An Optimized Multi-Paxos Protocol with Centralized Failover Mechanism for Cloud Storage Applications

Wenmin Lin[1], Hao Jiang[1], Nailiang Zhao[2(✉)], and Jilin Zhang[2]

[1] School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China
linwenmin@hdu.edu.cn, Jianghaokobe@163.com
[2] Network and Information Center, Hangzhou Dianzi University, Hangzhou, China
{znl,jilin.zhang}@hdu.edu.cn

Abstract. For typical Multi-Paxos protocol running on a cloud storage application, the failover mechanism is complex in terms of implementation. When the leader fails within a replica group, a new leader should be elected by broadcasting prepare requests over the replica group. Moreover, repairing new leader's missing log entries requires broadcasting prepare request as well. This introduces too much network cost and increase the latency to restore normal storage service at the same time. In view of this challenge, an optimization for Multi-Paxos protocol with centralized failover mechanism for cloud storage applications is proposed in this paper. Compared with typical Multi-Paxos protocol, failover mechanism and normal client requests handling logic are split, and been handled by two clusters respectively: A coordinator cluster is dedicated to handle failover issues as a central manager; while a data cluster only takes charge of data replication and storage regarding client commands. With the centralized failover mechanism in the new design, the centralized coordinator cluster maintains real-time status information of each replica group. And a replica with largest apply index value is elected as the new leader by coordinator cluster; while repairing missing log entries can be achieved with limited replica's bitmap information maintained by coordinator cluster as well. Comparison between two protocols is implemented and analyzed to prove the feasibility of our proposal.

Keywords: Centralized failover mechanism · Multi-Paxos · Replica group · Leader election · Leader repair

1 Introduction

Nowadays, increasing amount of applications are deployed in cloud, due to the convenience of "pay as you go" manner of using IT infrastructure. Among those applications, cloud storage is one of the most popular one. Cloud storage applications enable users to store data of their applications on cloud, instead of building their own storage infrastructures [1, 2]. As a typical distributed computing application, cloud storage

H. Gao et al. (Eds.): CollaborateCom 2018, LNICST 268, pp. 610–625, 2019.
https://doi.org/10.1007/978-3-030-12981-1_43

systems take advantage of replica technique to achieve fault tolerance and high availability, by storing user's data on multiple disks over the network, so as to make sure the data won't be lost as long as majority disks works probably [3].

As a typical distributed computing application, a cloud storage system can be treated as a set of distributed servers belong to one cluster. The servers work as a whole to handle client commands (i.e., write or read operations to store data and read stored data) [4]. Each sever can be described as a deterministic state machine that performs client commands in sequence. The state machine has a current state, and it performs a step by taking as input a client command and producing an output and a new state. The core implementation of a cloud storage system is to guarantee all servers execute the same sequence of state machine commands [5]. As a result, every cloud storage system can be modeled as a replicated state machine as shown in Fig. 1.

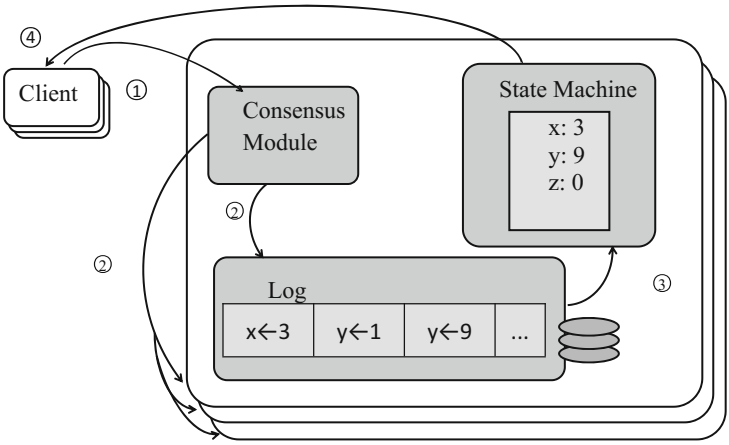

Fig. 1. Replicated state machine architecture [5].

Replicated state machines are typically implemented using a replicated log [4, 5]. Each server stores a log containing a series of client commands, which its state machine executes in sequence. Each log contains the same commands in the same order, so each state machine processes the same sequence of commands. Since the state machines are deterministic, each computes the same state and the same sequence of outputs. Keeping the replicated log consistent is the job of the consensus algorithm [16]. The consensus module on a server receives commands from clients and adds them to its log. It communicates with the consensus modules on other servers to ensure that every log eventually contains the same requests in the same order, even if some servers fail. Once commands are properly replicated, each server's state machine processes them in log order, and the outputs are returned to clients. As a result, the servers appear to form a single, highly reliable state machine.

There have been numerous researches on the consensus algorithm for replicated state machines. Among which Paxos is the dominated one over last decades: most implementations of consensus are based on Paxos or influenced by it. Representative

algorithms include Multi-Paxos [4], E-Paxos [6], as well as Raft [5]. The difference between Paxos and its variants vs. Raft is: Raft is strongly based on leadership mechanism, all client commands are handled by leader replica and other replicas work as followers; while for Multi-Paxos, leader is not necessarily required, but it always employs a distinguished leader to guarantee liveness of the algorithm; Moreover, E-paxos is totally leaderless to guarantee client latency for handling client commands in wide area environment. In this paper, we mainly focus on the optimization of Multi-Paxos regarding its failover mechanism.

