When Paxos Meets Erasure Code: Reduce Network and Storage Cost in State Machine Replication

Shuai Mu*, Kang Chen†, Yongwei Wu†, Weimin Zheng†
Tsinghua National Laboratory for Information Science and Technology (TNLIST)
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
Research Institute of Tsinghua University in Shenzhen, Shenzhen 518057, China
*shuai@cs.nyu.edu
†{chenkang, wuyw, zwm-dcs}@tsinghua.edu.cn

ABSTRACT

Paxos-based state machine replication is a key technique to build highly reliable and available distributed services, such as lock servers, databases and other data storage systems. Paxos can tolerate any minority number of node crashes in an asynchronous network environment. Traditionally, Paxos is used to perform a full copy replication across all participants. However, full copy is expensive both in term of network and storage cost, especially in wide area with commodity hard drives.

In this paper, we discussed the non-triviality and feasibility of combining erasure code into Paxos protocol, and presented an improved protocol named RS-Paxos (Reed Solomon Paxos). To the best of our knowledge, we are the first to propose such a combination. Compared to Paxos, RS-Paxos requires a limitation on the number of possible failures. If the number of tolerated failures decreases by 1, RS-Paxos can save over 50% of network transmission and disk I/O. To demonstrate the benefits of our protocol, we designed and built a key-value store based on RS-Paxos, and evaluated it on EC2 with various settings. Experiment results show that RS-Paxos achieves at most 2.5x improvement on write throughput and as much as 30% reduction on latency, in common configurations.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems–client/server; distributed applications; distributed databases; D.4.5 [**Operating Systems**]: Reliability–fault-tolerance; H.3.4 [**Information Storage and Retrieval**]: Systems and Software–distributed systems

Keywords

Paxos, erasure code, asynchronous message passing model, consensus, state machine replication

1. INTRODUCTION

Paxos-based state machine replication (SMR) has proven to be an effective approach to build highly reliable distributed services. For example, Google's Chubby[4] replicates important metadata through a couple of nodes, such as locks and configuration files in a storage system[7][9]. Since Gaios[3], Paxos has been used to replicate all user data instead of only metadata. Furthermore, systems such as MegaStore[2] and Spanner[8] use Paxos to replicate data across datacenters globally.

A key challenge of data replication through Paxos is the cost of network transmission and disk writes that need to be flushed to disk to tolerate crashes. And as the size of data increases, these costs increase. In an optimized Paxos instance, the value is required to be sent to a quorum of nodes at least once. And to tolerate more than minority crashes, all nodes need sync to disk on every acknowledged accept request in the accept phase. Both the network and the I/O costs can be expensive to achieve low latency and high throughput in a distributed system.

Erasure coding[22][18] is a very effective and common technique to reduce storage and network cost in data replication. Erasure coding encodes data objects into a configurable number of data fragments (including both original shares and redundant shares). From any large enough subset of these shares we can rebuild the original data objects. The redundancy rate of erasure coding depends on configuration. It is usually much smaller than making a full copy replication.

Can we extend Paxos to support erasure coding, instead of using the original value in Paxos? A naive approach to combine Paxos and erasure coding is to encode the original value into a majority of original data shares and a minority of redundant shares, and then send one coded data share to each acceptor. Because each share is smaller than the original value, network and disk I/O costs are reduced. However, this simple approach of injecting erasure code into Paxos is incorrect, mainly due to the asynchronous message passing model of Paxos (see details in Section 2.3).

In this paper, we examine the problem and present an improved protocol named RS-Paxos (Reed Solomon Paxos). By redefining the quorum number in each Paxos phase and correlated configuration of erasure coding, RS-Paxos incurs a huge reduction of message sizes, thus largely reducing network and disk I/O costs. A side effect of RS-Paxos is that it tolerates fewer failures than original Paxos in an instance.

But this is acceptable in realistic systems because most failures are one or two nodes. This is also the same assumption in EPaxos[20]. And by automatic reconfiguration of the state machines we can tolerate more failures in practice.

Most real-world Paxos systems takes the leader-follower design[4][8][6]. The leader acts as a distinguished proposer, in order to avoid live lock and save one message roundtrip. Leader serves all write requests and are in charge of proposing values. A follower usually redirects all requests to leader, unless the leader fails and it becomes the new leader. In certain cases, it can also serve read requests with relaxed consistency. RS-Paxos fits this pattern quite well. In RS-Paxos, the leader caches the original value itself, while sending coded shares to the followers. Both leader and follower only need to flush the coded shares into disks. A follower does not have to learn the original value immediately, since it redirects all consistent requests to the leader.

In the remainder of this paper, first we briefly go through the background knowledge and point out the incorrectness of an intuitive approach in Section 2. Next, Section 3 gives details of the design and algorithms of RS-Paxos. Section 4 presents a key-value store based on RS-Paxos. Section 5 is about implementation. We describe our experiments and evaluate the system in Section 6. Section 7 discusses about related work, and Section 8 concludes.

2. BACKGROUND AND PROBLEM

We begin by briefly describing the classic Paxos and erasure code algorithms, and an example showing the incorrectness of directly injecting erasure code into Paxos.

2.1 Paxos and SMR

The consensus problem requires a set of distributed processes[1] to agree on a single value in spite of possible failures. Paxos considers this problem in a partial-asynchronous system (partial-asynchrony is further explained in Section 3.1). Paxos assures that at most one value can be chosen (safety); if there are only a minority of faulty processes, all correct processes can eventually agree on a value (progress). Paxos does not tolerate Byzantine failures: A process may crash or fail to respond for an arbitrary long time; but it cannot respond in an undefined way; message corruption can be excluded by simple techniques such as checksums.

A single Paxos instance is barely useful in real systems. The more practical use is running multiple Paxos instances in a pre-defined order, such as to represent the state transitions of state machine replication (SMR). SMR is a classical approach to model and build a highly reliable and available distributed system. SMR aims to make a set of replicas execute the same commands (or make the same transitions) in the same order. For each state transition, a Paxos instance is run to decide what is the next command. Paxos instances can also run in parallel, as long as the decisions–commands are executed in the same order.

An unoptimized Paxos instance could go as following: When receiving client requests, a replica R picks the next unused Paxos instance, sends prepare messages containing a unique number to all replicas including itself. Upon receiving a majority of promises, R proceeds to send accept messages containing the command to all other replicas. If these messages are also confirmed by a majority of replicas, R then decides the command locally and notify all replicas. The majority rules can be replaced with read and write quorums. Any read quorum and write quorum must have a intersection part.

The canonical Paxos takes at least two roundtrips to commit a value. An important optimization in practice is Multi-Paxos. Multi-Paxos let one replica be the distinguished proposer and prepare a large amounts of instances at once, before the instances are actually used to propose values. This leader-follower variant of Paxos is widely taken in many systems such as Chubby, Spanner, etc.

