Fast Follower Recovery for State Machine Replication

Jinwei Guo[1], Jiahao Wang[1], Peng Cai*[1], Weining Qian[1], Aoying Zhou[1], and Xiaohang Zhu[2]

[1] Institute for Data Science and Engineering, East China Normal University Shanghai 200062, P.R. China
[2] Software Development Center, Bank of Communications Shanghai 201201, P.R. China
{guojinwei,jiahaowang}@stu.ecnu.edu.cn,
{pcai,wnqian,ayzhou}@sei.ecnu.edu.cn,
zhu_xiaohang@bankcomm.com

Abstract. The method of state machine replication, adopting a single strong Leader, has been widely used in the modern cluster-based database systems. In practical applications, the recovery speed has a significant impact on the availability of the systems. However, in order to guarantee the data consistency, the existing Follower recovery protocols in Paxos replication (e.g., Raft) need multiple network trips or extra data transmission, which may increase the recovery time. In this paper, we propose the **F**ollower **R**ecovery using **S**pecial mark log entry (FRS) algorithm. FRS is more robust and resilient to Follower failure and it only needs **one** network round trip to fetch the **least** number of log entries. This approach is implemented in the open source database system OceanBase. We experimentally show that the system adopting FRS has a good performance in terms of recovery time.

Keywords: Raft, State machine replication, Follower recovery

1 Introduction

Paxos replication is a popular choice for building a scalable, consistent and highly available database systems. In some Paxos variants, such as Raft [1], when a replica recovers as a Follower, it is ensured that its state machine is consistent with the Leader's. Therefore, the recovering Follower has to discard the invalid log entries (e.g., they are not consistent with the committed ones) and get the valid log records from the Leader. Figure 1 shows an example of log state of 3-replica system at a particular point in time. R1 is the Leader at that moment, and R3 which used to be the Leader crashes.

The *cmt* (*commit point*) indicates that the log before this point can be applied to local state machine safely, which is persistent in local. The *div* (*divergent point*) is the log sequence number (LSN) of the first log entry which are invalid

* Corresponding author.

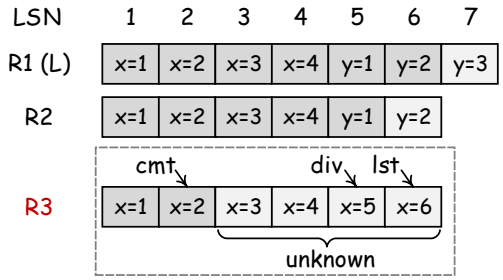

Fig. 1. An example of log state of 3-replica at one point in time.

in log order. The *lst* is the last LSN in the log. Note that the *cmt*, *div* and *lst* of R3 are 2, 5 and 6 respectively. Therefore, when R3 restarts and recovers, it can reach a consistent state only if it discards the log in the range $[5, 6]$, gets remaining log from the Leader (R1) and applies the committed log entries.

Unfortunately, R3 does not know the state of the log after the LSN point 3. The existing approaches of Follower recovery do not pay attention to locating the *divergent point* directly. In Raft [1], the recovering Follower discards the last entry in local by executing AppendEntries remote procedure call (RPC) repeatedly until it finds the *divergent point*[3]. This approach need numerous network interactions. The Spinnaker [2] truncates the log after *cmt* (which may be *zero* if it is damaged) logically and gets the write data after *cmt* from the Leader, which increases the number of transmitted log entries and need a complicated mechanism to guarantee the data consistency of log replacement.

In this work, we present the **F**ollower **R**ecovery using **S**pecial mark log entry (FRS) algorithm, which does not depend on *commit point* strongly and requires only one network round trip for fetching the minimum log entries from the Leader when a Follower is recovering. The following is the list of our main contributions.

- We give the notion of the special mark log entry, which is the delimiter at the start of a term.
- We introduce the FRS algorithm, explain why this mechanism works and analyze it together with other approaches.
- We have implemented the FRS algorithm in the open source database system OceanBase[4]. The performance analysis demonstrates the effectiveness of our method in terms of recovery time.

