FlexRaft: Exploiting Flexible Erasure Coding for Minimum-Cost Consensus and Fast Recovery

Mi Zhang, Qihan Kang, and Patrick P. C. Lee

***Abstract*—Consensus protocols like Paxos and Raft provide data consistency and fault tolerance for distributed services. Log replication in these protocols can be supported by erasure coding, which incurs lower redundancy than full-copy replication and significantly saves network and storage costs for overall performance improvements. However, existing consensus protocols with erasure coding cannot achieve the minimum network and storage costs during log replication. We propose FlexRaft, which dynamically varies the coding scheme used in Raft based on the server status to always achieve the theoretically minimum redundancy ratio, while maintaining the same liveness as in Raft. To address the issue of an inconsistent coding scheme between the leader and its followers, we specify the prerequisite of overwriting a log entry and also allow the leader and its followers to exactly track the coding scheme being used. We further extend FlexRaft into FlexRaft+, which provides a different storage layout to vary the coding scheme through a novel technique called re-encoding-free replication, so as to enable fast server recovery. We prove that both FlexRaft and FlexRaft+ maintain Raft safety. We implement a prototype of FlexRaft and FlexRaft+, atop which we build a distributed key-value store to show its efficacy. Experiments on Alibaba Cloud show that FlexRaft achieves the theoretically minimum network and storage costs in practice, and reduces the commit latency by 44.51% and 19.37% compared with state-of-the-art CRaft and HRaft, respectively. FlexRaft+ further reduces the commit latency when the coding scheme is being varied and improves the server recovery performance.**

***Index Terms*—Consensus, erasure coding.**

I. Introduction

CONSENSUS protocols coordinate multiple servers to provide reliable distributed services [18], [19], [26]. As failures are commonplace in distributed environments (e.g., server crashes, network partitioning, and message loss), consensus protocols act as a foundation for building distributed systems with high availability and strong consistency. To protect distributed services against failures, consensus protocols typically replicate commands across multiple servers, so that distributed systems can operate correctly when a minority of servers fail. Paxos [18], [19] and Raft [26] are two widely adopted consensus protocols in practical distributed systems [2], [7], [12], [29].

To guarantee strong consistency, consensus protocols record data operations as log entries and replicate them to all servers in a group. Each server maintains a log consisting of a sequence of commands in the same order, such that the state machines running in different servers can execute the same commands and output the same results. To tolerate F server failures, consensus protocols need to replicate log entries in $N = (2F + 1)$ servers. This incurs $2F$ times the network traffic from the leader to its followers and N times the storage cost as each server stores a full copy of the log entries. The *redundancy ratio* (i.e., the actual data size divided by the original data size) of full-copy replication is N times. Thus, the high redundancy ratio hinders the distributed systems to achieve low latency and high throughput.

Erasure coding is a well-known low-redundancy approach to achieving fault tolerance in storage systems. Reed-Solomon (RS) codes [27] are the most popular erasure codes deployed in practice. An RS(k, m) code divides a data object into k fixed-size *data chunks*, and performs encoding to generate m *parity chunks*; note that RS(k, m) is equivalent to full-copy replication when $k = 1$. The $k + m$ *coded chunks* (i.e., data and parity chunks) are stored in $k + m$ servers. When a server failure occurs, any k out of the $k + m$ coded chunks suffice to reconstruct the original content. The redundancy ratio of RS(k, m) is $(k + m)/k$, which is only $1/k$ times compared to full-copy replication. Thus, erasure coding significantly reduces network and storage costs.

Recent studies apply erasure coding into consensus protocols to save network and storage costs for high performance [15], [25], [31]. RS-Paxos [25] is the first work that adopts erasure coding to replicate the log entries in Paxos. To maintain safety (i.e., never return an incorrect result), RS-Paxos stores the chunks of a log entry in at least $F + k$ servers, such that the original data can be recovered from the remaining servers under F server failures. RS-Paxos trades *liveness level* (i.e., the number of tolerable failures) for better performance. To maintain liveness level F as in the original Raft, CRaft [31] converts to full-copy replication when there are fewer than $(F + k)$ healthy

Manuscript received 19 January 2024; revised 21 June 2024; accepted 10 August 2024. Date of publication 14 August 2024; date of current version 30 August 2024. This work was supported in part by the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant XDB44030200, in part by the Major Research Plan of the National Natural Science Foundation of China under Grant 92270202, and in part by the Research Grants Council of Hong Kong under Grant GRF 14214622 and Grant AoE/P-404/18. Recommended for acceptance by J. Lofstead. *(Corresponding author: Patrick P. C. Lee.)*

Mi Zhang is with the State Key Lab of Processors, Research Center for Advanced Computer Systems, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100045, China (e-mail: zhangmi@ict.ac.cn).

Qihan Kang is with the State Key Lab of Processors, Research Center for Advanced Computer Systems, Institute of Computing Technology, Chinese Academy of Sciences, University of Chinese Academy of Sciences, Beijing 101408, China (e-mail: kangqihan17@mails.ucas.ac.cn).

Patrick P. C. Lee is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China (e-mail: pclee@cse.cuhk.edu.hk).

Digital Object Identifier 10.1109/TPDS.2024.3443424

TABLE I
PERFORMANCE OF CRAFT AND HRAFT USING DIFFERENT ERASURE CODING SCHEMES FOR A GROUP OF FIVE SERVERS ($N = 5$)

(a) No server failure ($f = 0$)

k	Log Replication		Network Cost		Storage Cost	
	CRaft	HRaft	CRaft	HRaft	CRaft	HRaft
1	full-copy	full-copy	4	4	5	5
2	RS(2,3)	RS(2,3)	2	2	3	3
★ 3	RS(3,2)	RS(3,2)	4/3	4/3	7/3	7/3

(b) One server failure ($f = 1$)

k	Log Replication		Network Cost		Storage Cost	
	CRaft	HRaft	CRaft	HRaft	CRaft	HRaft
1	full-copy	full-copy	3	3	4	4
★ 2	RS(2,3)	RS(2,3)	3/2	3/2	5/2	5/2
3	full-copy	RS(3,2)+	3	5/3	4	8/3

(c) Two server failures ($f = 2$)

k	Log Replication		Network Cost		Storage Cost	
	CRaft	HRaft	CRaft	HRaft	CRaft	HRaft
★ 1	full-copy	full-copy	2	2	3	3
2	full-copy	RS(2,3)+	2	2	3	3
3	full-copy	RS(3,2)+	2	2	3	3

servers. HRaft [15] optimizes CRaft by replenishing some coded chunks in several healthy servers instead of switching to full-copy replication directly.

However, existing erasure-coded consensus protocols do not specify the optimal *coding scheme* (i.e., the choices of k and m) to minimize both network and storage costs during log replication, thereby failing to achieve the lowest redundancy ratio. The redundancy ratio $(k + m)/k$ depends on the value of k, as the value of $k + m$ is equal to the number of servers in a group (i.e., N) where each server stores a coded chunk of a log entry. When all servers in a group function correctly, we can choose the largest available value of k for log replication to minimize the network and storage costs. However, a larger k also implies that more servers (i.e., $F + k$) are required to store coded chunks for entry commitment. If we use the coding scheme with the largest k for log replication at the beginning, the network and storage costs cannot stay at the minimum when server failures occur. In the presence of server failures, neither CRaft nor HRaft achieves the minimum redundancy ratio, as CRaft degrades to full-copy replication while HRaft needs to store additional coded chunks in the remaining servers. Thus, existing erasure-coded consensus protocols cannot maintain the lowest redundancy ratio during log replication for entry commitment all the time.

Our insight is that the optimal coding scheme for entry commitment in the consensus protocol has a different value of k depending on the number of healthy servers in a group. For example, as shown in Table I in Section II-C, the optimal coding scheme (marked with a ★) for a Raft group of five servers is RS(3,2) and RS(2,3) when there is zero and one server failure, respectively. We argue that the coding scheme should be *dynamically* varied based on the latest server status (which can be estimated by heartbeat messages between the leader and the followers in Raft [26]). In contrast, both CRaft and HRaft employ a fixed coding scheme during log replication regardless of the server status.

In this paper, we propose a flexible erasure coding approach for Raft called FlexRaft, which *provably* minimizes both network and storage costs to commit log entries. FlexRaft varies the coding scheme and maintains the lowest redundancy ratio in the presence of server failures. It sets k as large as possible based on the number of healthy servers in a group. However, the varying coding scheme requires overwriting the coded chunks of a log entry with the new ones, thereby raising new consistency issues between the leader and its followers: (i) for the same log entry, the chunk stored in a follower can be mistakenly overwritten by an old coded chunk, and (ii) some followers may not successfully update the old chunks before the log entry is committed. Thus, FlexRaft specifies the prerequisite of overwriting a log entry, and modifies the *AppendEntries RPCs* (i.e., the commands initiated by the leader to replicate log entries (with arguments) or provide heartbeats (without arguments)) to ensure that the chunks stored in the followers are encoded with the same coding scheme as the leader. We consider how to recover a failed server and prove the safety of FlexRaft. We summarize our contributions as follows.

- We analyze the optimal coding scheme for consensus protocols under a different number of server failures. We show that given N' healthy servers in a Raft group ($N' \leq N$), the coding scheme with $k = N' - F$ achieves the lowest redundancy ratio while keeping the liveness level F.
- We propose FlexRaft to dynamically vary the coding scheme used for log replication in the Raft protocol based on the server status. To guarantee that the leader and its followers use the same coding scheme for a log entry, FlexRaft adds a restriction rule to avoid mistakenly updating a coded chunk in a follower, and also updates the AppendEntries RPCs so that the leader can exactly track the coding scheme of the stored chunks.
- We further extend FlexRaft with re-encoding-free log replication, referred to as FlexRaft+. FlexRaft+ provides a different storage layout to vary the coding scheme, so as to enable fast server recovery. We prove that both FlexRaft and FlexRaft+ guarantee Raft safety while maintaining the same liveness level F as Raft.
- We implement FlexRaft in C++ and build a distributed key-value store using RocksDB [6] atop FlexRaft. Experiments on Alibaba Cloud [1] show that FlexRaft reduces the commit latency by 44.51% and 19.37%, respectively, compared with CRaft and HRaft under two server failures for a group of seven servers. FlexRaft+ further reduces the commit latency of FlexRaft by 21.71% when a server failure occurs.

The source code of our FlexRaft prototype is now available at: https://github.com/ACS-Storage-Group/FlexRaft-Code.

II. BACKGROUND AND MOTIVATION

A. Basics of Raft Consensus

We first provide the background details of Raft [26]. We consider a Raft group of $N = (2F + 1)$ servers that can tolerate any F server failures ($F \geq 1$). Each server is in one of the

following three states: *leader*, *follower*, and *candidate*. In a normal situation, there is one leader in a Raft group and the remaining servers are followers. The leader handles all client requests and replicates *log entries* (i.e., the operations being executed); the followers respond to the requests from the leader and candidates, and redirect the client requests to the leader; the candidate state is used to elect a new leader. Raft divides time into *terms*, numbered with consecutive integers, and each term starts with a *leader election*.