After optimized, there are still two worth-noticing aspects of cost in Paxos. First, at least a full copy of the value need to sent to each replica, in the accept phase (the value sent in learn phase can be skipped, represented by a value id, assuming the target has received this value before). Second, in order to recover from failures, each replica has to log its decisions to disk whenever it responds to prepare and accept messages. In some cases, this can be avoided if we assume there is always a majority of correct processes (such as each process in a separate datacenter). But if we need to tolerate a majority crash such as in a power failures, this logging is necessary.

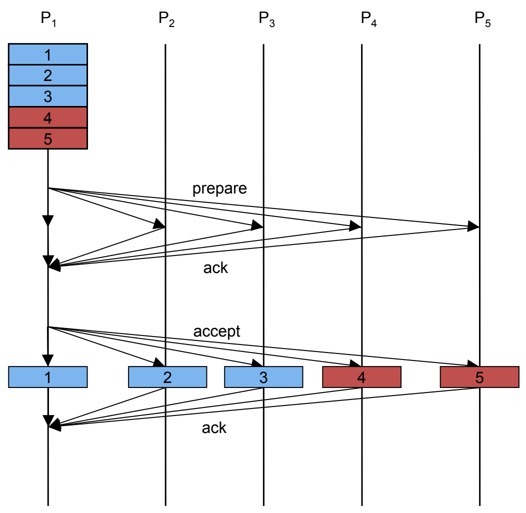


Figure 1: A naive combination of Paxos and erasure code. The configuration of erasure code is $\theta(3,5)$. The original shares (1∼3) are colored as blue ; the redundant shares (4∼5) are colored as red.

2.2 Erasure Code

Erasure coding is a very mature technique used in storage systems for data striping and fault tolerance. The principle of erasure coding is as follows. A data object is first divided into m equal-sized fragments called original data shares. Then, k parity fragments with the same size as original data shares are computed, called redundant data shares. This will generate a total of $n = m + k$ equal-sized shares. The erasure code algorithm guarantees any arbitrary m shares out of total n shares is sufficient enough to recon-

[1]In following sections we will use terms of *process*, *replica*, and *server* in a mixed manner, depending on context. They are synonyms in this paper.

struct the original data. Both m and k are positive values and configurable by users.[2]

Erasure code can reduce data redundancy in strict replication (full copy). Let r denote the data redundancy rate. For a strict replication with n copies, $r = n/1$. For erasure code, since each share is $1/m$ the size of the original data object, $r = n/m$. For example, if $n = 5$, $m = 3$, $k = 2$, the redundancy rate $r = n/m = 5/3$. The original data can be recovered as long as there are 3 replicas that are not permanently damaged. The space saved compared to full replication is $n - r = 10/3$ size of the original data object.

2.3 Problem with a Naive Approach

Since Paxos and erasure code can both be configured to be tolerant to a minority of failures, is it feasible that we can intuitively merge the two algorithms? The answer is no, it can be shown in a simple example.

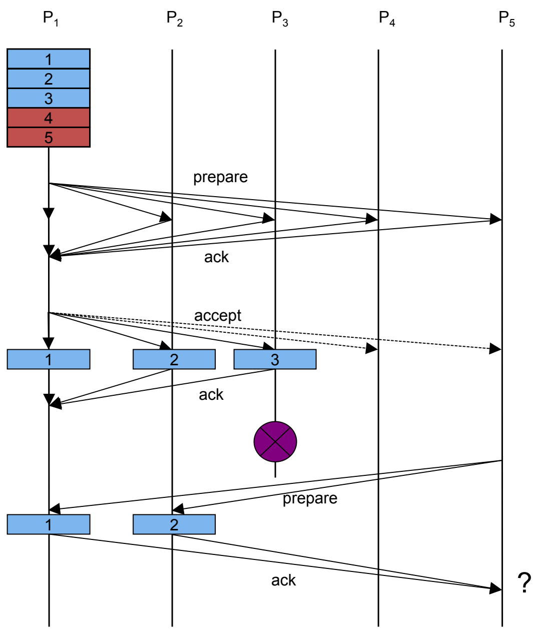


Figure 2: Incorrectness of an intuitive combination of Paxos and erasure code. After P_1 has decided the value, P_3 crashes, and P_5 tries to learn the value by sending prepare requests. The dotted line means the message is delayed or lost.

As shown in Figure 1, suppose there are 5 processes running Paxos (each processes act as proposer, acceptor, and learner at the same time). The Paxos protocol can tolerate any two crashes. The intuitive approach merging Paxos and erasure coding would be to configure erasure code as the same fault tolerant level as Paxos. In this example, say dividing the data into 3 original shares, and computing 2 redundant shares, which can be denoted as $\theta(3, 5)$. The original data can be recovered with any 3 pieces of the data shares.

[2]There are many types of erasure codes, mainly in two categories: optimal erasure codes and near optimal erasure codes. In this paper we only refer to Reed Solomon code, a type of optimal erasure code.

However, this intuitive approach won't tolerate two failures as expected. A simple example to reveal this is shown in Figure 2. One proposer P_1 successfully passes phase 1, and goes into phase 2. In phase 2, it sends accept requests to all replicas. Each of these requests contains a coded data share. After a certain amount of time, it receives 3 acknowledgements saying the proposal is accepted. According to Paxos protocol the value is now legally chosen. But here, if one of the replica P_3 fails after this, a learner P_5 would never be able to recover the value, even though it may discover that there was a value accepted. Since it cannot gather enough pieces of coded data shares, it is unable to recover the original value and re-propose it again.

The nature of the problem lies: Paxos is a consensus algorithm used for multiple processes to reach an agreement on the same value, not 5 different values in the above case. This is why we cannot directly use the same fault-tolerant level configured erasure code on the value proposed in Paxos. However, on careful analysis, we will find that the five values are not entirely independent to each other, they are still related. The key observation of this paper is that from any large enough subset of them we can recover the same value, which we will leverage below.

3. RS-PAXOS

We now go through the details of RS-Paxos. First we give a brief summary of the assumptions, models and goals of our protocol. Next the detailed protocol, followed by a brief summary of the proof structure. At last we give an example of RS-Paxos in a common case.

3.1 Preliminaries

The problem is considered in a partial-asynchronous system. The nodes in the system exchange information by sending messages to each other. The messages can be delayed, duplicated, or lost. But if a correct process repeatedly sends a message to another correct process, eventually the message will go through. A more formal definition of "eventually" is that only after an unknown period of time γ, the messages can be delivered within a timeout Δ if the source and the target are both non-faulty processes.

The processes in the system act as the following roles.

- **Proposer** proposes values.
- **Acceptor** votes on values that are proposed by proposers.
- **Learner** learns if a value is chosen based on the votes of acceptors.

These roles are only logical, a process can and usually act as multiple roles. Different systems may have different role assignments. For example, a system may assign clients as proposers and learners, and servers as acceptors and learners. However, in many practical systems such as Chubby, clients do not directly propose values, but ask one of the servers to propose as instead. This is also our model in this paper.

To be correct and useful, RS-Paxos must guarantee the following properties.

