operation appears as completed at every node or never invoked at all. If the operation execution points result in an order that gives the illusion of a serial execution in a single storage system, these points are referred to as linearization points and the actual execution is stated to be linearizable. When an application uses atomic or linearizable register for managing its data consistency, it's read and write operations are regulated according to the specification and a single storage illusion can be achieved. *Atomic consistency* is also known as *strict consistency* for its strict data consistency requirements.

2.1.2 Sequential Consistency

Sequential Consistency [19] allows execution where operations of all processes, whether concurrent in real time or not, appear in some sequential order and the local order of operations in each process is preserved. Every process sees the (write) operations on the register in the same logical order, although the order may differ from the order of operations issued at real time. In this model, each read operation returns the value of the last preceding write. The semantics of 'last' is different from that of the atomic one; since 'last' in this case refers to the most recent write according to the logically derived total order, whereas the last preceding write for atomic register is the most recent write in terms of real time occurrence. Sequential consistency is a bit weaker than atomic consistency, as it doesn't order events following global time. So it can't provide with a

single storage illusion that we ideally want. For example, sequential register allows the following execution in Figure 1 even if no single storage could behave that way.

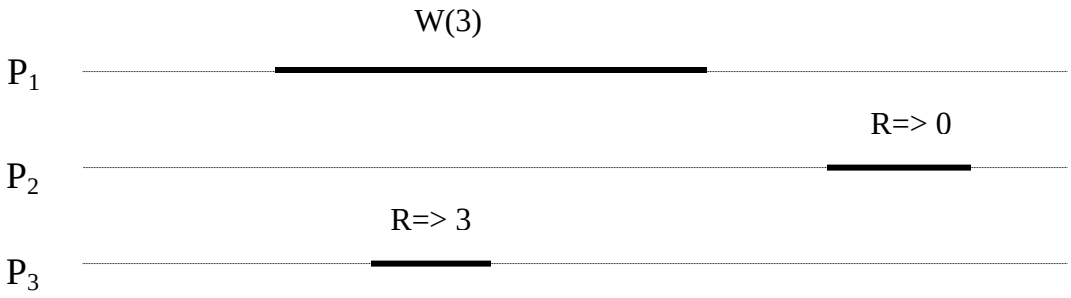


Figure 1: Sequential Execution

2.1.3 Regular Consistency

Regular Consistency [18] offers a weaker consistency model than the atomic one. In this model, read operation by any process returns the value of the last preceding write or the value being concurrently written. The notion of operations being last or concurrent is determined by their real time appearance in the execution. Failed write operation is considered to be concurrent with any read operation that doesn't precede it. Regular register is weaker than atomic register because it doesn't guarantee single storage illusion during any execution. It also differs from sequential register as it doesn't ensure the local ordering of operations in each process, therefore causality (see Section 2.1.4) at a single node can be violated in case of regular register when read is concurrent with a write (see Figure 2). Regular consistency only offers one kind of specification which is (1, N) regular register having single writer and multiple readers. The generalization of regularity for multiple writers has not found consensus in distributed computing literature [18].

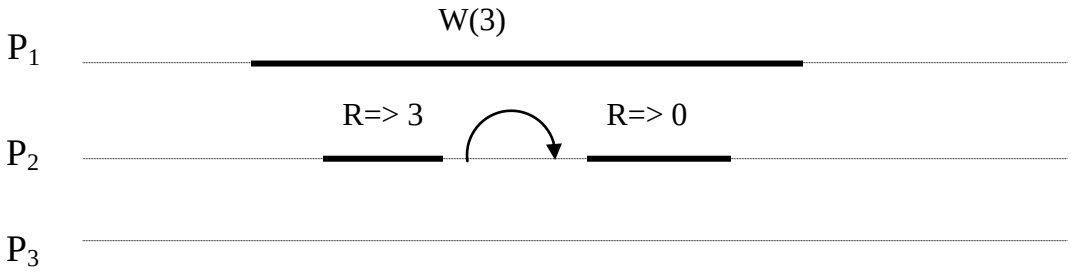


Figure 2: Regular Execution. Happened before relation observable in the same node

2.1.4 Causal Consistency

Causal Consistency [20] provides a consistency model where causally related shared memory operations are seen by all participating processes in the same order. Concurrent operations that are not causally related may be seen in different order by different processes. This model is weaker than the sequential one which ensures that all processes in the system will see all write operations in the same order. Two events are causally related if one event might be potentially caused by another event. Any of the following conditions have to happen for one operation to be causally related to another during an execution.

(i) If one operation A precedes another B in the same node, A might be causally related to B; which means there is a causal order between A and B where A comes before B.

(ii) If operation A in process p_1 involves writing a shared register x, later operation B (read or write) in another process p_2 consumes x; in that case operation A is causally related to operation B and A precedes B in the causal order.

(iii) Causal ordering is transitive. If A, B and C are three shared memory operations where A causally precedes B and B causally precedes C, then A causally precedes C.

As already mentioned, operations that are not causally related are considered to be concurrent.

2.1.5 Eventual Consistency

Eventual consistency [4] is a specific form of *weak consistency* which doesn't guarantee that a read from a shared register will return consistent value even after a successful write. What it guarantees is that in the absence of any new writes, all read operations from that register will eventually return the updated value. In this model, there exists a window of data inconsistency in between the time after the completion of a write operation and the time when the write propagates to all other replicas enabling any subsequent read to return the updated value. This time window is referred to as data inconsistency window. The length of the inconsistency window may vary depending on

factors such as data communication delays, load of the system at the time, churn and the number of processes involved in the shared memory system.

From a single client's perspective, when application (client) reads data from any replica during an execution, it is hard to define a clear semantic of the return value due to the inconsistent nature of the data being output from the underlying consistency model. To address this issue and to provide clients with a more useful model, *client-centric consistency* is introduced on the client side. The server side or *data-centric consistency* (similar to other models) remains that regulates the way updates propagate through the system and provides the consistency guarantee expected of it. Following are a few client-centric consistency models that regulate how the data updates are observed by applications using eventual consistency models.

- **Read Your Writes Consistency** – After a process writes to a shared register, future reads from the register by the same process will see the write and will never see an older value.
- **Writes Follow Reads Consistency** – After a process reads value V from a shared register R from a replica, the following write on R by the same process must go to the register from any replica that contains value V or a more recent value than V.
- **Monotonic Reads Consistency** – If a process reads from a shared register, any future reads on the register by the same process will return the same or a more recent value.
- **Monotonic Writes Consistency** – A write operation by a process on a shared register must be completed before any future writes by the same process on the register take place.

Applications use eventual consistency model for better performance, high system availability and scalability at the expense of getting intermittent inconsistency.

3. Related Work

This chapter covers a couple of large-scale data storage systems running in production in demanding distributed environments and gives insight of their respective data consistency models. Dynamo (see Section 3.2) implements a weaker eventually consistent data store and focuses on availability and partition-tolerance that would be useful for certain categories of web applications. The system also uses quorum technique and consistent hashing (explained in 3.2) which is of relevance in the context of the thesis prototype. PNUTS on the other hand gives a stronger data consistency model and addresses partition-tolerance, but suffers in the availability front in certain failure scenarios.

3.1 PNUTS

Yahoo!'s PNUTS [7] is a geographically distributed and replicated large scale data storage system currently being used by number of Yahoo! web applications. The system offers applications a relaxed consistency guarantee of its hosted data to make way for lower latency of operations, greater concurrency in data access and scalability to cope with ever-increasing loads in the web.

Although *eventual consistency* model adopted by Dynamo (see Section 3.2) is a good fit for many web services in Amazon platform [32] and satisfies the aforementioned properties, the model is vulnerable to exposing "dirty data" to applications under certain failure scenarios (e.g. partition) which may later be discarded during reconciliation. Eventual consistency guarantees all updates to reach all replicas eventually but doesn't guarantee on the execution order of the updates which may potentially be different at different replicas. For example, if someone updates his record in the order $U_1 \rightarrow U_2$ where U_1 modifies the access control list of the record and U_2 then adds some data meant to be read by the members of the new access list, then under this model it is possible for someone who were in the previous list but excluded in the new list to read the new data at some replica (where U_2 reaches before U_1). Although the consistency model guarantees

both updates to reach every replica eventually, there remains a window of vulnerability. For many web applications this model presents a weak and inadequate option regarding their data consistency. PNUTS offers a relaxed consistency model to applications that can live with slightly stale but valid data and guarantees no such creation or access of "dirty data" that may later require a roll back.

It has been observed [21, 7] 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 all updates to the record in the exact same order. Updates are however propagated to all replicas asynchronously to ensure lower latency and higher availability of its operations in a large scale distributed environment. The following Figure 3 (taken from [7], p.1279) explains the timeline consistency for a particular record.

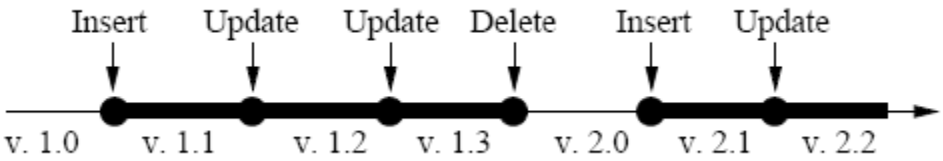


Figure 3: Timeline Consistency

(V. 2.1 indicates Generation =2, Version=1)

The events insert, update and delete of a given record are shown moving forward in a timeline. Between an insert and a subsequent delete, the bold line represents the time when the record is present in the data storage system. As the order of updates across all replicas is guaranteed to be same, the model ensures every replica evolve in the same way from one version to another and move forward in the timeline, although not in lockstep due to asynchrony in update propagation. A read of any replica in this model returns a 'consistent' version from the timeline. It may not always be the version resulting from the latest write, but potentially an older version of data which was correct at some point in the timeline.

PNUTS offers the following APIs (functionalities and semantics are explained in Section 4.2 which are shared by both PNUTS and the P2P Prototype, however the underlying

implementations are completely different) with varying degrees of consistency to provide applications with a useful handle for accessing a data store that relies on asynchronous replication.

- Read-any
- Read-critical (required_version)
- Read-latest
- Write
- Test-and-set-write (required_version)

The consistency model for the data storage system termed *per-record timeline consistency* is implemented by designating one replica of a record as master to which all updates from every replica are directed. The mastership is assigned on a per-record basis which means different records of the same data-table can have masters at different region. It is to be noted that the PNUTS key-value data store (which is horizontally partitioned into multiple tablets) is replicated in different geographic location and each region (cluster) hosts exactly the same set of data. A replica is only selected as master if it receives a majority of write requests originating from the same region. However the mastership of a record can adaptively migrate to another replica in case the workload shifts to the region hosting the replica. To keep track of the master, every replica of a given record maintains the identity of the current master as well as the origin of the last N (configurable) updates in its metadata field.

PNUTS uses Yahoo! Message Broker (YMB) which is a topic-based publish/subscribe system to help with its asynchronous replication. When an update from a client reaches the record's master, the master publishes it to the corresponding YMB cluster in the region. Once the publishing is done the update is considered 'committed'. YMB guarantees that any published message will be delivered to all topic subscribers asynchronously. It also provides a partial ordering for the published messages. Messages published in a particular YMB cluster will be delivered to all subscribers in the publish order, although no such ordering is guaranteed when messages are published in different YMB clusters. Master replica leverages these reliable publish properties of YMB to

implement its timeline consistency and publishes (commits) all updates of the record directed at it to the corresponding YMB cluster in the region. YMB ensures updates are asynchronously propagated to every replica and delivered in the commit order. When the system detects a change in mastership of a particular record, it also publishes identity of the new master to YMB.

The record-level mastering mechanism works for PNUTS as significant write locality is noticed on per-record basis in Yahoo!'s web workloads, meaning majority of the updates of a given record originates in the same datacenter. It has also been observed that different records have update affinity for different regions which justifies the record-level granularity. As most of the write operations are directed to the master in the same region, this mechanism minimizes the cost of write operations of a data record.

The *per-record timeline consistency* model presented in PNUTS provides a stronger guarantee than eventual consistency model, yet relaxed enough to work in a large scale distributed environment. Although asynchronous replication doesn't allow the data storage system to be strongly consistent, applications are however served with a variety of consistency handles to aid them extract data at different consistency levels. PNUTS is currently being used in production and serving social web and advertising applications. However the current version of PNUTS (unlike Dynamo) can't tolerate network partition in case an entire region having master replicas becomes unreachable and is only designed to work in stable and controlled environment. The detailed comparison of PNUTS against the peer-to-peer prototype is found on Section 7.1.

3.2 Dynamo

Amazon's Dynamo [21] is a highly available data storage system designed to provide a large number of applications (services) residing in Amazon's service oriented platform with an 'always-on' experience despite certain failure scenarios such as network partitions and massive server outages. In addition these services also have a stringent latency requirement even under high load. Dynamo uses *eventual consistency* model to manage its data consistency and is designed to be highly available especially for writes. As is the case for *eventual consistency*, Dynamo sacrifices data consistency under certain

failure scenarios or during high write concurrency to achieve higher system availability, better operational performance and scalability.

Dynamo's data model is a simple key-value data store as many of the Amazon's hosted services only require primary key based access to data. Nodes in the system form a ring structured overlay network and use consistent hashing [22] for data partitioning. The system exposes two simple operations - *get(key)* and *put(key, object, context)* to read from and write to the storage; *context* represents system metadata about the *object* e.g. version of the data. The *context* is kept alongside the corresponding *key* and *object* in the key-value store to help the system maintain its data consistency guarantee.

Dynamo replicates the data on multiple physical nodes and the replication factor (N) is configurable by applications. Upon joining the ring, a node is assigned a key range for which it is responsible. The node then additionally delegates the key handling responsibility to N-1 successor nodes by replicating the key range in those nodes. These N successive physical nodes constitute the preference list for each key in the key range (these nodes are preferred to handle the key) and additional nodes are included in the list to account for node failures. Dynamo is designed in such a way that every node in the system knows the preference list for any particular key, although this is not a scalable design.

A node handling a read or write operation for the requested key is known as the coordinator. Dynamo allows any node which is in the top N of the preference list for that requested key to be the coordinator. To maintain consistency among replicas, Dynamo uses quorum technique which involves two application configurable parameters: R and W. R is the minimum number of replicas that must take part in a successful read operation, where W is the minimum number of replicas that must participate in a successful write operation. The value of R and W are configured by applications in such a way that $R + W > N$; N is the total number of replicas of a data item in the system. In the absence of failures all R and W nodes are represented from the first N nodes in the preference list, with the overlapping quorum setting ensures strong consistency during read and write. However, nodes in the top N do fail or become unreachable and despite

trying circumstances highly available design of Dynamo makes sure quorum continues to work.

In Dynamo, read or write operations don't have to incorporate all R or W nodes from the first N nodes in the preference list. Rather the operations may involve the first N available nodes which may not be the top N nodes but nodes from lower down the preference list. This flexible quorum approach termed as 'sloppy quorum' is the reason why updates are never rejected even in the presence of network partitions, high concurrent writes or massive server outages. However this approach potentially introduces different versions of the same data in the system and data consistency suffers. Although Dynamo works with an asynchronous model where all updates reach all replicas eventually, some sort of write ordering is required to resolve the conflicting writes and make the data eventually consistent. Dynamo uses vector clock [23] to implement versioning of the data and to capture causality between different versions or 'replicas' of the same object. One vector clock is associated with every version of every data in the key-value storage. Vector clock which is essentially a vector of (node, counter) pairs can be quite useful in that it is possible to determine whether one data causally precedes another or both are concurrent by just comparing the vector clocks of the two data.

Dynamo is primarily designed for the applications that require high write availability; therefore conflicting versions are tolerated in the system during writes. However the divergent versions must be detected and eventually reconciled and this is done during reads. When a client intends to update a data object in Dynamo, it must specify in the context which version it is updating. The coordinator handling the write request generates a new vector clock from the local clock and the clock piggybacked in the client request. The resulting vector clock which associates itself with the client supplied data causally succeeds the local vector clock. The new version of data and the associated clock then overwrite the previous versions at the coordinator node. The coordinator then writes the new version of data (along with the new vector clock) to at least W-1 nodes from the preference list to complete the write. If the node participating in the write finds out the new version of the data to be causally unrelated to the existing version, both versions of

data are preserved along with their clocks. This results in version branching. If there is a causal order between the two versions the newer one is kept overwriting the older one.

When a coordinator receives a read request, it asks for all existing versions from the top N reachable nodes in the preference list and waits for at least R responses. If the versions received are all causally related, they are syntactically reconciled and returned to the client. In this case the highest version of data in the causal order subsumes all the previous versions. Concurrent versions which can't be syntactically reconciled are returned to the client for semantic reconciliation. The client being aware of the data schema and semantics merges the conflicting versions and hands it back to a coordinator. The reconciled data having a new version subsumes the conflicting versions and is written back to the system and eventually the data becomes consistent.

Amazon's Dynamo is a large scale highly available data storage system that uses eventual consistency as its data consistency model and is running in production with demanding applications e.g. Amazon's shopping cart service. It serves certain categories of applications that can tolerate inconsistent, out-of-order data under certain failure scenarios; however that demands data reconciliation from applications' part which is its weakness. In order to meet strict latency requirements, the system adopts full membership model and every node in the system keeps a full routing table as well as information on data partitioning responsibility for every node (by gossiping periodically) to avoid routing and lookup costs. However the design is expensive and limits scalability (therefore the current Dynamo only works with a few hundred nodes [21]) which is an important requirement for web applications.

4. System Design

This chapter describes the design of the prototype in greater details, also discusses the reasoning and motivation behind such design. Later in the chapter, the performance model of both the PNUTS and the prototype are presented and a comparative analysis is given.

4.1 Design Motivation and Assumptions

Even though the interfaces of PNUTS are preserved in the prototype and their intended functionalities and semantics are kept largely unchanged; the underlying mechanism to implement the APIs has a totally different approach than what is adopted in PNUTS. As mentioned earlier, in PNUTS all writes to a given record are forwarded to the master replica and master ensures that updates are applied to every replica in a particular order. Although serializing the writes through a single master is a convenient way to enforce ordering of the updates across all replicas and it works for PNUTS due to high write locality of Yahoo! Web workloads (see Section 3.2), the single master mechanism generally costs the scalability of a system which is an important requirement for large scale web applications; it also leads to uneven load distribution and makes the master replica a performance bottleneck.

The storage system prototyped in the project focuses on eliminating the limitations of a single master mechanism and follows a distributed approach where any node in the storage network receiving a client request can coordinate read and write operations using corresponding replicas. However, delegating the coordination role to just about any node in the network while maintaining the required data consistency, incurs additional complexity to the system. The peer-to-peer system is also designed to withstand churn of certain level which was not at all considered in the protected environment of PNUTS.

As for system assumptions, network partition wasn't considered during any execution. It is also assumed that majority of the nodes holding replica of a data item will not fail at any given instant. Security aspects are not featured in the design of the current prototype; its requirement might be evaluated in future.

4.2 System Architecture

The system is based on a peer-to-peer overlay network with scalable ring topology. Each peer has the exact same functional components (see chapter 5) and hosts data in (key, value) pairs in memory. In this prototype consistent hashing scheme [22] is used to partition the key-value data store and distribute them across multiple nodes in the system like Dynamo. The scheme, which works by hashing the data-key into a data-identifier and putting the corresponding record in the appropriate node (node-identifier ← successor (data-identifier)) in the ring, has good scalability in a sense that it only affects immediate neighbors when data handover is required during churn, while other nodes in the system remain unaffected. It is worth mentioning that even though data is 'replicated' at various nodes to aid fault-tolerance, availability and scalability, it is possible for different versions of data to exist in the system at any given instant. Figure 4 roughly depicts the prototype peer-to-peer system which implements the *Distributed Hash Table* (DHT) to manage its data.

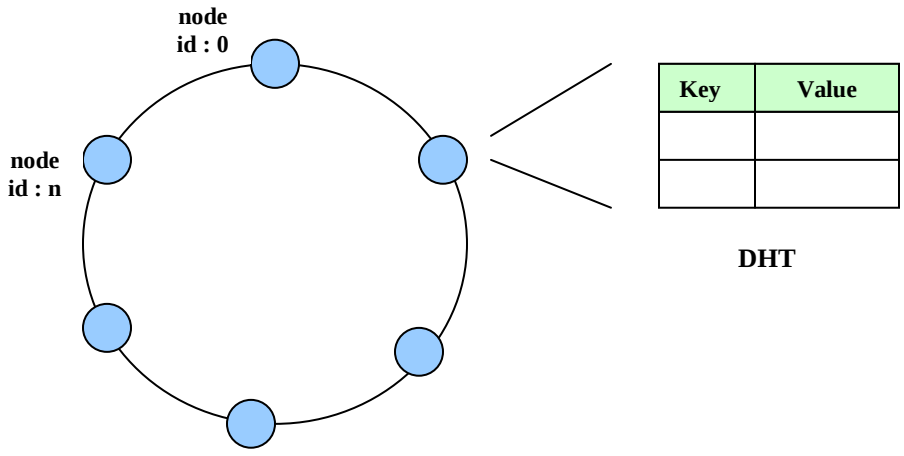


Figure 4: System Topology

Figure 5 takes an in-depth look into a peer and presents its major functional components that form the following layers of abstraction while it offers the read/write services.