For cloud storage systems running typical Multi-Paxos protocol, when the leader fails within a replica group, prepare requests are broadcast over the replica group to elect a new leader; and for each missing log entry in the new leader, repairing it requires broadcasting prepare requests to learn the missing log entry as well. This is both time consuming and introducing too much network cost. In view of this challenge, rather than electing a new leader in a distributed manner, we introduce a centralized failover mechanism to improve system performance of Multi-Paxos protocol in this paper. Briefly, we split failover mechanism and normal client commands handling logic in the new design, and each functionality is handled by a cluster respectively: A coordinator cluster is introduced as a central manager to handle failover issues; A data cluster only takes charge of data replication and storage regarding client commands. Moreover, failover mechanism consists of two phases: leader election and leader repair. The centralized coordinator cluster maintains real-time status information of each replica group. When the leader replica fails, a replica with largest apply index value is elected as the new leader by coordinator cluster; while repairing missing log entries can be achieved with limited replica's bitmap information maintained by coordinator cluster as well.

The reminder of this paper is organized as follows: Sect. 2 discusses related work on consensus algorithm for cloud storage applications. Section 3 highlights the problem of original failover mechanism with typical Multi-Paxos protocol. The details of centralized failover mechanism to optimize Multi-Paxos protocol is presented in Sect. 4. Section 5 compares our proposal and typical Multi-Paxos protocol with respect to message delay and message cost. Section 6 evaluate the performance of optimized Multi-Paxos protocol and typical Multi-Paxos protocol in terms of commit throughput. And Sect. 7 concludes the paper.

2 Related Work

Regarding distributed computing systems such as cloud storage applications [17, 18], the core of implementation is the consensus algorithm, to make sure each server belongs to the same cluster executes client commands in the same sequence [11–13]. There have been numerous researches related to consensus algorithms over last decades [14, 15], from which Paxos is the dominated one. Most implementations of consensus are based on Paxos or influenced by it. Among those consensus algorithms, they can be categorized as follows: (1) Lamport's Paxos [4, 8, 9], and its variants such as Multi-Paxos, Elaborations paxos (E-paxos) [6]; (2) Raft protocol [5], which is based on strong leadership mechanism.

The main difference between those consensus algorithm is the leadership mechanism: (1) Multi-Paxos does not necessarily requires a leader; and when there's no leader in a replica group, Multi-Paxos degrades to Basic-Paxos [10]. (2) E-Paxos is totally leaderless, which is designed to reduce remote client latency. It is a good candidate for wide-area data storage applications. (3) Moreover, Raft uses a strong form of leadership than other two consensus algorithms.

When failover happens, Multi-Paxos, E-paxos and Raft will behave differently. (1) For typical Multi-Paxos, a new leader will be elected by broadcasting prepare request in a replica group. And the leader is elected randomly, which is the first replica receives Prepare OK from majority replicas. Then missing log entry repair is conducted for leader by learning it from Prepare request as well. (2) For E-Paxos, since there's no leader in the replica group, when failover happens, a replica may only need to learn the decision for an paxos instance, since it has to execute commands that depend on that instance. The data repair process is similar to Multi-Paxos, so missing log entries with a replica is learnt from Prepare request as well. (3) For Raft, a leader must be elected before the system can handle more client request, that's because the leader handles all client requests (if a client contacts a follower, the follower redirects it to the leader). Raft uses randomized times to elect leaders, which adds a small amount of mechanism to heartbeats. The implementation simplifies the management of replicated log, and makes Raft easier to understand when comparing with other 2 protocols.

Our work is to optimize Multi-Paxos protocol in terms of system performance when replica failure happens. Compared with typical Multi-Paxos protocol, in this paper, we split the original distributed cluster to coordinator cluster and data cluster. The coordinator cluster is a central manager, which only takes charge of failover issues; while data cluster is only for handling client requests of data storage. This simplifies the implementation of Multi-Paxos protocol. Also, to avoid single point of failure, we made coordinator as a replica group as well, where existing consensus component is applied, so as to reduce complexity.

3 Preliminary Knowledge

3.1 How Multi-Paxos Protocol Works

Multi-Paxos is an optimization of basic Paxos protocol, which is similar to 2-phase commit protocol (i.e., the protocol consists of prepare phase and accept phase). When a replica R_i within a replica group receives a client command C_k, the two phases of basic Paxos protocol work as follows:

Prepare Phase: R_i first record C_k as the k-th client command in its local log, then broadcast *prepare_*C_k requests with proposal number R_i-k within the replica group. On receiving the *prepare_*C_k request for each replica R_j, it will send back *prepare_*C_k*_OK* response to R_i after checking it's ok to log C_k as the k-th log entry locally. If R_i receives *prepare_*C_k*_OK* response from majority replicas, it will enter Accept phase to make C_k as the k-th log entry in majority replicas.