- **Non-triviality**: Only proposed value can be chosen.
- **Stability**: Decisions can not be altered.

- **Consistency**: At most one value is chosen.
- **Liveness (Progress)**: Eventually a value is chosen.

The first three guarantees are usually called safety guarantees. Non-trivially is usually trivial. Stability means that for any process, if it decides a value at time t_1, then for any time $t_2 \geq t_1$, it will still decide on the same value. Consistency is the most important guarantee of safety, it means that for any two processes p_1 and p_2, if p_1 decides on a value v_1 and p_2 decides on v_2, v_1 must equal v_2.

The system can only make progress if there are a quorum of non-faulty acceptors, and at least one proposer and one learner that functions correctly.

3.2 Basic Protocol

The key principle of Paxos is that any two different majorities of processes have a non-empty intersection. And from that intersection a process can learn if a value is chosen or might yet be chosen. The insight of RS-Paxos is to increase the size of any intersecting set, so that it is possible to recover data from such intersections. Nevertheless, this approach unavoidably decreases the fault tolerance number of failed nodes.

Let N denote the number of acceptors; let F denote the number of failed acceptors that can be tolerated. Assume that proposers have different ids to identify themselves. Assume each proposer can generate a distinguished value id to identify the value to be proposed.

RS-Paxos shares many similarities with Paxos. The proposal in RS-Paxos includes following: 1) a ballot id, formed with the proposer id and a natural number, making it globally unique 2) a value id, to identify the value, which is also globally unique; 3) a coded data share, and the meta data of erasure code configuration.

RS-Paxos includes two phases:

Phase 1

(a) The proposer chooses a ballot id, sends a prepare request to at least a read quorum (denoted by Q_R) of acceptors.

(b) If an acceptor receives a prepare request with ballot id i greater than that of any prepare request which it has already responded to, then it responds to the request with a promise. The promise reply contains the proposal (if any) with the highest ballot id that it has accepted. The proposal contains a coded piece.

(c) The proposer waits until it collects Q_R promises. If no value is ever found accepted, then the proposer can pick up its own value for next phase. If any already accepted coded piece is found in one of the promises, the proposer then detects whether there are enough pieces in these promises to recover the original value. Next, the proposer picks up the recoverable value with highest ballot, recover it using erasure code and use it for next phase. If no value is recoverable, the proposer may also choose its own value.

Phase 2

(a) Based on the value v picked up in the previous phase, the proposer generates accept requests for at least a write quorum (denoted by Q_W) of acceptors. Every accept request should contain a coded piece of v. The piece is encoded with a configuration of $\theta(X, N)$, meaning that it divides the data into X original data shares, and computes $(N-X)$ redundant shares. After coding, the proposer sends these accept requests to acceptors.

(b) If an acceptor receives an accept request with ballot i, it accepts the proposal unless it has already responded to a prepare request having a ballot j greater than i.

(c) If the proposer receives Q_W of acknowledgements, the value is successfully chosen.

The two phase together with ballot i, we call it *round* i.

RS-Paxos is actually a superset of Paxos. In Paxos, $X = 1$. And if we take the canonical majority approach, $Q_R = Q_W = \lfloor N/2 \rfloor + 1$, $F = \lceil N/2 \rceil - 1$. In RS-Paxos, the size of the value in each accept request is about $1/X$ of the original value size in original Paxos.

The relationship between Q_R, Q_W, X, N is following:

$$Q_R + Q_W - X = N$$

This implies that any read quorum must have a non-empty intersection with any write quorum, which is a key to guarantee safety.

The relationship between Q_R, Q_W, X, F, N is following:

$$F = N - max(Q_R, Q_W) = min(Q_R, Q_W) - X$$

This tells us how to achieve the smallest data redundancy given a fault tolerate level. With a fixed F (then also a fixed $max(Q_R, Q_W)$), we have

$$X = min(Q_R, Q_W) - F$$

To get the maximum X, we need Q_W=Q_R (also common in practice). The larger X is, the smaller the data redundancy will be, the more network and I/O cost we can save.

3.3 Proof Framework

Due to space limitation, this paper only conveys a brief proof framework. But given our proofs here, one can construct a full proof such as in [16] [15].

Non-triviality is straight-forward. Since all values are from proposers in phase 1(a), a value can only be chosen if it has been proposed.

Consistency can be proved by the following proposition.

Proposition 1 For any two rounds with ballot ids i and j, and $j < i$, if a value v has been chosen or might yet be chosen in round j, then no acceptor can accept any code pieces except those of v in round i.

Proposition 1 is equivalent to the following proposition.

Proposition 2 For any two rounds with ballot ids i and j, and $j < i$, if an acceptor has accepted a coded piece of value v in round i, then no coded piece other than those of v has been or might yet be chosen in round j.

To prove Proposition 1, we only need to prove Proposition 2. To prove Proposition 2, we still need to prove another proposition first.

Proposition 3 For any two rounds with ballot ids i and j, and $j < i$, if a value v has been chosen or might yet be chosen in round j, then the value must be recoverable in phase 1(c) in round i.

Proof Sketch for Proposition 3 If a value v is chosen in round j, it means that at least Q_W acceptors have accepted one coded piece of that v. The proposer collects at least Q_R responses after phase 1(b). The intersection part of Q_R and Q_W is X, due to the protocol. It means from the Q_R promises, at least X of them are from the Q_W acceptors those have accepted the coded piece of v. Since the coding configuration is $\theta(X, N)$, it means we only need X part to recover the value. Thus, v must be recoverable in phase 1(c). In the phase 1 of round i, If v is not chosen yet, for it to be chosen there must be at least Q_W of acceptors that will accept the proposal in round j. This means these Q_W of acceptors must see the accept message of round j before the prepare message of round i, otherwise v cannot be chosen. Since Q_R and Q_W have X acceptors in common, the value must be recoverable in round i.

Now we can prove for Proposition 2 and Proposition 1.

Proof Sketch for Proposition 2 If an acceptor A has accepted value v from some proposer P with a round ballot i, there must be a read quorum Q_R of acceptors have promised to P with that ballot id i. If a different value v' is chosen or might yet be chosen in round j, P must be able to recover v' due to Proposition 3. If P recover such value v', it will use v' as the value in the accept phase. Since P proposes v to A, such v' must not exists. Proposition 2 and proposition 1 are proved.

Because acceptors log their states into persistent storage that can survive crash, proposition 1 and 2 also implies **stability**.

Each replica keeps sending message to one another until it gets response. As long as at least a $max(Q_R, Q_W)$ of replicas are non-faulty and messages eventually arrive, **liveness** is ensured.