2 Preliminaries

2.1 The overview of Paxos replication

Paxos replication, adopting the *strong leadership* and *log coherency* features of Raft, is introduced first. During normal processing, only the Leader can accept

[3] There is a optimization in Raft for reducing the number of network interactions, but the optimized approach does not find the *divergent point* directly yet.

[4] `https://github.com/alibaba/oceanbase/`

the write requests from clients. When receiving a write, the Leader generates a log record which contains the monotonically increasing LSN, its own *term_id* and updated data. The Leader replicates the log record to all Followers. When the entry is persisted on a majority of nodes, the Leader can apply the write to local state machine and update the *cmt* piggybacked by the next log message. The Followers only store successive commit logs, and replay the local log before the point $(cmt + 1)$ in order of log sequence.

The Leader sends heartbeats to all followers periodically in order to maintain its authority. If a Follower finds that there is no Leader, it becomes a Candidate and increases the current *term_id*, and then launches a Leader election. According to the newest log entry, a new Leader with the highest *term_id* and LSN can provide services only when it receives a majority of votes.

2.2 Handling log entries in unknown state

Recall that a replica node flushes a log entry to local storage first. Then the log entries, whose LSN is $\leq cmt$, can be safely applied to local state machine. However, the state of the log after *cmt* is unknown. In other words, a log entry whose LSN is greater than *cmt* may be committed or invalid.

Fortunately, a recovering Follower f can get the *cmt* from local—which is denoted by $f.cmt$—and ensure that the log entries whose LSN is equal to or less than this point are committed. Next, it needs to handle the local log after $f.cmt$ carefully. There are two ways to handle the log entries in unknown state:

- **Checking (CHK)**: The recovering Follower f gets the next log entry from the Leader l, and then checks whether it is continuous with local last log record. If not, it discards the log entry and repeats the above process; otherwise, it is back to normal.
- **Truncating (TC)**: The recovering Follower f gets the log after the local *cmt* from the Leader l. Then it replaces the local log after the *cmt* with received log, all of these operations should be done atomically.

It is clear that the CHK is simple and the TC is complicated, because the replacement operation of TC can be interrupted and more steps are required to handle each exceptions, e.g., if a recovering Follower fails again and the appending operation is not finished, it has to do the appending when it restarts. Both of approaches can not locate the *div* directly, which leads to more network round trips or more transmitted log entries in the Follower recovery. We will propose a new approach in the next sections, which needs only one network round trip for getting the minimum number of log entries.

3 The Special Mark Log Entry

In order to reduce the overhead of Follower recovery, we must provide additional mechanism, which can record necessary information used in the recovery of a

Follower. In this section, we introduce the special mark log entry and how a new Leader utilizes the special entry to take over the requests form the clients.

A special mark log entry S is the delimiter at the start of a term. Let S_i and $\mathcal{S}$ denote the special mark log entry of the term i and the set of all existing special entries respectively. And we use the notation $l.par$ to access the parameter named par of a log entry l (e.g., $S_i.lsn$ represents the LSN of special log entry S_i). In order to distinguish the other log entries from the special ones, we call them the normal operation log entries, which are the members of the set $\mathcal{N}$. And a mark flag is embedded in each log entry. An S, differing from the normal operation log entries, does not contain any operation data except the mark flag which is set to **true**.

When a replica is elected as a new Leader, a new term gets started. The new Leader must take some actions—which guarantee that the local log entries from previous term are committed—before it provides normal services. Therefore, the Leader using special mark log entry has to take following steps to take over:

(1) According to the new term id t, the Leader generates the special mark log entry S_t. More specifically, it produces a log record with a greater LSN and the new term id t, and sets its $mark\_flag$ to true. Then the Leader sends S_t to all other replicas.
(2) The Leader gets local *cmt*, and replays local log entries to this point in the background. It obtains the Followers' information and refreshes the *cmt* periodically until this point is equal to or greater than $S_t.lsn$. Note that the Leader can not service the requests from the clients in this phase.
(3) The Leader can safely apply the whole local log entries to local memory table. After that it can provide the clients with the normal services.

