CRaft: An Erasure-coding-supported Version of Raft for Reducing Storage Cost and Network Cost

Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang, *Tsinghua University*

https://www.usenix.org/conference/fast20/presentation/wang-zizhong

This paper is included in the Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-12-0

CRaft: An Erasure-coding-supported Version of Raft for Reducing Storage Cost and Network Cost

Zizhong Wang[†‡*] Tongliang Li[†‡*] Haixia Wang[‡] Airan Shao[†‡] Yunren Bai[†‡]
Shangming Cai[†‡] Zihan Xu[†‡] Dongsheng Wang[§†‡]

[†]*Department of Computer Science and Technology, Tsinghua University*

[‡]*Beijing National Research Center for Information Science and Technology, Tsinghua University*

Abstract

Consensus protocols can provide highly reliable and available distributed services. In these protocols, log entries are completely replicated to all servers. This complete-entry replication causes high storage and network costs, which harms performance.

Erasure coding is a common technique to reduce storage and network costs while keeping the same fault tolerance ability. If the complete-entry replication in consensus protocols can be replaced with an erasure coding replication, storage and network costs can be greatly reduced. RS-Paxos is the first consensus protocol to support erasure-coded data, but it has much poorer availability compared to commonly used consensus protocols, like Paxos and Raft. We point out RS-Paxos's liveness problem and try to solve it. Based on Raft, we present a new protocol, CRaft. Providing two different replication methods, CRaft can use erasure coding to save storage and network costs like RS-Paxos, while it also keeps the same liveness as Raft.

To demonstrate the benefits of our protocols, we built a key-value store based on CRaft, and evaluated it. In our experiments, CRaft could save 66% of storage, reach a 250% improvement on write throughput and reduce 60.8% of write latency compared to original Raft.

1 Introduction

Consensus protocols, such as Paxos [12] and Raft [14], can tolerate temporary failures in distributed services. They allow a collection of servers to work as a coherent group by keeping the commands in each server's log in a consistent sequence. These protocols typically guarantee *safety* and *liveness*, which means they always return correct results and can fully functional if no majority of the servers fail. Using these consensus protocols, commands can be properly replicated into each server in the same order, even if machine failures may happen. Google's Chubby [3] is one of the earliest systems using consensus protocols. In Chubby, metadata, like locks, are replicated through different nodes by Paxos. Since Gaios [2], consensus protocols have been used to replicate all user data (typically much larger than metadata) rather than only metadata. Recently, Raft and Paxos have been applied in real large-scale systems like etcd [8], TiKV [1] and FSS [11], to replicate terabytes of user data with better availability.

In such systems, data operations will be translated into log commands and then replicated into all servers by consensus protocols. Thus, data will be transferred to all servers, and then flushed to disks. In consensus problems, to tolerate any F failures, at least $N = (2F+1)$ servers are needed. Otherwise, a network partition may cause split groups to agree on different contents which is against the concept of consensus. Therefore, using consensus protocols to tolerate failures may cause high network and storage costs which can be around N times of the original amount of data. Since these protocols are now applied in large-scale systems and the data volume is growing larger, these costs become real challenges and they can prevent systems from achieving low latency and high throughput.

Erasure coding [16] is an effective technique to reduce storage and network costs compared to full-copy replication. It divides data into fragments, and encodes the original data fragments to generate parity fragments. The original data can be recovered from any large-enough subset of fragments, so erasure coding can tolerate faults. If each server only needs to store a fragment (can be either an original data fragment or a parity one), not the complete copy of the data, storage and network costs can be greatly reduced. Based on the above properties, erasure coding may be a good solution to the challenges of storage and network costs in consensus protocols. Erasure coding is deployed in FSS [11] for reducing storage cost. However, FSS uses a pipelined Paxos to replicate complete user data and metadata 5-ways before encoding. Therefore, extra network cost of FSS is still four times of the amount of data, which harms performance.

RS-Paxos [13] is the first consensus protocol to support erasure-coded data. Combining Paxos and erasure coding, RS-

[*]These authors contributed equally to this work.
[§]Dongsheng Wang (wds@tsinghua.edu.cn) is the corresponding author.

Paxos reduces storage and network costs. However, RS-Paxos has poorer availability compared to Paxos. RS-Paxos trades liveness to use erasure coding for better performance. In other words, a RS-Paxos-applied system of $N = (2F+1)$ servers cannot tolerate F failures any longer. This may be a serious problem since the system should tolerate enough failures. At the theoretical level, we tend to design a consensus protocol with the same level of liveness as Paxos and Raft. We examine this liveness problem and point out that this problem exists because the requirement of committing becomes stricter in RS-Paxos.

We present an erasure-coding-supported version of Raft, CRaft (Coded Raft). In CRaft, a leader has two methods to replicate log entries to its followers. If the leader can communicate with enough followers, it will replicate log entries by coded-fragments for better performance. Otherwise, it will replicate complete log entries for liveness. Like RS-Paxos, CRaft can handle erasure-coded data, so it can save storage and network costs. However, one major difference between CRaft and RS-Paxos is that CRaft has the same level of liveness as Paxos and Raft while RS-Paxos does not.

To verify the benefits of CRaft, we designed and built key-value stores based on different protocols, and evaluated them on Amazon EC2. In our experiments, CRaft could greatly save network traffic, leading to a 250% improvement on write throughput and a 60.8% reduction of write latency compared to original Raft. In addition, we proved that CRaft has the same availability as Raft.

In the remainder, first we briefly go through the background knowledge of Raft, erasure coding and RS-Paxos in Section 2. Next, we explain the details of our CRaft protocol in Section 3 and prove the safety property of CRaft in Section 4. Section 5 describes our implementation, experiments and evaluation. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Background

We begin by briefly describing Raft, erasure coding, RS-Paxos and then discuss RS-Paxos's liveness problem.

2.1 Raft

Raft [14] is one of the consensus protocols and it provides a good foundation for building practical systems. There are three server states in Raft, as shown in Figure 1. A leader is elected when a candidate receives votes from a majority of servers. A server can vote for a candidate only if the candidate's log is at least as up-to-date as the server's. Each server can vote at most once in each term, so Raft guarantees that there is at most one leader in one term.

The leader accepts log entries from clients and tries to replicate them to other servers, forcing the others' logs to agree with its own. When the leader finds out that one log entry accepted in its term has been replicated to a majority of servers, this entry and its previous ones can be safely applied to its *state machine*. The leader will commit and apply these entries, and then inform followers to apply them.

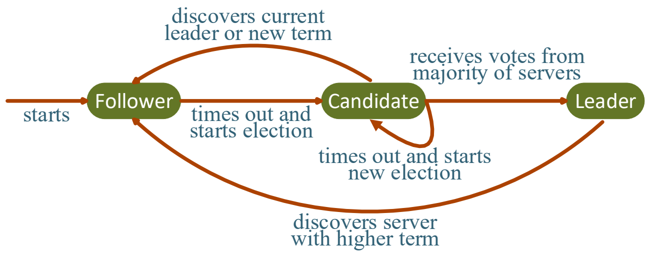
discovers current leader or new term
receives votes from majority of servers
starts
Follower
times out and starts election
Candidate
Leader
times out and starts new election
discovers server with higher term

Figure 1: Three server states in Raft [14]

Consensus protocols for practical systems typically have the following properties:

- *Safety*. They never return incorrect results under all non-Byzantine conditions.
- *Liveness*. They are fully functional as long as any majority of the servers are alive and can communicate with each other and with clients. We call this group of servers *healthy*.