Accept Phase: R_i initiates *Accept_*C_k requests and broadcast it within the replica group. On receiving *Accept_*C_k requests for each replica R_j, it will record C_k as the k-th log entry, and send back *Accept_*C_k*_OK* response after it checks there's no proposal number larger than R_i-k for the k-th log entry. Similarly, when R_i receives *Accept_*C_k*_OK* responses from majority replicas, it will mark C_k as committed; and broadcast *Commit_*C_k requests. Once R_j receives *Commit_*C_k request, it will mark C_k as committed if it has recorded C_k as the k-th log entry as well.

After a command get committed, it can be applied to state machine as long as it has no dependency on other commands, or all its dependency are resolved probably. Therefore, a response will be send back to client to indicate the success of executing C_k by the cloud storage application.

Compared with basic-Paxos protocol, a distinguished replica is elected as leader for Multi-Paxos protocol to improve system's performance by reducing 2-phase commit protocol to 1-phase commit protocol. As shown in Fig. 2, compared with basic Paxos protocol, "Prepare phase" is omitted in Multi-Paxos protocol, since only the leader replica makes proposal in the system. And there's only one phase(i.e., "Accept phase") during the execution of consensus algorithm.

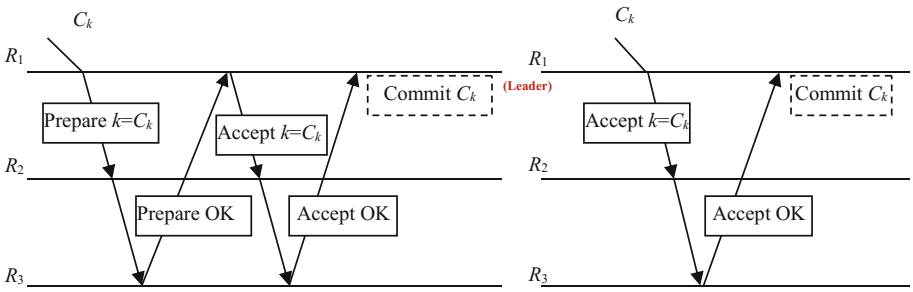

Fig. 2. The difference between basic-Paxos and Multi-Paxos

Multi-Paxos protocol could be treated as an optimization for basic Paxos protocol to address the live lock issue with basic Paxos protocol [4]. The live lock issue means for a same log entry (e.g., the k-th log entry), more than one replica issues *Prepare_*C_k requests within the replica group. In this scenario, according to the basic Paxos design, it's with great possibility that no command will be chosen as the k-th log entry.

Let's take the scenario in Fig. 3 to describe how Multi-Paxos protocol works. A client sends 3 commands C_1, C_2 and C_3 at the same time to the leader replica R_1, i.e., {C_1: "$x = v_1$", C_2: "$y = v_2$", C_3 = "$x^* = v_3$"}. C_1 and C_3 are updating the same key x; while C_2 is updating key y. Then R_1 will log C_1, C_2, and C_3 in sequence at its local log firstly, then broadcast *Accept_*C_1 messages regarding each command to each follower in the replica group. On receiving the *Accept_*C_1 request from R_1, R_2 and R_3 will record the C_1 in its local log; then send back *Accept_*C_1*_OK* message to R_1. Once R_1 receives Accept OK messages from at least follower replica, it will mark C_1 as committed, and broadcast *Commit_*C_1 request to all followers. And on receiving a *Commit_*C_1 message, a follower replica will mark C_1 as committed if it has already record C_1 in its local

log. Once C_1 is committed, it can be applied to the state machine and sends back to client that C_1 has already been recorded correctly. For C_2 and C_3, the workflow is similar to C_1.

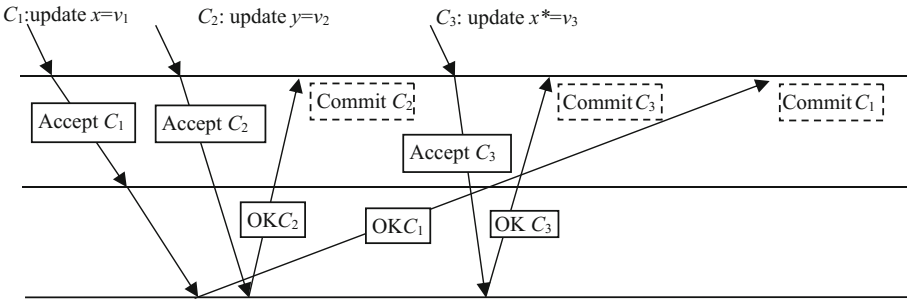

Fig. 3. A Multi-Paxos workflow example

As depicted in Fig. 3, the 3 commands (i.e., C_1, C_2 and C_3) are committed in out-of-order manner (i.e., the committing sequence is $\{C_2, C_3, C_1\}$), since the network delay may results the "*Accept_C*$_2$*_OK*" and "*Accept_C*$_3$*_OK*" messages arrives R_1 before "*Accept_C*$_1$*_OK*" message. And according to the protocol design, the applying sequence of three commands will be: $\{C_2, C_1, C_3\}$, since C_3 has dependency on C_1.