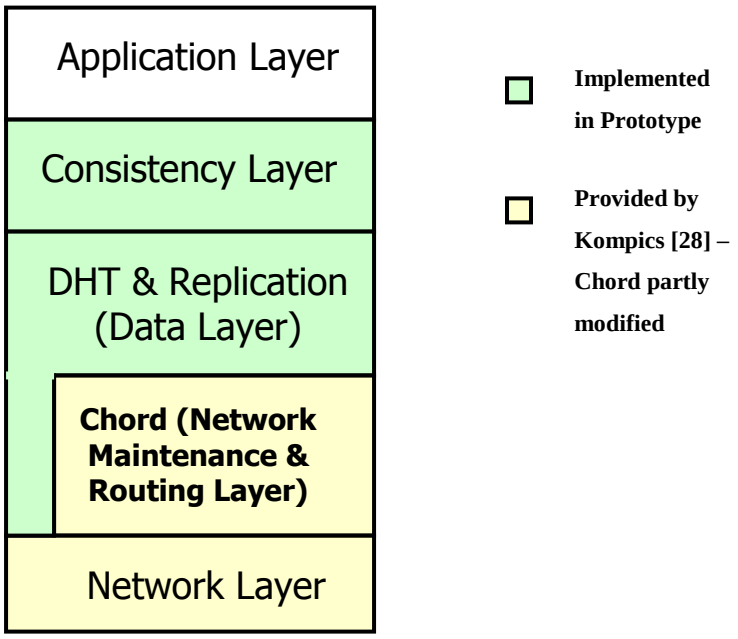


Figure 5: Major functional layers of a Peer during Read/Write

- **Application Layer** – Web-based or stand-alone distributed applications can invoke read and/or write APIs exposed by the consistency layer and access data from underlying storage with various consistency levels according to requirement.

- **Consistency Layer** – The layer implements the following read and write APIs that facilitate distributed data storage access managed by the lower data layer. The implementation details of these APIs are given in section 5.5.

  - *Read Any (key):* Read any API could possibly return an older version of the data even after a successful write; however this call has lower latency. Applications that prefer faster data access and performance and give less importance to data consistency can make use of the API.

  - *Read Critical (key, version):* This API returns data with a version that is at least equal to or higher (newer) than the required version. Using this API, applications can enforce read-your-writes consistency; meaning when a

user writes a record (which returns the version of the data in case write succeeds) and then needs to read a version that must reflect the update, this call should ensure that read guarantee.

- *Read Latest (key):* Read Latest API returns the latest version of the data, although it has higher latency compared to other reads. This API is useful for applications to which consistency matters more than performance
- *Write (key, data)*: This API is used to write data blindly, meaning node just overwrites a data item without reading it.
- *Test and Set Write (key, data, version)*: The API only writes data if the present data version at the storage is same as the version required by the client. This API can be used to serialize a transaction that first reads a data record, and then later come back to write to the record based on the previous read. If the version is found to be different than what is required for the write that implies other nodes have already changed the data. In that case *Test and Set Write* is rejected.

- **DHT Layer** – The layer hosts (key, value) data store that is part of the *Distributed Hash Table* (DHT). It also contains symmetrically replicated data (see Section 4.3) of other nodes. During read and write operations *Get (key, attachment)* and *Put (key, value, attachment)* interfaces, implemented in DHT layer, are invoked by upper consistency layer. The attachment object of the *Get* operation tells whether to get data or just version number (which is necessary for *Write*). In case of *Put* operation, the attachment indicates whether the data to be written belong to *Write* or *Test and Set Write*, as these two APIs operate differently at storage level. Besides writing data, *Put* operation is used to lock or unlock data and attachment object also carries that information. If data access turns out to be remote, *Get* and *Put* operation look up the key using underlying Chord layer. After the key is resolved, the requesting node contacts directly with the remote node through network layer to either store or retrieve data or version, or to lock or unlock data. The layer also provides functionality to manage data

relocation and recovery during nodes' arrival, departure or failure which has been discussed further in Section 5.3.

- **Chord Layer** – Chord [11] layer looks up any given key for the upper DHT layer in logarithmic time $O(\log_2 N)$ and returns the responsible node for the key. It maintains a routing table whose space cost is also logarithmic $O(\log_2 N)$. Here N represents maximum number of nodes in the network. This is unlike PNUTS (direct mapping approach [7]) and Dynamo (full membership model [21]) systems which are simpler routing techniques and not very scalable. The layer provided by Kompics enables nodes to join or leave the ring, also carries out periodic stabilization to keep network pointers correct (eventually) in the presence of churn.

- **Network Layer** – Network layer provides simple interfaces for sending and receiving messages to and from the network respectively. This layer works in the OSI application layer and should not to be confused with the OSI network layer which is part of the local operating system. A brief description on Kompics provided implementation of this layer is available in section 5.1.

4.3 Symmetric Replication and Data Recovery

Symmetric replication scheme [24] has been used in the system to replicate data at several nodes. It has a useful property that given a data identifier it is possible to lookup any individual replica of the data item, in other words replicas can be identified and requests can be directly sent to those using this scheme. This also leads to better load balancing as requests meant for other replicas don't need to be forwarded to the master node. As concurrent requests can be routed directly to the replicas, distributed voting is also possible here. Another advantage with this scheme is, each join or leave of a node only requires O(1) messages to be exchanged to restore replication degree.

Other replication technique such as *successor list replication* is suggested in [11] where data is replicated at successive nodes in the ring (used in Dynamo). In this scheme only the 'master' node having the first replica can be identified; other replicas can only be

reached using successor list of the master node. This scheme has major disadvantage in that the node holding the first replica becomes a performance bottleneck; since every request concerning the data item has to be sent to the node before it can be forwarded to any specific replica. It can also pose security threat as every request has to pass through the 'root' node. The scheme requires $\Omega(f)$ messages [24] following every join, leave or node failure to maintain a replication degree of f which is much costlier than symmetric replication. Another replication technique, m*ultiple hash functions* scheme has been used in DHTs like CAN [25], Tapestry [26] where f different hash functions are applied to the key to get f distinct (replica) identifiers, which are then used to replicate data at f nodes. The problem with the scheme is that it can't restore replication degree in the wake of replica failures. When a node fails, the successor needs to identify the replicas of the failed data items and recover data from those in an effort to maintain replication degree. As tracing replicas directly from lost data identifier is not possible in this scheme, only option available is successor hashing the failed keys with known hash functions to identify the replicas. Finding out which keys failed however requires inversing the hash functions, that is $Hash^{-1}$(data-identifier) = key, which is mathematically impossible [24].

Symmetric replication scheme that has been implemented in the DKS system [27], works by partitioning the identifier space into number of non-overlapping equivalence classes. All the identifiers in a single equivalence class are associated with each other. For simplicity congruence classes modulo f (replication degree) has been used in the prototype for partitioning the identifier space (same as [24]). A data item having an identifier of a given equivalence class is replicated at f different nodes, if each of those nodes is responsible for one distinct identifier of the equivalence class. To put it differently, if a node stores data item with a particular identifier of a given equivalence class, it also hosts data items with different identifiers belong to the same class. It is due to this symmetry of data at replicating nodes that a join or leave event involves interaction with only one node (neighbor); in other words only O(1) messages are required during data hand over. When a node fails, the successor node assumes responsibility of the identifier range (m, n) that the predecessor was responsible for before failure. In order to recover from loss of data, successor then selects another

identifier range (m′, n′) of equal length where both m and m′ are members of the same equivalence class. Similarly identifiers m+1 and m′+1 share a separate equivalence class and so on. Therefore as *symmetric replication* suggests, nodes responsible for the identifier range (m′, n′) should contain the same data set as those with identifier range (m, n). In this replication scheme, successor only from knowing failed identifier range can identify corresponding replicas and recover data to restore replication degree.

4.4 Performance Model Analysis

In this section we look into the design of APIs of both PNUTS and the peer-to-peer prototype. Even though the read and write APIs of both systems share identical interface, the respective underlying implementations vary quite a lot. As of now the detailed operational workflow of PNUTS APIs is not yet revealed, only a brief overview is given in [7] on the logical steps taken by the APIs involving few system components. This should be enough to construct performance models of the individual APIs of both systems and compare them at analytical level. In case of performance model for the prototype, the APIs are considered to follow policy 1 (see Section 6.3).

4.4.1 Read Any

The following diagram shows interaction between different entities in PNUTS during *read any* operation. *M* represents master replica and R_1, R_2 are just other replicas. Here *read any* operation reads from local region.

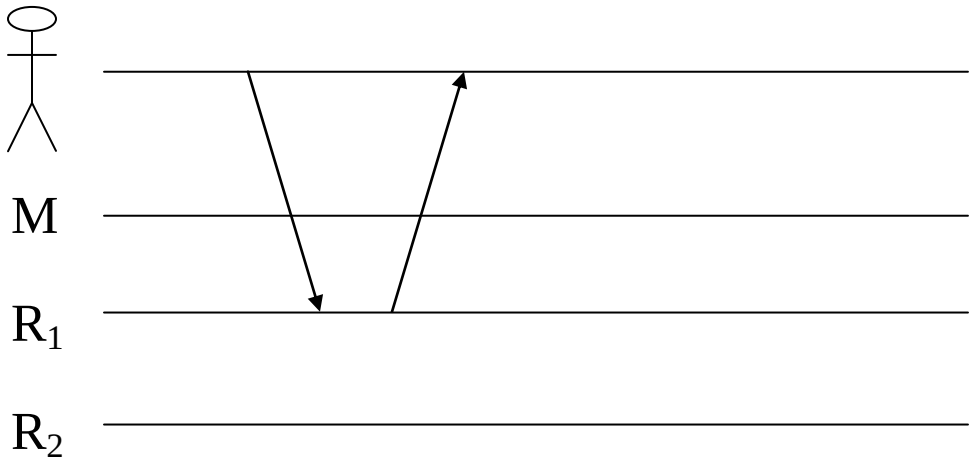


Figure 6: Interaction diagram for *Read Any* in PNUTS

For peer-to-peer prototype there is no master replica. Here request is broadcast to all replicas R_1, R_2, R_3 (policy 1) and R_2 returns the data first. Detailed description on implementation of the API can be found in Section 5.5.

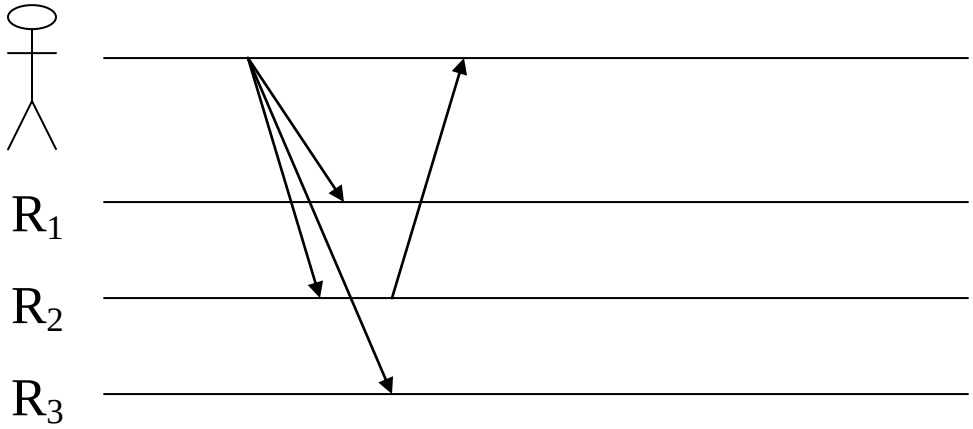


Figure 7: Interaction diagram for *Read Any* in P2P Prototype

4.4.2 Read Critical

Although the workflow of *read critical* operation isn't disclosed in [7], it can be guessed that request would be forwarded to the master replica, in case the required version is not available at local (requesting) region (as it is not possible to know which of the other regions has the required version). The following diagram imagines the worst case scenario; however in best case the request would be met locally (see Figure 6).

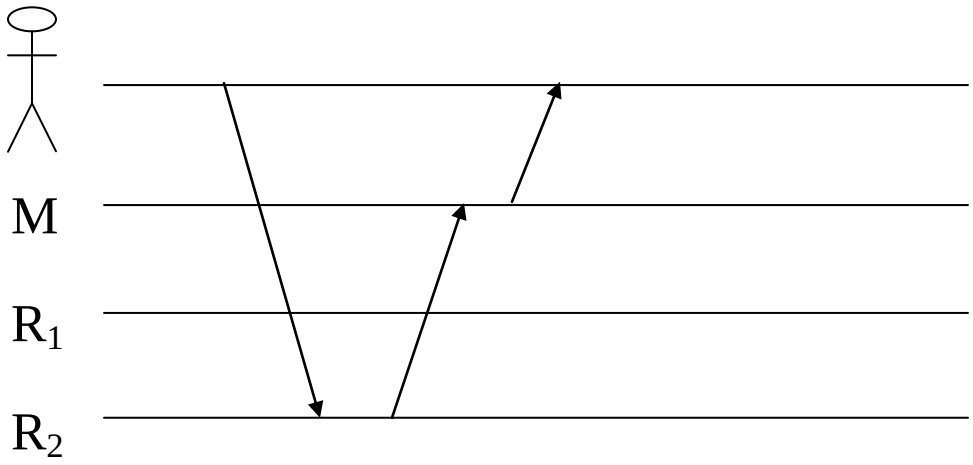


Figure 8: Interaction diagram for *Read Critical* in PNUTS

For peer-to-peer prototype the request is broadcast to all replicas and R_2 returns the data with required version earliest. Detailed description on implementation of the API can be found in Section 5.5.

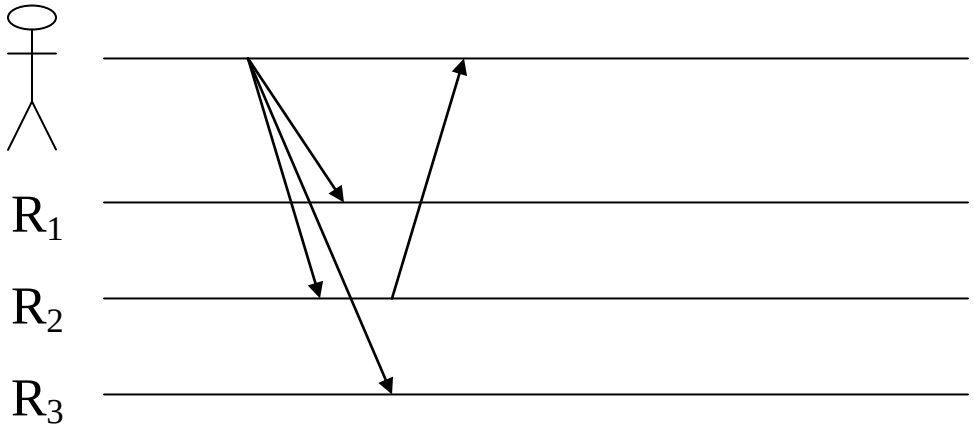


Figure 9: Interaction diagram for *Read Critical* in P2P Prototype

4.4.3 Read Latest

Even though it is not revealed how *read latest* is implemented in PNUTS, the following model is most probable because of the master based design PNUTS adopts. As master always has the last successful update, *read latest* request originating in any region is likely to be forwarded to the known master replica from where the latest value is retrieved.

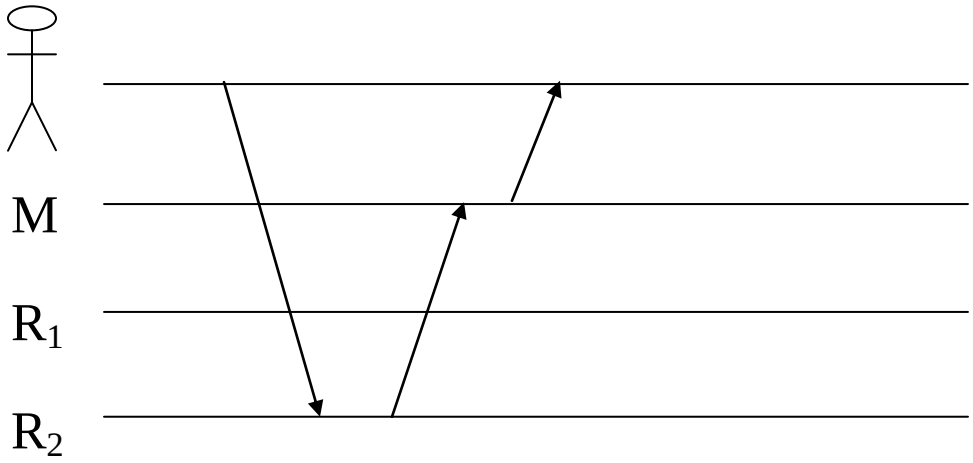


Figure 10: Interaction diagram for *Read Latest* in PNUTS

For peer-to-peer prototype, majority of successful responses has to be received by the requesting node to get the latest data. Detailed description on implementation of the API can be found in Section 5.5.

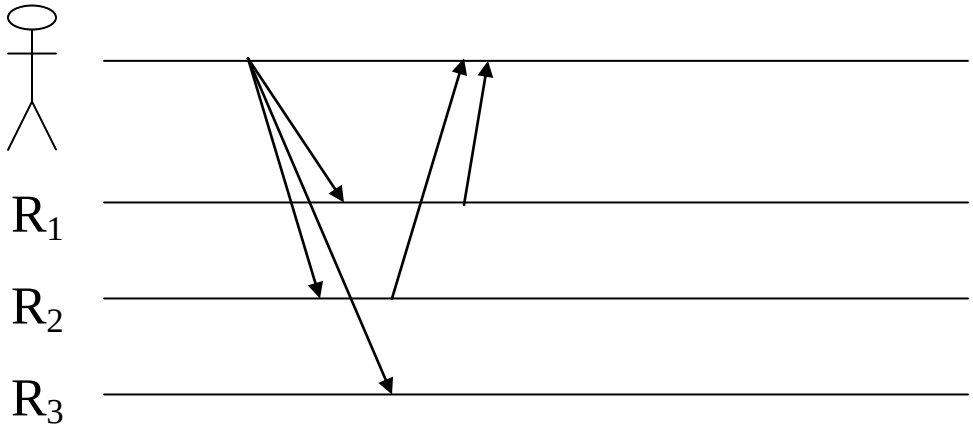


Figure 11: Interaction diagram for *Read Latest* in P2P Prototype

4.4.4 Write

For *write* operation in PNUTS, the request travels to master from originating region. Master then writes the update to Yahoo! Message Broker (YMB) and as soon as YMB acknowledges, the write is considered committed (discussed in Section 3.1). YMB the reliably delivers the update to all other replicas in asynchronous manner.

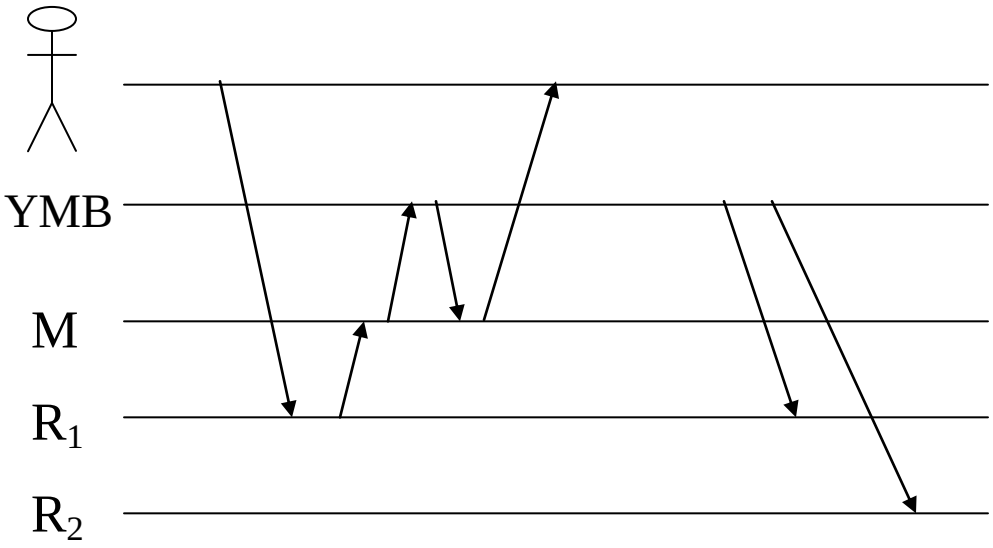


Figure 12: Interaction diagram for *Write* in PNUTS

For peer-to-peer prototype, after receiving majority of version responses in the first round, the requesting node sends write requests (with latest version) to all nodes in the next round. As soon as it receives majority acknowledgements the *write* operation is considered complete. Detailed description of the API can be found in Section 5.5.