The safety property in Raft is guaranteed by the Leader Completeness Property [14]: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.

Liveness is guaranteed by Raft's rules. Typically, the number of servers in systems using consensus protocols, N, is odd. Assume that $N = 2F+1$, then Raft can tolerate any F failures. We define a consensus protocol's *liveness level* as the number of failures that it can tolerate, so Raft has an F liveness level. Higher liveness level means better liveness. No protocol can reach an $(F+1)$ liveness level. If there exists a protocol with an $(F+1)$ liveness level, there can be two split groups of F healthy servers and these two groups can agree on different contents respectively, which is against the safety property.

Safety and liveness are the most important properties of consensus protocols. Raft can guarantee that the safety property always holds and it also reaches the highest possible liveness level F. Furthermore, Raft has been proved to be a good foundation for system building. According to these properties, we choose Raft as the basis to design our new protocol CRaft.

2.2 Erasure Coding

Erasure coding is a common technique to tolerate faults in storage systems and network transmissions. A large number of codes have been put forward, but Reed-Solomon (RS) codes [16] are the most commonly used ones. There are two configurable positive integer parameters in RS codes, k and m.

In this technique, data are divided into k fragments with equal sizes. Then, using these k original data fragments, m parity fragments can be computed by an encoding procedure. So there will be $(k+m)$ fragments generated from the original data. The magic of a (k,m)-RS code is that any k out of total $(k+m)$ fragments are enough to recover the original data, and that is how RS codes tolerate faults.

When a consensus protocol is applied, the number of the servers, N, is usually fixed. If each server only stores one fragment produced by a (k,m)-RS code whose parameters k and m are subject to $k+m=N$, storage and network costs can be reduced to $1/k$ compared to full-copy replication. However, how to guarantee the safety property and keep liveness as good as possible cannot be ignored.

2.3 RS-Paxos

Combining erasure coding and Paxos, RS-Paxos is a reform version of Paxos which can save storage and network costs. In Paxos, commands are transferred completely. However, commands are transferred by coded-fragments in RS-Paxos. According to this change, servers can store and transfer only fragments in RS-Paxos, so storage and network costs can be reduced. The complete description of RS-Paxos can be found in the RS-Paxos paper [13].

To guarantee safety and liveness, Paxos and Raft are based on the inclusion-exclusion principle as follows.

$$|A \cup B| = |A| + |B| - |A \cap B| \tag{1}$$

The inclusion-exclusion principle guarantees that there is at least one server in two different majorities of servers,[1] and then the safety property can be guaranteed.

The insight of RS-Paxos is to increase the size of the intersection set. Specifically, after choosing a (k,m)-RS code, the read quorum Q_R, the write quorum Q_W, and the number of the servers N, should fit the following formula.

$$Q_R + Q_W - N \geq k \tag{2}$$

Then if a command is *chosen* (like *committed* in Raft), at least Q_W servers have accepted it. If a server wants to propose its own command, it will contact at least Q_R servers in *Prepare* phase. Because of (2) and (1), at least k among this Q_R servers have a fragment of the chosen command. So the proposer can recover the original command by using the k fragments and then it proposes the chosen value rather than its own.

With the benefits of erasure coding, using RS-Paxos can greatly reduce storage and network costs when $k > 1$. However, RS-Paxos decreases the fault tolerance number of failed servers. As Theorem 1 shows, RS-Paxos's liveness cannot be as good as Paxos or Raft.

Theorem 1. *Liveness level of RS-Paxos, L_{RSP}, is always less than F when $k > 1$.*

[1]If $|A| > |A \cup B|/2$ and $|B| > |A \cup B|/2$, $|A \cap B| = |A| + |B| - |A \cup B| > 0$.

Proof. RS-Paxos works only if at least $\max\{Q_R, Q_W\}$ servers are alive, so $L_{\text{RSP}} \leq N - \max\{Q_R, Q_W\}$.

According to (2), we have

$$\max\{Q_R, Q_W\} \geq (Q_R + Q_W)/2 \geq (N+k)/2.$$

Therefore, $L_{\text{RSP}} \leq N - (N+k)/2 = F - (k-1)/2 < F$. □

RS-Paxos roughly solves the consensus problem with erasure coding, but it cannot reach an F liveness level any longer as Theorem 1 shows. RS-Paxos requires more healthy servers than Paxos or Raft to function. It is important to present a consensus protocol that not only supports erasure coding but also possesses the same liveness level as Paxos and Raft.

3 CRaft, a Reform Version of Raft that Supports Erasure Coding

Liveness is one of the most important properties of consensus protocols. However, the previous erasure-coding-supporting protocol, RS-Paxos, fails to reach an F liveness level. Thus, our goal is to design a new erasure-coding-supporting protocol (so it can save storage and network costs) that possesses an F liveness level. This new protocol is based on Raft, so it inherits the basic concepts in Raft.

To reduce network cost, leaders in the new protocol should be able to replicate their log entries to followers by using coded-fragments, like RS-Paxos. However, as Theorem 1 shows, a protocol with only coded-fragment replication method cannot reach an F liveness level. In fact, Theorem 2 shows that the complete-entry replication method in Raft is necessary for an F liveness level protocol.

Theorem 2. *When there are only $(F+1)$ healthy servers in an F liveness level protocol, an entry **e** can be committed only after the complete entry has been stored in all $(F+1)$ healthy servers' logs.*

Proof. If a healthy server S did not store complete entry $\boldsymbol{e}$ when $\boldsymbol{e}$ was committed, the protocol could not guarantee that it could work fully functionally in any $(F+1)$ healthy servers. Suppose only S and the previous unhealthy servers were healthy at the next moment, these $(F+1)$ currently healthy servers could not recover complete $\boldsymbol{e}$, then the protocol had to wait for other servers. So when $\boldsymbol{e}$ was committed, all $(F+1)$ healthy servers had this complete entry. □

Both coded-fragment replication and complete-entry replication are required in our new protocol. Using coded-fragment replication can save storage and network costs, while complete-entry replication can keep liveness.