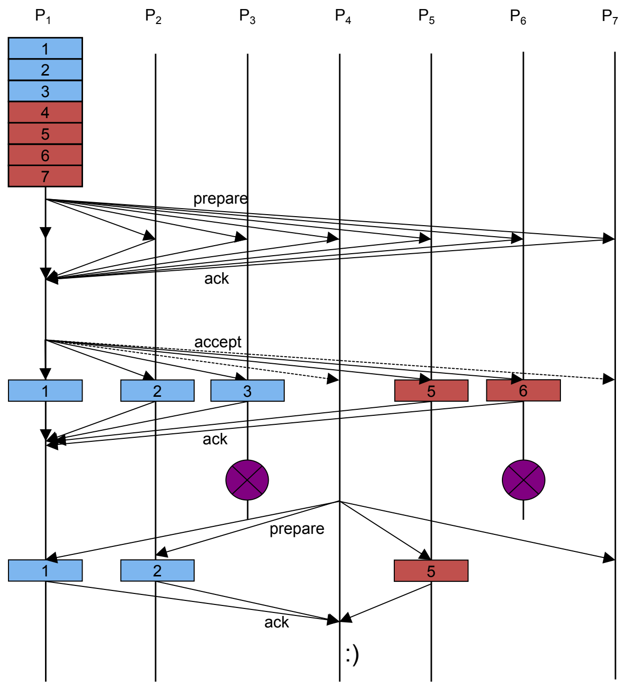


Figure 3: An example of RS-Paxos. N=7, Q_W=Q_R=5, X=3. With two lost accept messages and two replica crashes, the system is still safe.

3.4 An Example

Now we use a simple example to demonstrate the principle of of RS-Paxos. Suppose we have N=7 acceptors, and want to tolerate F=2 possible failures. Q_W and Q_R are both set to 5.

The procedure is shown in Figure 3. In phase 1, a proposer sends prepare to all acceptors. After it collects Q_R=5

N	Q_W	Q_R	X	F
7	4	4	1	3
7	5	3	1	2
7	5	4	2	2
7	5	5	3	2
7	6	2	1	1
7	6	3	2	1
7	6	4	3	1
7	6	5	4	1
7	6	6	5	1

Table 1: Various configurations when N=7. Given a fixed F, the configuration for maximum X is highlighted.

successful acknowledgements. It encodes the data with a coding configuration $\theta(3, 7)$. Each coded data share is $1/3$ size of the original data. Then it sends accept requests to everyone. Within each of these requests is a coded data share. Then it waits for Q_W=5 successful acknowledgments, after which the value is considered successfully decided.

After the coded value pieces have been accepted by 5 acceptors, if another proposer tries to propose, it will collect at least 3 coded data shares. From these data shares it can recover the original value and use that to propose again.

Table 1 shows about all possible configurations of RS-Paxos, when N=7. When a fault tolerance number F is chosen, there are different choices of Q_R, Q_W, and X. We highlight these lines in table that reaches the maximum X. With a maximum X, smallest amount of data $(1/X)$ is needed to be sent during phase 2.

4. A KEY-VALUE STORE BASED ON RS-PAXOS

In this section we go through our key-value store[3], which is designed to demonstrate the capacity of RS-Paxos. The rationales and techniques we use are mostly taken from previous Paxos-based systems[8][2] . This key-value store can be thought as a minimal functional component of these systems.

4.1 Architecture

The architecture of the key-value store is shown in Figure 4. Each a server has a persistent storage space attached. This storage space could be a local key-value datastore such as LevelDB and Redis. Or it could be distributed key-value storage such as HBase, depending on application requirements. Each server is responsible for one or more particular data shards. Of all the replicas in a shard, there is a *leader* replica which can provide consistent fast read and function as a *distinguished proposer*. Upon a client request (write), a leader start a new instance of Paxos, using it to commit a *write ahead log*. Once the Paxos instance commits, the leader commits the change log to its local persistent storage and returns to client. The Paxos instances of each data shard are committed and executed in a linearizable sequence.

[3]The system in this paper may seem closer to an *object store* to some readers. We just use the name of *key-value store* for simplicity.

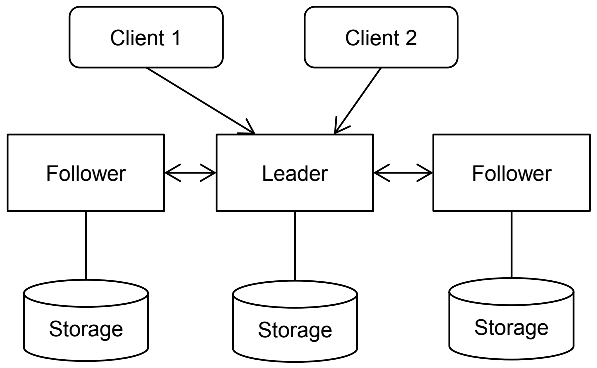


Figure 4: Architecture of the replicated key-value store

4.2 Data Shards and Paxos Groups

To improve throughput, data are partitioned into different shards. Operations on each shard run in a separate Paxos group. This is one common solution to reduce the unnecessary ordering inside a Paxos group, since canonical state machine replication requires all commands committed and executed in order to ensure linearizability.

To simplify, the number of shards are statically configured. The mapping relations of keys into shards is defined by a deterministic mapping function. The granularity of partition is controlled by users by configuring the number of shards and defining a proper mapping function.

4.3 Leader-leases

We use a very simple leader lease mechanism. Suppose the maximum time drift between different servers is δ. The leader maintains a lease which confirms its leadership in next Δ period of time. Each follower can only drops such lease in $\Delta + \delta$ of time.

The above leader lease mechanism can guarantee a solo leadership if all replicas always obey the time drift of δ. Spanner does this by introducing customized hardware (atomic clocks, etc.) and interfaces. In more common systems this is usually done by periodically time synchronization between replicas. Notice that even that leases are broken in rare circumstances and two replicas believe themselves to be leaders, (RS-)Paxos still guarantees safety of the system. But fast read may indeed return inconsistent data in such cases.

4.4 Data Operations

On client startup, it firstly gather the information that which replica is the leader of each data shard, and save this information in its local cache. Clients send their requests to the leaders. On most client requests (except fast read), the leader use RS-Paxos to commit the command as a write ahead log, then modify the actual data in its local storage.
Write. On write requests, the leader commit a log using RS-Paxos, containing operation type, key and value. Only the value are coded into pieces. This is for followers to conveniently tracking which keys are modified. Upon successful commits, the leader (which has the whole value) can write to its local storage, the follower (which only holds part of the value) also write to its local storage, but tag this value as incomplete. Notice that writes to local storage do not have flush to disks, because we already have a persistent write ahead log committed to disks by RS-Paxos.
Read. There are three kinds of reads. 1) *Fast read.* The leader can return values in its local storage. The results are correct as long as leader leases works correctly. If leader leases are malfunctioning, the results may be inconsistent. For example, if two replica both consider themselves as leaders, one of them may server client requests with outdated data from its local cache. We do not have particular solutions for this, neither do we make the problem worse, and it is not our main concern. 2) *Consistent read.* The leader can invoke a explicit Paxos instance, which works as a mark, to read the value. It returns the value on commit of this instance. This approach always return consistent value in spite of the correctness of leader leases. 3) *Recovery read.* This happens when a new leader comes up to replace the old one. Because the new leader only has a piece of the value, it needs to scan through its RS-Paxos history, find the most recent write to that key, and run an explicit RS-Paxos to gather enough pieces of the value before it returns to client. Recovery read can also function as *snapshot read* if the application requires a snapshot version from a non-leader replica.
Insert. For simplicity, the insert operations can be treated as regular writes.
Delete. Delete operations are treated as *write(key, NULL).*

4.5 Crash and Recovery

When a follower crashes within the fault tolerance level of the system. The system can still correctly server requests. Actually, the throughput of the system might increase because that fewer messages due to the crash.