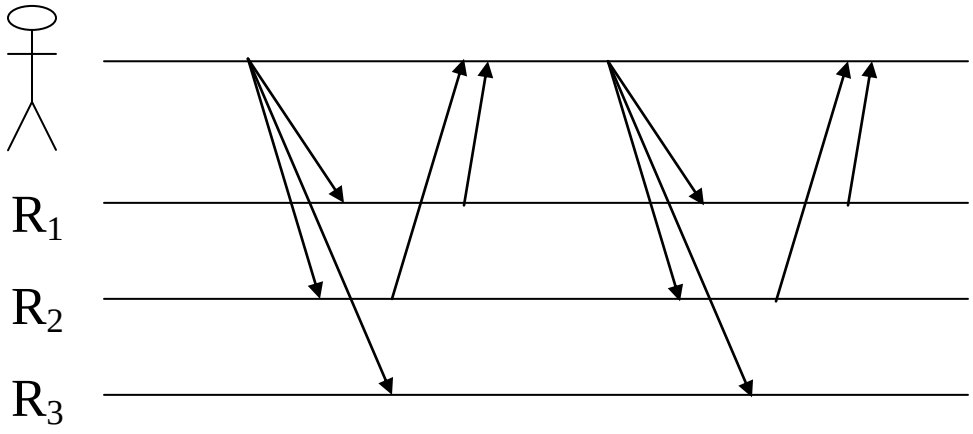


Figure 13: Interaction diagram for *Write* in P2P Prototype

4.4.5 Test and Set Write

The request travels to master from originating region and there the version number is checked. If version matches, master writes the update to YMB and as soon as YMB acknowledges, *test and set write* is considered committed. YMB the reliably delivers the update to all other replicas in asynchronous manner.

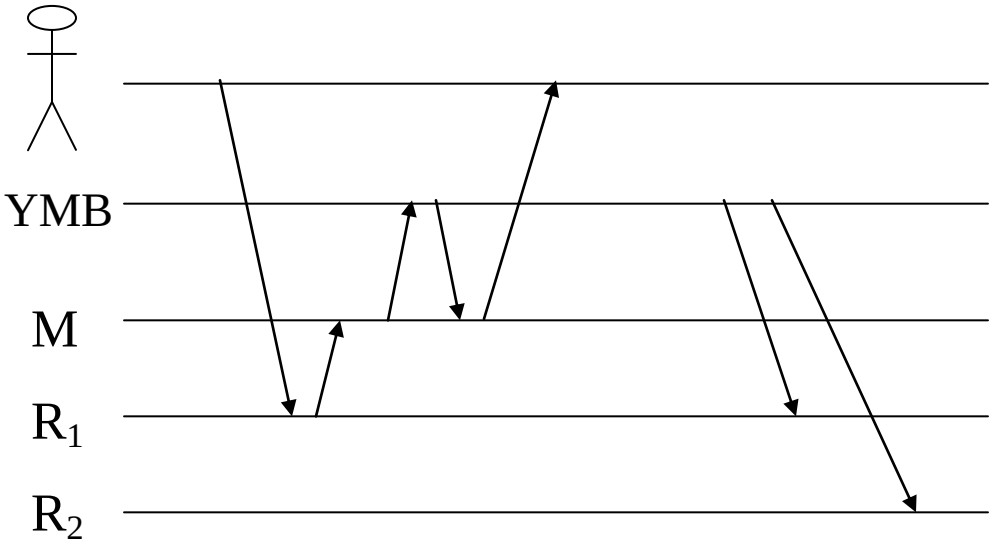


Figure 14: Interaction diagram for *Test and Set Write* in PNUTS

For the prototype, the requesting node sends lock requests in the first round. After receiving majority of lock responses (with version) and in case the version matches the required one, it sends write requests to all nodes. As soon as the node receives majority of acks, the *test and set write* operation is considered complete (see Section 5.5).

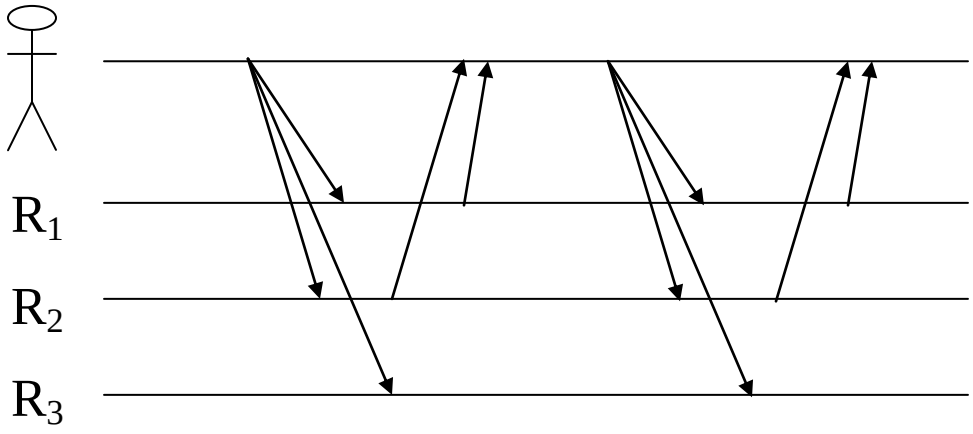


Figure 15: Interaction diagram for *Test and Set Write* in P2P Prototype

The logical steps (hops) and number of messages (not actual messages) involved in the analytical performance model of both systems are compared in the following table. Here in case of PNUTS, from requesting node to local region back and forth is considered 2 hops and 2 logical messages are involved. Traveling to master costs 1 more hop and entail 1 extra message. The asynchronous message propagation to replicas in Yahoo! Message Broker (YMB) is not included in message calculation. For the prototype, if the requesting node sends requests to all replicas and gets back response(s), it is considered 2 hops. Message counting ends, as soon as read or write operation returns at the requesting node for both systems. It is worth noting that in the end both systems costs about the same amount of messages proportional to the number of replicas in the system (taking into account the asynchronous messages propagating in YMB for PNUTS). Worst case scenario is considered for both the systems (for example, *read critical* has to travel cross region in case of PNUTS as data isn't available locally; for the peer-to-peer prototype majority responses has to be received before the required version is found) and replication degree is assumed 3.

API	P2P Prototype		PNUTS	
	Hops	**Message**	**Hops**	**Message**
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 1: Analytical comparison of two models of both systems

5. Implementation

The chapter presents overall system description at component level in the beginning. There is also a brief mention of the software framework that has been used to develop the component based system. The rest of the chapter discusses the individual components implemented in the prototype and gives insight into their functionalities and implementation logic.

5.1 System Overview at Component Level

Prototype implementation has been based on Kompics Peer-to-peer Framework [28] (version 0.4.2.7) which offers both a development as well as a simulation platform. Simulation environment is covered separately in the evaluation chapter; the development platform that has been leveraged during implementation is focused here. In Kompics framework, protocols and distinct functionalities can be implemented as event-driven components and can be easily composed to build complex distributed systems. Kompics components are reactive state machines that run concurrently and communicate by passing typed events (passive, immutable typed objects) through typed bi-directional ports (component interface acting as filter for events) connected by channels. Events can carry data and be extended as Messages when pushed into the network. Implemented in Java, Kompics framework offers a runtime environment as well as a component library to facilitate reuse of components.

In the prototype, *MyPeer* component is a major component that includes several sub-components such as *Consistency*, *DHT* and *Chord* among others. The component is mainly responsible in carrying out the functionality expected of the system. Figure 16 (taken from [29], p.18) shows the component architecture of an executable system which is designed to be deployed on nodes acting as peers (in a setup where peers running on different machines).

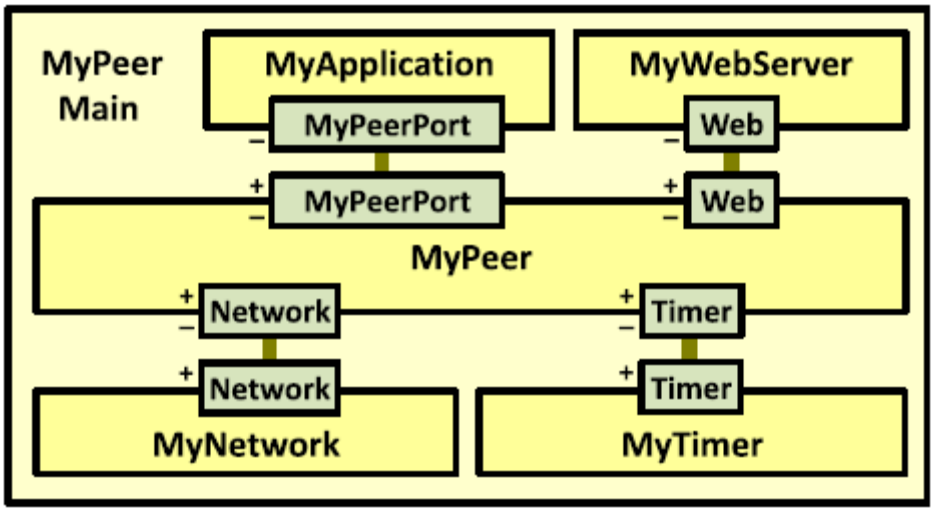


Figure 16: Component architecture of system designed for single peer deployment ('-' indicates component providing service, '+' indicates component using service. Complementary ports are connected by channels)

In this setup, the executable *MyPeerMain* component embeds the *Network*, *Timer*, *Web-server*, *Application* and *MyPeer* components. The *Network* component uses the Apache MINA [30] network library and manages automatic network connection as well as message serialization and compression, where the *Timer* component provides timer related services (set and cancel timeout). Through its *Web-server* component a peer exposes its status as HTML page. The *Application* component may embed a GUI and issues functional requests (or accept indications) to (or from) the *MyPeer* component through *MyPeerPort*.

5.2 *MyPeer* Component

The architecture of the *MyPeer* component is presented in greater detail in figure 17. The component and few of its sub-components e.g. the *MyConsistency*, *MyDHT* and *MyReplication* have been implemented from scratch; while other sub-components are already provided in the Kompics component library, of which the *Chord* component has to be modified partially in the prototype.

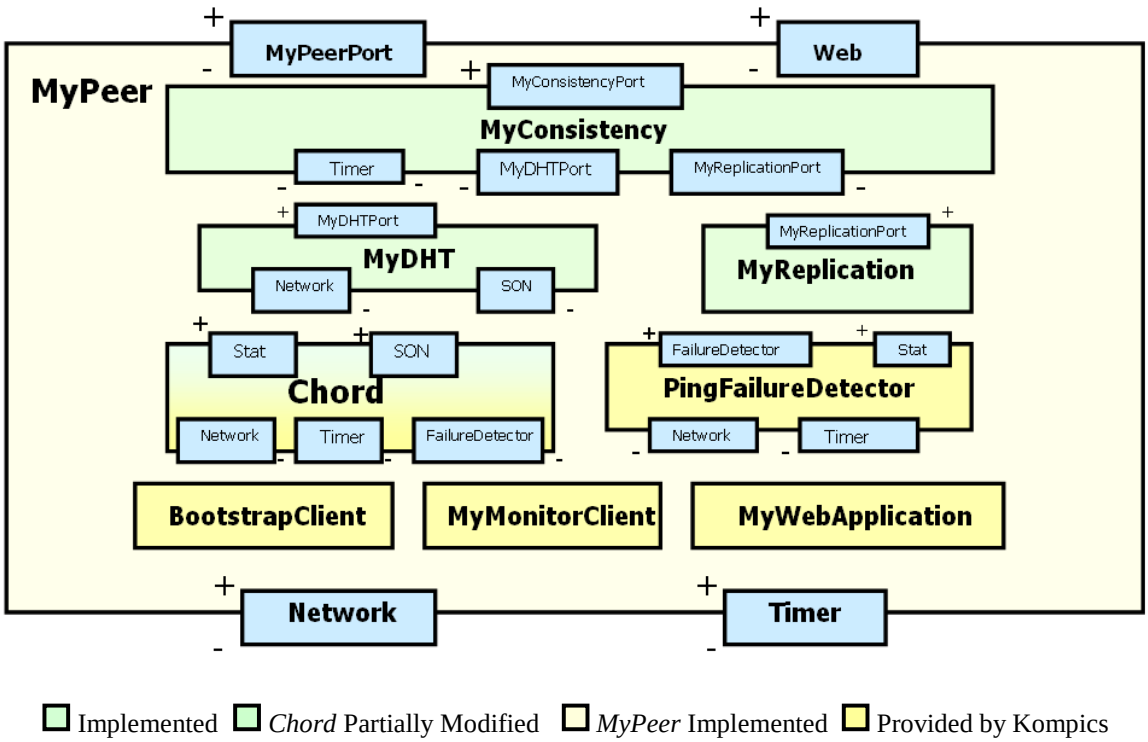


Figure 17: Detailed Architecture of *MyPeer* Component
(Some ports and channels are omitted for clarity)

When a peer starts, its bootstrap client component connects to a known *bootstrap server* to collect a list of peers already running in the system. The node then joins the overlay network via one of the existing peers and begins to send periodic keep-alive messages to the *bootstrap server*. The monitor client component periodically checks status of various components of the host peer and reports this to a known *monitor server* that aggregates the status of all the peers to compile a global view of the system (the monitoring option is mainly used for testing and might not be used in large deployments as it might not scale). Figure 18 (taken from [29], p.17) demonstrates the component architecture of both *bootstrap server* and *monitor server*. Web application component provides a web interface to the peer allowing users to access the status of peer components and enables users to issue interactive commands (although this component was not utilized in the prototype). Ping failure detector component implements an eventually perfect failure detector [18] which regularly sends ping messages to neighboring peers and expects their

pong replies. Chord component implements the Chord [11] structured overlay network (see section 4.2).

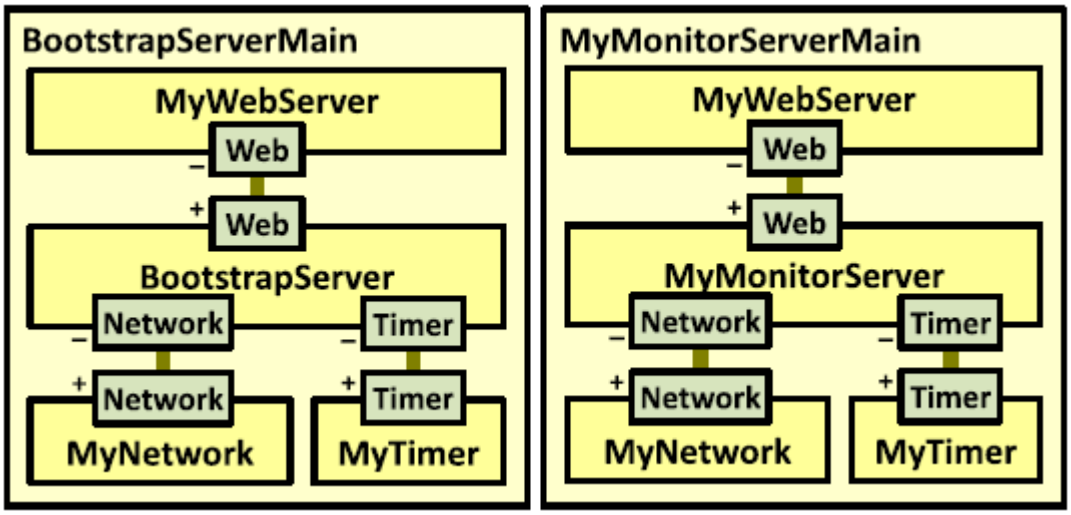


Figure 18: Generic Peer-to-peer Bootstrap and Monitoring Server

5.3 Replication and DHT Component

The *Replication* sub-component inside the *MyPeer* component implements symmetric replication scheme. When a numeric key (data or node identifier) is input, the component returns associated numeric keys from input's equivalence class. These associated keys, described in section 4.3, are used to identify and lookup individual replicas in the ring.

The *DHT* component being host of the key-value data store, processes read and write requests coming from the upper *Consistency* component. If the request is not meant to be processed locally, it is sent to a remote *DHT* component after looking up the key using the *Chord* component. It is worth mentioning that data store in the component is accessed in a serialized manner through synchronized Kompics handlers.

The *DHT* component also carries out data management (data relocation/replication) tasks during churn. When a new node *joins*, it becomes responsible for a section of keys in the ring. It then requests its successor for any data that is now its responsibility. After copying the data, the node sends acknowledgement to the successor which then deletes

duplicate entries from its own data store. During *leave*, node hands over all its data to its successor and gracefully exits the network. When a node detects *failure* of its predecessor, it needs to retrieve the data the failed node was hosting in order to maintain replication degree. The node carries out this task by first finding out the section of the ring (key range) the failed node was responsible for. The *DHT* component then decides on another key range of which every key is associated one-on-one with the failed key range. The component then resolves the new, associated keys using underlying Chord component, get nodes where replica should be found and fetch existing replicas from the nodes. This ensures that replication degree be maintained despite failure. In the prototype a simple mechanism is used during failure recovery which involves looking up every key in the new range and fetching replica (if exists) from the corresponding node. A more efficient mechanism (interval broadcast mechanism which doesn't lookup every key, rather covers intervals) will be introduced in future to reduce the number of unnecessary messages in the network.

5.4 Chord Component

Kompics provides the Chord component that implements the protocols introduced in [11]. However the implementation (join protocol) has been modified partly to accommodate the following scenarios that were encountered during experiments.

a) Before a new node (with new random identifier) joins the network, the protocol requires the node to find its successor. In a high churn environment it is possible, a node that has already failed might be returned as successor to the joining node. This anomaly arises from the fact that some node (last one in the iterative/recursive lookup) that finally resolves the key and returns the successor, couldn't detect its (successor's) failure. The new node subsequently sets up its successor pointer to the failed node and joins the network. It then attempts to update its own successor list by asking the supposed successor; although it doesn't get any reply. In the end, the node forms a separate ring itself parallel to the existing ring, which is not acceptable.

To prevent this multiple ring problem from happening, join protocol needs to be modified as follows. When a node receives its successor during join, instead of acknowledging the upper (application) layer straightaway, it delays the joining. The node then sends a message to the successor requesting successor list and sets a timer. If the successor returns the list before timeout, it is confirmed alive. Therefore the node can safely set up its successor(s) and join the network (acknowledges the upper layer). If no response is received within time, that indicates the successor might have failed and joining anyway would create a parallel ring. So the node aborts join and acknowledges about its unsuccessful attempt to the upper layer.

b) During bootstrap, the joining peer fetches a list of nodes (from bootstrap server) already inside the ring and uses one of them to join the network. In a high churn environment it is probable that the peer fails to receive any successor despite using up all the nodes in the list, as every single attempt to get a successor and join the network fails. This can happen if some nodes in the list returned by the bootstrap server may have already failed or some intermediary nodes involved in the (iterative/recursive) lookup is detected suspect resulting in the overall lookup failure. In such a case, instead of assigning itself as successor and risking a parallel ring (as some nodes could still be alive in the ring), the join is aborted.

5.5 Consistency Component

The five read and write APIs that provide various data consistency guarantees have been implemented as *ReadAny*, *ReadCriticial*, *ReadLatest*, *Write* and *Test&SetWrite* components. They are embedded in the *MyConsistency* component as shown in figure 19.

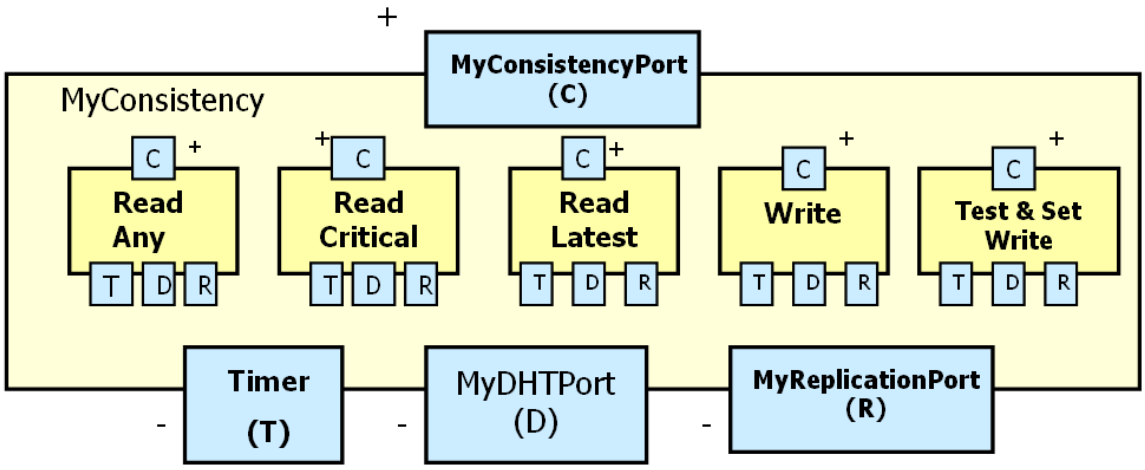


Figure 19: Detailed Architecture of *MyConsistency* Component

(Channels are omitted for clarity)

All these components receive their respective read or write events (requests) from the application component through the *MyConsistency* port. After getting a request, the read or write component tags it with unique identifier and sets a timer for it. If the request is timed out before a response is received, *failure* event is returned to the application component. Every read/write request of a data item attaches a numeric key (after hashing the primary key) whose associated keys (each represents one replica) are found from the replication component. The request is then pushed down to the *DHT* component to send it to different replicas (see Table 3 for different multicast policies). The read/write operations work in the following way.

- ***Read Any Component*** – The *Read Any* component sends read request to all replicas (*default* policy) and as soon as it receives the first successful response, it returns the result to application. If the data item is found locally, the operation immediately returns. In case no response is received within a time bound, the operation is timed out and a failure is returned. Operation also fails if all responses return failure. Failure could happen if data is not found at the expected node or looking up of a key fails at the Chord layer as there might be suspected nodes because of churn. Although the default policy is to send request to all replicas, an alternative design policy is tested where request is only sent to any two random replicas with a view to reducing the number of

network messages. The comparative analysis between the two policies is presented in section 6.4.