Raft adopts a strong form of leadership to simplify the management of log replication. The leader accepts log entries from clients and replicates them to other servers in the same Raft group by sending AppendEntries RPCs. Each log entry tracked by an *index* (a monotonically increasing number) stores an update to the state machine along with the term number when the leader receives the update. When a log entry is replicated to a majority of servers, the leader commits and applies the log entry and its previous log entries, and informs the followers to apply the log entries. Raft guarantees *leader completeness* property that the leader at any term has all committed entries in the previous terms.

B. Erasure-Coded Consensus Protocols

RS-Paxos [25] is the first protocol to combine erasure coding and consensus to reduce both network and storage costs. It divides the original data of a log entry into k chunks with equal sizes and encodes the k chunks into m parity chunks ($k > 1, m \geq 1$) using RS(k, m). It then sends one chunk to each acceptor for log replication. Any k chunks of data and parity chunks can reconstruct the original log entry. As the chunk size is a fraction (i.e., $1/k < 1$) of the total data size, RS-Paxos saves the network bandwidth cost and disk IOs. RS-Paxos is actually a superset of the vanilla Paxos [18], [19]. It requires that the intersection between the write quorum Q_w and read quorum Q_r should be not less than k to guarantee safety; that is, RS-Paxos should satisfy $Q_r + Q_w - N \geq k$ when using RS(k, m) for storing log entries, such that at least k chunks can be read from Q_r servers to recover the original data. With a larger k, it achieves more network and storage savings, but puts a higher requirement on the write and read quorum. Thus, compared to the vanilla Paxos using full-copy replication, RS-Paxos tolerates fewer failed servers ($< F$) in a group of $N = (2F + 1)$ servers.

CRaft [31] extends Raft with erasure coding like RS-Paxos, while maintaining the same *liveness level* (i.e., the number of failed servers tolerable by a protocol) as Raft. To address the liveness problem of RS-Paxos (i.e., tolerating fewer than F failures), CRaft uses erasure coding and full-copy replication jointly. When at least $F + k$ servers are running normally in a Raft group, CRaft uses RS(k, m) for log replication to reduce network and storage costs; otherwise, it switches to full-copy replication and keeps the liveness level F (i.e., CRaft switches to full-copy replication when the number of failed servers is larger than $N - F - k$). Although CRaft maintains the same liveness as the original Raft, switching to full-copy replication for log replication causes sharp performance degradation when the number of healthy servers reduces to less than $F + k$. Fig. 1 shows that CRaft switches from RS(3,2) to full-copy replication when one server fails in a group of five servers. Note that the leader in CRaft keeps a whole copy of each log entry for efficient reads (same for HRaft [15] as well). When a leader is newly elected in CRaft, the new leader performs a *LeaderPre* operation to deal with any *unapplied* coded chunk (i.e., not yet applied to the state machine) before the leader fully functions. During the LeaderPre operation, the leader attempts to recover any unapplied entries in sequence by collecting other coded chunks from its followers, and deletes the entries that cannot be recovered.

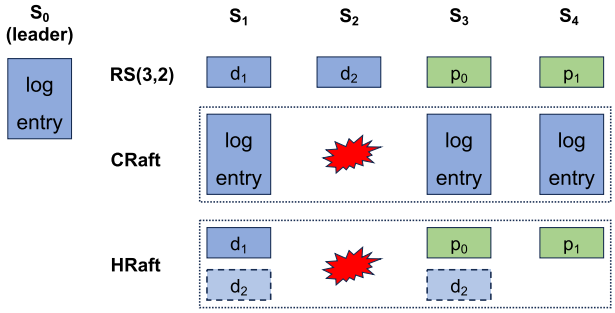


Fig. 1. Both CRaft and HRaft use RS(3,2) for log replication in a group of five servers ($N = 5$). When one server fails, CRaft switches to full-copy replication while HRaft replenishes the missing chunk.

HRaft [15] mitigates the sharp performance degradation in CRaft by replenishing some coded chunks in several healthy servers when failures occur. It adjusts the placement of the coded chunks and replicates some coded chunks to healthy servers instead of switching to full-copy replication if the number of failed servers is greater than $N - F - k$. When the leader receives p ($F \leq p \leq F + k - 1$) acknowledgments during log replication (i.e., there are $(N - 1 - p)$ failed servers), it chooses $(F + k - 1 - p)$ coded chunks and replicates them to F healthy servers before committing a log entry. In other words, some servers store multiple coded chunks of a log entry to guarantee safety (i.e., retrieving enough chunks for data reconstruction). Fig. 1 shows that HRaft stores the missing chunk in two servers under one server failure when $N = 5$. Although HRaft avoids switching to full-copy replication, HRaft cannot always maintain the minimum storage and network costs during log replication.

C. Motivating Example

We compare the network and storage costs of CRaft and HRaft with different erasure coding schemes under a different number of server failures. We use a group consisting of five servers ($N = 5$), which can tolerate at most two server failures ($F = 2$). The *network cost* is the bandwidth usage from the leader to its followers for a log entry to be committed. As the leader stores a full copy of the log, the *storage cost* is equal to the network cost plus one [31].

Table I shows the network and storage costs of CRaft and HRaft with different coding schemes where k ranges from 1 to 3. Here, we denote full-copy replication by $k = 1$. In the normal case that no server fails (i.e., $f = 0$), CRaft and HRaft introduce

the same network and storage costs when using the same coding scheme. Both CRaft and HRaft achieve the minimum network and storage costs with the largest k (i.e., $k = 3$) among all three coding settings. When using RS(3,2) for log replication, the network and storage costs are $4/3$ and $7/3$, respectively. If one server fails (i.e., $f = 1$), CRaft with $k = 3$ switches to full-copy replication as the number of healthy servers in the group is less than $(F + k)$ (i.e., 5). The network and storage costs of CRaft sharply increase to 3 and 4, respectively, the same as the original Raft. Under one server failure, HRaft with $k = 3$ still uses RS(3,2) while storing two coded chunks in two servers (denoted by RS(3,2)+). The network and storage costs of HRaft gradually increase to 5/3 and 8/3, respectively, lower than those of CRaft with $k = 3$ but higher than CRaft/HRaft with $k = 2$. For $k = 2$, both CRaft and HRaft can use RS(2,3) for log replication, which incurs the minimum network and storage costs (i.e., 3/2 and 5/2, respectively). When two servers fail (i.e., $f = 2$), CRaft with $k = 2$ and $k = 3$ switches to full-copy replication, while HRaft needs to store more coded chunks to keep the liveness level F. For CRaft and HRaft under two server failures, the network and storage costs are 2 and 3, respectively, equal to that of the original Raft. Thus, the largest k does not guarantee the minimum network and storage costs in all cases; for example, RS(2,3), rather than RS(3,2), incurs the lowest network and storage costs when one server fails. Although HRaft avoids switching to full-copy replication in the presence of server failures, it fails to achieve the minimum network and storage costs using a coding scheme with a fixed k.

III. DESIGN OF FLEXRAFT

FlexRaft dynamically adjusts the coding scheme for log replication according to the cluster status, so as to minimize both the network and storage costs while maintaining the liveness level F. We first describe the choice of coding schemes in FlexRaft (Section III-A). We then explain the issues raised by varying coding schemes and how FlexRaft addresses them (Section III-B). We further optimize FlexRaft (called FlexRaft+) to realize a re-encoding-free log replication under server failures (Section III-C). Finally, we introduce how FlexRaft and FlexRaft+ deal with server recovery (Section III-D) and provide the safety and liveness level guarantee (Section III-E).

A. Choice of Coding Schemes

We first define the terminologies and notations. We consider a Raft group of N servers, where $N = 2F + 1$, so as to tolerate up to F failures. Let N' be the total number of *healthy servers* in the group (i.e., the servers are alive and can communicate with each other and the clients [31]), and f be the number of server failures (i.e., $N' = N - f$). When a Raft group starts to work, all servers belonging to the group are healthy and the number of healthy servers is equal to the total number of servers in the group (i.e., $N' = N$ and $f = 0$). The number of healthy servers decreases when some servers crash or lose network connection, or increases when the failed servers rejoin the group or some new servers are added to replace the failed servers. Currently, we consider a Raft group of a fixed size where N does not change.

TABLE II
CODING SCHEME USED IN FLEXRAFT TO MINIMIZE NETWORK AND STORAGE COSTS UNDER A DIFFERENT NUMBER OF SERVER FAILURES

f	k	Log Replication	Network Cost	Storage Cost
0	$N-F$	RS(N-F, F)	$\frac{N-1}{N-F}$	$\frac{N-1}{N-F}+1$
1	$N-F-1$	RS(N-F-1, F+1)	$\frac{N-2}{N-F-1}$	$\frac{N-2}{N-F-1}+1$
2	$N-F-2$	RS(N-F-2, F+2)	$\frac{N-3}{N-F-2}$	$\frac{N-3}{N-F-2}+1$
...	...	...	...	...
F	1	full-copy	$N-F-1$	$N-F$

The Raft leader can estimate the number of healthy servers and update the value of N' based on the exchanges of the latest heartbeat messages with its followers.

Coding scheme chosen by FlexRaft: To minimize both the network and storage costs, FlexRaft dynamically chooses the coding scheme based on the number of healthy servers remaining in a group. Given N' healthy servers, FlexRaft uses RS(k, m) code where $k = N' - F$ (i.e., $N - f - F$) and $m = N - k$ for log replication. Here, we make $k + m = N$, so that each server in the group can store a coded chunk (either original data chunk or parity chunk) of a log entry. Note that $k = 1$ means that FlexRaft uses full-copy replication. FlexRaft chooses the available largest value of k (i.e., $N' - F$) given the number of healthy servers N', so as to replicate a log entry with the lowest redundancy ratio. We prove that FlexRaft always minimizes the network and storage costs for log replication below.

Theorem 1: When there are N' $(F + 1 \leq N' \leq N)$ healthy servers in a Raft group, FlexRaft always minimizes the network and storage costs for log replication.

Proof: To maintain safety, at least $F + k$ servers should store the chunks of a log entry before it can be committed, such that there are at least k chunks in any $F + 1$ servers [25], [31]. When N' healthy servers exist in a Raft group, at most N' servers store the chunks of a log entry to commit (i.e., $F + k \leq N'$). Thus, Raft can use RS(k, m), where $k \leq N' - F$, for log replication. The redundancy ratio of RS(k, m) is N'/k. As the network and storage costs decrease with the reduction of the redundancy ratio, a larger value of k incurs lower storage and network overhead given a fixed N'. Thus, FlexRaft always minimizes the network and storage costs for log replication by using the largest available value of k (i.e., $k = N' - F$). □

Table II shows the coding schemes, log replication methods, network cost, and storage cost of FlexRaft under a different number of server failures where f increases from 0 to F. Under f server failures, FlexRaft uses a coding scheme with $k = N - f - F$ for log replication where the network cost is $(N - f - 1)/k$ and the storage cost is $(N - f - 1)/k + 1$. Note that the network and storage costs increase with the number of server failures. When F servers fail, the only method for log replication is full-copy replication.