Next we will discuss the details of these two replication methods, and then we try to integrate them into a complete protocol, CRaft. To explain the details of CRaft, we first define some parameters. We assume that there are $N = (2F+1)$

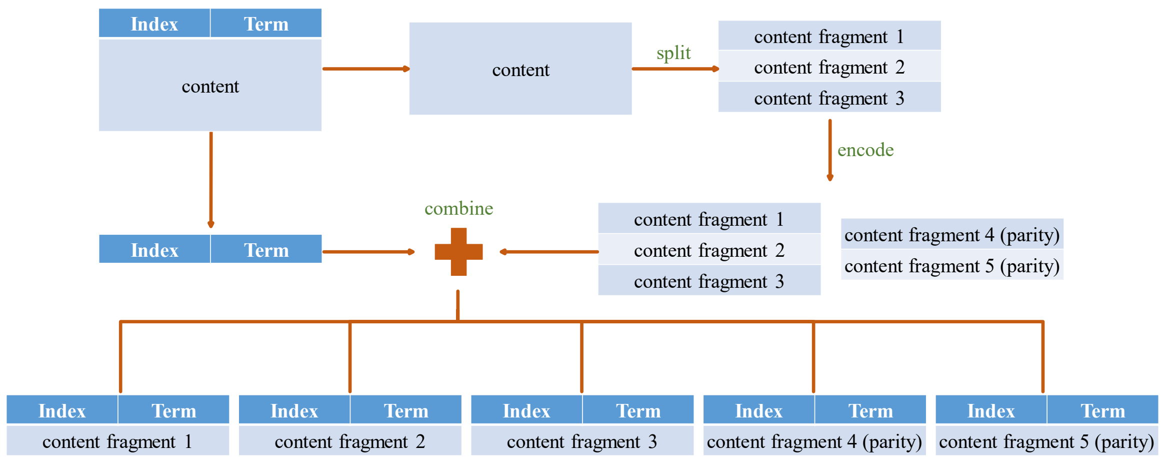

Figure 2: The encoding procedure in CRaft.

Table 1: Comparisons among Different Protocols

Performance Indicators	**Different Protocols**		
	CRaft	***Raft***	***RS-Paxos***
storage cost	$\mathbf{2F/k+1}$	$2F+1$	$\mathbf{2F/k+1}$
network cost	$\mathbf{2F/k}$	$2F$	$\mathbf{2F/k}$
disk I/O	$\mathbf{2F/k+1}$	$2F+1$	$\mathbf{2F/k+1}$
liveness level	$\boldsymbol{F}$	$\boldsymbol{F}$	$F-(k-1)/2$

servers in the protocol. Since CRaft should have the same availability as Raft, its liveness level should be F, which means that CRaft can still work when at least $(F+1)$ servers are healthy. We choose a (k,m)-RS code for CRaft. k and m should satisfy $k+m=N$, so each server in the protocol can correspond to one coded-fragment for each log entry. As Table 1 shows, CRaft supports erasure coding so it can save storage and network costs, while it possesses an F liveness level at the same time.

3.1 Coded-fragment Replication

When a leader in CRaft tries to replicate an entry by coded-fragment replication method, it first encodes the entry. In Raft, each log entry should contain its original content from clients and also its term and index in the protocol. When a CRaft leader tries to encode an entry, the content can be encoded into $N=(k+m)$ fragments by the (k,m)-RS code that the protocol chooses. Term and index should not be encoded, since they play important roles in the protocol. Figure 2 shows the encoding procedure.

After encoding, the leader will have N coded-fragments of the entry. Then it will send the corresponding coded-fragment to each follower. After receiving its corresponding coded-fragment, each follower will reply to the leader. When the leader confirms that at least $(F+k)$ servers store a coded-fragment, the entry and its previous ones can be safely applied. The leader will commit and apply these entries, and then inform followers to apply them. The commitment condition of coded-fragment replication is stricter than Raft's. This commitment condition also implies that a leader cannot use coded-fragment replication to replicate an entry and then commit it when there are not $(F+k)$ healthy servers.

When a leader is down, a new leader will be elected. If an entry is already committed, the election rule of Raft guarantees that the new leader at least has a fragment of the entry, which means the safety property can be guaranteed. Since at least $(F+k)$ servers store a fragment of a committed entry, there should be at least k coded-fragments in any $(F+1)$ servers.[2] So the new leader can collect k coded-fragments and then recover the complete entry when there are at least $(F+1)$ healthy servers, which means liveness can be guaranteed.

Figure 3 shows an example of coded-fragment replication and explains why the commitment condition becomes stricter in this replication method. If a leader can commit an entry when it only confirms that $F+1=4$ servers store the entry, new leaders may be unable to recover committed entries. Like the Index 3 entry in Figure 3, it should not be committed because only five servers stored it.[3] If it was committed, consider the situation that first three servers could not connect to other servers while other four servers were all healthy. CRaft should still be able to work because its liveness level is $F=3$. However, there were at most two fragments of the Index 3 entry in the healthy servers, so new leaders were not able to recover the complete entry. The protocol had to wait for the first three servers, which means liveness cannot be guaranteed.

In coded-fragment replication, followers can receive and

[2] According to (1), the number of the servers storing a fragment of a committed entry is at least $(F+k)+(F+1)-N=2F+1-N+k=k$.

[3] The entry can be committed only if at least $(F+k)$ servers store it. Since $F+k=3+3>5$, the Index 3 entry should not be committed.

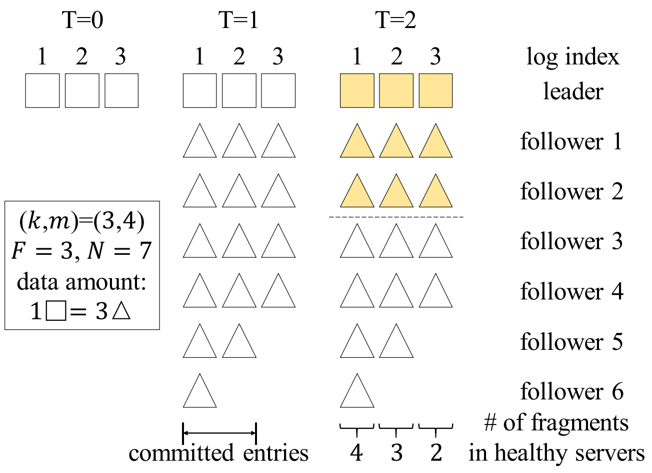

Figure 3: An example of coded-fragment replication. A square represents a complete entry, while a triangle represents a fragment of an entry. Yellow shadow means that the corresponding servers of the entries were not healthy. At $T = 0$, the leader got three entries and it tried to replicate them. At $T = 1$, followers received entry fragments with varying degrees of success. At $T = 2$, three servers, including the leader, failed.

store coded-fragments of entries. However, in Raft, followers must receive and store complete entries. According to the encoding procedure, the size of coded-fragments are about $1/k$ of the size of complete entries. So storage and network costs can be greatly reduced when using coded-fragment replication.

3.2 Complete-entry Replication

To reduce storage and network costs, leaders are encouraged to use coded-fragment replication. However, coded-fragment replication will not work when there are not $(F+k)$ healthy servers. When the number of healthy servers is greater than F and less than $(F+k)$, the leader should use complete-entry replication method to replicate entries.

In complete-entry replication, the leader has to replicate the complete entry to at least $(F+1)$ servers before committing the entry, just like Raft. Since the committing rule is the same as Raft, safety and liveness are not problems. However, since CRaft supports coded-fragments, the leader can replicate an entry by coded-fragments rather than the complete entry to remaining followers after committing the entry.