4 Follower Recovery

When a replica node recovers as a Follower from a failure, it has to take some measures to ensure that its own state is consistent with the Leader's. In other words, the recovering Follower has to discard the inconsistent log entries in local storage and get the missing committed log from the Leader. Then it can apply the log records to local memtable, so that the Follower can reach a consistent state with the Leader. The procedure of Follower recovery using special mark log entry (FRS) is shown as follows:

(1) The recovering Follower gets the *cmt* information and the last LSN *last_point* from local first. Then it obtains the last committed special log entry s (i.e., there exists a log entry l where $l \in \mathcal{N}$ and $l.lsn > s.lsn$). Two variables *start* and *end* are set to `max(`*commit_point*`,` $s.lsn$`)` and *last_point* respectively. The *start* indicates the last committed log entry's LSN, which can be figured out by the Follower itself. After that, the Follower sends a confirm request to the Leader with these information.
(2) When the Leader receives the confirm request from the Follower, it gets the embedded parameters *start* and *end* first. Then the Leader obtains the

first special log entry s in the log range $(start, end]$. If no one satisfies the requirement, the Leader replies a result value *zero* to the Follower; otherwise, it returns the LSN of the record s.

(3) When the Follower receives the response of the confirm request, it checks the result value v. If v is not *zero*, the Follower will first discard the entries from the local log after the index which is equal to $v - 1$. Then it gets the rest log from the Leader, and appends these entries to the local. If v is *zero*, the Follower will not copy with the local log.

When a Follower recovers from a failure, it first the above procedure to eliminate invalid log entries in local storage and to acquire the coherent ones from the Leader. Then it processes the log replication using the normal approaches described in Section 2.1. Note that the Follower can apply the log entries—whose LSNs are $\leq$ the local *cmt* —to local state machine in parallel with the recovery.

The FRS algorithm is applied not only to the restarting from a failure but also to the scenario where a Follower finds that a new Leader is elected. Generally, if the term is changed, the lease of previous Leader is expired, and the Follower will turn to Candidate and then convert to Follower again when it knows that the new Leader is not itself. In this case, the Follower executes the FRS algorithm actively. There is another case that a Follower receives a special mark log entry or a log entry containing a newer term value. In this scenario, the Follower dose not change its role, and it only discards the buffered log and executes the FRS algorithm as well.

5 Performance Evaluation

5.1 Experimental setup

We conducted an experimental study to evaluate the performance of the proposed FRS algorithm, which is implemented in OceanBase 0.4.2.

Cluster platform: We ran the experiments on a cluster of 12 machines, and each machine is equipped with a 2-socket Intel Xeon E5606 @2.13GHz (a total of 8 physical cores), 96GB RAM and 100GB SSD while running CentOS version 6.5. All machines are connected by a gigabit Ethernet switch.

Database deployment: The Paxos group (RootServer and UpdateServer as a member) is configured with 3-way replication and each of them is deployed on a singe machine in the cluster. Each pair of MergeServer and ChunkServer is deployed on one of the other 9 servers.

Competitors: We compare the FRS algorithm to other approaches described in the Section 2.2. In order to increase efficiency, the `CHK` approach is responsible for locating the *divergent point*. When finding the log index, the recovering Follower requests the log after *div* (including) as a group to the Leader. Therefore, the number of requests of `CHK` is about half of the results described in Raft. The Follower adopting the `TC` approach gets the log after *cmt* which is contained by one message package from the Leader.