Comparison to CRaft and HRaft: FlexRaft always achieves the minimum redundancy ratio, introducing lower network and storage costs than CRaft and HRaft. When the number of healthy servers N' is equal to or greater than $F + k$, all protocols can employ RS(k, m) for log replication, where the

network cost is $(N'-1)/k$. Suppose that we have Δf additional failed servers, such that the number of healthy servers N' is now less than $F+k$ (i.e., $N'=F+k-\Delta f$). The three protocols take different approaches to maintain the liveness level F: i) CRaft switches to full-copy replication, where the network cost increases to $2F-f$ [31]; ii) HRaft keeps using RS(k,m) and stores additional copies of some coded chunks, where the network cost is $(2F-f+F(f-N+F+k))/k$ [15]; iii) FlexRaft reduces the value of k based on the failure status and uses RS$(k-\Delta f,m)$ (for $k-\Delta f\geq 1$), where the network cost is $(F+k-\Delta f-1)/(k-\Delta f)$. Compared to the network cost when there are $(F+k)$ healthy servers (i.e., $(F+k-1)/k$), the increased cost of HRaft is $\Delta f(F-1)/k$, while that of FlexRaft is $\Delta f(F-1)/(k(k-\Delta f))$. Since we have $k-\Delta f\geq 1$, the increment of FlexRaft is less than that of HRaft. Also, as HRaft incurs less network traffic than CRaft [15], the network cost of FlexRaft is less than that of CRaft as well. Moreover, as the storage cost equals the network cost (from the leader to its followers) plus the storage cost of a full copy in the leader, the storage cost of FlexRaft is also less than those of CRaft and HRaft.

B. Varying the Coding Scheme

The main idea of FlexRaft is to choose the optimal coding scheme for log replication based on the cluster status. When failures occur during writes, FlexRaft adjusts the value of k if there are not enough servers storing the log entries.

Varying the coding scheme during log replication: When FlexRaft decides to vary the coding scheme for a log entry, the leader needs to re-encode the entry with the new coding scheme and send the new chunks to its followers. At the beginning of replicating a log entry, FlexRaft first performs encoding on the data using an initial coding scheme RS(k,m), where k is determined by the latest N' (Section III-A). Then the leader distributes the coded chunks to the followers and waits for the responses from the followers. Here, we map the chunk ID to the server ID consistently to determine which server should store which chunk for a log entry. When a follower receives a chunk from the leader, the follower appends the chunk to its log and returns a successful response to the leader. The leader collects the responses from the followers and decides how to perform log replication. If the leader receives at least $F+k-1$ successful responses, it continues to commit the log entry; otherwise, the leader varies the coding scheme to maintain safety since there are fewer than $F+k-1$ healthy followers for storing the log entry. FlexRaft then updates N' based on the responses from the followers and switches from the initial coding scheme to the new coding scheme RS(k',m') where $k'=N'-F$ and $m'=N-k'$. The leader re-encodes the log entry using the new coding scheme, and sends the new chunks to its followers. Each follower overwrites the old chunks with the new chunks. If the leader receives at least $F+k'-1$ successful responses, it can commit the log entry; otherwise, the leader continues to adjust the coding scheme by decreasing the value of k and performs the above process until it successfully replicates the log entry. The smallest value of k is one (i.e., full-copy replication).

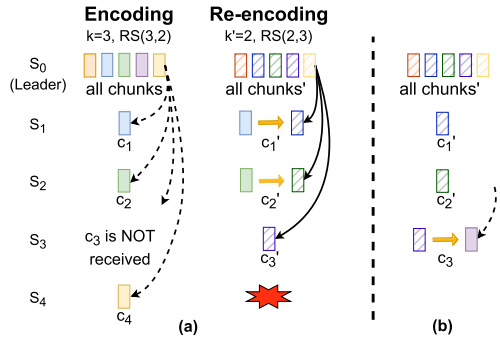


Fig. 2. The coding scheme varies during log replication in a group of five servers ($N=5$).

Fig. 2(a) shows how the coding scheme varies during log replication in a Raft group of five servers. At the beginning, the leader S_0 encodes the log entry using RS(3,2) and sends four chunks to its followers. However, the leader only receives three responses from servers S_1, S_2, and S_4, since S_3 does not receive any chunk (e.g., due to network failures). The leader S_0 then varies the coding scheme to RS(2,3), re-encodes the log entry, and sends the new chunks to its followers. The followers S_1 and S_2 overwrite the old chunk with the new chunk, S_3 stores the new chunk directly, and S_4 does not store the new chunk as it now crashes. After receiving successful responses from servers S_1, S_2, and S_3, the leader S_0 can commit this log entry safely. However, varying the coding scheme raises new challenges to the correctness of log replication.

Challenge 1: the chunk in a follower can be mistakenly overwritten by an old chunk of the same log entry: As the coding scheme varies during log replication in FlexRaft, the servers in a group may store the chunks encoded with different coding schemes for a log entry. Thus, the chunk stored in a follower may be an old one due to chunk overwrites. For example, as shown in Fig. 2(b), the followers S_1, S_2, and S_3 are expected to store the new chunks c_1', c_2', and c_3', respectively, after the re-encoding with RS(2,3). However, due to network delays, the request containing the old chunk c_3 reaches the follower S_3 after S_3 stores the new chunk c_3'. S_3 then overwrites c_3' with c_3, but the leader has received a successful response from S_3 that it stores c_3'. In this case, the chunk stored in a follower is inconsistent with that stored in the leader; in other words, the chunks belonging to a log entry are now encoded by different coding schemes. Thus, FlexRaft should guarantee that the chunks of a log entry stored in the followers are encoded with the same coding scheme as specified by the leader.

Prerequisite of overwriting a log entry: To avoid mistakenly overwriting a chunk in a follower, FlexRaft requires the follower to check the value of k with the chunk before overwriting a log entry: if the value of k with the new chunk is less than the current value of k, the follower overwrites the current chunk with the new one and responds to the leader; otherwise, the follower can ignore this message containing the new chunk. The reason is that the value of k always decreases when the coding scheme varies during log replication, and the new chunk to overwrite is

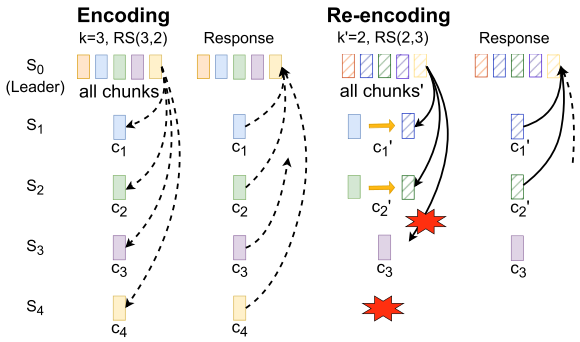


Fig. 3. Responses of varying coding schemes during log replication in a group of five servers ($N = 5$).

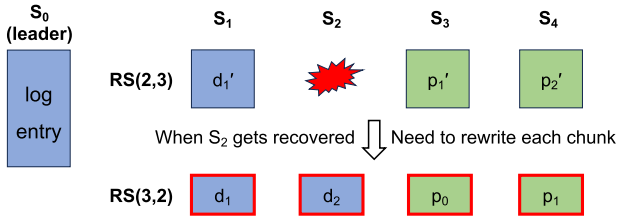


Fig. 4. Code conversion to the optimal coding scheme for the minimum storage overhead. When server S_2 gets recovered, the coding scheme switches from RS(2,3) to RS(3,2).

encoded from a coding scheme with a smaller k. Thus, in Fig. 2, if the follower S_3 compares the value of k with the chunk c_3 (i.e., $k = 3$) to the one with the chunk c_3' (i.e., $k = 2$), it will refuse to update the current chunk and the above inconsistent case will not occur.

Challenge 2: some followers may not overwrite the old chunks successfully before the leader commits the log entry. Although each follower checks the value of k before overwriting its chunks, the chunks stored in the follower may still use a different coding scheme from the leader's. One possible case is that the leader requires the followers to overwrite their existing old chunks, yet some followers fail to replace the old chunks with the new chunks. Fig. 3 depicts such an example in a group of five servers ($N = 5$). The leader S_0 encodes a log entry with RS(3,2) and sends the coded chunks to its followers. All followers store the chunks successfully and return successful responses to the leader. However, the follower S_3's response does not reach the leader in time (e.g., due to network failures). The leader S_0 only receives three successful responses from S_1, S_2, and S_4 before it decides to vary the coding scheme. Thus, the leader S_0 re-encodes the log entry with RS(2,3) and distributes the new chunks to the followers. As the AppendEntries RPC from the leader to the follower S_3 is lost and the follower S_4 crashes this time, only the followers S_1 and S_2 overwrite the old chunks with the new chunks and return successful responses to the leader again. However, the leader S_0 mistakenly thinks that three followers have stored the new chunks and treats log replication as successful, since it also receives the last successful response from S_3. That is, the new chunks being re-encoded are stored in only two followers, while S_3 still stores the old chunk. The main reason is that the successful response from a follower to the leader in Raft, which consists of the currentTerm and a true flag if the follower contains the entry matching the index of the previous log entry (i.e., preLogIndex) and the term of prevLogIndex entry (i.e., prevLogTerm), fails to indicate which server stores which coded chunk. Thus, the leader cannot distinguish the exact number of stored chunks belonging to the same coding scheme for a log entry from the responses of the followers.

Updating AppendEntries RPCs: To make a follower's log stay consistent with the leader's, we add *prevK* (i.e., the value of k of the prevLogIndex entry) and *ChunkInfo* (i.e., a tuple of the log index and the value of k) to the AppendEntries RPCs. We include prevK in the request of the AppendEntries RPCs to allow a follower to detect the chunks encoded with a different coding scheme. When receiving an AppendEntries RPC request, the follower checks the value of prevK after checking the term and prevLogTerm. If the follower finds that the value of k of the entry in prevLogIndex does not match prevK (i.e., storing a chunk encoded with an old coding scheme), the follower can return false; the leader then decrements nextIndex and retries to send AppendEntries RPCs. For example, when S_3 rejoins the group as a follower, it can compare prevK and find that the old chunk c_3 should be overwritten. Moreover, we include the ChunkInfo of the log entries in the requests and responses of the AppendEntries RPCs. Thus, the leader sends log entries containing ChunkInfo (included in each log entry) to the followers and each follower responds to the leader with the ChunkInfo of the chunks stored, such that the leader can determine whether the chunks encoded with the latest coding scheme have been stored in enough followers. For example, by checking the ChunkInfo in the received responses, the leader S_0 in Fig. 3 can find that one response should not be accounted as a successful response to replicate the new chunks, as the chunk c_3 rather than the new chunk c_3' is stored in S_3. Thus, the leader will decrease the value of k (i.e., reducing to full-copy replication) and retry to replicate the log entry. Thus, by adding prevK and ChunkInfo to the AppendEntries RPCs, FlexRaft guarantees that the chunks belonging to a committed log entry are encoded with the same coding scheme and correctly and safely stored in the followers.