- ***Read Critical Component*** – The *Read Critical* component sends read request to all replicas (*default* policy) and whenever it receives data item with version greater than or equal to the required version, the operation returns. Read critical operation fails if all responses return failure which can be attributed to receiving older version or data not being found at the expected node or key lookup failure at the Chord layer. Operation also fails in case it times out while waiting for successful response. An alternative design policy e.g. *majority* policy (sending requests to only majority of replicas) has been introduced for this operation which is expected to involve less number of messages at the cost of fewer successful responses. The two policies are compared and the result is shown in section 6.4.

- ***Read Latest Component*** – The *Read Latest* component sends requests to all replicas (*default* policy) and waits for majority of successful replies. As soon as it receives the majority, it finds the latest version of data and returns it to application. As successful writes involve writing to distinct majority, read latest (R) always overlaps with write (W) that is R + W > N (no of replicas) as shown in figure 20 and guarantees client with the latest data.

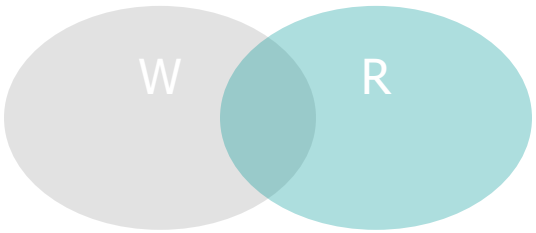


Figure 20: Read Latest and Write overlaps

Read latest operation fails if the operation times out or majority of responses return failure. An alternative *majority* based design policy has also been examined where requests are sent to majority of replicas as opposed to all. The comparative result is shown in section 6.4.

- ***Write Component*** – To handle a write request from application, *Write* component first sends version requests to all replicas and waits for majority of responses. Reading from majority quorum ensures latest version number is read (as write also uses majority quorum), the component then adds one to the highest number, tags the new version number with application supplied data and writes it to all replicas. For new inserts, version 0 is returned as replica doesn't have data. If majority of writes are acknowledged, the write operation is considered successful and operation returns. It is worth mentioning that the term 'majority' implies majority of replicas comprised of distinct nodes. It is possible that two or more replica keys point to the same physical node as nodes may not be uniformly distributed in the ring. So the same node may respond to two or more requests (meant for different replicas), all of which should be considered as one in the majority count. If two or more distinct nodes try to write data with the same version number, node with highest identifier in the ring wins.

  Write to a replica fails for any of the following reasons - key lookup failure in the Chord layer because of suspected nodes; failure to write data or get required version number as the data item might be locked (by a test and set write operation); or a more recent data (with greater version number) may have been concurrently written which denies writes involving older version. Operation as a whole can fail due to lack of version numbers or write acknowledgements (lack of majority), it can also time out if not completed within time.

- ***Test and Set Write Component*** – After receiving a test and set write request, the component sends lock requests to all replicas. Each replica then locks the data item in case it is not already locked by another operation and sends back successful lock response to the requester piggybacking the version number (stored at replica) with it. After receiving majority of lock responses, the component tests whether the latest data version (retrieved from majority)

matches the required version. If they match, application data is written back to the locked replicas with new version number (after one being added to the recent version); the replica immediately unlocks the data and returns acknowledgement. As soon as majority of write acknowledgements are received, the operation is considered successful and it returns to application. It is to be mentioned that even if data is locked read is allowed to ensure high read availability. However '*Write*' or other '*Test and Set Write*' operations are not allowed to access locked data, even version request is denied.

There are several reasons why test and set write fails at a particular replica. Failure to lock data which is already locked by another operation, failure to resolve key in the Chord layer, or not finding data expected at a replica due to churn are worth mentioning. Operation as a whole can fail if the recent version retrieved from quorum of lock responses doesn't match the required version; or majority of lock responses or write acknowledgements is not received within time bound and operation times out. When an operation fails, the replicas that have already been locked as part of the unsuccessful operation need to be unlocked and this is done by the component that initiated the operation. Late lock requests that were not part of the majority can also arrive at a replica even after the operation is complete. This causes the replica to lock the data (in case data is open to lock) and return the lock response back to the requesting node even though the node is already done with that particular operation. Upon receiving the late lock response (lateness is detected as the request doesn't relate to any pending operation), the component at the requesting node acts on it and sends unlock request to the replica which unlocks the data.

6. System Evaluation

The chapter starts with a discussion about the components involved in the simulation. The experiments to be conducted are outlined; the experimental setup is also described briefly. Finally the results are presented and system behavior and performance is observed and analyzed under various system settings.

6.1 System Simulation

Kompics provides simulation platform on which the same executable component that builds up an individual peer (one peer per machine set up) can be executed unaltered (with same code) in simulation mode with few additional components included. Therefore it is very convenient and rapid to move the real execution to simulation platform. Figure 21 (taken from [29], p.19) shows the complete component architecture of the system to be executed in simulation mode.

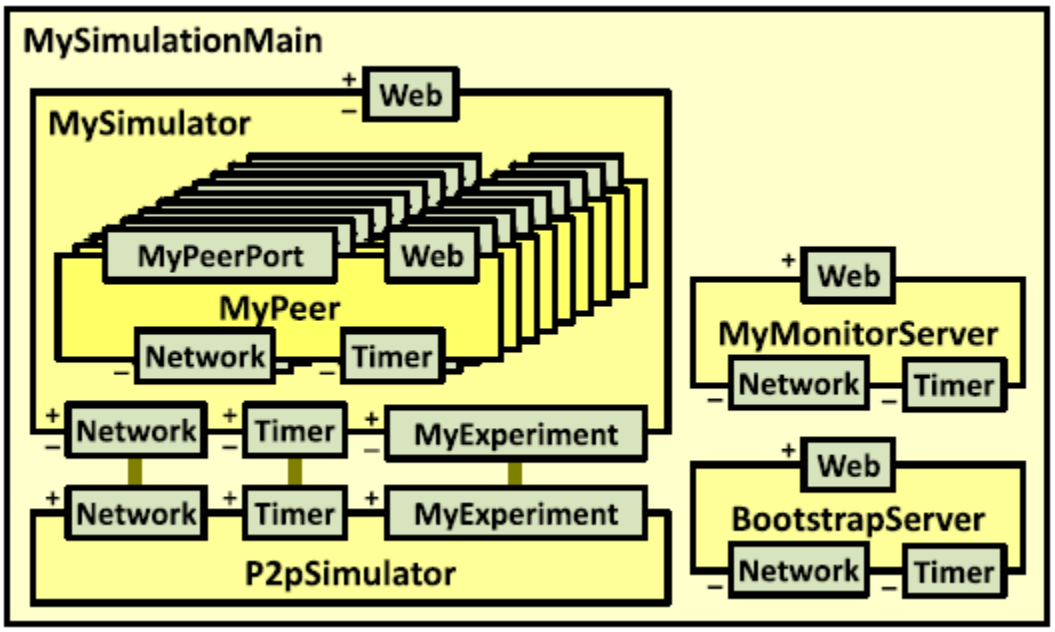


Figure 21: Component Architecture for the whole System Simulation

As described in section 5.2, the *MyPeer* component performs all the functionalities intended for a peer and is designed to be deployed on different physical nodes (see Figure 16). In simulation mode, the system-specific simulator component (*MySimulator*) spawns a number of these components to represent the number of peers in the system. The *P2pSimulator* component provided by Kompics component library implements a generic discrete-event simulator. It interprets an experiment scenario[1] and issues system specific events to the *MySimulator* component through *MyExperiment* port. The events or commands instruct the *MySimulator* component to create and start a new peer (for join), stop and destroy an existing peer (for leave, fail), direct a peer through its *MyPeerPort* to execute read-write operations, or collect simulation data for system evaluation. The *P2pSimulator* also provides the *Network* and *Timer* abstractions (see Section 5.1) required by the *MyPeer* components. The main executable component (*MySimulationMain*) also includes the *MonitorServer* and *BootstrapServer* components (see Figure 18) and represents the complete architecture of the system in simulation mode. It is to be mentioned that all peers and servers execute within a single OS process in simulated time.

6.2 Evaluation Plan

As described in section 4.2, the memory-based key-value storage system is hosted on top of a peer-to-peer overlay network that is completely distributed, can tolerate different churn rates and may receive variable read-write request loads. Several system parameters have been identified in such an environment that can potentially affect the performance of the five read and write APIs that are providing access to the storage with different consistency levels. The following table enlists the experiments to be carried out and the corresponding parameter that will be varied in each of the experiments. Their impact on certain distributed system properties will be observed using relevant performance metrics.

[1] A scenario can be defined as a parallel and/or sequential composition of stochastic processes (e.g. boot, churn, data collect etc.); stochastic process is a finite random sequence of events with a specified distribution of inter-arrival times. See [29] for further details.

System Parameters to vary	Properties to be tested
Network Size: System starts with certain number of nodes that is configurable. In this experiment system performance will be observed for different network sizes.	Scalability
Churn Rate: Mean life time of nodes that maps to churn rate will be varied to see its impact on the system. Lifetime based churn model is discussed later in the section.	Dynamism
Replication Degree: System performance will be observed by varying number of data-replicas in the system. The parameter is pre-configured before running the system.	Fault Tolerance/Availability
Request Rate: The read and write request load having a mean inter-arrival time are generated from different nodes. The rate (inter-arrival time) will be varied to measure its impact on system performance.	Scalability/Availability
***Read-Write Ratio*:** Read to Write request ratio will be varied to see if any inter-dependency between read and write exists in the system.	Request Inter-dependency

Table 2: Experiment list and parameters to vary

The following performance metrics, associated with each API, are measured after every simulation run:

- **Mean Latency**: The time duration between a request and its corresponding response i.e. latency is recorded and averaged with respect to each API. Only successful operations are considered for latency measurement. It is to be mentioned that Kompics simulate network and uses traces from the KING latency dataset [28] to make simulations resemble running on real Internet.

- **Operation Success Ratio**: The metric is defined as the ratio of the number of successful operations (reasons for operation failure are mentioned in Section 5.5) to total number of operations issued, with respect to each API.

- **Message Count**: All system specific messages received over network by every peer are aggregated and counted until the end of an experiment. The count not only includes get/put messages at DHT layer, key lookup messages at chord layer that are generated in response to incoming read/write requests or generated to perform data maintenance task following a churn event (join, leave, failure); but also incorporates 'infrastructure' messages e.g. periodic failure detector messages, bootstrap messages, messages for periodic chord network stabilization and routing table update etc. that are passed among the nodes to keep the system going. Volume of the messages not only depends on incoming request rate and churn in the system, but also varies on how system is configured. Therefore message count itself isn't so much of an important indicator as it is the growth of the messages during experiment. The growth of messages is an important issue for the scalability of the system.

- **Churn Count**: Churn count shows the degree of dynamism in the system. To represent churn in the prototype lifetime based churn model [31] is implemented. In this model, a node upon joining the network is assigned a random lifetime drawn from some distribution and after the lifetime expires the node either fails or leaves the network. To prevent the network size from reducing to zero, the failed or left node is immediately replaced by a newly joined node that is assigned

another random lifetime. In the experiments only node failure will be simulated, leave won't be considered. Leave will be included in the next version; as the code for leave in the current prototype needs further testing. To get the picture of churn level in the system, (failure, join) event pairs will be counted (as one failure event is immediately followed by a join event). It will be observed how much the level of churn affects system performance.

6.3 Experimental Setup

- Initially 100 nodes join the ring (default setting). Then 100 writes are carried out in a churn free environment to initialize the system with data. The node and data identifier space are configured to be of size 2^{13}.
- An environment is assumed for all experiments where load (read/write request) is generated from different nodes uniformly across the network and demand for data items is also assumed uniform (request initiating node and data identifiers are drawn from uniform distribution).
- All the experiments in Table 2 will be subject to two different design policies. In the first (default) policy, read/write requests are sent to all replicas; in the second, read requests are sent to only two random replicas in case of *read any*. As mentioned in [33] two random choices give a better load balancing. For *read critical* and *read latest*, requests are sent to majority. Second policy is expected to generate fewer messages, but may have less success ratio compared to the first. The following table summarizes the policies.

API	Policy 1 (Default)	Policy 2
Read Any	All	Random two
Read Critical	All	Majority
Read Latest	All	Majority
Write	All	All
Test&SetWrite	All	All

Table 3: Request Multicast Policies

- It has been observed that user lifetime distribution in existing peer-to-peer systems is often heavy-tailed [31]; that is most users spend short period of time inside the network while handful others stay much longer and exhibit server-like behavior. Shifted Pareto distribution, $X = \beta((1-x)^{-1/\alpha} - 1)$, which is heavy-tailed has been used in the churn model to draw node lifetime (X). Here, x is uniformly distributed random value between 0 and 1, β is the mean lifetime and shape parameter $\alpha = 2$.

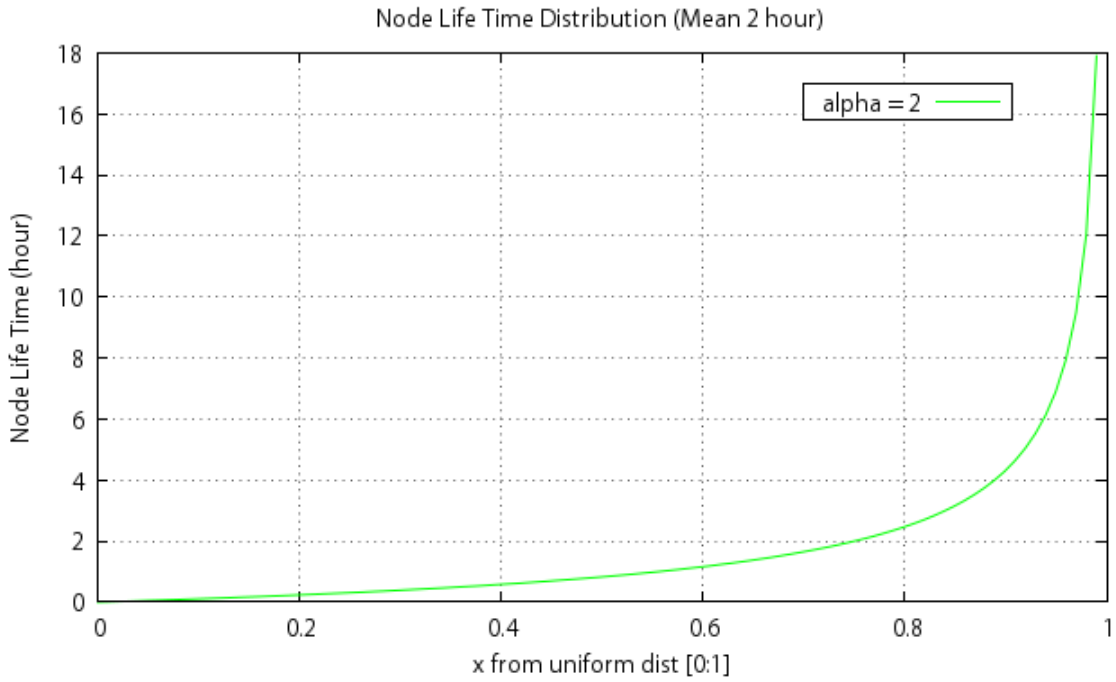


Figure 22: Pareto Distribution for Node Life Time

- Every experiment is set up to run for 24 hours (simulated time). Node joining time during bootstrap is not considered in this period. Each execution is repeated 12 times with different seeds and the results are averaged to get the final data with higher accuracy.
- In the experiments default replication degree is 5, default value for mean node life time is set to 2 hours and default read percentage is 60%. Read/write operations are generated from exponential distribution with default mean inter arrival time of 2 seconds.

6.4 Performance Results

Several experimental simulations are conducted varying system parameters as described in Table 2. The results of the experiments are presented in the following few sections.

6.4.1 Varying Churn

The peer-to-peer system is targeted to serve applications in a dynamic environment. In this experiment system is run with various mean node life time and its performance in terms of latency and success ratio is observed under both policies. As already described

in the life time based churn model, smaller mean life time implies higher level of churn. As mean node life time is increased churn level in the system gets lower.

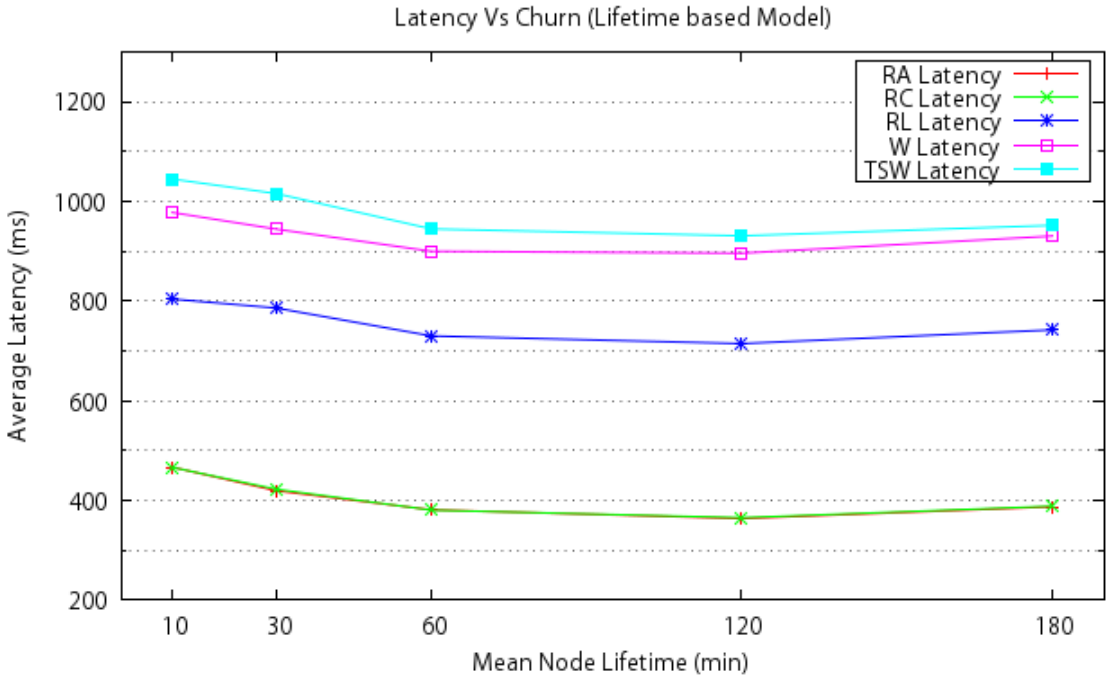
Figure 23(a) demonstrates the impact on latency under policy 1. As can be seen average latency for each API doesn't vary too much even though churn level in the system changes (latency appears higher during high churn). *Read any* and *read critical* operations perform much faster than *read latest* operations which is expected. *Read latest* shows more latency as it needs to wait for responses from majority, whereas *read any* and *read critical* return as soon as they get the data. Although *read critical* latency is expected to be higher than *read any* (as is the case for PNUTS) because of stricter requirement for certain version, both turn out identical under policy 1. The reason might be data is always written to majority of replicas and other replicas also get data without much delay. As request is broadcast to all replicas (policy 1) for both *read any* and *read critical*, it is likely the first reply comes from a node which has the required version (version doesn't matter for *read any*). So in terms of latency, these two APIs don't make much of a difference under policy 1. This is unlike PNUTS where data update propagates asynchronously to replicas from master. Therefore different replicas can have different versions at any given time. Which means *read critical* request may sometimes have to bear costly cross-region latency which is never the case for *read any* as data is served from local region.

Two writes in the system incorporate the highest latency (under policy 1) as they involve extra communication rounds. However *test and set write* displays little more latency than *write* during churn. This is due to the fact that *test and set write* needs to check data version in its operational workflow and until the requesting node gets majority of successful replies containing (any) version, it waits. Amidst churn it is possible some replicas may fail to return version number because of unavailability of data, therefore getting majority might take some time as successful replies are interspersed with failed responses. *Write* on the other hand doesn't depend on the availability of data and system just responds with 0 as version number in case data is not found (system considers it insert). Therefore *write* operations are expected to form majority set sooner and show lower latency than *test and set write* during churn.