If the error is fixed and the crashed server comes back online, it is essential that it is able to recover all its states including all including the maximum ballots it replied to and all the values it accepted. Otherwise the system will misbehave. That's why it needs to log all these decisions into disks before sending out the reply.

On recovery, the server needs to catches up all the chosen values in the instances it has missed. In Paxos, it only needs to ask for any other server that is aware of these values. In RS-Paxos, only the leader is aware of these decisions. Therefore, the leader needs to re-code the data and sends the corresponding fragment to the recovering server.

If a leader crashes, there will be a time window during which the server cannot server any new requests until a new leader takes over. Leader election can be a complex issue. Deciding a new leader is actually also a consensus problem. We simply use another Paxos instance to decide the new leader. After the new leader starts to serve requests in the system, it needs to perform recovery read to rebuild a local cache, in order to server fast read requests from clients. This will affect system performance. Write requests, however, are not influenced by this. When a new write request arrives, the leader can simply issue a new RS-Paxos instance containing the write request, even if it has not observed the previous value of this key.

4.6 Reconfiguration

Our design includes a classical way to reconfigure Paxos-based the state machines, aka *view change.* On adding a new replica or removing a current replica, a special Paxos instance *view change* is proposed, containing the new view of the replicas. Each view change will give the Paxos group a new *epoch* number. Each epoch number represents a distinguished configuration of the system. Each Paxos instance should be attached with a correct epoch number to guarantee the quorum calculation corresponds to a correct view.

The view change also leads to new configuration of erasure coding. For example, the system currently has N=5 replicas, quorum Q=4, coding configuration $\theta(3,5)$. Now a new replica is added to the system, and the new configuration becomes N'=6, Q'=5, $\theta'(4,6)$. Strictly, the system needs to issue new RS-Paxos instances to re-code the data with the new configurations.

RS-Paxos contains a few optimization to reduce this cost. First, if the new configuration keeps the same k in the new $\theta(k,m)$, it is unnecessary to resend the original coded fragments again in the new RS-Paxos instances. For example, a previous configuration of N=5, Q=4, $\theta(3,5)$ is changed to N'=5, Q'=4, $\theta'(3,3)$. In this case there is no need to re-spread the data. The system only needs to launch an instance to ensure that every replica has its own data share.

Another important optimization also aims to avoid re-sending and recoding all data. If the quorum in the new configuration is greater than the number of original shares in old configuration, i.e. $Q' \geq X$, the system only needs to confirm that every server is already holding its data share. For example, with the old configuration N=5, Q=4, X=3, and a new configuration N'=4, Q'=3, X'=2, the system only needs to confirm every server holds all its data shares correctly when applying a view change. The insight of this optimization is that the fault tolerance level rules that at most F=$N-Q$ errors can be tolerated during an RS-Paxos instance. If the value is chosen at each server and each data share is stored correctly, the actual fault tolerance becomes $N-X$.

5. IMPLEMENTATION

Our prototype is implemented using C and C++. The core RS-Paxos framework is implemented in C. The key-value store part is implemented in C++. Since RS-Paxos is a superset of Paxos, we chose to firstly build a Paxos implementation, and then modified it to adapt RS-Paxos protocol.

RPC. We built an asynchronous RPC module for message passing between processes. It uses TCP as transmission protocol. This layer ensures that common network errors are handled properly, such as occasional packet loss and duplicate. Unit tests showed that it can complete over 1 million batched `ADD` operations in 1 second between two servers (using single CPU core each) in our local clusters.

Paxos. We built a Paxos implementation from scratch. Our Paxos implementation includes common optimization such as: 1) The leader do a batch prepare for a large amount of instances before it actually use the instances; 2) The commit message is delayed and bundled together and sent off the critical path of leader.

Erasure Code. We chose to use Zfec[25] as the erasure coding library, instead of on our own. A recent performance evaluation[21] shows that Zfec performs well and has relatively low cost.

6. EVALUATION

This section presents an experimental evaluation of RS-Paxos as the core protocol in a key-value store. The experiments include a series of micro-benchmark and a dynamic workload, including both intra and inter datacenter situations.

6.1 Setup

To test for how our key-value store functions, we configure the system with 5 replicas within a Paxos group (N=5). We have to make a trade-off in choosing the number of replicas inside a Paxos group. The benefits of RS-Paxos is more obvious as the number of replicas increase, but it is impractical in real systems to have a 100-replica Paxos group. If the size is very small, for example a 3-replica Paxos, RS-Paxos has no win over Paxos because it has to set X=1 to tolerate a failure, making it no different to Paxos. At last, we chose N=5, which is a common configuration in practical systems[4].

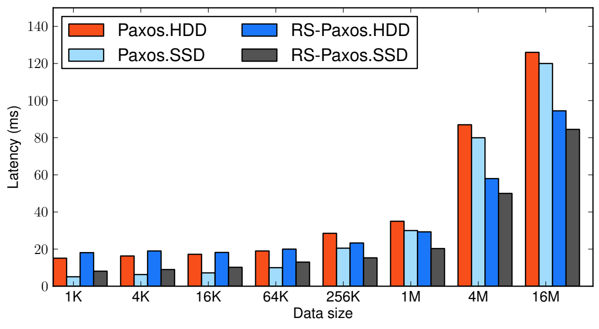
Both read and write quorum size Q=4, which means X=3, theoretically the message size should be about 1/3 about the original size (if the value is large enough). If one replica fails, the system is configured to change to a new quorum Q=3, and change to a new erasure coding configuration with X=2. This strategy allows the system tolerates two uncorrelated failures, given enough time for view change.

To increase parallelism, the whole key space is partitioned into 100 Paxos groups, following the typical configuration in practical Paxos systems[8]. A replica is configured to be the leader, which is responsible for all client write requests and is able to do fast read. In spite of our configuration, finer granularity of data sharding is totally possible. In real systems there could be thousands of machines and millions Paxos groups, and each server serves a certain number of Paxos groups[8]. But since scalability is not our concern in this paper, our full replication configuration is sufficient to observe the improvements of RS-Paxos to Paxos.