3.2 Problem Statement

Regarding the typical Multi-Paxos protocol design, when the leader replica fails, failover mechanism will be triggered to elect a new leader. And before new leader is elected, Multi-Paxos protocol degrades to basic Paxos: client requests should go through both prepare and accept phases to get committed and applied in the system. The new leader is elected during processing new client requests C_k: On receiving a client request C_k, a replica R_j will first checks whether itself is the leader, if yes it will skip prepare phase, and enter Accept phase directly. Otherwise, R_j will execute prepare phase and accept phase in sequence: it will first broadcast a *Prepare_C*$_k$ request to check whether there's any other proposal regarding the k-th log entry. Once it receives "*Prepare_C*$_k$*_OK*" responses from majority replicas within the replica group, it will initiate "*Accept_C*$_k$" request among the replica group. Once R_j receives "*Accept_C*$_k$*_OK*" response from majority replicas, R_j will promote itself to leader. Otherwise, it will learn which replica is the new leader when it receives *Commit_C*$_k$ request later.

The problem with failover mechanism in typical Multi-Paxos includes 3 parts: (1) Huge network cost: electing a new leader requires broadcasting prepare request over the replica group; (2) Reduced system performance: Multi-Paxos degrades to basic Paxos during the failover period, and it is with great possibility that no leader will be elected, when each replica tries to issue a prepare request regarding the same log entry due to the live lock issue [4]. (3) Data syncing on leader would increase client latency if the new leader has large missing items: after a new leader is elected, the number of missing items

on the new leader would be very large compared with other follower nodes, since the leader is elected randomly. As a result, data syncing on leader node would be quite time consuming, so the client latency of writing operations would be increased.

In view of those challenges, we propose a centralized failover mechanism for Multi-Paxos protocol in this paper. Instead of electing new leader by broadcasting Prepare requests over the cluster, we introduce a designated coordinator taking charge of failover when there's node fails happens in the cluster. With the central coordinator, the 3 problems we analyzed above could be solved accordingly: (1) communication cost is saved since the new leader is elected by coordinator; (2) a new leader will always be elected as long as the majority nodes are not failed; (3) the coordinator will try to elect a follower node with most items as new leader as possible. As a result, the cost of data syncing is saved, so as to reduce client latency on writing operations.

4 A Centralized Failover Mechanism for Multi-Paxos Protocol in Cloud Storage Applications

Motivated by the problem discussed in Sect. 3, the optimized Multi-Paxos protocol is discussed in details in this section. In our proposal, instead of electing the new leader by broadcasting Prepare requests in the distributed cluster, we apply a centralized failover mechanism to optimize the system performance for Multi-Paxos protocol.

4.1 The Architecture of Centralized Failover Mechanism for Multi-Paxos Protocol

The architecture of our optimized Multi-Paxos protocol is depicted in Fig. 4. The implementation of cloud storage are split into two parts: Data Cluster and Coordinator Cluster respectively. Data Cluster is for handling normal data storage logic (i.e., log replication and data storage); while Coordinator Cluster is for failover issues handling.

Definition 1 (Coordinator cluster). Coordinator Cluster works as a centralized manager to handle failover issues. By collecting status report from each replica group, it maintains the status of each replica group, thus it an detect the running status of each replica, and trigger failover mechanism when leader replica fails.

The design of the centralized coordinator is inspired with 2-Phase Commit protocol as discussed in [7]. However, original 2 Phase Commit protocol faces the challenge of typical single point of failure. To achieve high availability, we make the coordinator as a cluster consisting of $2F' + 1$ nodes. Moreover, each node is running consensus algorithm to achieve fault tolerance as well.

Definition 2 (Data cluster). Data cluster is a collection of physical hosts for handling data storage requests, where replicated state machine is implemented, to make sure client requests are handled in the same sequence over several replicas.

As shown in Fig. 4, a data cluster consists of x physical machines, and for each machine p_i, there're a set of multiple processes running consensus algorithm for data storage. A piece of data is stored on $2F + 1$ physical machines as replicas to achieve fault tolerance. Moreover, each $2F + 1$ processes consists of a replica group as shown in Definition 3.

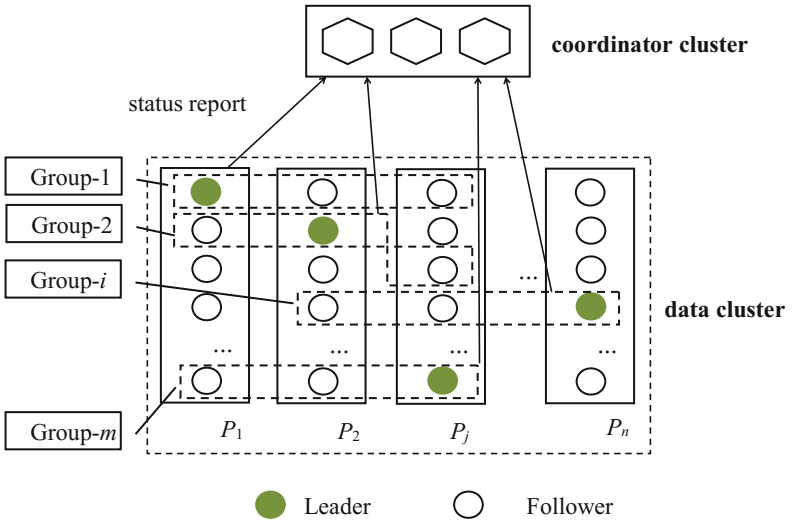

Fig. 4. The architecture of optimized Multi-Paxos with centralized failover mechanism

Definition 3 (Replica group). A replica group consists of $2F + 1$ processes from $2F + 1$ physical hosts. A process is a backup for each other within the same replica group, which is running Multi-Paxos protocol. At one moment, there's one leader elected by coordinator in the group, and other replica works as followers.