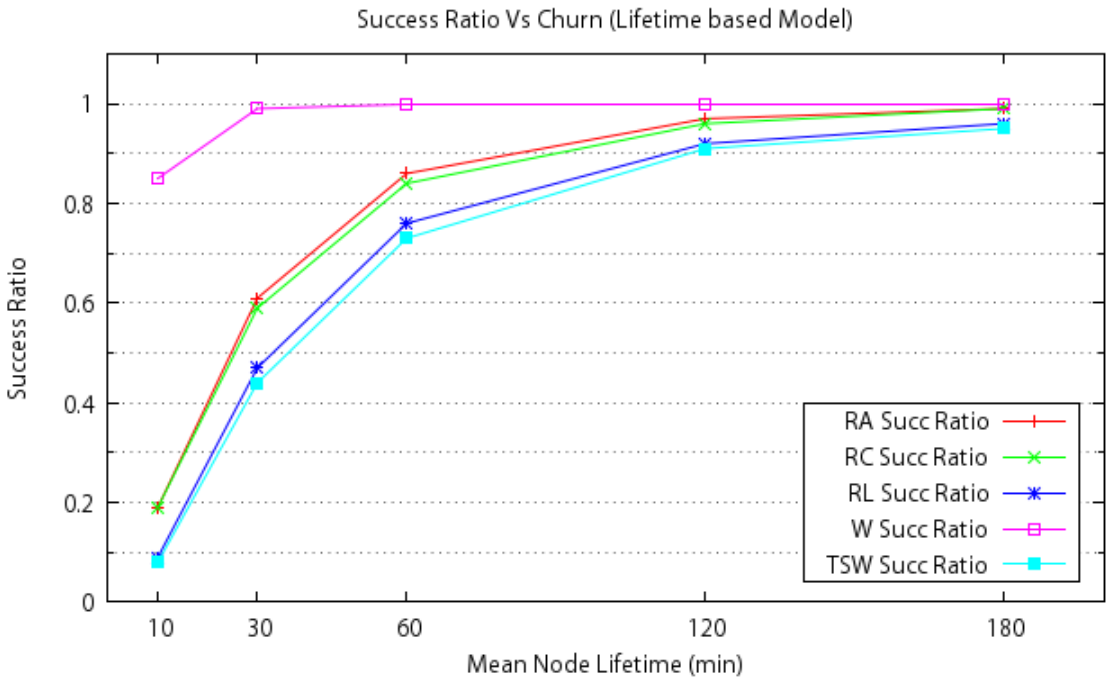
Figure 23(b) shows the effect of churn on operations' success ratio under policy 1. As can be seen, success ratio apart from *write* appears pretty low for small node lifetime (during high churn); but improves exponentially as mean lifetime increases (churn level decreases). It reaches above 90% for every operation when mean node lifetime is set 2 hours. As mentioned in Section 5.5, there could be several reasons behind operations' failure during churn. After analyzing the logs, the primary cause has been identified as the unavailability of data at expected node, when data is required during read or write operation. The scenario is explained in sufficient details in Section 6.4.3. Another major reason that contributes to operation failure is the error in lookup of replica id because of suspected nodes. This can happen when one of the participating nodes in the lookup process can't resolve the id (can't find the closest preceding node as relevant entries are suspected) or returns an already failed node due to its incorrect routing table; resulting in the whole lookup failure. As we know in the Chord protocol, stabilization of routing table occurs periodically where one entry is fixed at a time; so it takes time to get rid of incorrect entries. However when the churn is low, routing table eventually stabilizes and lookup starts to succeed which is not the case during high churn as routing table continues to have incorrect entries and operations fail in larger degree.

As shown in figure 23(b), *test and set write* and *read latest* display the lowest success ratio among all operations (in policy 1) as both of them require majority quorum to continue and this quorum can't be reached at times due to churn. *Read any* and *read critical* on the other hand only require a single successful reply, therefore entail better success ratio. Success ratio however is pretty high in case of *write* irrespective of churn level in the system. As mentioned above, one of the major reasons for reads and *test and set write* operations not being successful is the unavailability of required data at the anticipated node. For *write* operations, when data is not found at the expected node, system consider the write as insert (new write) and return 0 as version number instead of returning failure (see *write* component in Section 5.5). As a result, *write* always shows high success ratio despite churn. However its success ratio drops noticeably during high churn as operation failures in this case are mainly caused by lookup error due to suspected nodes. It is to be added the metric also includes 100 successful inserts

(performed in a churn-less environment to initialize the system with data); but compared to the number of *writes* issued during simulation that spans one day, this number is insignificant and shouldn't skew the statistics.



(a)



(b)

Figure 23: Effect of Churn under Policy 1

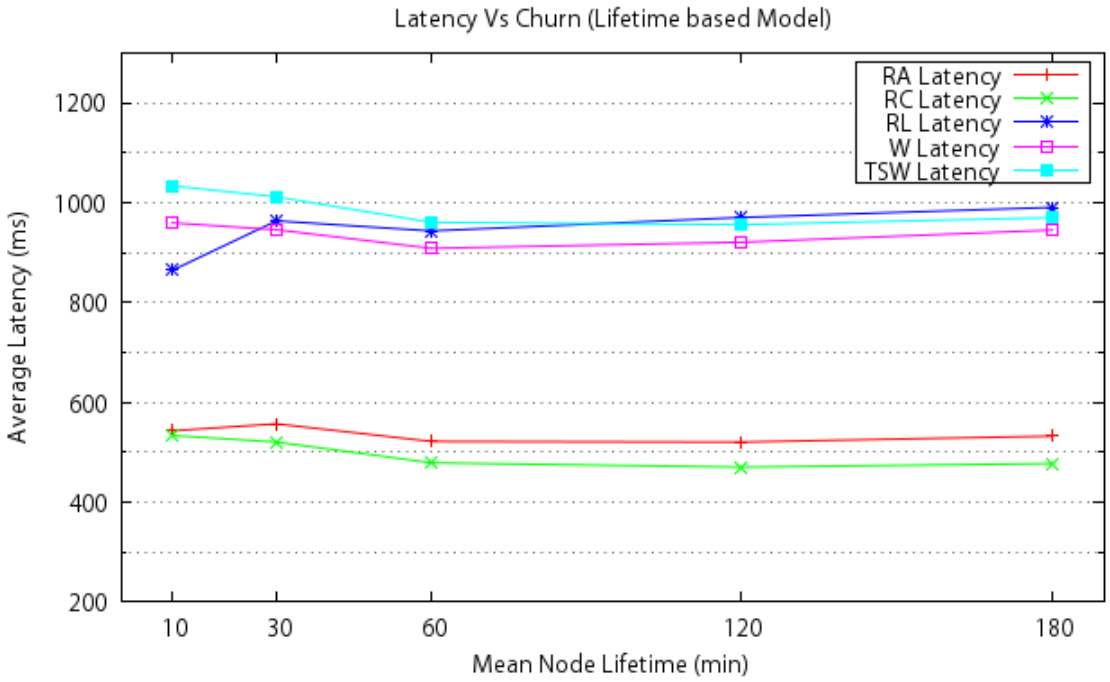
(a) Latency vs Churn (b) Success Ratio vs Churn

As shown in figure 24(a) the effect on latency under policy 2 is found to be similar to that of policy 1 in the sense that they don't vary much despite churn level in the system differs. However, read APIs in policy 2 show much higher latency compared to what is observed in policy 1. The reason could be read request is broadcast to all replicas under policy 1 which make it possible for the read operations to tap into the fast performing replicas (in terms of lower round trip time and/or faster processing at the node) and get an early successful reply (for *read any* and *read critical*) or get a majority comprising of faster nodes in case of *read latest*. This is not the case for policy 2, as read requests are sent to selective random replicas which may not be the fast performing ones.

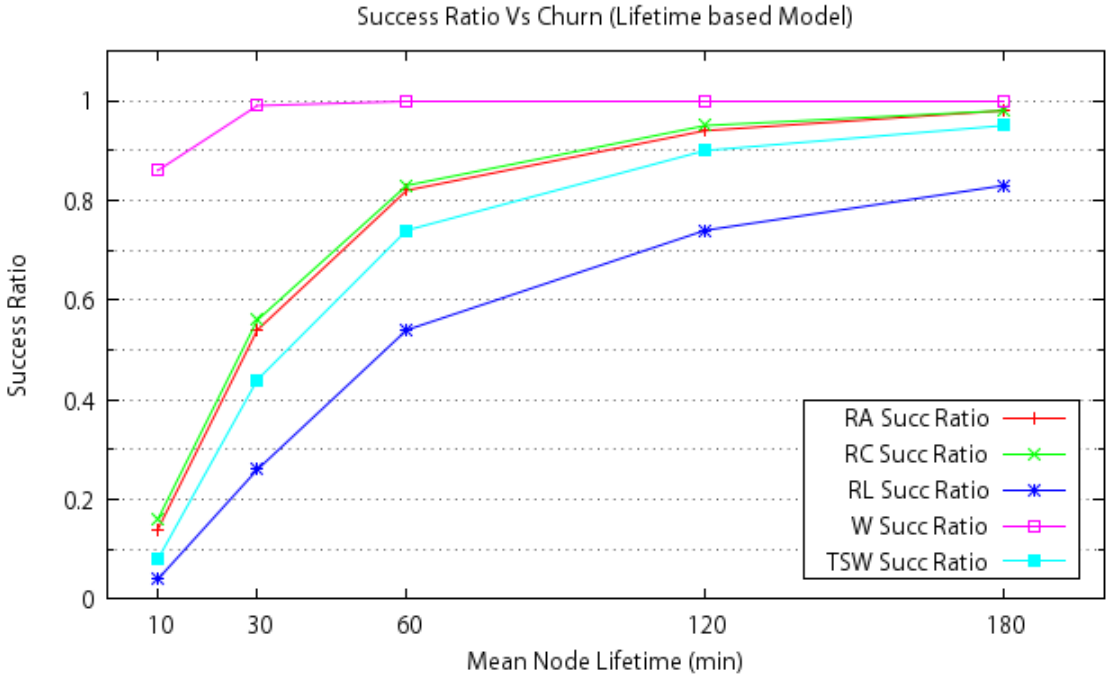
One thing noticeable in figure 24(a) is that *read any* displays higher latency than *read critical* under policy 2 (these two APIs have identical latency under policy 1). The reason could be *read any* selects only two random replicas as opposed to random majority for *read critical*; which means *read critical* has better probability to get a faster replica than *read any*. *Read latest* also shows equal or higher latency than the writes which is in stark contrast with the result of policy 1 where writes understandably have maximum latency. This is due to the fact that *write* and *test and set write* make use of the faster replicas as requests are broadcast to all. *Read latest* on the other hand sends requests to random majority some of which may not be the fast performing ones, then waits for each of their responses. This means operation's latency is all dependent on the delay caused by the last arriving reply. Therefore despite having fewer communication rounds than writes, *read latest* under policy 2 ends up with latency comparable to writes in the presence of slow performing replica(s).

Figure 24(b) shows the effect of churn on success ratio under policy 2. Result suggests that growth of success ratio is similar to that of policy 1, except for the *read latest* success ratio which is comparatively low. As already mentioned, *read latest* in policy 2 only sends request to random majority. If just one replica out of the majority doesn't yield successful response, which is very much possible during churn, the whole operation fails. So *read latest* under policy 2 seems to be pretty sensitive to churn, shows much worse performance (both latency and success ratio) and shouldn't be a design choice when there is churn in the system (*read latest* with policy 1 is much better design).

Another result that is visible here is that *read any* and *read critical* success ratio is slightly lower than those under policy 1 (around 5% which isn't insignificant considering the volume of requests) when churn is on the higher side. However the gap reduces gradually as churn level decreases. This is expected as *read any* and *read critical* read from fewer replicas in policy 2 which means possibility of operation failure during churn is more compared to policy 1 where operations read from all replicas. Finally *read any* success ratio is observed just a bit lower than *read critical* in policy 2 (as *read any* only uses random two whereas *read critical* reads from majority), which is not the case for policy 1.



(a)



(b)

Figure 24: Effect of Churn under Policy 2

(a) Latency vs Churn (b) Success Ratio vs Churn

Figure 25 illustrates the effect of churn on volume of messages in the system under both policies. As expected, policy 1 results in more messages in the system than policy 2 as requests are sent to more replicas. As can be seen, the amount of messages drops considerably as node lifetime increases (churn decreases). This is because during churn as nodes join or fail in the system, lot of messages related to data handover or data recovery are sent over the network (see Section 5.3). Therefore, more churn implies more messages in the network. One small 'anomaly' at the beginning of the curve can be seen where policy 2 involves more messages than policy 1. After investigating the logs it is found that policy 2 in that case experienced more churn (around few hundreds) than policy 1, therefore caused more messages. It is to be added that the total message count (accumulation of system specific messages over the length of the experiment run) isn't so much of an important indicator as it is the growth of messages with respect to churn; since message count would even give whole different value if system were configured differently (discussed in Section 6.2). Figure 26 confirms the fact that as nodes' mean lifetime increases, churn level in the system drops exponentially. Since policy doesn't play any part on the level of churn, churn under both policies gives almost identical curve.

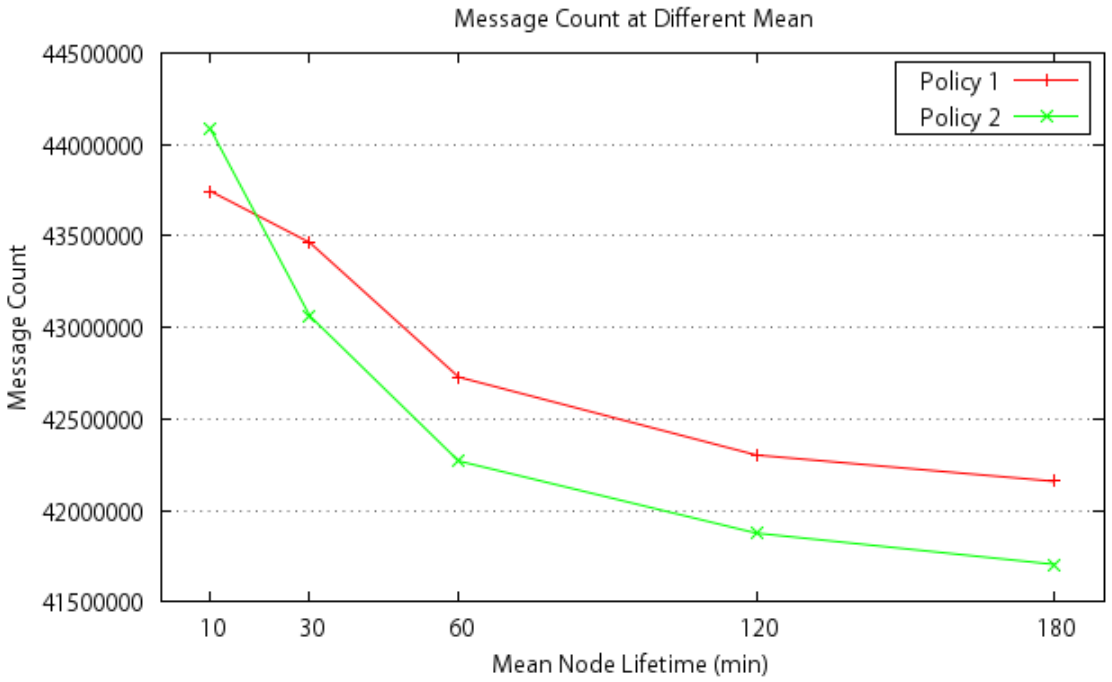


Figure 25: Message growth with respect to churn level under both policies

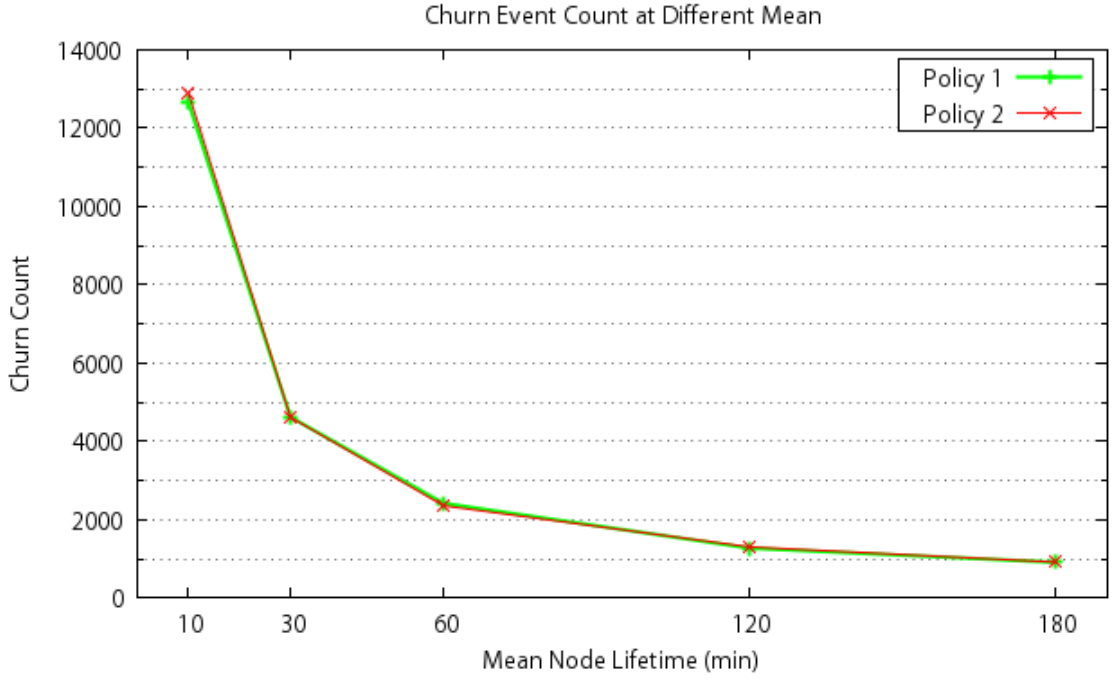
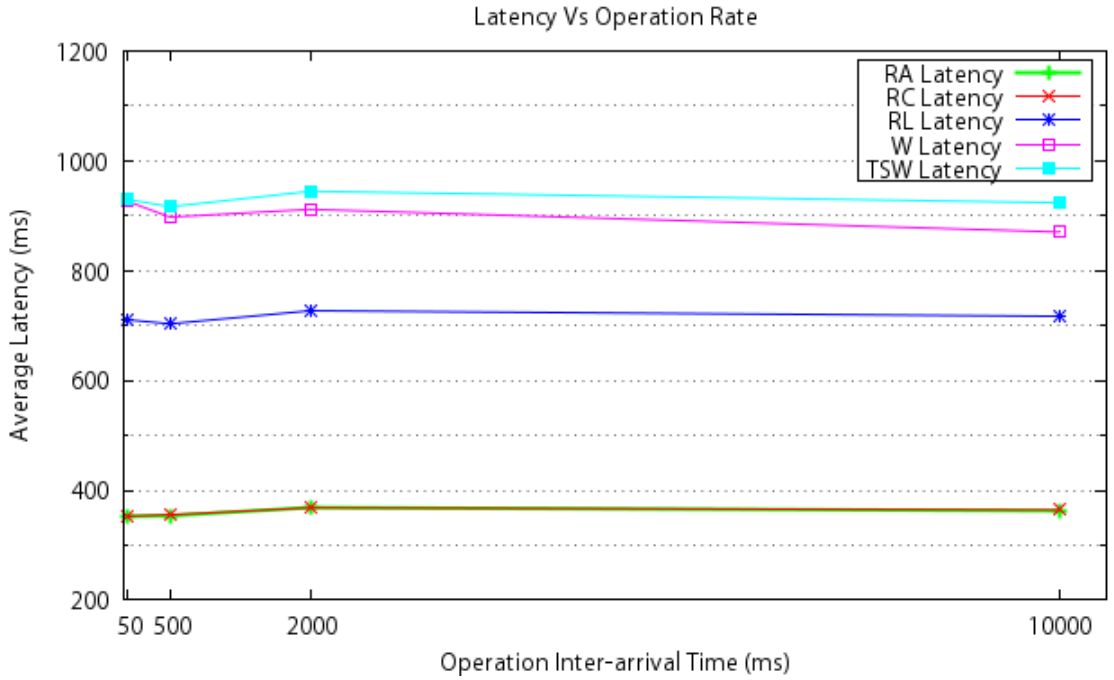


Figure 26: Churn level at different lifetime

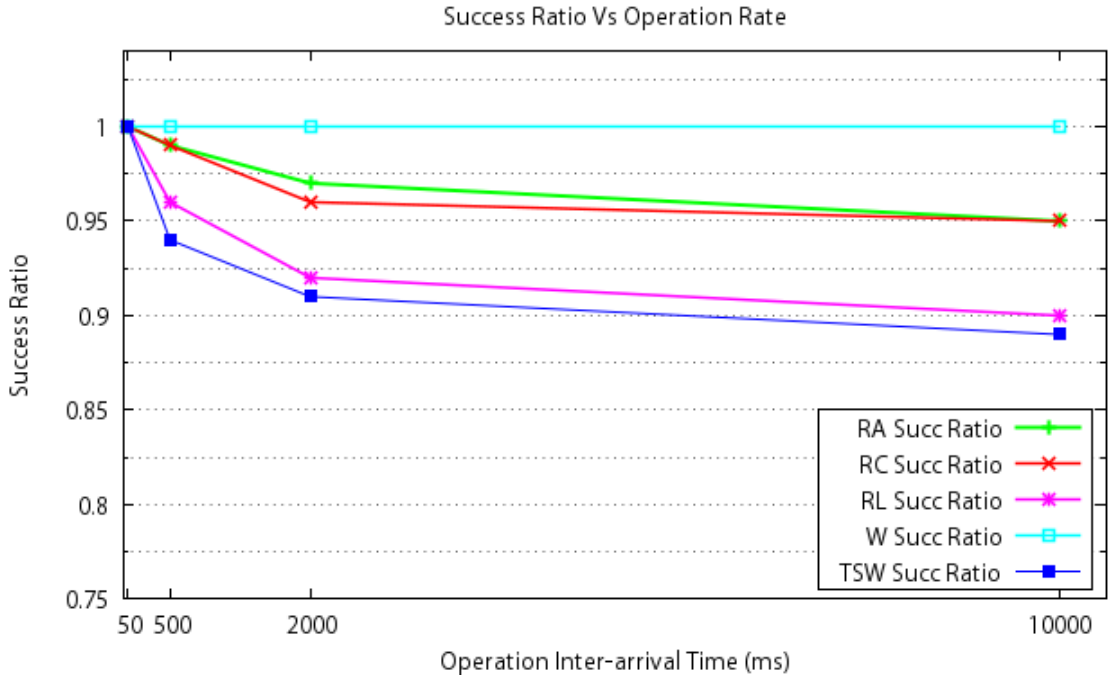
6.4.2 Varying Request Load

System can experience variable request load in a dynamic environment. The following experiment is conducted to observe how system performs under different load scenarios. Requests are generated using exponential distribution with a mean operation inter-arrival time. Rate of incoming requests (load) will be more for low mean inter-arrival time and it will decrease as inter-arrival time increases.