In practical implementations, there are many strategies to replicate an entry via complete-entry replication. Define an integer parameter $0 \le p \le F$. The leader can first send complete copies of an entry to $(F+p)$ followers and then send coded-fragments to remaining $(F-p)$ followers. A smaller p means less storage and network costs, but it also means a higher probability to have longer committing latency (if no F out of $(F+p)$ answers return in time, more rounds of communications may be required before commitment). When $p = F$, the strategy becomes the same as Raft's replication method. Figure 4 shows different strategies when $p = 0, 1, F$. In our implementation for experiments, we choose $p = 0$.

3.3 Prediction

Using coded-fragment replication rather than complete-entry replication can achieve better performance, if both methods can replicate successfully. A greedy strategy is that the leader always tries to replicate entries by coded-fragment replication. If it finds out that there are not $(F+k)$ healthy servers, it turns to replicate the entry by complete-entry replication. However, if the leader already knows that the number of healthy servers is less than $(F+k)$, the first replication attempt via coded-fragments is meaningless.

Choosing the replication method accurately can reach the best performance. However, the leader cannot be sure about the status of other servers. So it can only predict how many healthy servers it could communicate with when it tries to replicate an entry. The leader can use the most recent heartbeat answers to estimate the number of healthy servers. This prediction should be accurate enough.

When a leader tries to replicate an entry, it should use this prediction method to determine how to replicate. If the number of most recent heartbeat answers are not less than $(F+k)$, the leader should use coded-fragment replication first, and then it tries complete-entry replication if coded-fragment replication does not work. Otherwise, the leader directly uses complete-entry replication. Figure 5 concludes this process.

It is worth noting that this prediction is independent of the method that the leader chose to replicate last entry. It only relies on the most recent heartbeat answers. So it is quite possible that a leader used complete-entry replication to replicate the last entry and then it automatically chose coded-fragment replication to replicate a new entry.

3.4 Newly-elected Leader

Both replication methods can guarantee safety and liveness when leaders have all complete entries. However, when a leader is newly elected, it is likely that the newly-elected leader's log does not have complete copies but only coded-fragments of some entries. These incomplete entries are not guaranteed recoverable when there are only $(F+1)$ healthy servers. If some of these unrecoverable entries have not been applied by the newly-elected leader, the leader has no way to deal with these entries. The leader cannot send *AppendEntries* RPCs containing any one of these entries to the followers who need them,[4] so these unrecoverable entries will retain unapplied. According to the rules of Raft, the leader's new entries received from clients cannot be replicated to the followers as

[4]CRaft inherits Raft's RPCs [14], the only difference between their RPCs is that entries can be encoded in CRaft's *AppendEntries* RPC.

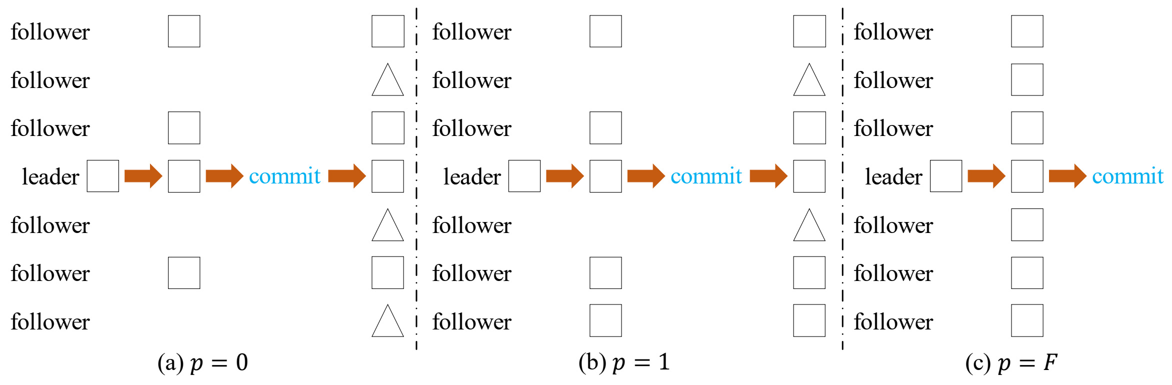

Figure 4: Examples of complete-entry replication with parameter $p = 0, 1, F$ when $N = 7$. A square represents a complete entry, while a triangle represents a fragment of an entry.

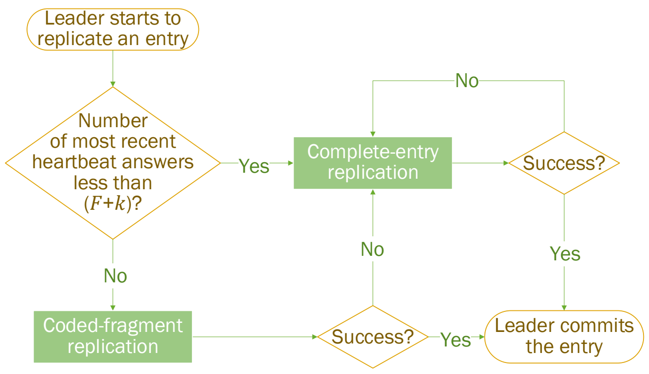
Leader starts to replicate an entry
Number of most recent heartbeat answers less than $(F+k)$?
Yes
No
Complete-entry replication
Success?
No
Yes
Coded-fragment replication
Success?
Yes
No
Leader commits the entry

Figure 5: Flow chart of log entry replication.

well. So the protocol fails to function fully and the protocol's liveness property cannot be guaranteed. Therefore, some extra operations are required to guarantee liveness.

The coded-fragments in the newly-elected leader's log can be applied or unapplied by the leader. If a coded-fragment is applied, the entry must have been committed by a previous leader. According to the commitment condition of two replication methods, at least k coded-fragments or one complete copy of the entry are stored in any $(F+1)$ servers. So the leader can always recover this entry when there are $(F+1)$ healthy servers. However, if a coded-fragment is unapplied, no rules can guarantee that this entry can be recovered when there are $(F+1)$ healthy servers.

To deal with unapplied coded-fragments, newly-elected leaders in CRaft should do the *LeaderPre* operation, before they can become fully-functioned leaders.

When a leader is newly-elected, it first checks its own log, finds out its unapplied coded-fragments. Then it asks followers for their own logs, focusing on the indexes of the unapplied coded-fragments. At least $(F+1)$ answers (including the new leader itself) should be collected or the new leader should keep waiting. The new leader should try to recover its unapplied coded-fragments in sequence. For each entry, if there are at least k coded-fragments or one complete copy in $(F+1)$ answers, it can be recovered, but not allowed to be committed or applied immediately. Otherwise, the new leader should delete this entry and all the following ones (including complete entries) in its log. After recovering or deleting all the unapplied entries, the whole *LeaderPre* operation can be done. During *LeaderPre*, the newly-elected leader should keep sending heartbeats to other servers, preventing them from timing out and starting new elections.