The replicas of each group is deployed on different physical machines, so as to make sure exception of one physical host has least impact on the cloud storage system. Also, each replica sends its running status periodically to the coordinator for leader election.

4.2 A Centralized Failover Mechanism for Multi-Paxos Protocol

In this section, the centralized failover mechanism for Multi-Paxos protocol is introduced in details. Table 1 lists some key terms and the definition.

Table 1. Key terms and notification for optimized Multi-Paxos protocol.

Term	Definition
G_i	The i-th replica group
G_i-R_j	The j-th replica of the i-th replica group
C_k	The k-th client request
G_i-R_i-*AppIndex*	The apply index of replica G_i-R_j
G_i-*StatusSet*	The status report collection of the i-th replica group
G_i-R_j-*Bitmap*	The bitmap of log entries existence for G_i-R_j
t	Timeouts for replica group status probing
F	The maximum number of failed replicas within a replica group

Definition 4 (Apply index G_i-R_i-*AppIndex*). R_i-*AppIndex* is the index of a replica R_i evolves in a Multi-Paxos protocol. It indicates that commands C_1, C_2, … $C_{Ri\text{-}AppIndex}$ are already applied from local log to the state machine.

Definition 5 (Status report G_i-R_j.*Status*). G_i-R_j.*status* is the status report of replica R_j from G_i, it can be formulated as G_i-*Rj.status* = <*i*, *j*, *role*, G_i-R_i-*AppIndex*, *health status*>. *i* is the replica group number, *j* is the replica number, role indicates whether the replica is a leader or follower, G_i-R_i-*AppIndex* is the apply index of G_i-R_j, and health status indicates whether the replica running properly.

The state diagram of the coordinator is shown in Fig. 5. When the system starts, coordinator will enter probing state to collect each replica's status report from each replica group. If it detects there's no leader in current replica group, it will trigger failover mechanism, which consists of two phases: leader election and leader repair.

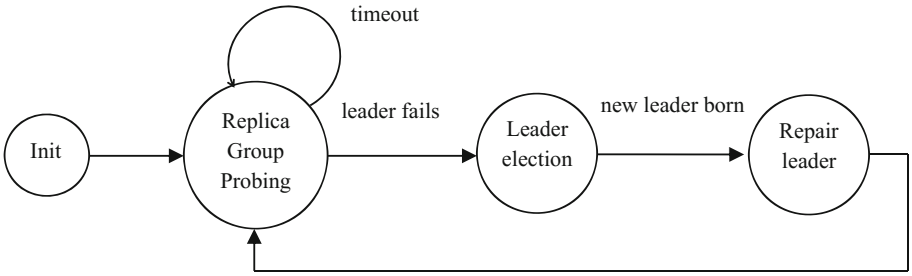

Fig. 5. The state machine of centralized coordinator

4.2.1 New Leader Election

There're three cases when new leader election happens: (1) When the system starts: all replicas within a same group are default to be follower at the beginning. Leader election is triggered to elect a leader for each replica group. Since at this moment, all replicas are the same, the coordinator will randomly assign a replica as the leader. (2) a leader replica report unhealthy status during system running, such as shortage of memory, to indicate it can't perform properly as a leader anymore. (3) a leader does not send status report to leader within timeout t. For case (2) and case (3) the coordinator need to elect a new replica with largest apply index as the new leader. This is to make sure the new leader has as much log entries as possible. This is to save time cost to learn all possible missing log entries. As a result, the replica group will take least time to recovery data storage service to clients.

Please be noted that if a follower reports unhealthy status report or lose connection to the coordinator, new leader election won't triggered. The distributed cloud storage system works fine as long as there's quorum replica works fine within a replica group. Otherwise, the system will break if over *F* replicas are failed.

Algorithm 1 summarizes the new leader election algorithm. Take the case in Fig. 6 for example. A replica group G_i has 3 replicas R_1, R_2 and R_3, where R_1 is the leader. If R_1 reports unhealthy status report or lose connection during the system running, new leader election is triggered. Since R_2's apply index R_2-*AppIndex* = 5, while R_3's apply

index = 3, R_2 will be elected as the new leader. Since both R_2 and R_3 are working properly, the storage service will be restored once R_2 get repaired with all missing log entries.

Algorithm 1. New leader election

```
1. foreach t timeouts:
2.   foreach replica group Gi:
3.     check whether Gi-StatusSet is collected from each replica
4.     if yes:
5.       foreach status report in Gi-StatusSet:
6.         check the health status of each replica Gi-Rj:
7.         case 1: Gi-Rj is running OK
8.               keep probing
9.         case 2: Gi-Rj has health problem
10.              if Gi-Rj is leader:
11.                elect replica with largest Rj-AppIndex as the new leader
12.       endforeach
13.     if no:
14.       check which replica loses the status report
15.       if the leader status is missing and majority replicas are healthy:
16.         elect the replica with largest Rj-AppIndex as the new leader
17.   end foreach
18. endforeach
```

4.2.2 Repair New Leader with Missing Log Entries

When a new leader is elected, we need to repair the new leader to fill all missing log entries before restoring client request handling service.