C. Log Replication Under Server Failures

Code conversion for the long-term minimum storage overhead: The minimum redundancy ratio is achieved with RS$(F + 1, F)$ code when all servers in the group are healthy. Although FlexRaft minimizes the network and storage costs by varying the coding scheme during log replication, the log entries committed in the presence of server failures are encoded with a smaller k than $F + 1$. To further reduce the storage cost in the long term, we can convert the log entries stored with a smaller k to the RS$(F + 1, F)$ code when all servers are healthy ($N' = N$). It needs to re-encode log entries and update the chunks stored in all servers, which inevitably introduces additional encoding overhead and a large amount of network traffic. Fig. 4 shows the process of code conversion in a group of five servers ($N = 5$).

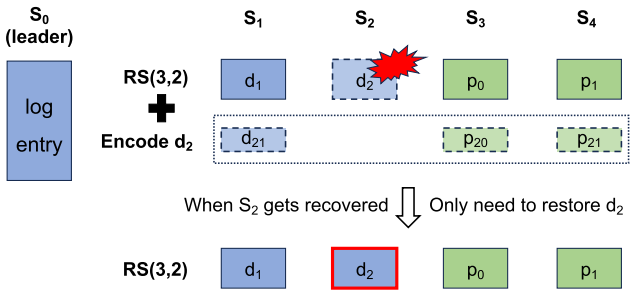


Fig. 5. An example of re-encoding-free log replication under one server failure in a group of five servers ($N = 5$), which maintains the minimum network and storage costs when varying the coding scheme directly.

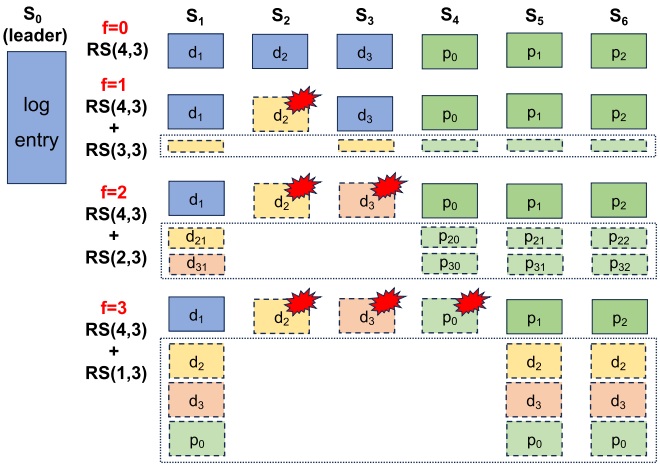


Fig. 6. An example of re-encoding-free log replication under a different number of server failures in a group consisting of seven servers ($N = 7$).

FlexRaft uses RS(2,3) for log replication under one server failure (e.g., server S_2 crashes), where each chunk is half of the original log entry size. Note that we do not show the chunk that should be stored in the leader S_0, since the leader stores a full copy of each log entry for log replication and answering client requests. When server S_2 is recovered, each follower needs to change the stored chunk (framed in red) as the coding scheme varies from RS(2,3) to RS(3,2) to achieve the minimum redundancy ratio. The leader S_0 needs to perform re-encoding with RS(3,2) and distribute the new chunks to its followers. That is, the code conversion process needs to re-encode and distribute the new chunks, thereby incurring additional computation overhead and increasing the network cost by $(N-1)/(N-F)$. Thus, such code conversion incurs additional re-encoding overhead and network traffic for long-term storage savings.

Re-encoding-free log replication under server failures: To reduce the code conversion overhead during server recovery, we propose *re-encoding-free log replication* under server failures and call the extension FlexRaft+. By re-encoding-free, we mean that FlexRaft+ does not need to perform re-encoding to switch to the optimal coding scheme when all servers in the group are healthy. The key idea of FlexRaft+ is to keep the optimal coding scheme unchanged when some servers fail, by storing the missing chunks (i.e., the chunks that should be stored in the failed servers) with erasure coding. Unlike FlexRaft, which directly varies the coding scheme with a decreasing k, FlexRaft+ adds some additional coded chunks to replace the missing chunks to maintain the same liveness level and the lowest redundancy ratio. Note that some existing code conversion techniques [21], [22], [23], [32] in distributed storage systems consider how to reduce the bandwidth and I/O during code conversion, they do not consider how code conversion is applied to consensus protocols.

Fig. 5 shows an example of re-encoding-free log replication in a group of five servers when one server S_2 fails. The healthy followers still store the chunks encoded with the optimal coding scheme RS(3,2), i.e., servers S_1, S_3, and S_4 store chunks d_1, p_0, and p_1, respectively. For the chunk d_2 that should be stored in the failed server S_2, FlexRaft+ encodes the chunk with RS(2,2) and distributes the new coded *subchunks* (i.e., generated from a chunk) to the remaining healthy servers. That is, FlexRaft+ divides d_2 into two subchunks d_{20} and d_{21}, computes two parity subchunks p_{20} and p_{21}, and stores the four subchunks in the remaining four servers. As the leader S_0 already stores the full copy of the log entry, we do not need to store additional chunks and subchunks in the leader. The network traffic of encoding d_2 under RS(3,2) equals that of converting to RS(2,3) directly. When server S_2 is recovered, FlexRaft+ only needs to restore chunk d_2 in S_2 and remove the subchunks for encoding d_2 in other servers, so it avoids modifying the chunks stored in all servers.

FlexRaft+ maintains the same liveness level and the lowest network and storage costs during log replication as varying the coding scheme directly in FlexRaft, while enabling a fast server recovery. In a group of N ($N = 2F + 1$) servers, FlexRaft+ always adopts the optimal coding scheme (i.e., RS$(F+1, F)$) for log replication and stores each missing chunk with RS$(F + 1 - f, F)$ under f server failures. We prove that FlexRaft+ maintains Raft safety and liveness level with the same redundancy ratio as FlexRaft in Section III-E. Fig. 6 shows how FlexRaft+ performs log replication under a different number of server failures in a group of seven servers ($N = 7$). When there is no server failure ($f = 0$), FlexRaft+ only needs to store the chunks encoded with the optimal coding scheme RS(4,3). If one server S_2 fails, FlexRaft+ encodes the missing chunk d_2 with RS(3,3) and stores the coded subchunks. Under two server failures ($f = 2$), FlexRaft+ stores the additional subchunks encoded with RS(2,3) for the missing two chunks d_2 and d_3. When three servers in the group fail, FlexRaft+ replicates the missing chunks in the remaining healthy servers, as RS(1,3) actually means 4-way replication. In this way, FlexRaft+ can directly recover the failed server and use the optimal coding scheme when all servers are healthy. When generating the subchunks under server failures, FlexRaft+ needs to perform additional encoding operations, which slightly increase the write latency (as shown in Figs. 9 and 10). Thus, FlexRaft+ makes a trade-off between server recovery performance and write latencies.

Compared to HRaft, FlexRaft+ takes a different approach to store the missing chunks under server failures. FlexRaft+

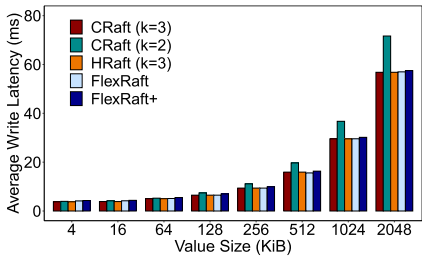


Fig. 7. Write latency in normal cases when $N = 5$.

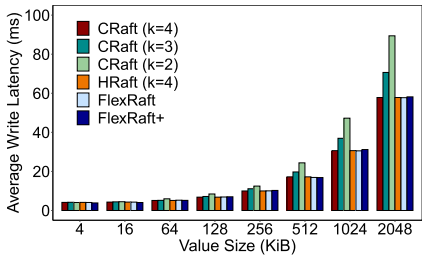


Fig. 8. Write latency in normal cases when $N = 7$.

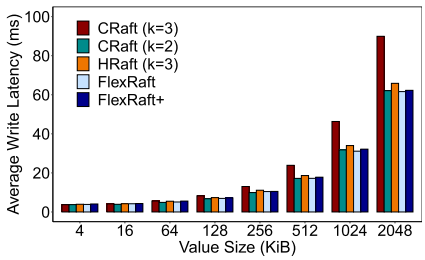


Fig. 9. Write latency under one server failure when $N = 5$.

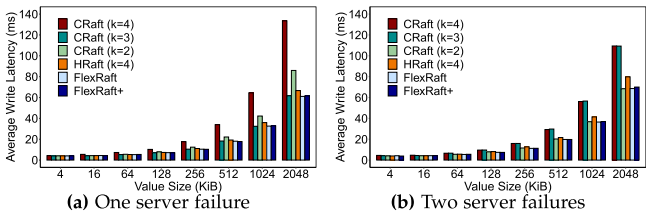


Fig. 10. Write latency under some server failures when $N = 7$.

encodes the missing chunks with a proper coding scheme to keep the lowest redundancy ratio of FlexRaft, while HRaft replenishes the missing chunks with replication and hence introduces a higher redundancy ratio. Thus, FlexRaft+ can achieve the lowest network and storage costs under a different number of server failures.

D. Server Recovery

Handling leader failures: When the current leader fails, the process of leader election starts and a new leader is elected. For the committed log entries, the newly elected leader has at least one chunk according to Raft election rules [26], and the leader can recover the complete log entry by collecting other chunks from its followers since we require that any $F + 1$ servers store k chunks. For the unapplied entries, the new leader should perform the LeaderPre operation (Section II-B) before it can become a fully-functioning leader [31]. Note that the chunks stored in the new leader may not use the same coding scheme when the log entries are committed, even though the leader's log is up-to-date. To deal with this issue, the leader recovers the entries among (commitIndex, lastLogIndex], i.e., the entries between the highest index of log entry known to be committed and the index of the last log entry, and replaces the chunks with the complete log entries during the LeaderPre operation. As the leader stores the ChunkInfo for each log entry, the leader can decide whether a log entry is recoverable based on the value of k. For those entries that cannot be recovered, the leader removes them from the log. The leader can process the client requests normally after the LeaderPre operation.

A newly elected leader may need to perform the decoding operation when serving read requests to an object. If the leader only has a coded chunk for a data object to be read, the leader needs to retrieve other coded chunks to decode the original object. The leader then stores a full copy of the recovered object for further reads to avoid decoding every time. That is, the leader handles multiple reads to an object with at most one decoding operation. Note that CRaft and HRaft perform the same read procedure as FlexRaft and FlexRaft+. These Raft protocols with erasure coding make a trade-off between the network and storage costs during commitment and the computation overhead during reads.