Figure 6 shows examples of *LeaderPre*. In Figure 6, $N = 5$ and $k = 3$. S1 committed the first two entries and then crashed, and other servers had only applied the first entry. S2 was elected as a new leader, and it would do *LeaderPre*. It first asked followers about the entries in Index 2 and Index 3. In Figure 6(a), after receiving answers from itself, S3 and S4, it tried to recover the two entries. There were three fragments of the Index 2 entry and two fragments of the Index 3 entry. So S2 should recover the Index 2 entry and delete the Index 3 entry. While in Figure 6(b), S3, S5 and S2 itself all had the Index 2 entry and the Index 3 entry. So S2 could recover both of them. Though the uncommitted Index 3 entry would be handled differently if S2 collected answers from different groups of servers, the committed Index 2 entry would be guaranteed to be recovered by *LeaderPre*.

After adding *LeaderPre*, the Leader Append-Only Property in Raft has an exception: deletion in *LeaderPre*. In original Raft, the original Leader Append-Only Property is the key to prove safety, so it is necessary to prove that *LeaderPre* will not harm safety. The proof can be found in Section 4.

There are two major reasons that leaders in original Raft do not delete entries. First, leaders have no way to find out whether an unapplied entry was committed by old leaders or not. Second, even though an entry is unapplied, leaders can still replicate it to followers since it has the entry's complete copy, so there is no need to delete it. In CRaft, if there are enough fragments of an unapplied entry, the new leader can recover it and be able to replicate it. Otherwise, the new leader can conclude that this entry is uncommitted. Unrecoverable

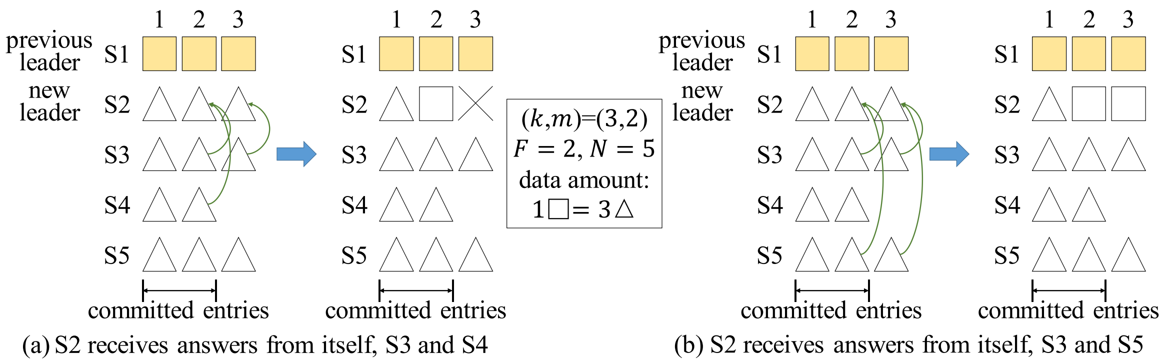

Figure 6: Examples of *LeaderPre*. A square represents a complete entry, while a triangle represents a fragment of an entry.

entries may harm CRaft's liveness, but unrecoverable also means uncommitted, so it is reasonable to delete them.

Based on Raft, CRaft provides two different replication methods for supporting erasure coding while keeping liveness. A prediction based on the most recent heartbeat answers helps the leader to choose replication method. In addition, to guarantee liveness, *LeaderPre* can help newly-elected leaders deal with unapplied coded-fragments.

3.5 Performance

The advantages of CRaft are shown in Table 1. Using a (k,m)-RS code, CRaft has advantages in reducing storage and network costs. In CRaft, ideally, only coded-fragments are needed to be transferred between the leader and followers, which indicates that the network cost can be saved to $1/k$. With this huge saving, CRaft can reach a much shorter latency and a higher throughput compared to original Raft.

The major difference between CRaft and RS-Paxos is liveness. To tolerate F failures, CRaft only needs to deploy $(2F+1)$ servers. However, RS-Paxos needs to deploy at least $(2F+3)$ servers. With the same parameter k in erasure coding, less servers required means that CRaft can save more storage and network costs compared to RS-Paxos.

One of the major concerns is the extra consumption when a leader is newly-elected. The new leader has to collect entry fragments if there are some behind followers. However, storage and network costs of CRaft in the worst situations are basically the same as Raft in any situations. Also, in most cases, the first new leader can replicate the old entries to all behind followers, so each entry only needs to be collected once extra. This harms the performance a little, but network cost is still greatly reduced generally, compared to Raft.

LeaderPre latency may affect election time, so it may affect the protocol's availability. This kind of latency is possibly affected by the number of the new leader's unapplied entries. However, a new leader can get brief information of its unapplied entries first and then collect them later. The time consumption of communicating brief information is quite short so that *LeaderPre* latency will not harm the protocol's availability seriously.

It is optional that a newly-elected leader first collect the whole state machine by communicating with its healthy followers. This operation is helpful to reduce read latency in the future, while it may significantly increase election time so that it may harm the protocol's availability. So there is a trade-off between using it or not.

If there are far behind followers, we recommend that the followers should catch up with the leader entry by entry when they become healthy again. Snapshots can be used to compact logs in CRaft. However, the deployment of snapshots can be much more complex than original Raft, since different servers store different fragments in CRaft.

Encoding time can be a problem too. However, many studies showed that encoding time is short enough compared to transfer time in practical systems [6]. It is worth having a slightly longer encoding time to reduce network cost.

4 Safety Argument

The key of safety in Raft is the Leader Completeness Property. Since we add a new operation *LeaderPre* in CRaft, we have to prove that the property still holds. First we give the proofs of the Log Matching Property and its two related lemmas, then we use them to prove the Leader Completeness Property.

Lemma 1. *A server S has a log entry* ***e****, and* ***e*** *was first added into the protocol in Term T, then* ***e*** *and its previous entries in S's log now are the same as the entries in leader$_T$'s log when* ***e*** *was first added into the protocol.*

Proof. Guaranteed by contents in *AppendEntries* RPC [14]. Noticing deletions in *LeaderPre* always delete the newest part in a log, this Lemma can be proved by the same induction technique in the Raft paper [14]. □

Theorem 3. *Log Matching Property: if two logs contain an entry with the same index and term, then the logs are the same in all entries up through the given index.*

Proof. As Lemma 1 holds, these two logs are the same as the leader's log when the entry was first added into the protocol. □

Lemma 2. *A server S has a log entry **e**, then entries with the same term and smaller index are in S's log.*

Proof. A leader cannot add entries to its log until *LeaderPre* is done. When ***e*** was accepted, entries with the same term and smaller index must be in the leader's log. According to Lemma 1, Lemma 2 holds. □

Theorem 4. *Leader Completeness Property: if a log entry **e** is committed in a given term (Term T), then **e** will be present in the logs of the leaders for all higher-numbered terms, and **e** will not be deleted in any higher-numbered term's LeaderPre.*

Proof. We assume that the Leader Completeness Property does not hold, then we prove a contradiction. Since indexes are positive integers, there is a log entry ***e*** with a smallest index that breaks the property.

Consider two kinds of events: one, Leader$_U$ $(U > T)$ does not have ***e*** at the time of its election; and two, ***e*** is deleted in *LeaderPre* by Leader$_U$ $(U > T)$.