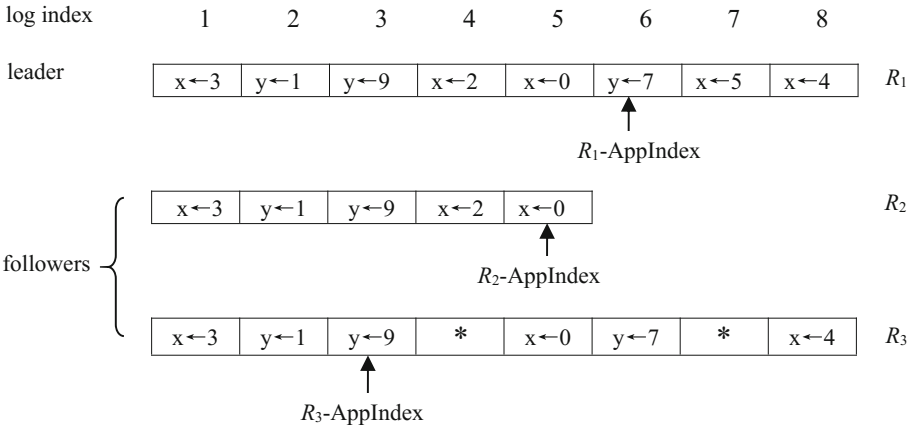

Fig. 6. Example for log entries when failover is triggered

Definition 6 (Bitmap of a replica's log entries G_i-R_j-*Bitmap*): G_i-R_j-*Bitmap* is the bitmap of replica's log entries. For i-th bit in G_i-R_j-*Bitmap*, the Boolean indicates whether R_j has the i-th log entry or not.

Definition 7 (logEntry summary G_i-*logEntry*): For a replica group G_i, G_i-*logEntry* is the log entry summary in G_i. It is an array consisting of $2F + 1$ entries, and G_i-*logEntry* [i] means for the i-th log index, which replica has the log record.

When a new leader is elected, some log entries may be missing due to network delay or missing sync messages from former leader. Before restoring cloud storage service, we need to make sure all committed client commands should be ready in the new leader. As a result, the coordinator should calculate which replica to find the missing log entries. We introduce a bitmap for each replica R_j, where i-th bit indicates whether R_j has the i-th client command. With bitmap from majority replicas, the coordinator can calculate the global G_i-*logEntry* as a summary for log storage within G_i. Moreover, here only majority replicas' bitmap is required due to the design of Multi-Paxos: a command can only be committed after it is accepted by majority replicas in the group. Therefore, the new leader can always obtain the missing log entry once it get committed by former leader.

Algorithm 2. repair new leader with missing log entries in replica group G_i

1. coordinator send bitmap request to F+1 replicas within G_i (except the failed former leader)
2. with the F+1 bitmaps, coordinator calculates the value of G_i-*logEntry*
3. coordinator send G_i-*logEntry* to the new leader
4. **foreach** missing log entry $record_k$ in the new leader's log:
5. check G_i-*logEntry* to get replica R_j, and R_j has $record_k$ in its log
6. send a request to R_j for $record_k$
7. fill the missing log entry with R_j's response
8. **end foreach**

Algorithm 2 summaries the algorithm to repair a new elected leader with missing log entries. Take the scenario in Fig. 5 for example, when R_1 fails, R_2 will elected as the new leader. With bitmap information from both R_2 = {1, 1, 1, 1, 1, 0, 0, 0} and R_3 = {1, 1, 1, 0, 1, 1, 0, 1}, G_i-*logEntry* = {R_2, R_2, R_2, R_2, R_2, R_3, NULL, R_3}. Compared with former leader R_1, R_2's missing log entry is $record_6$. And with G_i-*logEntry*, it can obtain $record_6$ from R_3.

5 Comparison Analysis

In this section, we evaluate our proposal (*OMP*) by comparing it with the typical Multi-Paxos protocol (*MP*) with respect to when failover is triggered. As introduced in Sect. 4, when the coordinator detects there's exception with current leader in a group, failover mechanism is triggered to elect a new leader and repair log entries for the new leader. Two factors are considered in the comparison: message delay to restore normal service; as well as the message cost to repair the new leader with missing log entries.

5.1 Comparison of Message Delay

The message delay means how many round trips is required for the coordinator to choose a new leader and repair missing log entries for the new leader.

For our optimized Multi-Paxos protocol, message delay consists of following parts:

(a) 0.5 round trip to obtain heartbeat messages sent from coordinator to all nodes;
(b) 1 round trip to get bitmap messages required by coordinator;
(c) 1 round trip to get missing log entries, since leader already know where to find missing log entries from G_i-*logEntry*.

As a result, the message delay of the optimized Multi-Paxos protocol is a *const* as listed in formula (1):

$$messageDelay(OMP) = 2.5 \tag{1}$$

While for typical Multi-Paxos protocol, message delay consists of following parts:

(a) $\sum_{i=0}^{m} \alpha^i$ round trips to elect a new leader. α is the probability that conflict happens when electing a new leader. m is the total times of conflict. In optimal case, there's only 1 round trip to elect a new leader, if majority nodes replies with Prepare OK for the first replica initiating Prepare request.
(b) n round trip of Prepare requests to obtain the missing log entries on leader, to broadcast Prepare request to all replicas within the same group.