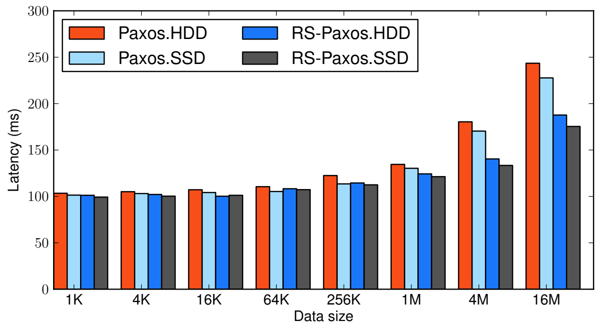
For comparison, we also test for a key-value store based on Paxos. The group size is also set to 5. Although strictly speaking, a 5-node Paxos offers stronger fault-tolerance than RS-Paxos. It can tolerate two concurrent failures, while RS-Paxos can only tolerate one at a time. Another possible configuration is to set up a 3-node Paxos group, which tolerates exactly one failure. Theoretically, the data redundancy of a 3-node Paxos group is 3/1; the data redundancy of a 5-node RS-Paxos group is 5/3. Therefore, a 5-node RS-Paxos is still better than a 3-node Paxos. The transformation from RS-Paxos to Paxos is very easy, simply to configure the quorum to be a simple majority and turn off the erasure coding function. In our evaluation, our major considerations include latency, throughput, computational cost and failure recovery.

Our experiment is run on Amazon EC2 platform. We use the `extra-large` EC2 VM instances in the `us-east-1` region. Each VM has 7GB memory and 8 virtual CPU cores, and connected by a gigabyte Ethernet network. 5 VMs are deployed as servers, and 10 VMs are deployed as clients. Each client VM serves up to 100 logical clients.

The above configuration shows the performance of RS-Paxos compared to Paxos in a local cluster environment. Inspired to see how RS-Paxos performs in various environments, we also emulate a wide-area deployment by adding a 50 ± 10ms delay to the network interface, thus having a



(a) Local cluster



(b) Wide area

Figure 5: Micro-benchmark: Average Latency

100 ± 20ms latency for a message roundtrip. Also the bandwidth is limited to 500Mbps. We choose this rather than deploy in actual different EC2 datacenters, in order to keep the ability of a large bandwidth, to emulate the private network connecting different datacenters in enterprises.

To evaluate the performance of RS-Paxos in various storage environment, we use two kinds of EBS volumes in our benchmarks. One is regular EBS volume, with around 100 IOPS, representing traditional hard drives; the other is high-performance EBS volume, with over 4,000 IOPS, representing high performance solid-state drives. We will use .HDD and .SSD as postfix to distinguish in the following evaluation, such as RS-Paxos.SSD.

6.2 Micro-Benchmark

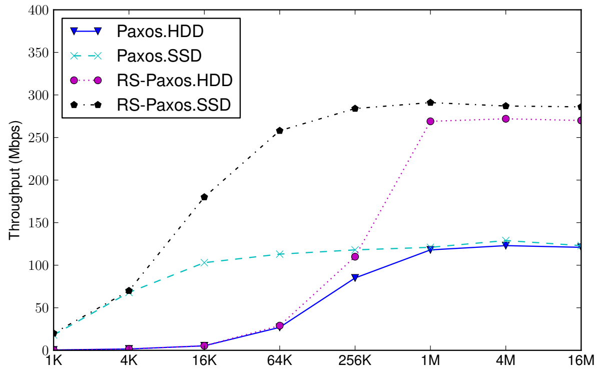
Our major concern is the performance of write requests, including the possible encoding overhead and the potential benefits of smaller amount of network transmission and disk flushes. And since we almost have no overhead on read requests compared to Paxos, we should supposedly have similar read performance. Accordingly, in this benchmark we test RS-Paxos with various sized write requests, to its maximum throughput. The value size ranges from 1KB to 16MB. We believe that larger sized data are usually chopped into smaller chunks, such as 16MB each.

6.2.1 *Latency*

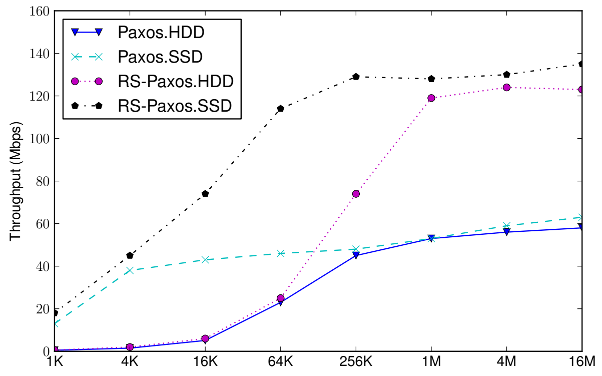
Figure 5 shows the latency of various sized write requests in both local cluster and wide area. When measuring latency for a given size request, there is a fixed cost that the client send the request to the server, and the server reply to client when it finishes. Since this cost is identical for both Paxos and RS-Paxos, we remove it from our results for better comparison of cost that matters.

In the local cluster, when the request size is small (lower than 64KB), latency of both Paxos and RS-Paxos is dominated by file system flushes. Thus SSD can commit within 10ms, while HHD takes 20-30ms. RS-Paxos performs slightly worse than Paxos, due to extra computational time for coding. But for data object larger than 256KB, RS-Paxos has an obvious advantage. It achieves a 20%-50% lower latency, because it reduces the number of network packets and the number of the disk I/Os.

In wide-area deployments, the network becomes the dominating factor in latency. The CPU cost is hardly affecting RS-Paxos. RS-Paxos performs almost the same as Paxos at small size requests. And with larger objects, the advantage of RS-Paxos is more obvious, saving more than 50ms.



(a) Local cluster



(b) Wide area

Figure 6: Micro-benchmark: Throughput

6.2.2 *Throughput*

Since RS-Paxos can largely reduce the amount of data needed to be sent over network and flushed into disk, it is expected to have much better throughput than Paxos, especially for large writes. This is proved in this benchmark. Figure 6 show the write throughput of the system in both local cluster and wide-area.

In any deployment, the system is more disk-bounded for small writes. This is more obvious in HDD than SDD, which is easy to understand because HDD has a much lower capability of handling small writes than SDD. When the data size is small, RS-Paxos performs no better than Paxos because it has the same amount of disk seeking and writing, touching the limit of disk access rates.

For HDD deployment, as data size grows larger than 64KB, the system becomes both network and disk bounded. And disk cost lies mainly on writing, rather than section seeking with small objects. In such case RS-Paxos performs about 2.5x better than Paxos, which is a giant improvement. For SDD deployment, this turning point comes smaller between 4KB and 16KB, due to its better performance with small writes.

6.2.3 CPU Cost

The major CPU cost in the system can be categorized as follows: ① Paxos logic; ② system calls such as epoll; ③ thread switching and synchronization; ④ encoding and decoding; ⑤ marshalling and unmarshalling. The overhead that RS-Paxos has over Paxos is mainly on ①④⑤. We try to measure and compare the CPU cost of the system by periodically sampling the CPU usage in the micro-benchmarks. In different setups, the CPU cost fluctuates around 10∼20% per core, and RS-Paxos barely shows an observable overhead compared to Paxos. This is reasonable since such storage system is severely network and disk bounded, rather than CPU bounded. Our observation does not conflict with the fact that erasure coding may cause more overhead in other systems, because the amount of data the system handle per second is far smaller than it can compute in erasure coding. Even with the maximum throughput, the amount of data the system only needs to encode is less than 50MB. Moreover, it proves that it is fair to trade CPU time for a better system throughput in such storage systems.