Assume that event one first appears. According to assumption, leaders between Term T and Term U had ***e*** at the time of their own elections, and ***e*** was never deleted in *LeaderPre*. So ***e*** was never deleted from anyone's log since Term T. Leader$_T$ replicated ***e*** on at least $(F+1)$ servers (no matter which replication method Leader$_T$ used), and Leader$_U$ received votes from at least $(F+1)$ servers. Since $(F+1)+(F+1) = N+1 > N$, at least one server both accepted ***e*** from Leader$_T$ and voted for Leader$_U$. This server must have accepted ***e*** from Leader$_T$ before voting for Leader$_U$, otherwise it would reject Leader$_T$'s *AppendEntries* request. Since ***e*** was never deleted since Term T, this voter had ***e*** and voted for Leader$_U$ at the same time. So Leader$_U$'s log must have been as up-to-date as the voter's. If the voter and Leader$_U$ shared the same last log term, then Leader$_U$'s log must have been at least as long as the voter's. According to Lemma 2, Leader$_U$'s log must have ***e*** and this is a contradiction. Otherwise, Leader$_U$'s last log term must have been larger than the voter's. Since ***e*** was in the voter's log, Leader$_U$'s last log term, P, was larger than T. According to assumption, in Term P, Leader$_P$'s log had ***e***. According to Lemma 1, Leader$_U$'s log must have ***e*** and this is a contradiction.

So event two must appear earlier than event one. According to assumption, leaders after Term T had ***e*** at the time of their own elections, and ***e*** was never deleted in *LeaderPre* before. So ***e*** was never deleted from anyone's log since Term T. Since ***e*** was deleted in *LeaderPre*, there was an unrecoverable entry $\boldsymbol{e}_1$. If $\boldsymbol{e}_1$ was not ***e***, since ***e*** was deleted, the index of $\boldsymbol{e}_1$ must be smaller than ***e***'s. Because ***e*** was committed by Leader$_T$, and $\boldsymbol{e}_1$ had a smaller index than ***e***, so $\boldsymbol{e}_1$ had been committed. Then $\boldsymbol{e}_1$ broke the Leader Completeness Property and had a smaller index than ***e***, this is a contradiction. So ***e*** was deleted because it was unrecoverable. In Term T, Leader$_T$ replicated ***e*** to at least $(F+1)$ servers by complete copies, or at least $(F+k)$ servers by coded-fragments. Since ***e*** was never deleted from anyone's log since Term T, According to (1), there were at least one complete copy or k coded-fragments in any $(F+1)$ answers. Then ***e*** was recoverable and this is a contradiction.

Then the contradiction is completely proved. The Log Matching Property guarantees that future leaders will also contain entries that are committed indirectly (not by its term's leader). So, the Leader Completeness Property holds. □

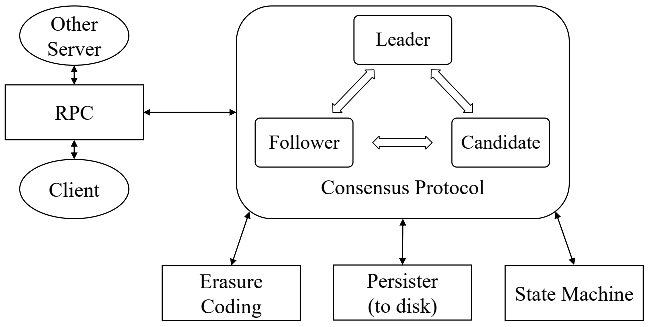

Figure 7: The structure of each server in our key-value store.

After proving the Leader Completeness Property, we can conclude the State Machine Safety Property effortlessly.

Theorem 5. *State Machine Safety Property: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.*

Proof. Suppose T is the lowest term in which any server applies an entry at the given index i. If a server applied an entry at Index i in Term U, the entry's term must be the same as the term of the Index i entry in Leader$_U$'s log. According to the Leader Completeness Property, the term of the Index i entry in Leader$_U$'s log should be identical to the term of the Index i entry in Leader$_T$'s log. Since T is constant when i is given, the State Machine Safety Property holds. □

5 Experiments and Evaluation

To evaluate our protocol, we first designed a key-value store based on Raft. Then we modified it to adapt CRaft. Since RS-Paxos is based on Paxos but not Raft, it is difficult to compare RS-Paxos with CRaft or Raft directly. We took the insight of RS-Paxos and implemented an equivalent protocol named *RS-Raft* onto our key-value store. The parameters in RS-Raft have the same meanings as the ones in RS-Paxos, which are described in Section 2.3. We ran experiments on the key-value store with different protocols to present an evaluation.

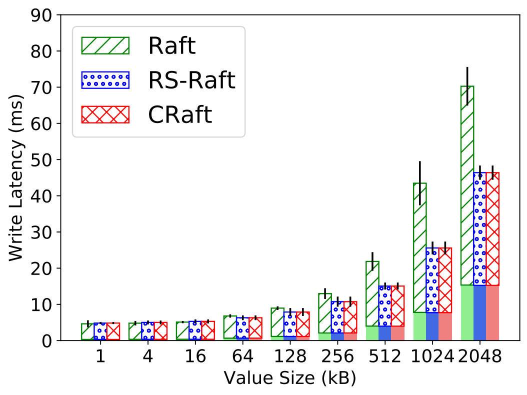

Figure 8: Latency in different value sizes when $N = 5$.

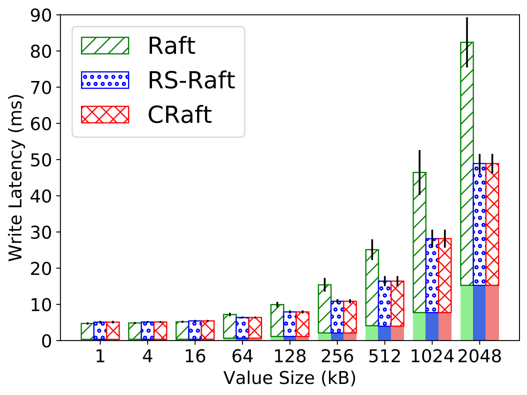

Figure 9: Latency in different value sizes when $N = 7$.

5.1 Key-value Store Implementation

The key-value store we design supports three kinds of operations: *Set*, *Append* and *Get*. *Set* and *Append* operations must be logged, while *Get* operations are not. The keys were accessed uniformly in our experimental workloads. Followers can just store fragments of their entries to reduce storage cost. However, the leader should keep complete copies of entries to ensure performance of *Get*. After a new leader is elected, if there is a *Get* operation and the new leader only has a fragment of the data, it should first force at least $(k-1)$ followers' log to catch up with its own, then collect enough data fragments from them and decode the data. If the leader can directly respond to client's *Get*, we call this operation a *fast read*. Otherwise, if the leader should collect fragments from followers first, we call this operation a *recovery read*.

We used C++ to implement our key-value store. The structure of each server in our key-value store is shown in Figure 7. The consensus protocol can be Raft, CRaft and RS-Raft. We used RCF 3.0 [18] to implement RPC, and we chose TCP as transmission protocol. Jerasure 2.0 [15] is the library that we used for erasure coding.