As a result, the message delay of the typical Multi-Paxos protocol is $n + 1$ as listed in formula (2):

$$messageDelay(MP) = n + \sum_{i=0}^{m} \alpha^i \tag{2}$$

5.2 Comparison of Message Cost

Suppose there're n missing log entries in the new leader node. Message cost include the message produced when a new leader is elected and missing log entries are repaired.

For our proposed optimized Multi-Paxos protocol, the message cost consists of following 4parts:

(a) $2F + 1$ heartbeat messages, from which the coordinator detects leader failure;
(b) $F + 1$ bitmap messages from majority replicas, with which the coordinator decide which replica to be new leader;
(c) 1 log entry summary message (i.e., G_i-*logEntry*) merged with $F + 1$ bitmap messages in (b) is sent to the new leader;
(d) n request messages made by the new leader, to require missing log entries from G_i-*logEntry* in (c);

Combined with (a)–(d), the message produced during failover is $(2F + 1) + (F + 1) + 1 + n = 3F + n + 3 = 3 * (F + 1) + n$.

Furthermore, the message produced for optimized Multi-Paxos could be reduced for step (d) when $n > F$. Since we already know the missing log entry exists in one of the $F + 1$ majority nodes, the maximum request messages in step (d) is $F + 1$. Therefore, the message cost for OMP is $(2F + 1) + (F + 1) + 1 + (F + 1) = 3(F + 1)$ when $n > F$.

$$messageCost(OMP) = 3 * (F + 1) + n \text{ if } n \leq F \tag{3}$$

$$\text{messageCost}(OMP) = 3 * (F + 1) \quad \text{if } n > F \tag{4}$$

For typical Multi-Paxos protocol, the message cost includes 2 parts:

(a) $2 * F * \sum_{i=0}^{m} \alpha^i$ Prepare requests to elect a new leader. Similar to formula (1), α is the probability that conflict happens when electing a new leader. m is the total times of conflict.
(b) $2 * F * n$ requests to repair n missing log entries in the new leader.

Combined (a) and (b), the total message produced by failover mechanism with typical Multi-Paxos protocol is $2 * F * \sum_{i=0}^{m} \alpha^i + 2 * F * n = 2F * \left(\sum_{i=0}^{m} \alpha^i + n\right)$:

$$messageCost(MP) = 2 * F * \left(\sum_{i=0}^{m} \alpha^i + n\right) \tag{5}$$

Table 2 summaries the comparison result, which indicates our optimized Multi-Paxos protocol works better than the typical Multi-Paxos protocol, in terms of both message delay and message cost.

Table 2. Comparison summary of difference between OMP vs. MP

Protocol	Message delay	Message cost
Multi-Paxos (*MP*)	2.5	$3 * (F + 1) + n$ if $n \leq F$ $3 * (F + 1)$ if $n > F$
Optimized Multi-Paxos (*OMP*)	$n + \sum_{i=0}^{m} \alpha^i$	$2 * F * \left(\sum_{i=0}^{m} \alpha^i + n\right)$

6 Evaluation

We evaluated the optimized Multi-Paxos protocol against typical Multi-Paxos protocol, using three replicas for each replicated state machine. The protocols are implemented with Golang [19] and running on Mac OS 10.13.16. For the coordinator node, we apply the existing open source library etcd (which implements Raft protocol) [20] as the centralized manager to monitoring data cluster and electing leader replica.

A client on a separate instance sends batched requests to both three-replica group in loop. And the client requests are initiated by using a replicated key-value store where client requests are updates (i.e., write operations). For each protocol, the evolution of commit throughput in the three-replica setup that experiences the failure of one replica is recorded, as depicted in Fig. 7.