Recovering a failed follower in FlexRaft and FlexRaft+: When a server resumes normally, FlexRaft restores the committed log entries with the same coding scheme (i.e., the same value of k) to avoid code conversion while FlexRaft+ can directly restore the chunks encoded with the optimal coding scheme. For a committed log entry, the recovery costs of both FlexRaft and FlexRaft+ are $1/k$ of the original data size, but FlexRaft with a smaller value of k incurs higher storage and network costs than FlexRaft+. FlexRaft+ can then remove the subchunks encoded for the missing chunks directly. Both FlexRaft and FlexRaft+ guarantee that the recovered server stores all previous log entries before it is counted as a healthy server to function. After the recovery process completes, FlexRaft and FlexRaft+ increase the number of healthy servers (i.e., N'), and vary the coding scheme accordingly to replicate log entries later.

E. Safety Guarantee

We first show that FlexRaft guarantees Raft safety by proving the Log Matching Property and the Leader Completeness Property. We then prove that FlexRaft+ also maintains Raft safety and the same liveness level with the same redundancy ratio as FlexRaft.

Theorem 2: Log Matching Property: if two logs contain an entry with the same index and term, then the logs are identical (either a full copy or a coded chunk of the original proposed data) in all entries up through the given index.

Proof: The original Raft protocol maintains two properties to constitute the Log Matching Property: 1) if two entries in different logs have the same index and term, then they store the

same command, and 2) if two entries in different logs have the same index and term, then the logs are identical in all preceding entries. The first property stands because a leader creates at most one entry with a given log index in a given term, and log entries never change their position in the log. FlexRaft works as in Raft, although it varies the coding scheme when the server status changes. The coded chunk stored with the same index and term may use an old coding scheme (which will be updated during processing AppendEntries RPCs and the LeaderPre phase), but stands for the same command. The second property in Raft is guaranteed by a simple consistency check performed by the AppendEntries RPCs. FlexRaft also follows the AppendEntries RPC rules in the original Raft protocol. When the coding scheme varies for the entry with the same index and term, FlexRaft adds a restriction rule to avoid mistakenly overwriting a log entry, and includes prevK and ChunkInfo in the AppendEntries RPCs to make the followers use the same coding scheme as the leader. Thus, the Log Matching Property still holds where the data is a coded chunk when using erasure coding for the log entry. □

Theorem 3: Leader Completeness Property: 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.

Proof: As FlexRaft extends the LeaderPre operation in CRaft, CRaft guarantees that if a log entry e is committed in a given term, 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 [31]. We then prove that the committed data is recoverable using the right coding scheme in FlexRaft. For each committed entry, FlexRaft guarantees that there are at least $F + k$ coded chunks stored in the cluster. Thus, the leader can always recover a committed entry by collecting k chunks from other servers. Therefore, the Leader Completeness Property stands in FlexRaft. □

Theorem 4: FlexRaft+ maintains Raft safety and the liveness level (F) with the same redundancy ratio as FlexRaft.

Proof: We first prove that FlexRaft+ has the same redundancy ratio as FlexRaft. Compared to FlexRaft, FlexRaft+ keeps the optimal coding scheme $\text{RS}(F+1, F)$ and stores each missing chunk with $\text{RS}(F+1-f, F)$ instead of directly switching to $\text{RS}(F+1-f, F)$ under f server failures. For simplicity, we denote $F+1$ and $F+1-f$ by k and k', respectively. The redundancy ratio of FlexRaft with $\text{RS}(k', F)$ is $\frac{k'+F}{k'}$. For FlexRaft+, the redundancy ratio depends on two parts: the original chunks encoded with $\text{RS}(k, F)$ and the subchunks added for the missing chunks. Assuming the log entry size is one, each original chunk size is $\frac{1}{k}$ and each subchunk size is $\frac{1}{k} \cdot \frac{1}{k'}$. The redundancy ratio is equal to the total data size divided by the log entry size, i.e., $(\frac{1}{k} \cdot (k + F - f) + \frac{1}{k} \cdot \frac{1}{k'} \cdot (k' + F) \cdot f)/1 = \frac{k'+F}{k'}$. Thus, the redundancy ratio of FlexRaft+ is also $\frac{k'+F}{k'}$, which equals that of FlexRaft.

To prove that FlexRaft+ maintains Raft safety and the same liveness level F as FlexRaft, we only need to prove that FlexRaft+ can recover the original log entry under any F server failures. That is, FlexRaft+ also achieves the liveness level F by tolerating any F server failures. We first prove that FlexRaft+ can recover the f missing chunks (i.e., encoded with $\text{RS}(k, F)$) under any F server failures. As each missing chunk is encoded with $\text{RS}(k', F)$, there are at least k' subchunks left under any F server failures, which are sufficient to recover each original missing chunk. We then show that FlexRaft+ can recover the complete content of any committed Raft log entry. Suppose that a log entry is stored in N' ($N' = N - f$) servers before the entry is committed. When at most F server failures occur, there are at least $N' - F = k'$ alive servers where each server stores one chunk and f subchunks. According to the above proof, we can successfully reconstruct the f missing chunks. Given k' chunks stored in the alive servers and f recovered chunks, we can gather at least $k' + f = k' + (k - k') = k$ chunks that are encoded with $\text{RS}(k, F)$, which enables the recovery of the original log entry. Thus, FlexRaft+ maintains Raft safety and the liveness level F , while having the same redundancy ratio as FlexRaft. □

IV. Implementation

We implement a prototype of FlexRaft from scratch in C++ and also support FlexRaft+ with some modifications of FlexRaft. We leverage the ISA-L library [4] for erasure coding operations and use RCF 3.0 [5] for interprocess communication. To evaluate FlexRaft, we also build a distributed key-value store based on RocksDB v7.3.1 [6]. Each server has a constantly running FlexRaft module, a RocksDB engine as a state machine, and a background working thread to apply committed entries to the state machine. The whole system contains about 6K LoC.

Log entry: We modify the structure of the log entry by adding an *EntryType* flag and the *ChunkInfo*, a tuple of (log index, k). The EntryType flag in a log entry's metadata indicates whether the command in this entry is a complete copy or a coded chunk. If this entry is encoded, the ChunkInfo identifies the coded chunk uniquely, where k is the encoding parameter used for this entry.

Log replication and commitment: The leader determines the coding scheme for log replication by counting the number of healthy servers. The leader sends heartbeat messages (i.e., empty AppendEntries RPCs) to its followers every 100 ms. To track the server status, the leader records the time point when it receives RPC messages (RPC requests and responses) from other servers. If the leader detects that some servers have not sent any messages for some time (e.g., 200 ms), the leader considers these servers as unhealthy and determines the encoding parameter by $k = N' - F$. The leader then encodes the original data into $k + m$ chunks, sends the coded chunk to each healthy follower using AppendEntries RPCs, and waits for responses from the followers. If the leader receives more than $F + k - 1$ success responses within a configured time limit (e.g., 1 s), the leader checks the ChunkInfo in the responses and commits this entry if at least $F + k$ servers (including itself) successfully store this entry. Otherwise, the leader decreases the encoding parameter k by one, re-encodes the entry, and repeats the above process until the log entry gets committed.

Processing in followers: Upon receiving an AppendEntries request, the follower checks the metadata of this entry to decide whether to store it or not. For the entry that has the same term and index as the latest one stored in the follower's log, the follower overwrites the old entry using the newly-received one only if the

new entry is encoded with a smaller k; otherwise, the follower ignores this request. The follower overwrites the old entries by trimming its log and appending the new entries. If the log entry has a different term or index, the follower appends the new entry. After storing the log entry, the follower returns a response containing the ChunkInfo to the leader.

FlexRaft+ implementation: We implement FlexRaft+ by making some modifications to FlexRaft. For FlexRaft+, each log entry carries both the original chunk and added subchunks. Each subchunk is labeled by *SubChunkInfo*, a tuple consisting of *ChunkId* that presents which chunk the subchunk is encoded from and *SubChunkId* that marks the position of the subchunk in its stripe. For example, the subchunk d_{21} in Fig. 5 can be uniquely identified by SubChunkInfo of (2, 1), meaning that this subchunk is encoded from the chunk d_2 and it is the second subchunk belonging to the substripe encoded with RS(2,2). Each SubChunkInfo consists of two 4-byte integers (i.e., ChunkID and SubChunkID), accounting for a total of 8 bytes of storage space. The number of subchunks (and SubChunkInfo) contained in one AppendEntries RPC is equal to the number of failed servers during log commitment. For example, for the failure cases where $f = 1, 2, 3$, the leader incurs 8, 16, and 24 bytes of extra space, respectively, in each AppendEntries RPC call. The extra payload size is negligible (less than 1%) to the AppendEntries RPC call. We leverage the field *ChunkInfo* introduced in FlexRaft to indicate the change of coding scheme. If the leader detects that one more follower crashes during the log commitment, it decreases k by one and adds more subchunks encoded from the newly missing chunk. Each server in FlexRaft+ separates the storage of the original chunks and added subchunks (i.e., storing them in different files) to reduce the overhead of removing the subchunks during server recovery. Thus, the follower only needs to overwrite the subchunks when receiving a log entry with a smaller k.

To recover a failed follower, the leader directly sends the original coded chunk to the failed server. For example, the leader sends d_2 to S_2 during server recovery as shown in Fig. 5. Once the leader has received the responses from the failed server that the coded chunks have been stored, the leader notifies other healthy servers to remove the subchunks, such that the overall storage cost is reduced. An exceptional case is that the leader may not contain the original full entry and it has no corresponding coded chunk for the failed server (e.g., this entry has been committed by a previous leader). In this case, the leader notifies the failed follower to contact other healthy servers to collect enough subchunks and recover the chunk; after the follower successfully restores the chunk, the follower then notifies other followers to remove the corresponding subchunks and replies to the leader that the follower recovery completes.

V. EVALUATION

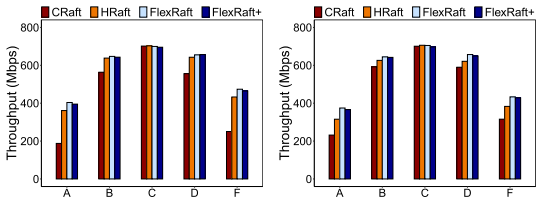
In this section, we show the I/O performance of the key-value store atop FlexRaft, the breakdown performance, the overhead, and the scalability of FlexRaft. We run all experiments on Alibaba Cloud [1]. The cluster consists of at most 11 servers to run key-value service and one individual server to send client requests. Each server is a cloud instance equipped with an Xeon CPU of 8 vCPU, 32 GiB DRAM, and 512 GiB ESSD cloud drive (about 7800 IOPS). In each experiment, each Raft server spawns two threads in two vCPUs to execute the Raft protocol and apply log entries, and starts a thread on-the-fly to handle any incoming RPC request. Each client sends read/write requests with a single thread that runs in a vCPU. The network bandwidth is 1 Gbps. We compare FlexRaft and FlexRaft+ with CRaft and HRaft in the normal case and when some servers fail.