5.2 Setup

We ran experiments on the configurations of $N = 5$ and $N = 7$, which are reasonable choices when using consensus protocols supporting erasure coding. k was set to 3, so the erasure code we used is a $(3,2)$-RS code (when $N = 5$) or a $(3,4)$-RS code (when $N = 7$).

In $N = 5$ configuration, $F = 2$, so Raft and CRaft can tolerate any two failures. We chose $Q_R = Q_W = 4$ for RS-Raft, so it can tolerate one failure. In $N = 7$ configuration, $F = 3$, so Raft and CRaft can tolerate any three failures. We chose $Q_R = Q_W = 5$ for RS-Raft, so it can tolerate two failures.

Our experiments were run on Amazon EC2 platform. We used six (when $N = 5$) or eight (when $N = 7$) instances, one of them played the role of clients and the other instances were servers. Each instance has two virtual CPU cores and 8 GiB memory. The network bandwidth of each instance is about 550 Mbps. The storage devices we used are Amazon EBS General Purpose SSDs, each with 80000 IOPS and 1750 MB/s throughput.

5.3 Evaluation

We evaluated the protocols by measuring write latency, write throughput, network cost, liveness level and recovery read latency. Each experiment is repeated at least 100 times.

5.3.1 Latency

Figure 8 and Figure 9 show commitment latency of various value-sized write requests with error bars. Operations with a value size that larger than 2 MB can be solved by splitting it into multiple *Append* operations. Each latency consists of two parts. The part at the bottom with shadow in Figure 8 and Figure 9 is communication time from clients to the leader. This part of time is only influenced by value size. The other part is latency from the moment that the leader starts the entry to the moment that the leader commits it, and it is the part that we focus.

When value size is lower than 128 kB, three protocols perform evenly. In these situations, latency is mainly dominated by disk I/O. Since data amount is too small, even though CRaft and RS-Raft can save the amount of data flushed to disks, the I/O time usage remains almost the same, so there is not much difference between these protocols on latency.

When value size becomes larger, the advantage of CRaft and RS-Raft can be revealed. Network traffic and disk I/O both affect latency. Since CRaft and RS-Raft save network cost and disk I/O greatly, they reduce 20%–45% of latency compared to Raft when $N = 5$, and 20%–60.8% when $N = 7$.

5.3.2 Throughput

Since CRaft and RS-Raft can save the amount of data transferred and flushed to disks, they are expected to have bet-

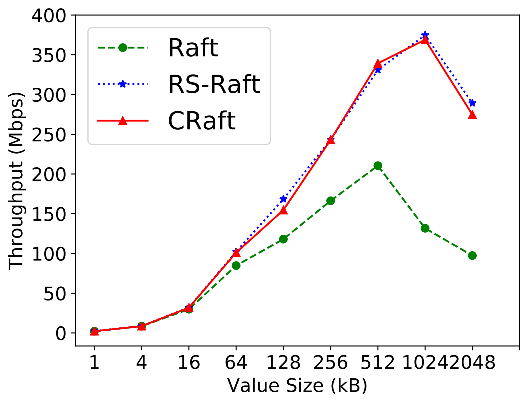

Figure 10: Throughput in different value sizes when $N = 5$.

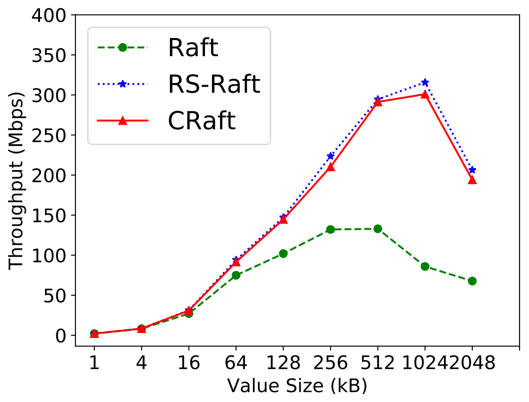

Figure 11: Throughput in different value sizes when $N = 7$.

ter throughput than Raft. We simulate the situation that 100 clients raise write request, and evaluate throughput of the leader. Figure 10 and Figure 11 show the experiment results.

The results show that CRaft and RS-Raft can improve throughput compared to Raft. They can reach about 150%–200% improvements when value size is relatively large.

With value size grows larger, throughput first increases and reaches a peak, then it will fall. Throughput will fall because of network congestion. How to prevent this network congestion problem is interesting, but it is not our concern in this paper. We compare the peak throughput of these three protocols. CRaft and RS-Raft can have a 180% improvement on write throughput when $N = 5$ and 250% when $N = 7$. Also, the throughput peaks of CRaft and RS-Raft both appear much later than Raft's. This is another advantage of CRaft and RS-Raft because of their reductions on network cost.

RS-Raft's throughput can be slightly better than CRaft's when the numbers of servers are equal, because more *AppendEntries* replies are needed before commitment in CRaft. However, it is unfair to compare these two protocols' throughput in such way, since RS-Raft's liveness is worse than CRaft's. To tolerate two failures, seven servers are required when using RS-Raft, while only five servers are required when using CRaft. So it is fairer to compare RS-Raft's throughput when $N = 7$ with CRaft's throughput when $N = 5$. According to Figure 10 and Figure 11, in this comparison, CRaft has an advantage.

5.3.3 Network Cost

We monitored the amount of data transferred from the leader to directly prove that our protocol can save network cost. In this experiment, clients raised a write request every 70 ms. Figure 12 shows the monitoring results when $N = 7$. The leader in Raft transfers about 250% of data amount compared to the leader in CRaft. This result directly proves that CRaft can greatly reduce network cost. However, ideally, when $k = 3$, the ratio between the amount of data transferred from a Raft leader and a CRaft leader should be close to 300%. The gap between 250% and 300% may be caused by costs that are not generated by the consensus protocols.

5.3.4 Liveness

The major difference between CRaft and RS-Raft is liveness. CRaft can tolerate any two failures when $N = 5$, and it can tolerate any three failures when $N = 7$. Though we choose the parameters for RS-Raft to reach its highest possible liveness level, RS-Raft can only tolerate one failure when $N = 5$, and it can only tolerate two failures when $N = 7$.

Figure 13 shows the throughput of different protocols when the number of healthy servers changes in $N = 7$ experiments with error bars. RS-Raft performs very well when the number of healthy servers is no less than 5, but it cannot work when the number is 4. CRaft performs just like RS-Raft when the number of healthy servers is 6 or 7, while it performs worse than RS-Raft when the number is 5. This is because CRaft can only use complete-entry replication when the number of healthy servers is 5, so its throughput is degraded. However, CRaft can still work when the number of healthy servers is 4, just like Raft. And this proves CRaft's liveness advantage to RS-Raft.