Figure 27(a) and 28(a) illustrate the effect of load on operations' latency under both policies. As can be seen, latency for each operation doesn't change even though load in the system varies; in other words system scales well despite increasing load. Read latencies however display higher value under policy 2 compared to policy 1, as discussed in Section 6.4.1. Load effect on operations' success ratio for both policies is shown in figure 27(b) and 28(b). Apart from *read latest* in policy 2 (explained in Section 6.4.1), all operations maintain success ratio of 90% or more at different loads. One interesting aspect of the result is that when system experiences maximum load (mean inter-arrival time of 50 ms), operations show their highest success ratio. This is due to the fact that frequency of writes is very high in this setting; as a result data is available most of the times reads are attempted. As mean inter-arrival time increases (load decreases), writes become less frequent and churn in the system causes some data not to be found at the expected node; which is one of the reasons mentioned in Section 6.4.1 for operation failure. As a result success ratio drops a little and steadies around 90% or more, which is acceptable. Success ratio for all operations can be improved even further by taking some measures (modifying some part of implementation) that will be described in the next section. The experimental result in terms of latency and success ratio suggests that under variable load, the system performs well and it is both scalable and available.



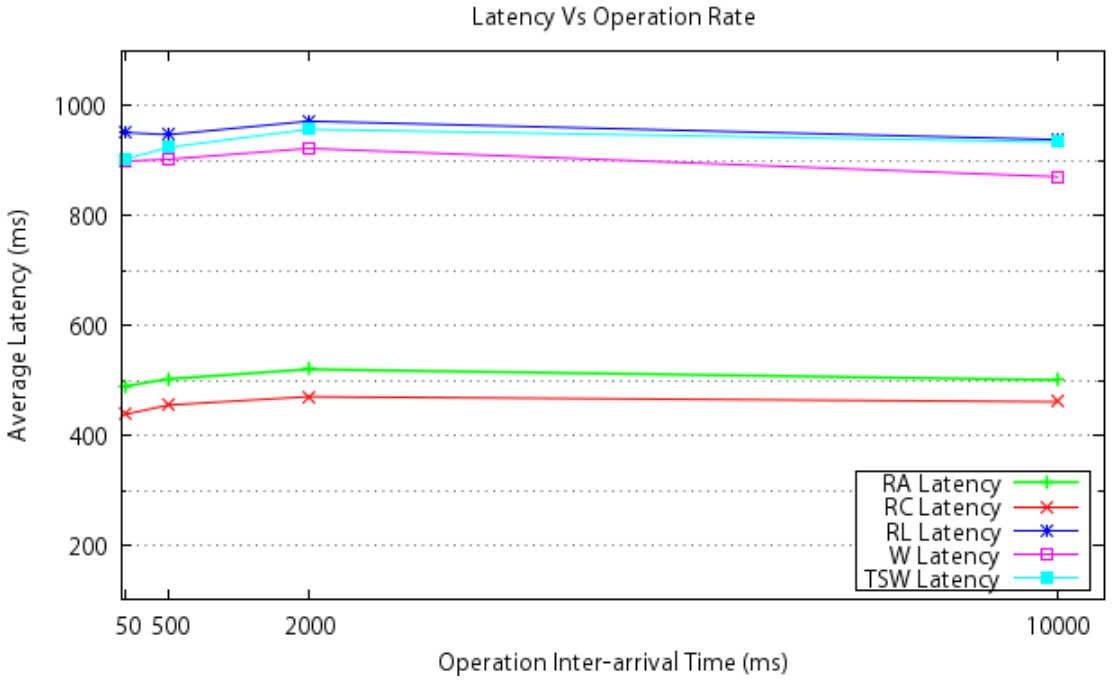
(a)



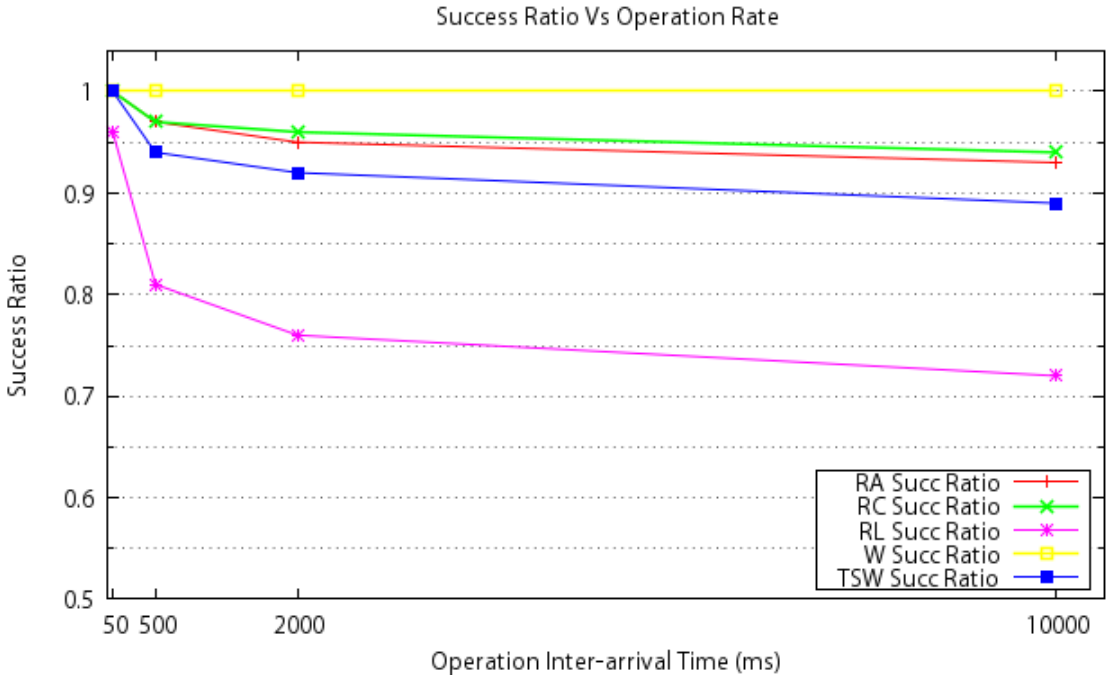
(b)

Figure 27: Effect of Request Load under Policy 1

(a) Latency vs Operation Rate (b) Success Ratio vs Operation Rate



(a)



(b)

Figure 28: Effect of Request Load under Policy 2

(a) Latency vs Operation Rate (b) Success Ratio vs Operation Rate

Figure 29 reflects on message level in the system under both policies during the experiment. As expected, system experiences a lot of messages when rate of incoming requests is high. Volume of messages is observed more for policy 1 than policy 2 because of policy design (explained in Section 6.4.1). However as request load decreases, number of messages drops and eventually settles at some level. The steady level of messages can be attributed to 'infrastructure' messages (see Section 6.2) that is present in the system even when there is no load.

Figure 29: Message count for two policies at different Operation Rate

6.4.3 Varying Network Size

In this experiment, system is run with different number of nodes and its performance in terms of latency and success ratio is observed for various network size. The point of the experiment is to investigate whether system scales when number of nodes in the network increases manifold. Figure 30(a) and 31(a) show the effect on operations' latency under both policies. As can be seen latency goes higher for all operations as network gets larger. However the increase is logarithmic and should be tolerable to many applications. The reason for this increase is that operations use underlying Chord layer to lookup keys and average cost of lookup is $\frac{1}{2}\log_2(N)$ [11] where N is the actual number of nodes (N

doesn't refer to the fixed ring size which is 2^{13} in the current setup) uniformly distributed in the network. This cost adds up to the latency cost and pushes it higher as network size scales up. Figure 30(b) and 31(b) illustrate how growth in network size affects operations' success ratio under both policies. As can be seen success ratio of all operations other than *write* decreases with the increase in number of nodes, which is not really ideal. The dip is even more for policy 2 as operations read from fewer nodes. To explain the scenario, relevant logs are analyzed from which the following information (Table 4) is extracted.

Network Size	Failed/Detected/ Missed	Missed %
100	1292/1281/11	0.85%
500	7031/6978/53	0.75%
1000	14181/14079/102	0.71%
1500	21209/21033/176	0.82%

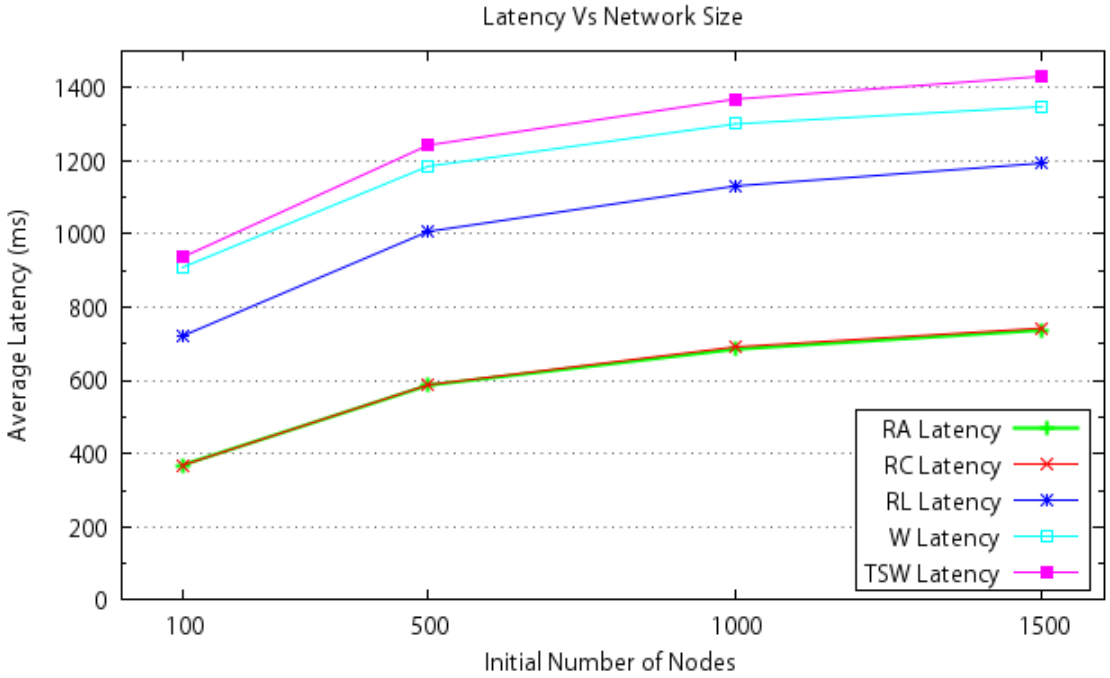
Table 4: Failure detection error

As already mentioned churn model that is used in the prototype is lifetime based and as the number of nodes grows, churn i.e. node failure in the system increases as well (see Figure 32(b)). After analyzing the logs it is found that some of the failures (very few but not negligible) are not detected at all and as Table 4 suggests number of misses (not detecting failures) goes up for larger network because node fails more in that setting. As a result data recovery is not attempted in those cases by successor nodes and data associated with those failures are left with one less replica. This loss of data has a negative impact further down the road on operations' success ratio in a sense that data won't be found when required and success ratio might come down as system continues to run, which is clearly evident in figure 30(b) and 31(b). It is worth noting that if the system were to run for longer period than one day (current setup); more failures would

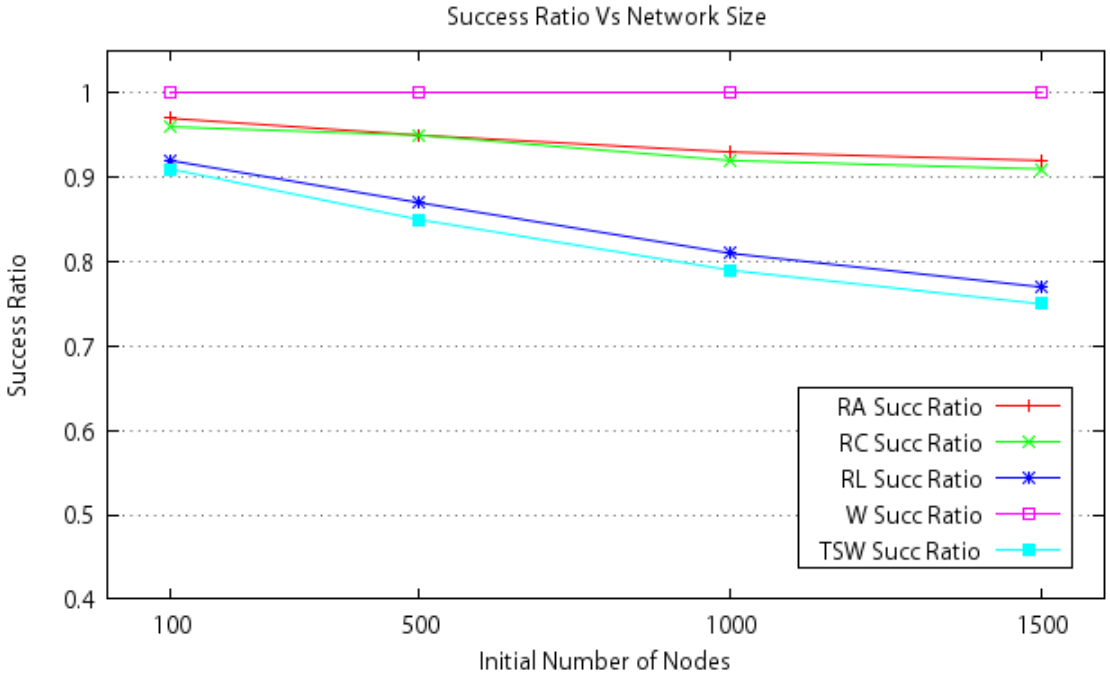
occur, very few of which might go undetected contributing to further data loss and lesser success ratio.

However this problem can be remedied by fixing how system detects failure in the current implementation. Successor node is in charge of detecting failure of predecessor in the current version and as is evident, it sometimes misses failure of its predecessor (one of the reasons could be concurrent join and failure in the same area of the ring which causes the successor to miss failure of its predecessor). The chance of failure detection can be improved further if every node is conferred with the responsibility of detecting failure of both its immediate neighbors (both predecessor and successor, instead of just predecessor in the current setup). In other words instead of being watched over by one node, a node would now be watched over by two nodes on either side of it; both can detect its failure and notify each other. That way even if successor node somehow fails to detect the failure of its predecessor, another node on the other side of the failed node might be able to pick up its failure and notify the successor. Successor then would act on it and recover data from other replicas in the system. Operation failures caused by data loss in the system can be reduced this way and success ratio could be improved further; although the trade-off here is more messages in the system. However messages would increase linearly with the increase in number of nodes just as figure 32(b) demonstrates and wouldn't be too much of a problem. It is to be added that success ratio of all other experiments listed in the chapter can be bettered if the probabilistic measure (stated here) is taken; since many operation failures can be traced back to the particular problem of data not being found at expected nodes and improving on failure detection with a view to reducing data loss can contribute to lower these scenarios.

Summing up the experiment, system is scalable in terms of latency as latency growth is observed logarithmic, but success ratio slightly drops with the increase in network size. However the situation can be improved by modifying the failure detection with a probabilistic measure which is left for future work.



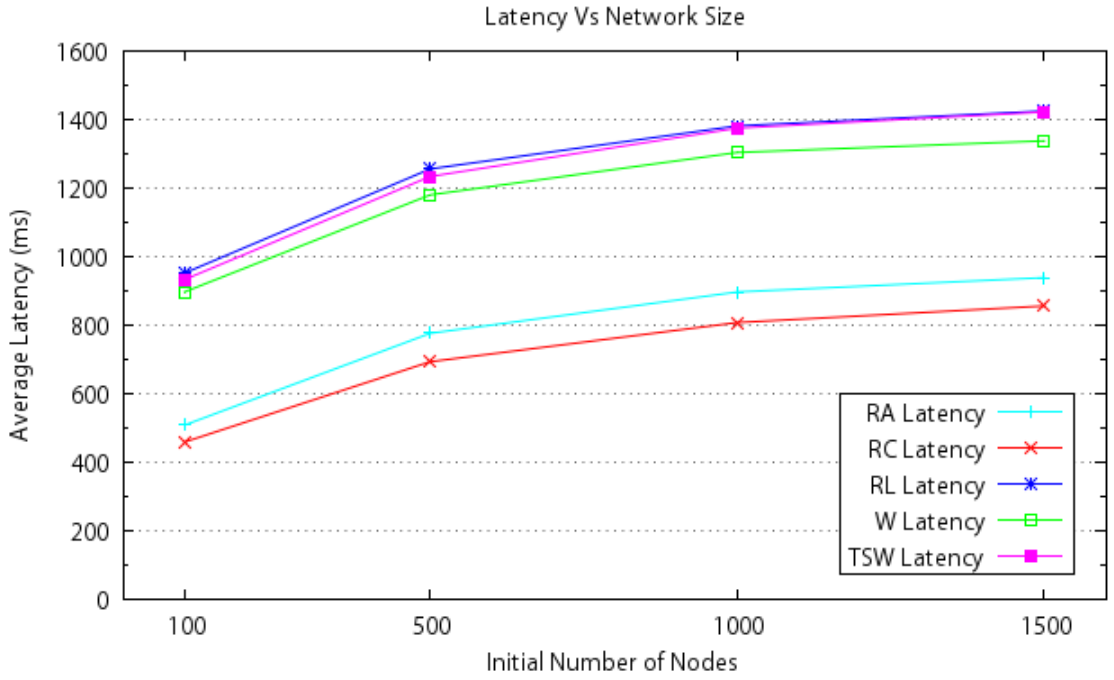
(a)



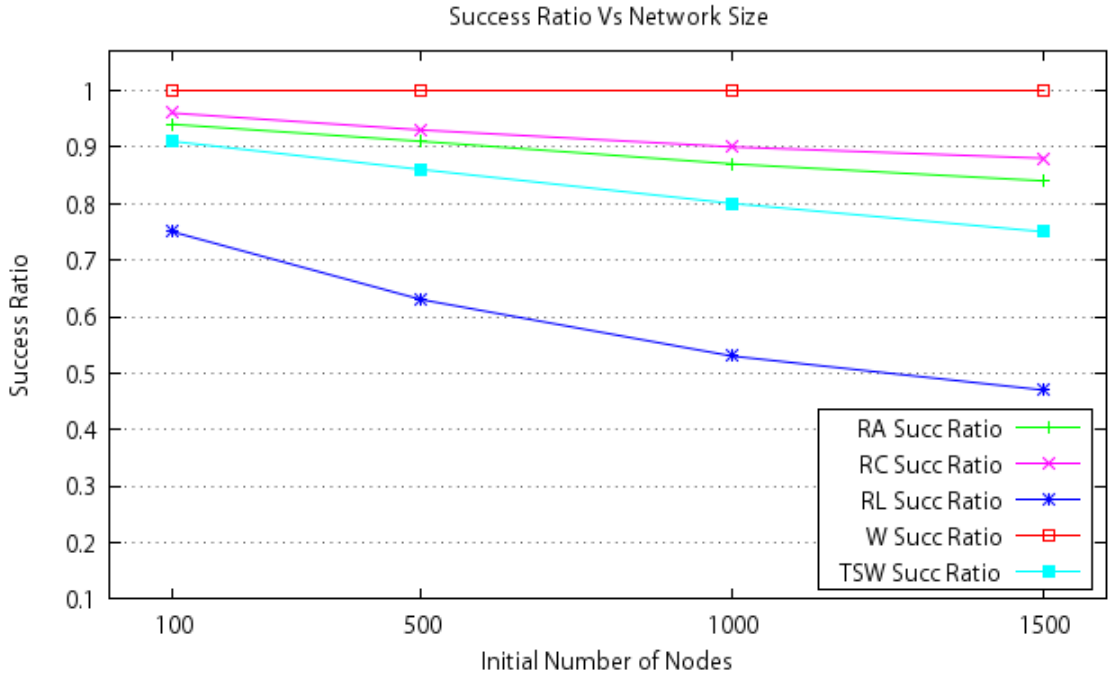
(b)