A. Write Performance

Normal write latencies: We first compare the write latencies of FlexRaft and FlexRaft+ with CRaft and HRaft in normal cases when no server fails. The Raft group has five or seven servers (i.e., $N = 5$ or $N = 7$, respectively, where $N = 2F + 1$). For the coding scheme, we configure all possible values of k for CRaft and use the largest k for HRaft; FlexRaft and FlexRaft+ choose the optimal coding scheme automatically. When $N = 5$, CRaft can use RS(3,2) or RS(2,3) as its coding scheme; when $N = 7$, CRaft can use RS(4,3), RS(3,4), or RS(2,5). HRaft uses the largest value of k, i.e., RS(3,2) when $N = 5$ and RS(4,3) when $N = 7$. The key size is 16 bytes (same for the following experiments), and the value size varies from 4 KiB to 2 MiB. For each value size, the client generates 10000 PUT requests.

Figs. 7 and 8 show the write latencies of CRaft, HRaft, FlexRaft, and FlexRaft+ under different value sizes when $N = 5$ and $N = 7$, respectively. When $N = 5$, CRaft ($k = 3$) has a lower latency than CRaft ($k = 2$), reducing the latency by 4.05-20.73% since CRaft ($k = 3$) saves the network bandwidth cost. FlexRaft and HRaft achieve the same performance as CRaft ($k = 3$) since both use the largest k to minimize the redundancy ratio. When $N = 7$, CRaft ($k = 4$) achieves the lowest latency among all three configurations. Compared to CRaft ($k = 3$) and CRaft ($k = 2$), CRaft ($k = 4$) reduces the latency by up to 18.15% and 35.32%, respectively. The latencies of FlexRaft, FlexRaft+, and HRaft are close to that of CRaft ($k = 4$) under the same value size since they use the same coding scheme with the lowest redundancy ratio. Thus, both FlexRaft and FlexRaft+ minimize the network and storage costs for log replication in normal cases.

Write performance under server failures: We evaluate the write performance under one server failure when $N = 5$. We use the same configuration as the above experiment. Fig. 9 plots the write latencies of different value sizes under one server failure. As the number of healthy servers is below $F + k$, CRaft ($k = 3$) converts to full-copy replication and HRaft replicates two additional coded chunks, incurring higher network and storage costs. In this case, FlexRaft varies the coding scheme to RS(2,3), achieving the lowest latency as CRaft ($k = 2$). Compared to CRaft ($k = 3$) and HRaft, FlexRaft reduces the latency by at most 32.82% and 8.43%, respectively. FlexRaft+ achieves similar write performance to FlexRaft, because both of them incur the minimum network and storage costs. As FlexRaft+ needs to perform encoding once more (i.e., generating the subchunks) and store the subchunks separately, FlexRaft+ increases the write latencies of FlexRaft by 3.27% on average under different value sizes.



(a) One server failure **(b)** Two server failures

Fig. 11. Performance under YCSB workloads when $N = 7$.

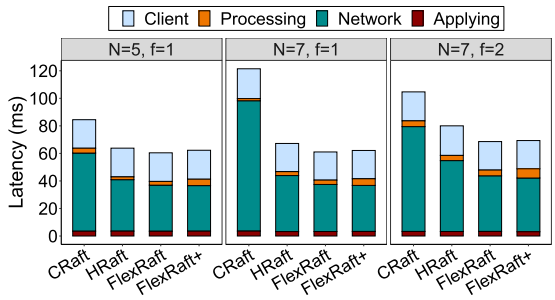


Fig. 12. Breakdown performance of a put operation.

We then measure the write performance under a different number of server failures when $N = 7$. Fig. 10(a) and (b) show the write latencies under one server failure and two server failures, respectively. When one server fails, FlexRaft and CRaft ($k = 3$) achieve the lowest write latencies among all coding schemes, while CRaft ($k = 4$) has the highest latencies as it converts to full-copy replication. FlexRaft reduces the latencies of CRaft ($k = 4$), CRaft ($k = 2$), and HRaft by up to 54.28%, 28.97%, and 9.25%, respectively. When there are two failed servers, FlexRaft uses RS(2,5) for log replication to minimize the redundancy ratio and incurs the minimum network traffic and storage overhead. Both CRaft ($k = 4$) and CRaft($k = 3$) switch to full-copy replication, while HRaft stores more coded chunks. Compared to CRaft ($k = 4$) and HRaft, FlexRaft reduces the write latency by 37.32% and 14.10%, respectively when the value size is 2 MiB. The write latency of FlexRaft+ is similar to that of FlexRaft as they incur the same network and storage costs, where FlexRaft+ increases the latency by at most 3.26% and 2.89% under one server failure and two server failures, respectively. The small latency increase in FlexRaft+ comes from the additional coding overhead of subchunks, which is negligible as the network transfer time accounts for the majority of the total processing time (shown in Section V-C).

B. Performance Under YCSB Workloads

We compare the performance of FlexRaft and FlexRaft+ to HRaft and CRaft under YCSB [13] workloads when $N = 7$. We configure the largest value of k ($k = 4$) for CRaft and HRaft. We launch four client threads to generate 10000 requests following a Zipf distribution with a Zipfian constant of 0.99. Here, we fix the value size as 2 MiB.

Fig. 11 shows the throughput of CRaft, HRaft, FlexRaft, and FlexRaft+ under YCSB workload A (50% reads, 50% updates), B (95% reads, 5% updates), C (100% reads), D (95% read-latest, 5% inserts), and F (50% reads, 50% read-modify-writes), when there is one failed server and two failed servers, respectively. Here, we do not show the results of workload E (95% scans, 5% updates) as our prototype does not support scan operations currently. Compared to CRaft and HRaft, FlexRaft increases the throughput of write-heavy workloads (A and F) by 37.28-115.14% and 9.47-18.76%, respectively, because FlexRaft minimizes the network and storage costs during writes. For read-only workload (C), all protocols achieve similar throughput as they directly read data from the leader. For read-intensive workloads (B and D), FlexRaft improves the throughput of CRaft and HRaft by 8.90-17.88% and 1.40-5.74%, respectively. FlexRaft+ achieves similar performance to FlexRaft under different YCSB workloads, where the throughput of FlexRaft+ is 2.26% lower at most (when the fraction of write operations is high) due to its additional encoding overhead in the presence of server failures. Thus, FlexRaft achieves the best performance by reducing the network and storage costs during writes.

C. Microbenchmarks

We study the breakdown performance of a single PUT operation in the presence of server failures. We divide the whole process of a PUT operation into the following parts: (i) the communication latency between the client and the leader (denoted by *client*); (ii) the total time spent in handling the request and encoding (denoted by *processing*); (iii) the latency of sending AppendEntries RPCs in parallel to replicate an entry (denoted by *network*); and (iv) the time of applying a log entry to the state machine (denoted by *applying*). Note that the sum of the processing time and network latency is indeed the *commit latency* of a log entry. The client time is calculated by deducting the processing, network, and applying times from the total response time observed in the client (i.e., the time from sending a request until receiving the response). In common cases, there are one RTT between the client and the leader, and one RTT between the leader and each of its followers. Note that the network time varies due to the difference in payload size. We fix the value size as 2 MiB and send 10000 PUT requests from the client. We configure the largest value of k for CRaft and HRaft to reduce the redundancy ratio, i.e., $k = 3$ when $N = 5$, and $k = 4$ when $N = 7$.

Fig. 12 shows the breakdown performance of CRaft, HRaft, FlexRaft, and FlexRaft+. CRaft, HRaft, and FlexRaft have similar client, processing, and applying latencies, as they go through the same workflow. FlexRaft+ incurs 53.76-70.91% higher processing latencies than FlexRaft due to the additional encoding overhead of generating subchunks on the leader. However, the encoding impact on the overall write latency in FlexRaft+ is negligible, as the processing latency only accounts for about 5.18% of the total write latency. The network latency, depending on the network bandwidth cost, determines the overall write latency. For $N = 5$, FlexRaft reduces the network latencies of CRaft and HRaft by 41.02% and 10.45% under one server failure, resulting in 39.95% and 8.19% reduction of commit latencies.

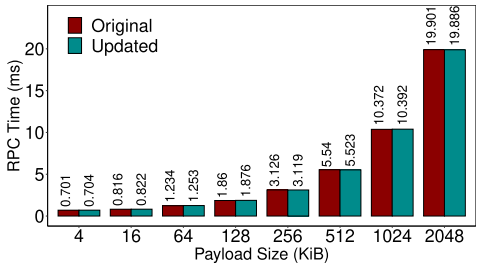


Fig. 13. RPC time of AppendEntries requests.

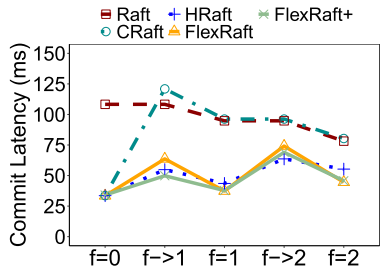


Fig. 14. The overhead of varying the coding scheme.

For $N = 7$, compared to CRaft and HRaft, FlexRaft reduces the commit latencies by 61.20% and 14.26%, respectively under one server failure, and 44.51% and 19.37%, respectively under two server failures. The reduction of commit latencies confirms the saving of network traffic; for example, FlexRaft reduces the network bandwidth costs of CRaft and HRaft by 50% and 20% in theory under two server failures when $N = 7$, respectively. FlexRaft+ achieves the same network latencies as FlexRaft does as they transfer the same size of payload for each entry to be committed. Thus, FlexRaft and FlexRaft+ minimize the network bandwidth and storage costs using a flexible coding scheme for log replication.

D. Overhead of FlexRaft

The overhead of updated AppendEntries RPCs: We compare the updated AppendEntries RPCs in FlexRaft to the original RPCs in Raft. We measure the completion time of one AppendEntries RPC (including one request and one response) carrying one log entry, labeled as *RPC time*. We vary the payload size of the log entry from 4 KiB to 2048 KiB. Fig. 13 plots the RPC time of the original and the updated AppendEntries. The RPC time of the updated AppendEntries is almost the same as that of the original across different payload sizes. FlexRaft only adds 16 B to the request and 8 B to the response to include the EntryType and ChunkInfo, which are negligible compared to the payload sizes. As FlexRaft+ reuses the AppendEntries RPCs, the overhead of the updated AppendEntries in both FlexRaft and FlexRaft+ is negligible.