5.3.5 Recovery Read

One of our concerns is that *recovery read* will take too much time compared to *fast read*. The leader in Raft always does *fast read*, but sometimes new leaders in CRaft may have to do *recovery read*. Noticing that a new leader only needs to do at most one *recovery read* to a specific key. If a leader needs to handle several *Get* operations of one key in its term, only the first time it has to do *recovery read*. So the existence of *recovery read* may not harm performance too much when servers do not crash too often. We made a new leader handle a *Get* operation in different protocols, and then we repeated this *Get* operation nine more times and calculated average latency. The results are shown in Figure 14. CRaft takes at

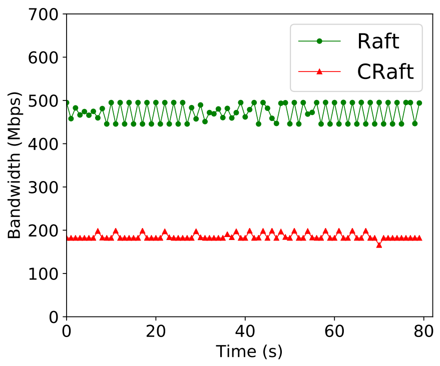

Figure 12: The leader's network consumption in Raft and CRaft when $N = 7$.

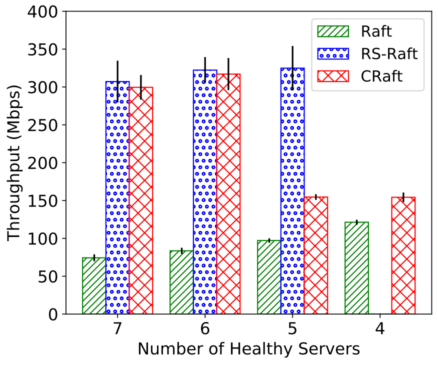

Figure 13: Throughput when the number of healthy servers changes in $N = 7$ experiments.

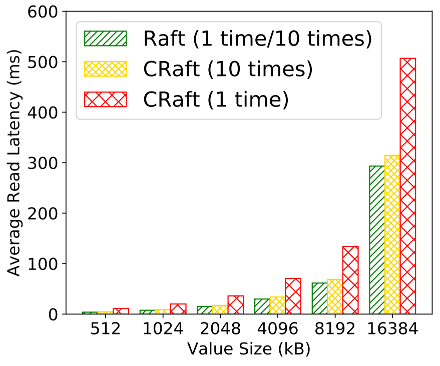

Figure 14: Average latency of *Get* operations when $k = 3$. Only the first *Get* operation harms performance in CRaft.

most 140% more time compared to Raft handling the first *Get* operation. However, time usages of ten operations between different protocols become close enough. So we can conclude that extra time usage of *recovery read* is acceptable.

6 Related Work

Many systems use consensus protocols to provide highly reliable and available distributed services. In early years, most of them use Paxos to achieve consistency, like Chubby [3] and Spanner [5]. After the presence of Raft, many systems are using it for understandability, such as etcd [8] and TiKV [1].

Recent years, consensus protocols are not only used to replicate small size database records, but also files and data objects. Using Paxos, Gaios [2] builds a high performance data store. To prevent the service from compromising availability, FSS [11] uses a pipelined Paxos to replicate both user data and metadata. Also, etcd and TiKV use Raft to consistently distribute user data to different servers. This kind of systems are target systems of CRaft.

Erasure coding is first developed in network transmission area and now it is applied in many distributed storage systems, such as Ceph [19], HDFS [17] and Microsoft Azure [10]. The most focus problem about erasure coding now is that its recovery cost is too high compared to simple replication, and there are many works trying to solve this problem [7, 10]. Our work does not focus on this area, but we have another contribution on erasure coding. The methods that most systems replicate erasure-coded fragments are similar to using the two-phase commit protocol [9]. This kind of methods have a high probability to fail in an asynchronous network, while CRaft can still work well in this situation.

RS-Paxos [13] is the first consensus protocol supporting erasure coding. However, it cannot reach the best liveness level and it misses important details to build a practical system. Our new protocol CRaft solves the above problems. Giza [4] uses metadata versioning to provide consistency for erasure coding objects. However, its method mainly focuses on safety and ignores liveness when transferring user data. Liveness can be optimized by using CRaft.

7 Conclusions

We presented CRaft, an erasure-coded version of Raft. CRaft is based on Raft while it extents Raft to support erasure coding. With the help of erasure coding, storage and network costs can be greatly reduced.

The previous erasure-coding-supporting protocol, RS-Paxos, fails to retain an F liveness level like Paxos or Raft. CRaft solves this problem. In other words, to tolerate F faults, CRaft only needs $(2F+1)$ servers while RS-Paxos needs more. So CRaft can save more storage and network costs.

We analyzed the performance of different protocols and we concluded that CRaft can reduce storage and network costs most while it has the best liveness. We designed a key-value store and ran experiments on it. The results show that CRaft can reduce 60.8% of latency and improve throughput by 250% compared to Raft. In the future, we will attempt to implement CRaft onto practical systems.

Acknowledgments

We thank all reviewers for their insightful comments, and especially our shepherd Darrell D. E. Long for his guidance during our camera-ready preparation. We also thank Yinxing Hou, Hongmin Lou and Rui Mao for helpful discussions. This work is supported by the National Key Research and Development Program of China (Grant No. 2016YFB1000303) and the Guangdong Province Key Research and Development Program of China (Grant No. 2018B010115002).

References

[1] TiKV Authors. TiKV: A distributed transactional key-value database. 2019. https://tikv.org/.

[2] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, pages 141–154, 2011.

[3] Michael Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 335–350, 2006.

[4] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. Giza: Erasure coding objects across global data centers. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*, pages 539–551, 2017.

[5] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8:1–8:22, 2013.

[6] Loic Dachary. Ceph Jerasure and ISA plugins benchmarks. 2015. https://blog.dachary.org/2015/05/12/ceph-jerasure-and-isa-plugins-benchmarks/.

[7] Alexandros G. Dimakis, Brighten Godfrey, Yunnan Wu, Martin J. Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, 2010.

[8] The etcd authors. etcd: A distributed, reliable key-value store for the most critical data of a distributed system. 2019. https://etcd.io/.

[9] Jim Gray. Notes on database operating systems: operating systems: an advanced course. *Lecture Notes in Computer Science*, 1979.

[10] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*, pages 15–26, 2012.

[11] Bradley C. Kuszmaul, Matteo Frigo, Justin Mazzola Paluska, and Alexander (Sasha) Sandler. Everyone loves file: File Storage Service (FSS) in Oracle Cloud Infrastructure. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, pages 15–32, 2019.

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

[13] Shuai Mu, Kang Chen, Yongwei Wu, and Weimin Zheng. When Paxos meets erasure code: Reduce network and storage cost in state machine replication. In *Proceedings of the 23rd International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC'14)*, pages 61–72, 2014.

[14] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference, (USENIX ATC '14)*, pages 305–319, 2014.

[15] James S. Plank and Kevin M. Greenan. Jerasure: A library in C facilitating erasure coding for storage applications version 2.0. Technical report, Department of Electrical Engineering and Computer Science, University of Tennessee, 2014. http://jerasure.org/jerasure-2.0/.

[16] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of The Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[17] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST2010)*, pages 1–10, 2010.

[18] Delta V Software. Remote call framework. 2019. http://www.deltavsoft.com/index.html/.

[19] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 307–320, 2006.