Figure 30: Effect of Network Size under Policy 1

(a) Latency vs Network Size (b) Success Ratio vs Network Size



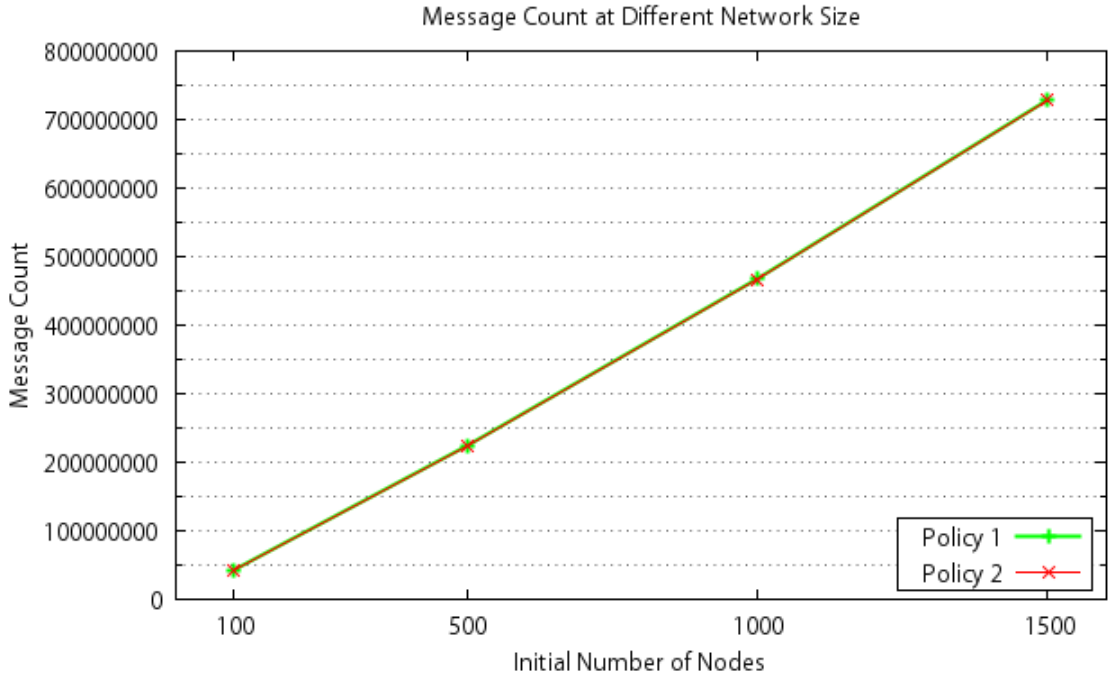
(a)



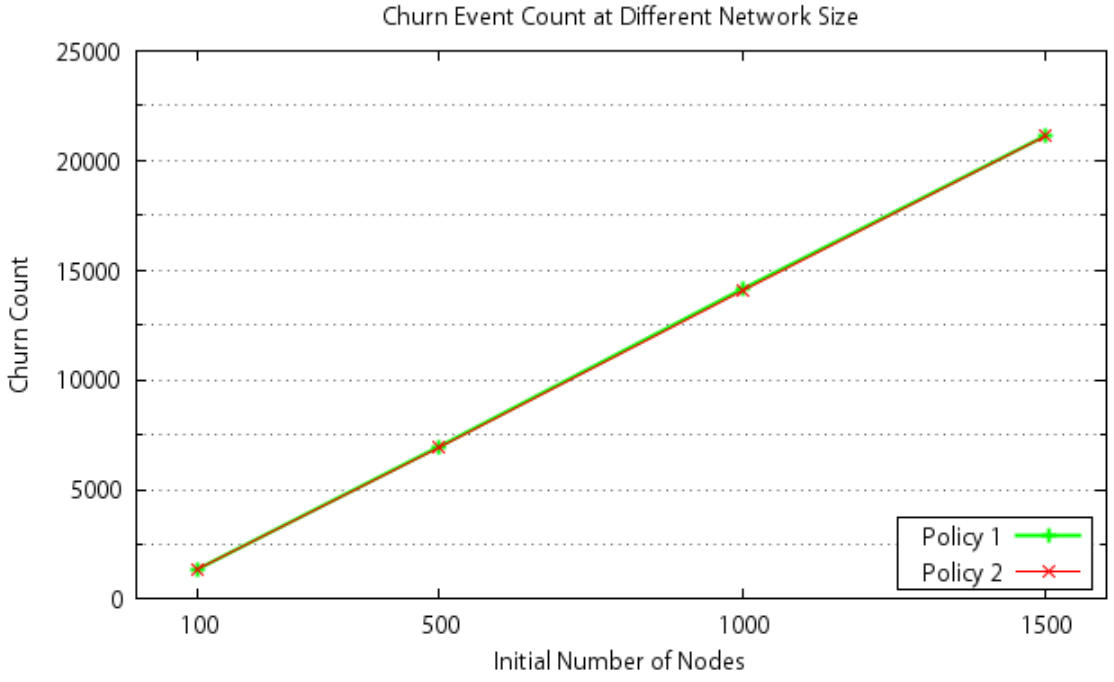
(b)

Figure 31: Effect of Network Size under Policy 2

(a) Latency vs Network Size (b) Success Ratio vs Network Size



(a)



(b)

Figure 32: Message and Churn observation for different network sizes under both policies

(a) Message count (b) Churn count

6.4.4 Varying Replication Degree

Replication of data is required to improve fault tolerance, availability as well as scalability of the system. In this experiment system is run with different replication degree and its effect on system performance is observed. Figure 33(a) illustrates the effect on operations' latency under policy 1. *Read any* and *read critical* latency is highest when there is no replication (single copy), but decreases appreciably as more replicas are added to the system. Because both these operations return after receiving the first successful reply, no replication means request initiating node has to wait for the only reply and must accept whatever latency that comes with it. More replicas increase the probability to have faster nodes in the pool from which it can receive the quickest reply and reduce latency (all replicas receive read request in policy 1). For *read latest*, *write* and *test and set write* operations latency doesn't improve much; rather they get little slower as replication degree increases. This is because in all three operations requesting node has to wait for majority of responses; as the number of replicas grows, majority increases as well resulting in more waiting time. For policy 2, replication effect on latency is shown in figure 34(a). Result suggests that latency varies the same way as policy 1 when replication degree changes. However due to policy design, latency of all reads show higher value compared to policy 1, especially noticeable is the *read latest* latency which is even higher than the writes (explained in Section 6.4.1).

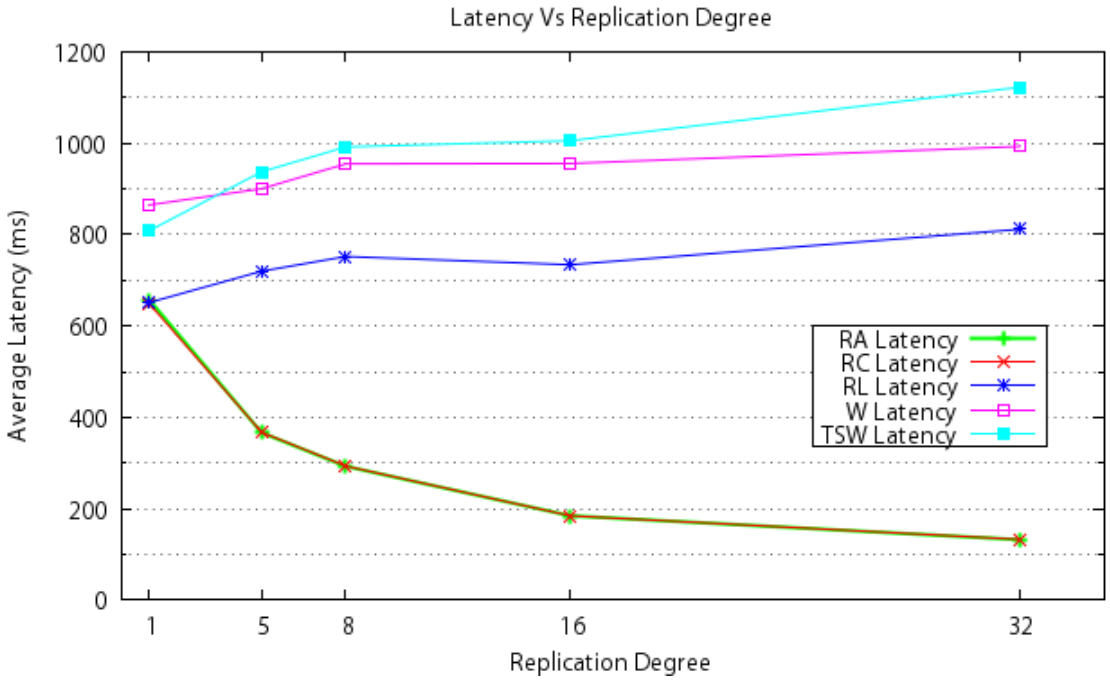
Operations' success ratio under both policies is presented in figure 33(b) and 34(b). Apart from *read latest* under policy 2, replication effect on success ratio is observed to be similar for both polices. The metric gives poor result when replication degree is one (single copy), but steadies around 90% or more when system is configured to have multiple replicas. *Write* operation always maintains high success ratio irrespective of replication degree in the system as it doesn't require availability of data (explained in Section 6.4.1). The poor success ratio of other operations in case of single replica is due to the fact that once a node fails its data can't be recovered, since there are no other copies available in the system. As nodes keep on failing because of churn, loss of data occurs without recovery and operations apart from *write* fail in large percentage as required data can't be found at the expected node. But success ratio (barring *read latest* in

policy 2) continues to get better and reaches close to 100% as replication degree increases which is expected since system can increasingly tap more replicas to get data. However there is a slight drop in success ratio when replication degree goes from 16 to 32. The reason could be, for a network size of 100 nodes (default setup) the replication degree of 32 is little too many. If one node fails or gets suspected in this setup, lot of data items are affected since too many replicas of different data items are hosted in each node. As a result operations (especially *read latest* in policy 2 that requires all requests to be replied with successful response) suffer from unavailability of data and success ratio gets lower when replication degree crosses a certain 'threshold'. To confirm the point another experiment is run (under policy 1) by increasing the network size to 200 and keeping replication degree of 32. It turns out success ratio improves and gets close to 100%. The following table gives a summary of the experiment.

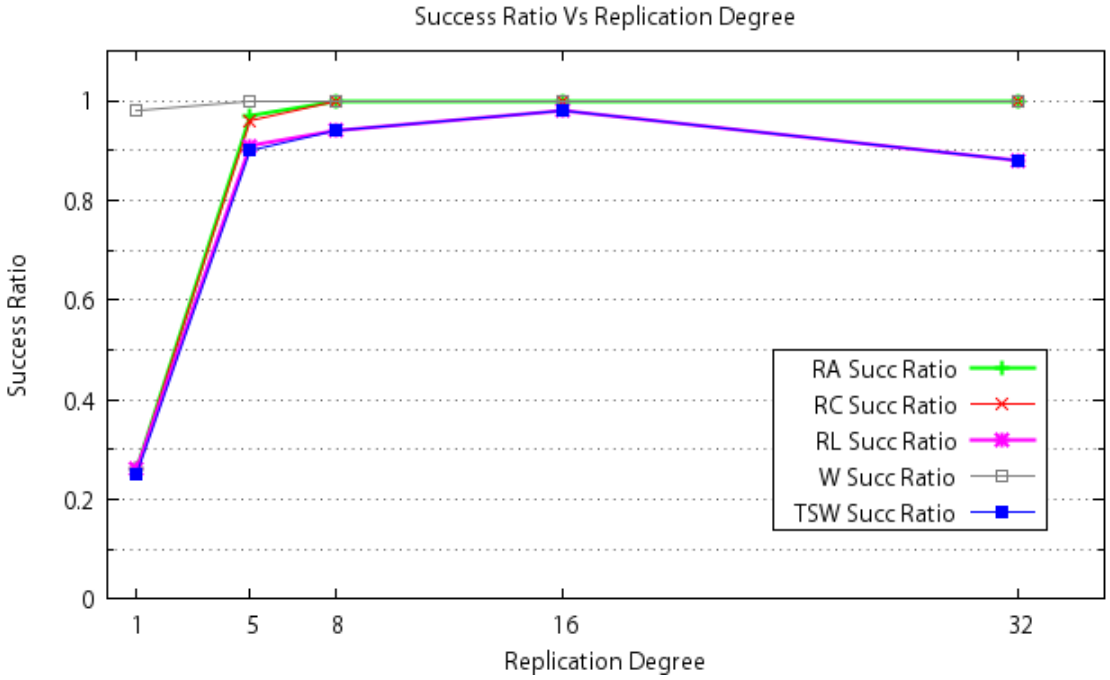
API	**Success Ratio**	
	Network Size (100)	**Network size (200)**
Read Any	1.0	1.0
Read Critical	1.0	1.0
Read Latest	0.88	0.99
Write	1.0	1.0
Test and Set Write	0.88	0.99

Table 5: Success Ratio Observation for two network sizes
(Replication Degree 32, Policy 1)

Figure 35 demonstrates the effect of replication degree on message volume of the system. As expected messages increase linearly with the increase in replication degree since operations have to interact with more replicas. Policy 1 also incorporate more messages than policy 2. Finally the result of the experiment suggests that increasing of replication degree improves system performance; however there seems to be a certain 'threshold' crossing which will deteriorate performance. Finding out the limit for different network size in a methodical way is left for future work.



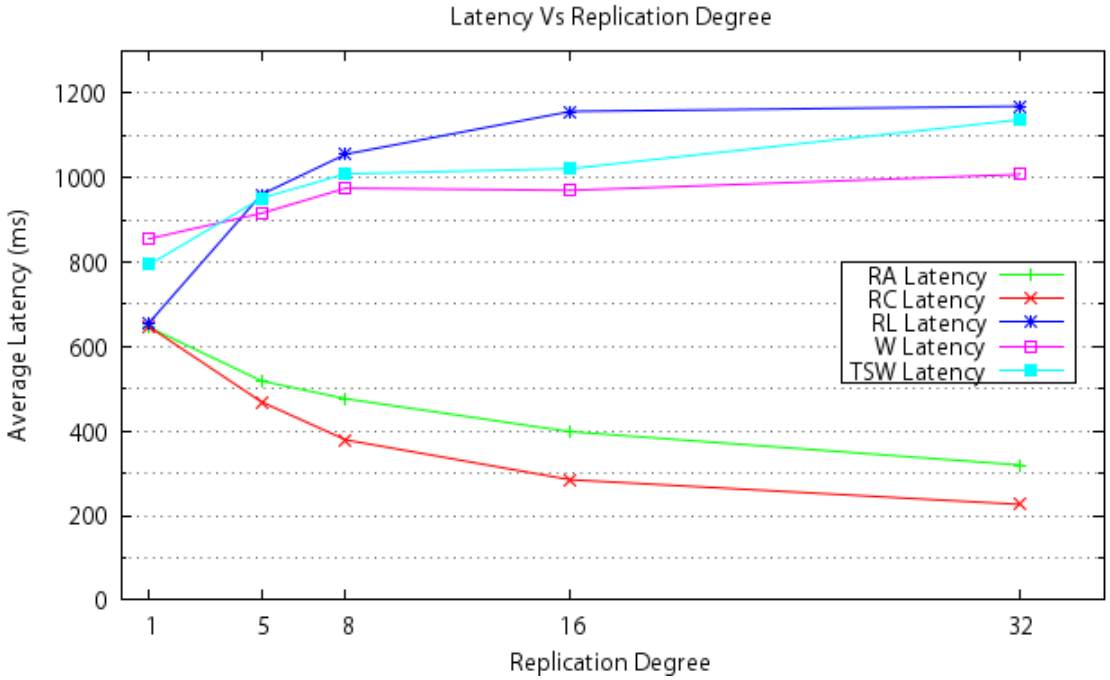
(a)



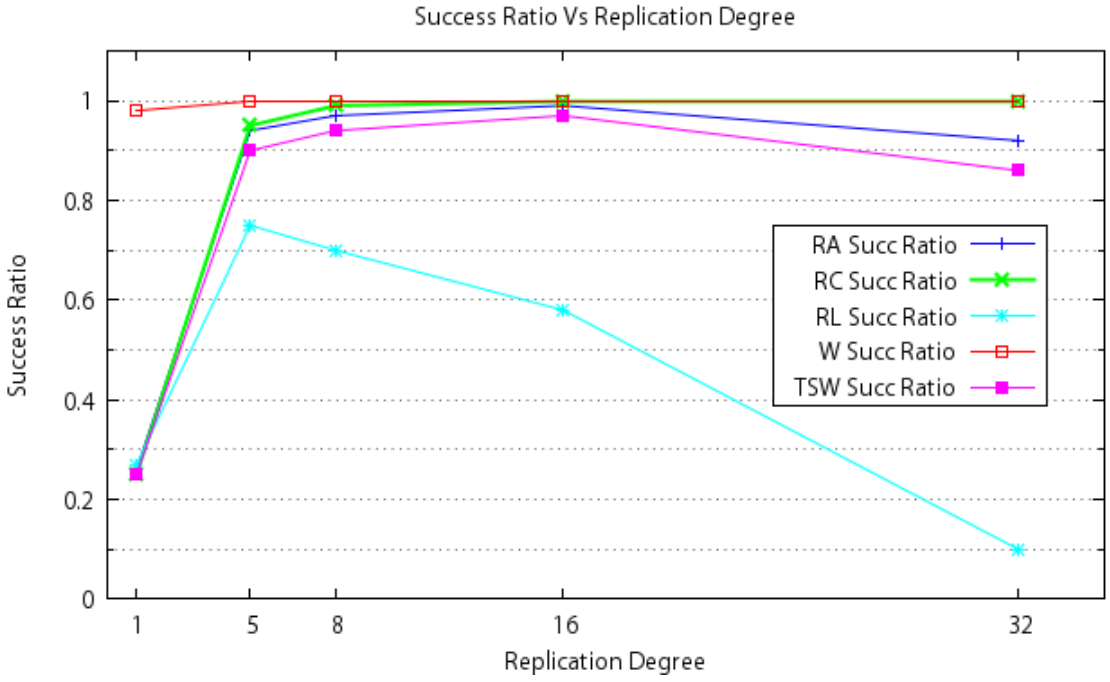
(b)

Figure 33: Effect of replication degree under Policy 1

(a) Latency vs Replication Degree (b) Success Ratio vs Replication Degree



(a)



(b)

Figure 34: Effect of replication degree under Policy 2

(a) Latency vs Replication Degree (b) Success Ratio vs Replication Degree

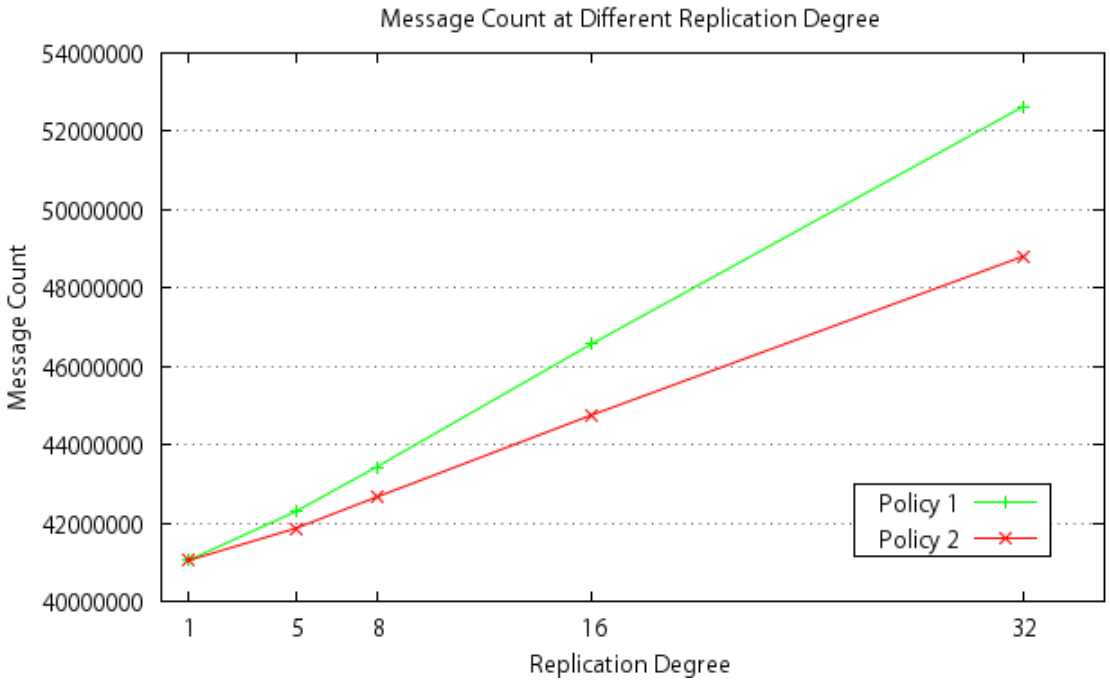
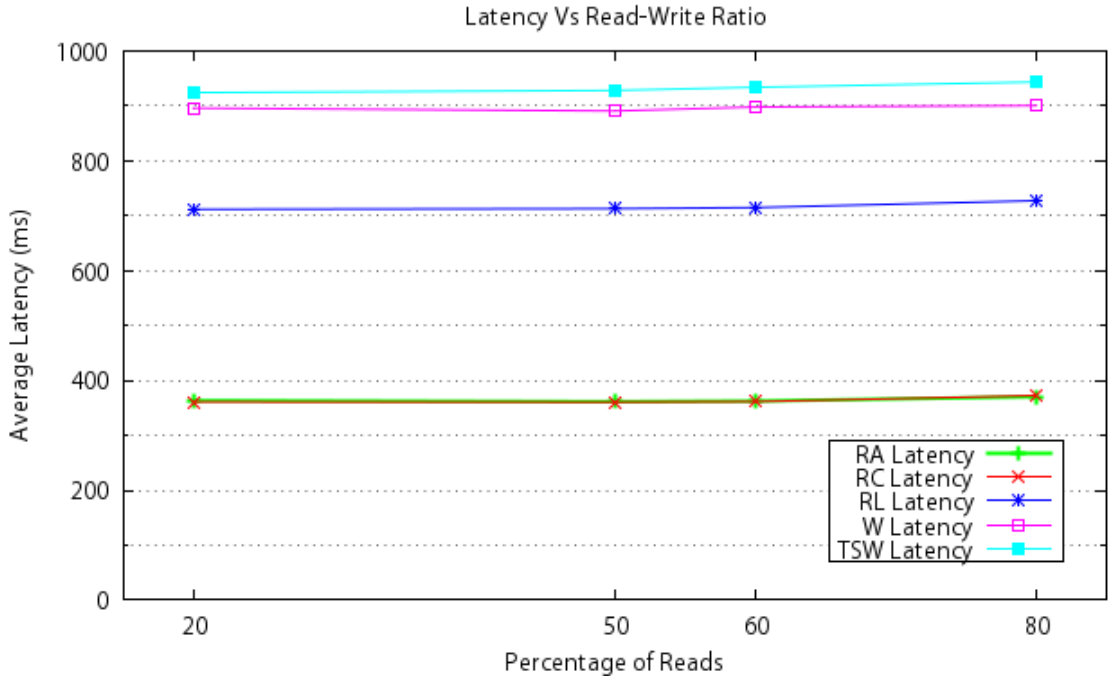