The overhead of varying the coding scheme: We study the overhead of varying the coding scheme in FlexRaft and FlexRaft+. We measure the write latency under a different number of server failures for a group of seven servers (i.e., $N = 7$). Here, we fix the value size as 2 MiB. Fig. 14 shows the average commit latencies of Raft, CRaft, HRaft, FlexRaft, and FlexRaft+ over 10,000 times. Note that the x-axis in Fig. 14 shows the timeline of different server failure cases: (i) all servers are healthy (i.e., $f = 0$), (ii) one of the servers fails during log replication (i.e., $f \rightarrow 1$), (iii) the server status of all servers stabilizes under a server failure (i.e., $f = 1$), (iv) one additional server fails during log replication (i.e., $f \rightarrow 2$), and (v) the server status of all servers stabilizes under two server failures (i.e., $f = 2$). All three protocols with erasure coding achieve the same latency using RS(4,3) in normal cases, reducing the commit latency of Raft by 69.03%. When a server fails during log replication (i.e., $f \rightarrow 1$), CRaft reduces to full-copy replication, while HRaft needs to store one more chunk in three followers; FlexRaft re-encodes the log entry with RS(3,4) and distributes the new chunks to the remaining followers; FlexRaft+ only needs to encode the missing chunks with RS(3,3) and store the subchunks in five servers. The additional network costs of CRaft, HRaft, FlexRaft, and FlexRaft+ are 5, 3/4, 5/3, and 5/12 respectively. Thus, the write latency of CRaft is the largest, which is 11.67% higher than that of Raft and about four times the normal latency with erasure coding. HRaft and FlexRaft increase the normal write latency by 63.45% and 90.28%, respectively, but this only occurs once when the server fails during writes. FlexRaft+ achieves the lowest commit latency in the presence of failure, which is 9.11% and 21.71% lower than those of HRaft and FlexRaft, respectively. Like HRaft, FlexRaft+ does not vary the coding scheme directly and overwrites existing chunks on the follower nodes, which helps reduce the commit latency of FlexRaft. Compared to HRaft, FlexRaft+ reduces the network and storage costs by only storing the encoded subchunks for the missing chunk instead of replicating the missing chunk.

When the server status remains unchanged under one server failure, FlexRaft achieves the lowest commit latency and reduces the latency of CRaft and HRaft by 61.20% and 14.26%, respectively. FlexRaft+ achieves a similar performance to FlexRaft, because they incur the same network and storage costs. Note that CRaft performs the same as Raft when one server fails. When one more server fails during writes (i.e., $f \rightarrow 2$), FlexRaft switches to RS(2,5), resulting in 11.96% higher latency than HRaft; FlexRaft+ encodes the two missing chunks with RS(2,3), increasing the latency by 8.46% compared to HRaft. When the number of failed servers increases from one to two, HRaft performs the best by replicating the missing chunks directly, while both FlexRaft and FlexRaft+ need to perform re-encoding. When there are two failed servers, FlexRaft reduces the latencies of CRaft and HRaft by 44.51% and 19.37%, respectively. FlexRaft+ also achieves a similar performance to FlexRaft under two server failures. The commit latency of CRaft under two server failures is lower than that under one failure since CRaft has one fewer follower storing a full copy of the log entries which reduces the network and storage costs. While FlexRaft incurs additional network transfer by varying the coding scheme when a server fails during the log replication, FlexRaft achieves the lowest latency by minimizing the network and storage costs when the cluster status remains stable (i.e., $f = 0, 1, 2$). As

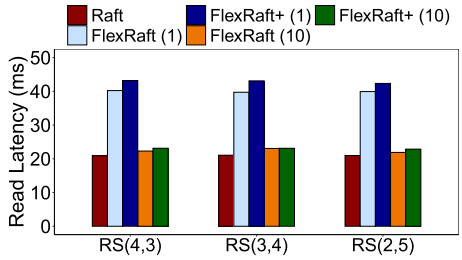


Fig. 15. Decoding overhead during reads.

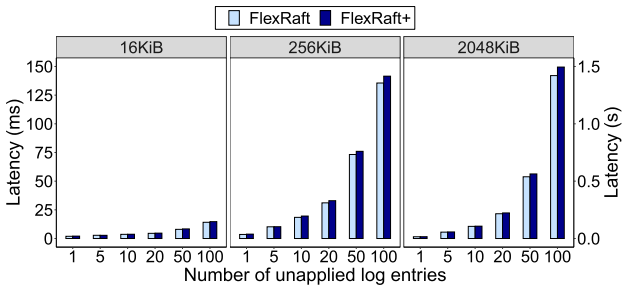


Fig. 16. Duration of leaderpre operations.

FlexRaft+ adopts the re-encoding-free log replication instead of varying the coding scheme directly, FlexRaft+ achieves the best performance under the stable cluster status and when the server failure first occurs (i.e., $f \rightarrow 1$).

Decoding overhead during reads: We compare the read performance of FlexRaft and FlexRaft+ to that of Raft. Fig. 15 shows the read latencies of Raft, both FlexRaft and FlexRaft+ at the first time to read (denoted by FlexRaft (1) and FlexRaft+ (1), respectively), and both FlexRaft and FlexRaft+ with 10 read times (denoted by FlexRaft (10) and FlexRaft+ (10), respectively) under different coding schemes when $N = 7$. Compared to Raft, FlexRaft (1) increases the read latencies by 88.60-92.35%, while FlexRaft (10) only increases by 4.34-6.62%. FlexRaft+ incurs slightly higher decoding overhead than FlexRaft as it needs to first decode the missing chunks and then recover the original value. The read latency of FlexRaft+ (1) is 101.81-106.42% higher than the read latency of Raft, while that of FlexRaft+ (10) is only 8.98% higher than that of Raft due to the data cache on the leader node. Thus, the decoding operation has a limited negative impact on the read performance when the leader does not crash frequently (i.e., the leader already stores a full copy for read requests).

E. Server Recovery

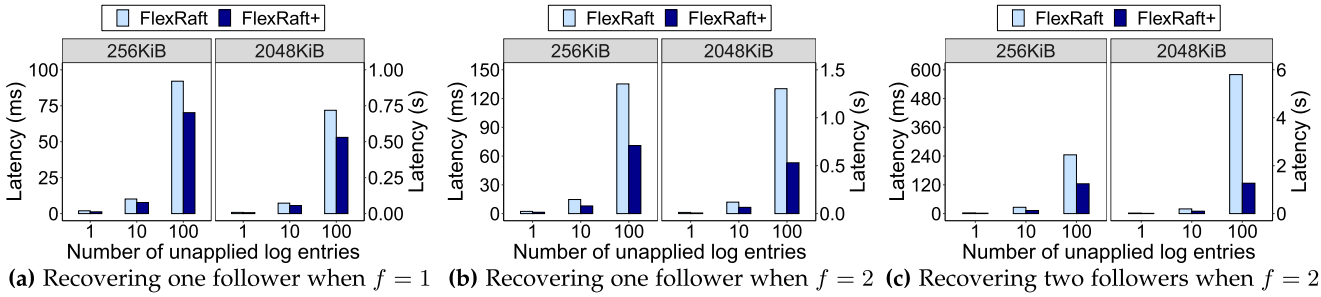
Duration of the LeaderPre operation under the leader failure: We measure the duration of the LeaderPre operation when the leader fails and a new leader is elected; here, we consider $N = 7$. As the LeaderPre operation only recovers the unapplied log entries, we vary the number of unapplied log entries (from 1 to 100) with different value sizes. Fig. 16 shows the duration of the LeaderPre operation, which increases with the number of unapplied log entries. For the small values (e.g., 16 KiB), it takes about 4 ms and 14 ms for FlexRaft to recover 10 and 100 log entries, respectively. For the values with a medium size (e.g., 256 KiB), the LeaderPre operation in FlexRaft lasts about 18 ms and 136 ms when there are 10 and 100 unapplied log entries, respectively. For the large values (e.g., 2 MiB), it takes about 106 ms for FlexRaft to recover 10 entries and 1.4 s to recover 100 entries. Compared to FlexRaft, FlexRaft+ increases the LeaderPre duration by 4.19% on average due to its additional decoding overhead. Thus, the newly-elected leader in both FlexRaft and FlexRaft+ can quickly complete the LeaderPre operation, making a limited impact on system availability.

Follower recovery: We compare the follower recovery latency of FlexRaft and FlexRaft+ to show the recovery efficiency of re-encoding-free log replication adopted by FlexRaft+. Fig. 17 plots the follower recovery latency under different server failures when $N = 7$. When one follower fails, FlexRaft+ reduces the recovery latency of FlexRaft by 19.62-35.42%, because FlexRaft+ can directly send the missing chunk with a smaller k (i.e., using the optimal coding scheme) to the follower. If two follower servers fail, FlexRaft+ reduces the recovery latency of FlexRaft by 39.06-59.28% and 42.43-67.16% when recovering one follower or both failed followers, respectively. The reduction in recovery latency of FlexRaft+ comes from the re-encoding-free log replication scheme, which reduces the data transfer size from $1/(F+1-f)$ to $1/(F+1)$ of the log entry to recover a failed follower. The recovery performance improvement of FlexRaft+ over FlexRaft when recovering two failed followers is higher than the theoretical improvement, because the large data size to transfer in FlexRaft triggers the re-transfer process due to the network timeout sometimes. Therefore, FlexRaft+ greatly improves the follower recovery performance by using the re-encoding-free log replication.

Storage cost in the followers: To demonstrate the storage efficiency of FlexRaft+, we measure the storage cost of different consensus protocols. As each protocol needs to store a whole copy of each log entry in the leader, we compare the storage cost of the committed log entries in the followers. In this experiment, we first make the leader commit 10,000 log entries of 1 MiB in the presence of server failures; we then restart one failed follower to trigger the follower recovery process; finally, we calculate the total storage cost by summing up the storage costs of all followers. Fig. 18(a) and (b) show the storage costs of CRaft, HRaft, FlexRaft, and FlexRaft+ when $N = 7$ and $N = 9$, respectively. FlexRaft+ achieves the lowest storage cost in all failure cases. Compared to FlexRaft, FlexRaft+ reduces the storage cost by 27.58-37.89% and 19.56-25.66% for $N = 7$ and $N = 9$, respectively. FlexRaft+ achieves a lower storage cost than FlexRaft, as it sends less data to the recovered follower (i.e., a chunk of $1/(F+1)$ in FlexRaft+ and a chunk of $1/(F+1-f)$ in FlexRaft). FlexRaft+ reduces the storage cost of HRaft by 12.49-21.43%, because FlexRaft+ only stores the coded subchunks rather than the original chunks.

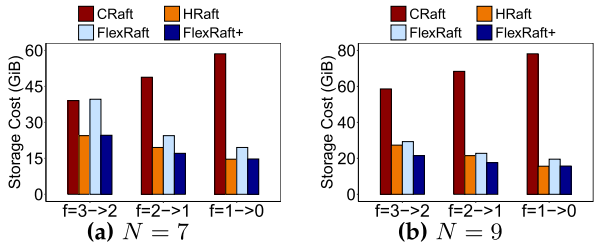
F. Scalability of FlexRaft

We evaluate the performance of CRaft, HRaft, and FlexRaft with a larger number of servers in a group. As all three protocols achieve the lowest commit latency using the largest available



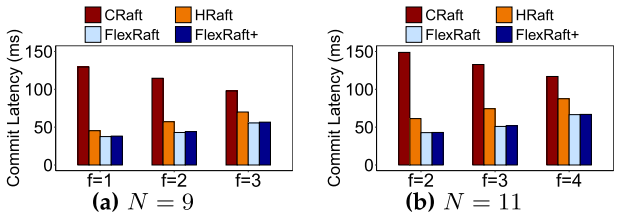
(a) Recovering one follower when $f = 1$ **(b)** Recovering one follower when $f = 2$ **(c)** Recovering two followers when $f = 2$

Fig. 17. Duration of follower recovery under different server failures.