6.3 Macro-Benchmark

To evaluate the performance of RS-Paxos in various workload, we built a macro-benchmark following Intel's COSBench[27]. The benchmark include four dynamic workloads, with different size ranges and kinds of requests.

One dimension to distinguish the workloads is object size:

- `SMALL` objects. Size range: 1KB∼100KB.
- `LARGE` objects. Size range: 1MB∼10MB.

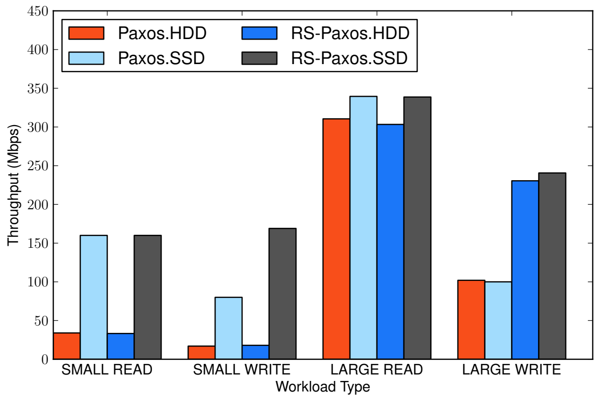
Another dimension is read write ratio:

- `WRITE` intensive. Read write ratio is 1:9.
- `READ` intensive. Read write ratio is 9:1.

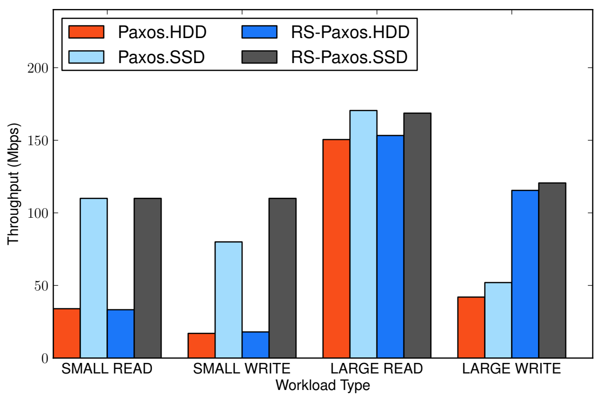
Each combination of the two dimensions above represents a dynamic workload. Every workload simulates the characteristics of real world workload. For example, the `SMALL-READ` workload represents a web hosting service, and the `LARGE-WRITE` workload represents an enterprise backup service.

In this benchmark throughput is our major concern. The results are as shown in Figure 7. In both local cluster and wide area, for small objects, the throughput of SSD is much better than HDD. This is true for both Paxos and RS-Paxos, which proves the same conclusion as in micro-benchmark, that small object size request is mainly disk bounded. Also, for large objects, the difference between HDD and SSD is much less obvious, since it is limited by bandwidth.

In either case, the read performance of RS-Paxos is almost identical to Paxos. This fits our expectation because Paxos



(a) Local cluster



(b) Wide area

Figure 7: Throughput in different dynamic workloads

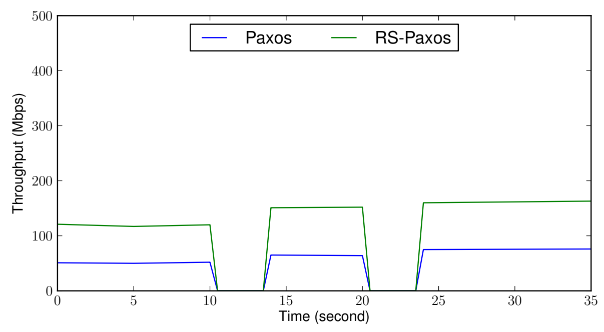
and RS-Paxos share the same mechanism in reading. Both of them do a fast read from local copy as consistent read, when leader leases holds up.

RS-Paxos performs much better than Paxos in the `LARGE-WRITE` workload, for both HDD and SSD. It also performs better in `SMALL-WRITE` workload, for SSD. This case suggests that as the disk performs increases, the advantage of RS-Paxos is more obvious, and the threshold of object size to observe that advantage will decrease, unless another boundary is touched, such as network or CPU.

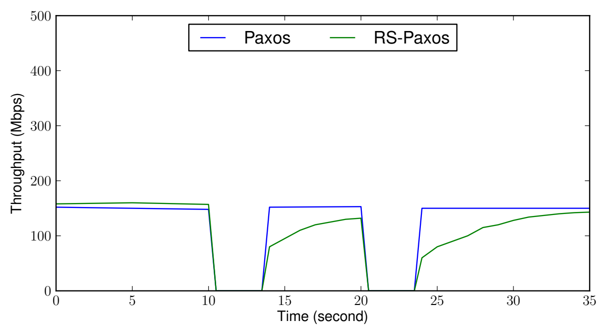
6.4 Availability

In this part we evaluate the the behavior of RS-Paxos under uncorrelated failures. In our configurations, RS-Paxos can tolerate 2 uncorrelated failures, under the conditions that there is enough time for the system to perform view change between the failures.

To reduce the statistical influences of possible time skew on different machines in our observation, we choose to do this evaluation in the wide-area deployment. The test flow goes as following. Initially we keep the server fully loaded. Next we shut down the leader replica R_1. After a while another replica R_2 should become leader. Then we shut down R_2. During this procedure we keep observing system throughput on the next coming leading replica.



(a) Write intensive workload



(b) Read intensive workload

Figure 8: Fail-over time. The first crash is triggered at 10s; the second crash is triggered at 20s.

We are also interested in different kinds of workloads. Because if it is read-intensive workload, a new leader must perform recovery read on client read requests. So the test is done twice, once with write intensive and once with read intensive workload.

Results are as shown in Figure 8. For both RS-Paxos and Paxos, when the current leader is killed, there is a time window where system throughput drop to zero. After the rest replicas wait for the lease to timeout, a new leader will be elected before the system goes back to normal. This time period is the same for RS-Paxos and Paxos, since RS-Paxos does not incur any overhead in design for view change.

After the leader is elected, there is a period during which the system throughput climbs to its maximum. For write intensive workload, RS-Paxos has almost the same recovery time as Paxos because RS-Paxos can directly handle writes without recover the previous value. Notice after each failure, the throughput for write workload actually becomes higher than before. This is because the amount of message to finish a (RS-)Paxos instance decreases with fewer nodes in the system.

For read intensive workload, it takes a longer time for RS-Paxos to climb up to its maximum throughput. This is because the new leader replica does not hold the actual object value in its local storage. So the new leader has to perform a recovery read for every missing object. The cost of a recovery read is similar to a write.