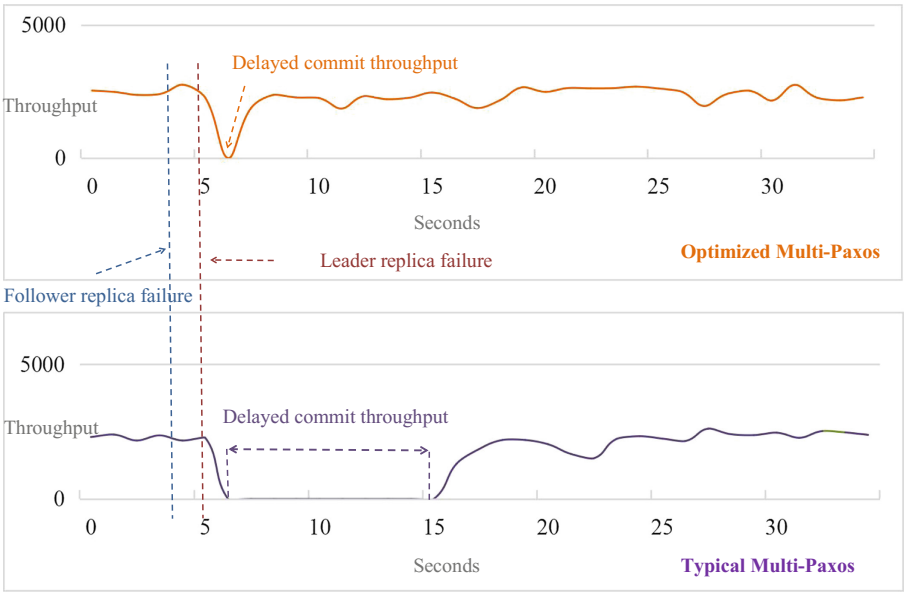

Fig. 7. Commit throughput evolution when one replica fails

As shown in Fig. 7, under normal cases, the commit throughput of both optimized Multi-Paxos protocol and typical Multi-Paxos protocol are almost identical; Moreover, a follower replica fails has no impact on the system's performance. This is due to both protocols are using a stable leader to handle client requests. Furthermore, both protocols suffer from availability issue when the leader replica fails. This is reasonable as well, since a distributed system running either protocol can't process client requests until a new leader is elected.

The difference between two protocols is the impact on commit throughput when the leader of three replicas fails. In our implementation, optimized Multi-Paxos takes about 1 s to elect a new leader and recovery client request handling service; while Multi-Paxos protocol takes about 10 s to restore the service. And the reason for this difference is: when failover happens, for our proposed optimized Multi-Paxos protocol, a new leader is elected by the coordinator node when it loses heartbeat information of the leader, which takes 1 round of message; while in typical Multi-Paxos protocol, new leader is elected during broadcasting prepare requests, which takes at least 1.5 round of message loop among each replica pair within the group. Please be noted, as discussed previously, it's of great opportunity that no leader can be elected in Multi-Paxos protocol due to the mutual stomping issue [4]. Under such circumstances, the commit throughput of typical Multi-Paxos protocol will be delayed infinitely.

7 Conclusion

In this paper, we proposed an optimization with centralized failover mechanism for Multi-Paxos protocol, so as to improve performance of cloud storage applications. Compared with original design of typical Multi-Paxos protocol, failover mechanism and data handling logic are split to different clusters. A coordinator cluster is introduced as a central manager to handle failover issues; while data cluster only takes charge of log replication for data storage. In the new design of failover mechanism, a replica with largest apply index value is elected as new leader; and repair missing log entries is conducted with limited replica's bitmap information. Finally, comparison between two protocols is analyzed to prove the feasibility of our proposal.

Acknowledgement. This paper is supported by The National Key Research and Development Program of China (No. 2017YFB1400601), National Natural Science Foundation of China (No. 61872119), Natural Science Foundation of Zhejiang Province (No. LY12F02003).

References

1. Zeng, W., et al.: Research on cloud storage architecture and key technologies. In: Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human, pp. 1044–1048. ACM, Korea (2009)
2. Arokia, R., Shanmugapriyaa, S.: Evolution of cloud storage as cloud computing infrastructure service. IOSR J. Comput. Eng. **1**(1), 38–45 (2012)
3. Ousterhout, J., Agrawal, P., Erickson, D., et al.: The case for RAM cloud. Commun. ACM **54**, 121–130 (2011)
4. Lamport, L.: Paxos made simple. ACM SIGACT News **32**(4), 18–25 (2001)
5. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of the ATC 2014, Usenix Annual Technical Conference, pp. 1–18 (2014)
6. Moraru, I., Andersen, D.G., Kaminsky, M., There is more consensus in Egalitarian parliaments. In: SOSP, pp. 358–372 (2013)
7. Gray, J., Lamport, L.: Consensus on transaction commit. ACM Trans. Database Syst. **31**(1), 133–160 (2006)
8. David, M.: Paxos Made Simple. http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf
9. Chandra, T., Griesemer, R., Redstone, J.: Paxos made live - an engineering perspective. In: ACM PODC, pp. 1–16 (2007)
10. Lamport, L.: Fast Paxos. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-112.pdf
11. Rao, J., Shekita, E.J., Tata, S.: Using paxos to build a scalable, consistent, and highly available datastore. Proc. VLDB Endow. **4**(4), 243–254 (2011)
12. Ailijiang, A., Charapko, A., Demirbas, M.: Consensus in the cloud: paxos systems demystified. In: 2016 25th International Conference on Computer Communication and Networks, pp. 1–10 (2016)
13. Marandi, P.J., et al.: The performance of Paxos in the cloud. In: Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems, pp. 41–50 (2014)
14. Kirsch, J., Amir, Y.: Paxos for system builders: an overview. In: Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware, pp. 1–5 (2008)

15. Wang, C., Jiang, J., Chen, X., Yi, N., Cui, H.: APUS: fast and scalable Paxos on RDMA. In: Proceedings of the 2017 Symposium on Cloud Computing, pp. 94–107 (2017)
16. Lamport, L., Malkhi, D., Zhou, L.: Reconfiguring a state machine. SIGACT News **41**(1), 63–73 (2010)
17. Xu, X., et al.: An IoT-oriented data placement method with privacy preservation in cloud environment. J. Netw. Comput. Appl. **124**, 148–157 (2018)
18. Xu, X., Fu, S., et al.: Dynamic resource allocation for load balancing in fog environment. Wirel. Commun. Mob. Comput. **2018**, 15 (2018)
19. GoLang. https://github.com/golang/go
20. Etcd. https://github.com/etcd-io/etcd