(a) $N = 7$ **(b)** $N = 9$

Fig. 18. Storage cost in the followers.



(a) $N = 9$ **(b)** $N = 11$

Fig. 19. Commit latency of CRaft, HRaft, FlexRaft, and FlexRaft+ with a larger number of servers in a group.

value of k, we compare their commit latencies under a different number of server failures. We set the payload size of each log entry as 2 MiB and plot the average commit latencies over 10,000 times. Fig. 19 shows the commit latencies of CRaft, HRaft, and FlexRaft under a different number of server failures in a group of $N = 9$ and $N = 11$ servers. For a group of 9 servers, FlexRaft reduces the commit latency of CRaft and HRaft by 43.20-71.00% and 17.14-24.87%, respectively. Also, FlexRaft reduces the commit latency of CRaft and HRaft by 43.12-71.45% and 24.01-31.47%, respectively, when more than one server fails in a group of 11 servers. FlexRaft+ achieves almost the same commit latencies as FlexRaft due to the same network and storage costs. The evaluation results demonstrate that both FlexRaft and FlexRaft+ achieve higher performance than CRaft and HRaft for a large-scale cluster in the presence of server failures.

VI. RELATED WORK

Improving the performance of consensus protocols: Distributed systems employ consensus algorithms to provide high reliability and availability for upper-layer applications. Paxos [18], [19] is one commonly used consensus protocol in distributed systems, such as Chubby [9] and Spanner [9]. Many variants of Paxos [8], [11], [20], [24], [28], [34] have been proposed in the literature to improve the performance of Paxos-based systems. Mencius [20] proposes Round-robin Paxos with rotating leaders to alleviate the single-leader bottleneck. EPaxos [24], [28] extends vanilla Paxos in a decentralized fashion to achieve optimal commit latency. SDPaxos [34] presents a semi-decentralized replication, which overlaps the separated replicating and ordering processes to achieve one-round-trip latency under three/five-replica configurations. WPaxos [8] proposes a multi-leader Paxos to achieve low latency and high throughput in WAN deployments. PigPaxos [11] decouples the decision-making from the communication at the leader using an in-network aggregation and piggybacking technique.

Raft [26] is a widely used consensus algorithm designed for easy understanding and implementation, which is equivalent to Multi-Paxos. There have been many optimizations proposed to improve the performance of the original raft protocols in recent years [10], [16], [17], [30]. ParallelRaft [10] realizes a parallelized version of Raft, which removes the original Raft's strict serialization for high I/O concurrency. HovercRaft [17] extends the Raft protocol by eliminating CPU and I/O bottlenecks to achieve both scalability and fault tolerance. KV-Raft [30] introduces commit return and immediate read into vanilla Raft, to accelerate the write and read performance of distributed key-value storage systems respectively. NB-Raft [16] increases the parallelism and throughput of Raft by enabling multiple entries from the same client to be processed in parallel.

Optimizing consensus algorithms with erasure coding: Some studies [15], [25], [29], [31] extend the replication-based Paxos and Raft protocols with erasure coding (based on RS codes [27]) for higher performance with lower overhead. Erasure coding has been widely applied in distributed storage systems [3], [14] to protect data against server failures with a low redundancy ratio. The adoption of erasure coding in consensus algorithms introduces a new approach to improve the overall system performance by reducing the redundancy overhead. RS-Paxos [25] is the first consensus protocol that combines erasure coding into Paxos protocol, but reduces the liveness level. Pando [29] leverages erasure coding in geo-distributed storage to reduce costs for preserving consistency. CRaft [31] applies erasure coding to Raft while keeping the same liveness level as the original Raft protocol, but it degrades to full-copy replication

when the number of failed servers exceeds a certain threshold. HRaft [15] addresses the degradation problem of CRaft by maintaining additional coded data in healthy servers instead of switching to full-copy replication when server failure occurs.

VII. Conclusion

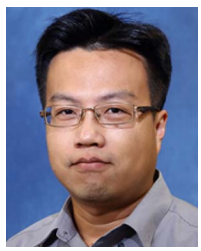
This paper proposes FlexRaft which minimizes the network and storage costs by dynamically adjusting the coding scheme used for log replication in Raft. It uses the optimal coding scheme with the available largest k based on the server status. When varying the coding scheme during writes, FlexRaft restricts the overwriting of the coded chunks and updates the AppendEntries RPCs to make sure that all servers store the right coded chunks. We further propose re-encoding-free log replication in FlexRaft+ to enable fast server recovery. Our evaluation results demonstrate that both FlexRaft and FlexRaft+ minimize the network and storage costs for log replication in Raft, while FlexRaft+ enables fast server recovery.

References

[1] Alibaba cloud, 2024. [Online]. Available: https://us.alibabacloud.com/
[2] etcd: A distributed, Reliable key-value store for the most critical data of a distributed system, 2024. [Online]. Available: https://etcd.io/
[3] HDFS 3.1, 2024. [Online]. Available: https://hadoop.apache.org/release/3.1.1.html
[4] ISA-L, 2024. [Online]. Available: https://github.com/intel/isa-l
[5] RCF, 2024. [Online]. Available: https://www.deltavsoft.com/
[6] RocksDB, 2024. [Online]. Available: https://github.com/facebook/rocksdb/tree/v7.3.1
[7] TiKV: A distributed transactional key-value database, 2024. [Online]. Available: https://tikv.org/
[8] A. Ailijiang, A. Charapko, M. Demirbas, and T. Kosar, "WPaxos: Wide area network flexible consensus," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 1, pp. 211–223, Jan. 2020.
[9] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2006, pp. 335–350.
[10] W. Cao et al., "PolarFS: An ultra-low latency and failure resilient distributed file system for shared storage cloud database," *Proc. VLDB Endowment*, vol. 11, pp. 1849–1862, 2018.
[11] A. Charapko, A. Ailijiang, and M. Demirbas, "PigPaxos: Devouring the communication bottlenecks in distributed consensus," in *Proc. ACM Int. Conf. Manage. Data*, 2021, pp. 235–247.
[12] Y. L. Chen et al., "Giza: Erasure coding objects across global data centers," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 539–551.
[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[14] C. Huang et al., "Erasure coding in windows azure storage," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2012, Art. no. 2.
[15] Y. Jia, G. Xu, C. W. Sung, S. Mostafa, and Y. Wu, "HRaft: Adaptive erasure coded data maintenance for consensus in distributed networks," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2022, pp. 1316–1326.
[16] T. Jiang et al., "Non-blocking raft for high throughput IoT data," in *Proc. IEEE Int. Conf. Data Eng.*, 2023, pp. 1140–1152.
[17] M. Kogias and E. Bugnion, "HovercRaft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2020, Art. no. 25.
[18] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.
[19] L. Lamport et al., "Paxos made simple," *ACM Trans. SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.
[20] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machines for WANs," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2008, pp. 369–384.
[21] F. Maturana, V. C. Mukka, and K. Rashmi, "Access-optimal linear MDS convertible codes for all parameters," in *Proc. IEEE Int. Symp. Inf. Theory*, 2020, pp. 577–582.
[22] F. Maturana and K. Rashmi, "Convertible codes: Enabling efficient conversion of coded data in distributed storage," *IEEE Trans. Inf. Theory*, vol. 68, no. 7, pp. 4392–4407, Jul. 2022.
[23] F. Maturana and K. Rashmi, "Bandwidth cost of code conversions in distributed storage: Fundamental limits and optimal constructions," *IEEE Trans. Inf. Theory*, vol. 69, no. 8, pp. 4993–5008, Aug. 2023.
[24] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proc. ACM Symp. Operating Syst. Princ.*, 2013, pp. 358–372.
[25] S. Mu, K. Chen, Y. Wu, and W. Zheng, "When paxos meets erasure code: Reduce network and storage cost in state machine replication," in *Proc. ACM Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2014, pp. 61–72.
[26] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2014, pp. 305–320.
[27] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, 1960.
[28] S. Tollman, S. J. Park, and J. Ousterhout, "EPaxos revisited," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2021, pp. 613–632.
[29] M. Uluyol, A. Huang, A. Goel, M. Chowdhury, and H. V. Madhyastha, "Near-optimal latency versus cost tradeoffs in geo-distributed storage," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2020, pp. 157–180.
[30] Y. Wang, Z. Wang, Y. Chai, and X. Wang, "Rethink the linearizability constraints of raft for distributed key-value stores," in *Proc. IEEE Int. Conf. Data Eng.*, 2021, pp. 1877–1882.
[31] Z. Wang et al., "CRaft: An erasure-coding-supported version of raft for reducing storage cost and network cost," in *Proc. USENIX Conf. File Storage Technol.*, 2020, pp. 297–308.
[32] S. Wu, Z. Shen, P. P. Lee, Z. Bai, and Y. Xu, "Elastic reed-solomon codes for efficient redundancy transitioning in distributed key-value stores," *IEEE/ACM Trans. Netw.*, vol. 32, no. 1, pp. 670–685, Feb. 2024.
[33] M. Zhang, Q. Kang, and P. P. Lee, "Minimizing network and storage costs for consensus with flexible erasure coding," in *Proc. Int. Conf. Parallel Process.*, 2023, pp. 41–50.
[34] H. Zhao, Q. Zhang, Z. Yang, M. Wu, and Y. Dai, "SDPaxos: Building efficient semi-decentralized geo-replicated state machines," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 68–81.

Mi Zhang received the BEng degree in software engineering from Shandong University, in 2014, and the PhD degree in computer science and engineering from the Chinese University of Hong Kong, in 2019. She is now an assistant researcher with the Institute of Computing Technology, Chinese Academy of Sciences. Her current research interests include distributed storage systems and storage reliability.

Qihan Kang received the BEng degree in compute science and technology from the University of Chinese Academy of Sciences, in 2021. He is now a master's degree with the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include key-value stores and disaggregated memory systems.



Patrick P. C. Lee received the BEng degree (first-class honors) in information engineering from the Chinese University of Hong Kong, in 2001, the MPhil degree in computer science and engineering from the Chinese University of Hong Kong, in 2003, and the PhD degree in computer science from Columbia University, in 2008. He is now a professor of the Department of Computer Science and Engineering, Chinese University of Hong Kong. His research interests are in various applied/systems topics on improving the dependability of large-scale software systems, including storage systems, distributed systems and networks, and cloud computing.