Figure 35: Replication Effect on message volume under both policies

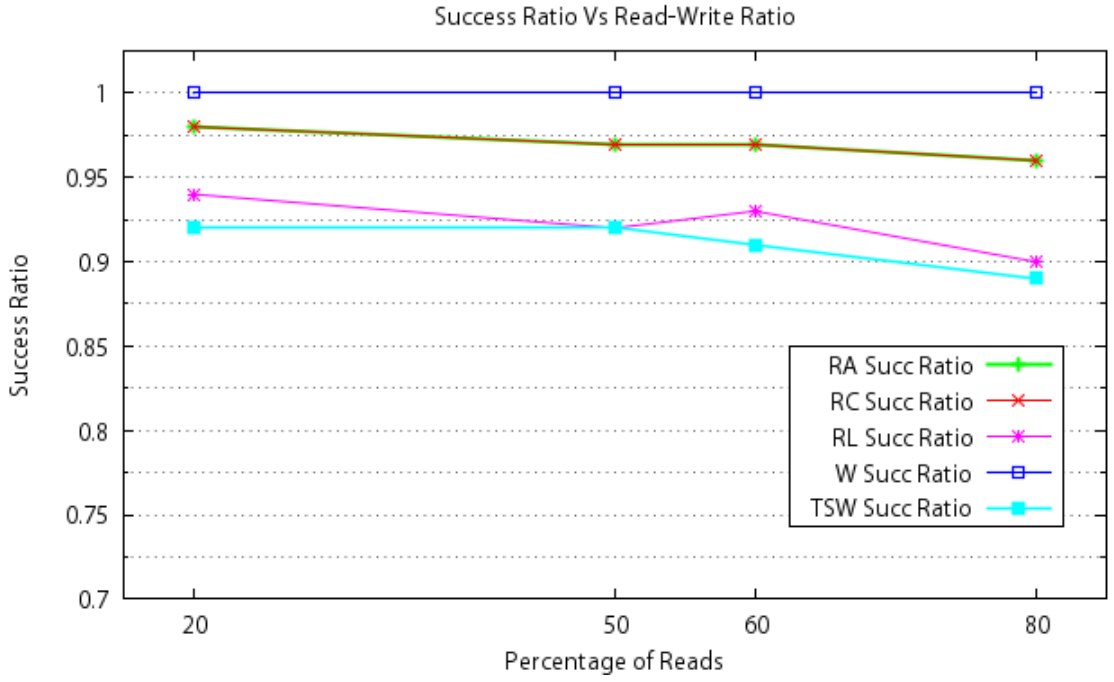
6.4.5 Varying Read-Write Ratio

System can be read-oriented (more read than write) or write-oriented depending on applications' usage. The experiment investigates whether the system favors one usage over another; in other words if the system performs any differently when it has more reads than writes as opposed to having more writes than reads.

Figure 36(a) and 37(a) show that latency for each API doesn't change even if read-write ratio is varied in the system under both policies. This confirms that it doesn't matter if the system has more reads or writes, the operations don't really affect each others' performance when it comes to latency. Success ratio also remains steady over 90% for all operations and doesn't vary too much under both policies, except for *read latest* in policy 2 which steadily hovers around 75% (low success ratio has been explained in Section 6.4.1); this can be seen in figure 36(b) and 37(b). One thing to notice is that success ratio dips only slightly when read-percentage goes higher (write-percentage lower). This could be due to the fact that the more frequent the write, the less possibility there is of data not being found at expected node which can contribute to an operation failure during churn.



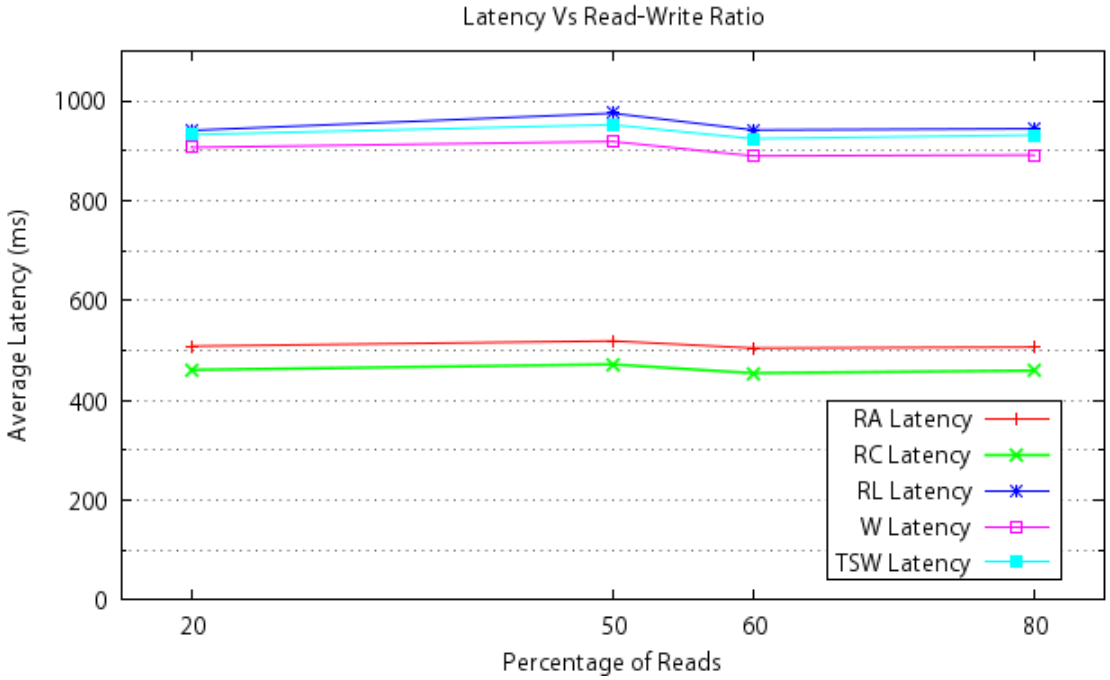
(a)



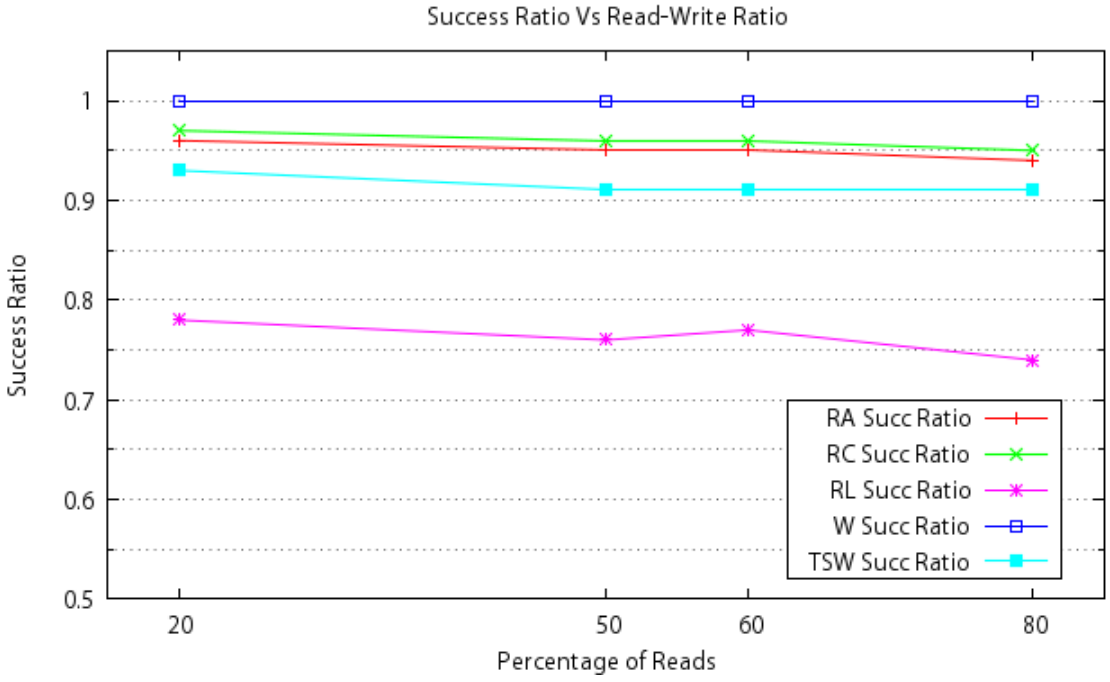
(b)

Figure 36: Effect of read-write ratio change under Policy 1

(a) Latency vs Read-Write Ratio (b) Success Ratio vs Read-Write Ratio



(a)



(b)

Figure 37: Effect of read-write ratio change under Policy 2

(a) Latency vs Read-Write Ratio (b) Success Ratio vs Read-Write Ratio

Figure 38 demonstrates the effect of the experiment on volume of messages in the system. As can be seen, policy 1 causes more messages in the system than policy 2 which is expected. It is also observed that number of messages decreases with the increase in read percentage. This is because write operations entail more communication rounds than reads; therefore more write means more messages in the network.

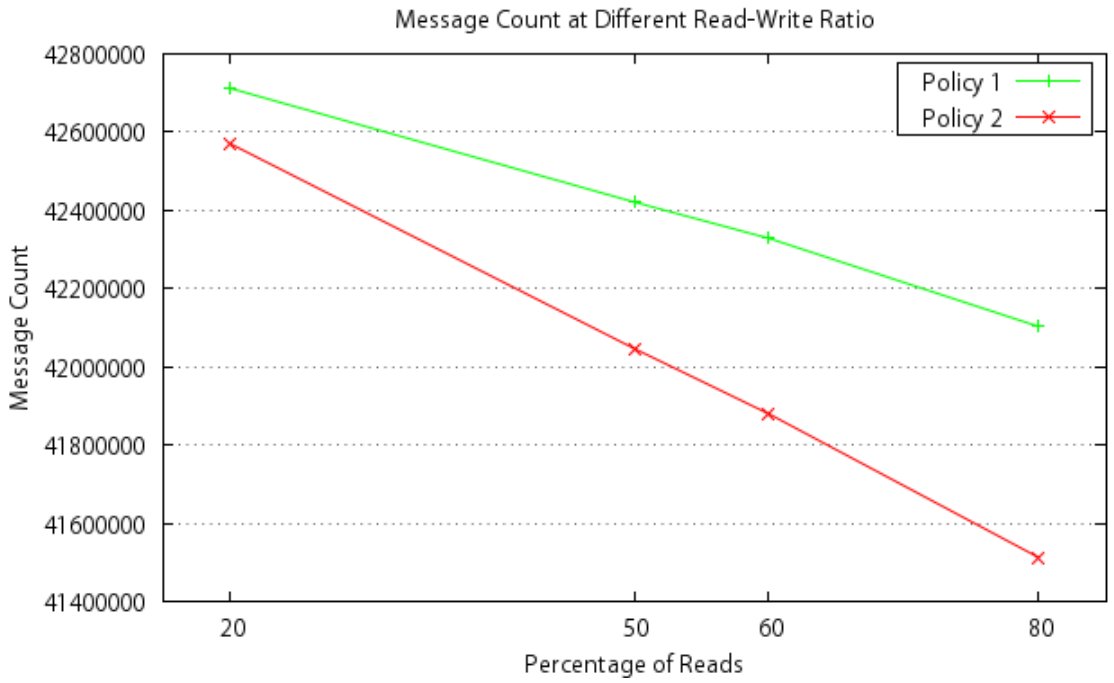


Figure 38: Effect of ratio change on message volume under both policies

7. Conclusions

The thesis presents a memory based key-value data storage prototype intended to be deployed on peer-to-peer infrastructure. It exposes a range of read and write APIs that guarantee various degrees of data consistency. The system performs well in terms of latency and success ratio of its operations despite churn in the system; as simulation suggests latency isn't affected and operations' success ratio gets even better as churn level in the system decreases. The performance of the system turns out acceptable as well during high request load. The system is also scalable as latency scales logarithmically and success ratio remains high even though number of nodes increases manifold. However as discussed in Section 6.4.3 success ratio for all operations can be improved even further if the failure detection problem (in which very few, less than 1%, failures go undetected resulting in data not recovered and replication degree being reduced) can be addressed. A solution is proposed in the section to counter the problem which could be applied in the next version of the prototype. The system also tolerates replica failure as long as majority of replicas remain alive at any given time. However data is recovered following a failure detection and system continues to maintain replication degree (apart from a small percentage already mentioned) which is evident in the high success ratio of operations despite churn in the system. The storage system is therefore robust, fault-tolerant, scalable and can work in a dynamic environment with higher degree of availability.

The storage also provides stronger data consistency guarantee using majority based quorum technique than what could be achieved in DHTs [12]. As writing is done to majority of replicas, reading from majority can ensure latest data being read. However as mentioned in [14], lookup inconsistency can potentially produce disjoint or partitioned write sets (exceeding replication degree) which means two writes won't be overlapped in such scenario resulting in data inconsistency. But, it is also shown in [14] that the probability to have such inconsistent data is very very low and the probability that data consistency would be guaranteed is more than 99%. This guarantee would be well acceptable to Web 2.0 applications that can tolerate relaxed consistency.

Two request multicast policies (see Table 3) have been tested in the prototype to investigate the trade-off between performance and availability on one hand and message volume in the system on the other. As expected, Policy 1 inspires better performance as more replicas are tapped but costs more messages in the system; whereas Policy 2 results in less volume of messages with reduced performance. It is up to the applications which one to choose (or even combination of two could be tried) as API specific results are available concerning two policies. However from simulation results, it appears Policy 2 is not suitable for *Read Latest* API since it turns out very sensitive to churn and performance is observed unacceptable.

7.1 Comparison with PNUTS

- PNUTS data store operates on a controlled and stable environment comprising of a handful of datacenters distributed in several geographic regions and focuses on cross-datacenter replication. The peer-to-peer prototype on the other hand can work with a massive amount of nodes and can tolerate high degree of churn. PNUTS would beat the peer-to-peer storage in latency if both operate in stable environment. Peer-to-peer system suffers from higher lookup cost compared to a system with few datacenters and simpler routing. However in a dynamic environment where nodes can join or fail or leave at relatively high rate, peer-to-peer storage would perform well.
- Since PNUTS uses master based approach for data consistency and update ordering, in case a data center in charge a region fails as part of a major outage, all the master replicas residing in the region, the amount of which is expected to be huge, would become unreachable. Such scenario would cause massive amount of read and write requests to fail. This will not be the case for peer-to-peer system which is completely distributed and doesn't depend on any specific replica as there is no master replica concept.
- PNUTS heavily depends on a separate pub-sub service, Yahoo! Message Broker (YMB) for replication and recovery. In such a scenario where YMB fails, the system would cease to work (only *read any* would work). Peer-to-peer system

doesn't rely on any external system and therefore not vulnerable to any component's failure outside the system.

7.2 Future Work

- As the storage system will be deployed on peer-to-peer infrastructure, there might be a need to ensure data security and protect personal information using cryptographic means. In future such mechanisms would be evaluated.
- In the current prototype only node failure in the context of churn is implemented; leave wasn't considered as code for handling leave needs further testing. In the next version of the prototype, leave (graceful node exit) will be included.
- The solution proposed in Section 6.4.3 to improve on failure detection which in turn would increase system performance and operation success ratio would be incorporated in the next version.
- The storage system is designed and developed for Web 2.0 applications that can tolerate relaxed consistency. The majority based quorum technique gives probabilistic guarantee [14] which is enough for such applications. However in future stronger (atomic) consistency guaranteed by Paxos systems [13] could be investigated whether they are relevant to the system requirements and what are the performance implications. But Paxos systems generally limit scalability and performance as they incorporate extra communication rounds to ensure consensus among replicas.

References

[1]. Times. http://www.time.com/time/magazine/article/0,9171,1569514,00.html

[2]. World Bank. http://blogs.worldbank.org/publicsphere/how-scalable-web-20-changing-world-disaster-management

[3]. Gilbert, S., and Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News 33(2)*.

[4]. Vogels., W. (2008). Eventually Consistent. *ACM Queue, 6(6)*.

[5]. Bernstein, P., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

[6]. Helland., P. (2007). Life beyond distributed transactions: an apostate's opinion. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR '07)*. CA, USA.

[7]. Cooper, B. F. et al. (2008). PNUTS: Yahoo!'s Hosted Data Serving Platform. In *Proceedings of the Thirty-Fourth International Conference on Very Large Data Bases (VLDB '08)*, pp. 1277-1288. Auckland, New Zealand.

[8]. Lakshman, A., and Malik, P. (2010). Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review, 44(2),* pp. 35-40.

[9]. BBC News. http://www.bbc.co.uk/news/technology-12214628

[10]. BBC News. http://www.bbc.co.uk/news/technology-13693791

[11]. Stoica, I. et al. (2003). Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking (TON), 11(1),* pp. 17-32.

[12]. Dabek, F. (2005). A Distributed Hash Table. Doctoral Dissertation, MIT — Massachusetts Institute of Technology.

[13]. Schintke, F. et al. (2010). Enhanced Paxos Commit for Transactions on DHTs. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid '10)*, pp. 448-454.

[14]. Shafaat, T., M., Haridi, S., Ghodsi, A. et al. (2008). On Consistency of data in structured overlay networks. *CoreGRID Integration workshop.*

[15]. Lamport, L. (2001). Paxos Made Simple. *ACM SIGACT News, 32(4),* pp. 18–25.

[16]. Misra, J. (1986). Axioms for Memory Access in Asynchronous Hardware Systems. *ACM Transactions on Programming Languages and Systems (TOPLAS), 8(1),* pp. 142-153.

[17]. Herlihy, M. P., and Wing, J. L. (1990). Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS), 12(3),* pp. 463-492.

[18]. Guerraoui, R., and Rodrigues, L. (2006). *Introduction to Reliable Distributed Programming*. Germany: Springer.

[19]. Lamport, L. (1979). How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers, 28(9),* pp. 690-691.

[20]. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., and Hutto, P.W. (1995). Causal Memory: Definitions, Implementation and Programming. *Distributed Computing, 9(1),* pp. 37-49.

[21]. DeCandia, G. et al. (2007). Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the Twenty-First ACM Symposium on Operating Systems Principles (SOSP '07),* pp. 205-220.

[22]. Karger, D. et al. (1997). 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 (STOC '97)*, pp. 654-663.

[23]. Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *ACM Communications, 21(7),* pp. 558-565.

[24]. Ghodsi, A., Alima, L. O., Haridi, S. (2005). Symmetric Replication for Structured Peer-to-Peer Systems. In *Proceedings of the 2005/2006 International Conference on Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P '05)*, pp. 74-85. Trondheim, Norway.

[25]. Ratnasamy, S. et al. (2001). A Scalable Content Addressable Network. *ACM SIGCOMM 2001*, pp. 161-172.

[26]. Zhao, B.Y. et al. (2004). Tapestry: A Global-scale Overlay for Rapid Service Deployment. *IEEE Journal on Selected Areas in Communications, 22(1),* pp. 41-53.

[27]. Distributed k-ary System (DKS): A Peer-to-Peer Middleware, 2005. http://dks.sics.se

[28]. Kompics: A Message-passing Component Model for Building Distributed Systems, 2010. http://kompics.sics.se

[29]. Arad, C. (2010). Kompics Programming Manual and Getting Started Tutorial. http://kompics.sics.se/trac/attachment/wiki/WikiStart/kompics-tutorial.pdf

[30]. Apache MINA network application framework, 2010. http://mina.apache.org

[31]. Leonard, D., Rai, V., Loguinov, D. (2005). On Lifetime-Based Node Failure and Stochastic Resilience of Decentralized Peer-to-Peer Networks. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '05)*, pp. 26–37, New York, USA.

[32]. Amazon.com. http://www.amazon.com

[33]. Mitzenmacher, M. (2001). The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel and Distributed Systems archive, 12 (10)*. NJ, USA.

TRITA-ICT-EX-2011:178

www.kth.se