7. RELATED WORK

In the last decade, Paxos[14][13] has become a de facto standard for state machine replication[23]. Many practical systems choose to use Paxos to synchronously replicate their states across multiple nodes. Chubby[4] is one of the earliest industrial implementations of Paxos. It uses Paxos to replicate critical metadata such as configuration files and system views.

Later on, there are more and more systems using Paxos to replicate data, as well as metadata. Many research database systems[1][24][26] use Paxos (at least as an option) to replicate data records, in a single site or across sites. Google has published two systems, MegaStore[2] and Spanner[8]. Both of them use Paxos to replicate data record globally.

Besides database records which are usually of small size, there are also systems using Paxos to replicate larger data, such as files and data objects. This is also currently the target system of RS-Paxos. Gaios[3] presents a design of building a high performance data store using Paxos-based replicated state machines. There are also other file system designs[1] using Paxos as the replication module.

The original Paxos is more of a research theory than a real-system protocol. It is seldom used without any optimization. There are various optimization directing on different dimensions to reduce network messages. The most common optimization is Multi-Paxos[6]. A long-lived leader can act as the distinguished proposer. It can issue prepare requests for next coming Paxos instance before it is executed. RS-Paxos is totally compatible with Multi-Paxos, and we have adopted it in our implementation and evaluation.

Some Paxos-based systems do not take the leader-follower approach, such as Round-robin Paxos, proposed in Mencius[19]. Each participant in the system proposes in an independent set of instances which are pre-defined. On execution, these instances are ordered in a round robin manner. In such case every participant can propose without conflict, getting rid of the leader bottleneck. It is feasible to merge Round-robin Paxos and RS-Paxos together. But a replica may have to issue a recovery read to read the value proposed by other replias.

There are other optimizations those can not be easily combined with RS-Paxos, like Fast Paxos. Fast paxos allows a client directly sends its proposal to each accepter, bypassing the leader. It saves a message trip than sending the proposal to the leader and let leader re-propose it to acceptors. Fast Paxos has a fast quorum and regular quorum. Normally it runs in fast quorum, on conflicts it backs off to a regular quorum. The reason it is hard to combine RS-Paxos and Fast Paxos is that they both modify the quorum definition. It is still possible, but may require cautious work to guarantee the correctness of combination.

The same goes with EPaxos[20]. EPaxos requires client send requests to a nearby server. And it allows every server node to propose value without conflict in common cases. Every proposal is attached with extra dependencies. Based on these dependencies, each server can execute instances in a consistent and efficient manner. It is also not a simple job to combine EPaxos and RS-Paxos, because EPaxos alters the quorum definition as well.

IO batching is also an important engineering technique used in practical Paxos systems and other systems[6], in order to reduce disk and network cost to improve throughput. Usually the server would delay all disk write requests for a small time window (say 10ms), save all the write log in a cache, and then flush them together. This is a good utilization of disk resources, especially when disk performs badly handling small writes. The same batching techniques also goes with RPC, with similar principles. Batching is also an orthogonal optimization to RS-Paxos.

Erasure code[22][18] has been used in many distributed systems such as [12][5][17], in order to reduce storage and network cost. When doing replication, these systems have stronger assumptions about messages passing model and failures. They assumes a more "synchronous" model of the network. If two servers are both healthy, messages must be delivered within a timeout. If the message cannote be delivered within the timeout, the server must have failed. This model does not work properly when there is long message delay or message loss. In another point view, RS-Paxos also points out how to do erasure coding correctly in an asynchronous network.

There are many recent works on erasure coding optimizations[10][11], including reduced the size of data shares, efficient recovery mechanisms, etc. These works have very different perspectives from our work. However, we believe RS-Paxos can benefit from them with careful revisit and combination. On the other hand, rethinking these works in an asynchronous messaging passing model may also lead to promising results.

8. CONCLUSION

In this paper we pointed out a new direction to optimize Paxos protocol. By combining coding techniques into Paxos we can largely save the cost of network transmission and disk flushes. We summarized the possible problems in intuitive approaches, analyzed the requirements for safety guarantees of Paxos in an asynchronous message passing model, and gave an improved protocol RS-Paxos, combing Reed Solomon code with Paxos. We designed and built a key-value store based on RS-Paxos. The experiment results of RS-Paxos is very promising. It shows that RS-Paxos improves throughput by 2.5x in common configurations, with reasonable extra cost and a minor relaxation on fault tolerance level.

9. ACKNOWLEDGEMENTS

The authors are greatly indebted to Jinyang Li, for giving us valuable suggestions and help revise the paper. We thank our shepherd Charles Killian and the anonymous reviewers for their feedback. We also thank Feng Ye, Wenlei Zhu, Pin Gao for their contribution to our Paxos codebase.

This work is supported by National Basic Research (973) Program of China (2011CB302505), Natural Science Foundation of China (61373145, 61170210), National High-Tech R&D (863) Program of China (2012AA012600, 2013AA01A213), Chinese Special Project of Science and Technology (2013zx01039-002-002), Ministry of Education-China Mobile Funding: MCM20123021.

10. REFERENCES

[1] M. K. Aguilera, A. Merchant, M. A. Shah, A. C. Veitch, and C. T. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP*, pages 159–174, 2007.

[2] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.

[3] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 11–11. USENIX Association, 2011.

[4] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350, 2006.

[5] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[6] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC*, pages 398–407, 2007.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data (awarded best paper!). In *OSDI*, pages 205–218, 2006.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8, 2013.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, pages 29–43, 2003.

[10] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure coding in windows azure storage. In *USENIX ATC*, volume 12, 2012.

[11] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: Minimizing i/o for recovery and degraded reads. In *USENIX FAST*, 2012.

[12] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[13] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[14] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[15] L. Lamport. Generalized consensus and paxos. 2004.

[16] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[17] J. Li and B. Li. Erasure coding for cloud storage systems: A survey. *Tsinghua Science and Technology*, 18(3):259–272, 2013.

[18] W. Lin, D. M. Chiu, and Y. Lee. Erasure code replication revisited. In *Peer-to-Peer Computing, 2004. Proceedings. Proceedings. Fourth International Conference on*, pages 90–97. IEEE, 2004.

[19] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machine for wans. In *OSDI*, pages 369–384, 2008.

[20] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372. ACM, 2013.

[21] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, Z. Wilcox-O'Hearn, et al. A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST*, volume 9, pages 253–265, 2009.

[22] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM Computer Communication Review*, 27(2):24–36, 1997.

[23] F. B. Schneider. The state machine approach: A tutorial. In *Fault-Tolerant Distributed Computing*, pages 18–41, 1986.

[24] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[25] Z. WILCOX-OHEARN. Zfec 1.4. 0. *Open source code distribution: http://pypi. python. org/pypi/zfec*, 2008.

[26] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291. ACM, 2013.

[27] Q. Zheng, H. Chen, Y. Wang, J. Zhang, and J. Duan. Cosbench: cloud object storage benchmark. In *ICPE*, pages 199–210, 2013.