Benchmark: We adopted YCSB [3]—a popular benchmark with key-value workloads from Yahoo—to evaluate our implementation. Since we pay attention

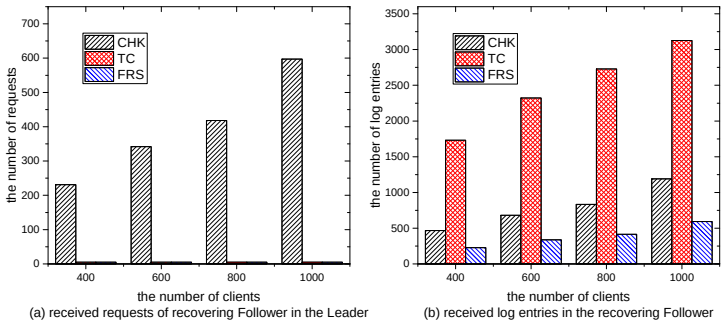

Fig. 2. The statistics of a recovering Follower, which used to be a Leader and failed in different workloads measured by the number of clients.

to the Follower recovery relying on log replication, we modify the workload to have a read/write ratio of 0/100. The clients, which request the writes to the database system, are deployed on the MergeServer/ChunkServer nodes. the size of each write is about 100 bytes value.

5.2 Experimental results

To measure the Follower recovery, we need to kill a replica node and then restart it. Therefore, each experimental case is conducted as follows first. We ensure that the three replicas work normally and about one million records are inserted into the system. Next, we make the Leader r disconnect from the other replicas before its lease expires in 2 seconds. When r loses the leadership and a new Leader is elected, we kill and restart r. Then r recovers as a Follower.

Follower recovery statistics: We first measure the statistic of the Follower recovery in terms of received requests in the Leader and received log entries in the recovering Follower. Therefore, we get the results by adding code to cumulate the corresponding values in the program.

Figure 2 shows the statistics of a recovering Follower, which fails in different workloads denoted by different numbers of clients. As the number of clients increases, we find that the size of the log after local *cmt* or *div* becomes larger, which can lead to more requests and received log entries in the Follower recovery. Figure 2(a) shows that `CHK` approach needs hundreds of requests to locate the *div* point, and the other approaches need only few network interactions in the recovery phase. Figure 2(b) shows that the `TC` transmits the most log entries containing the unnecessary ones. Since the log transmitted in locating *div* is not preserved, the number of log received are about twice as the `FRS`'s. All of these results conform to the analysis described above.

Follower recovery time: Note that the recovery of a Follower has two phases: handling log in unknown state and applying the log to local machine state. Since the applying phases of all the Follower recovery approaches are the same, we only

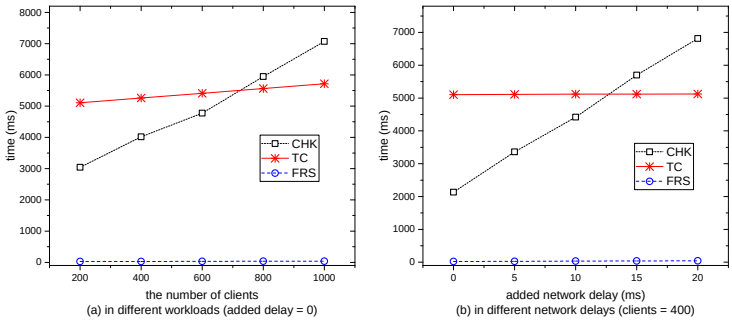

Fig. 3. Follower recovery time with invalid *cmt*.

measure the time of handing the uncertain log entries. In this experiment, the recovering Follower fails when the number of clients (workload) is 400.

Figure 3 shows the Follower recovery time with invalid *cmt* when the system is in different workloads or network delays. We use Linux tool `tc` to add network delay in the specified ethernet interface of the recovering Follower. We find that the recovery time of `FRS` is shortest and is not impacted by workloads and network delays at all. As the workload increases in Figure 3(a), the result of `CHK` was linearly correlated with the number of clients, which indicates that the higher workload can lead to more time of handling a request in the Leader. A recovering Follower adopting `TC` needs to get the whole log from the Leader. Due to the long time of transforming the whole log, the recovery time of `TC`—which is between 5s and 6s—is not desired. The trend in Figure 3(b) is similar to 3(a). Therefore, we can conclude that the `FRS` has the best performance in terms of recovery time, no matter what the workload or the network delay is, and no matter whether the *cmt* is valid.

6 Related Work

State machine replication (SMR) [4], a fundamental approach to designing fault-tolerant services, can ensure that the replica is consistent with each other only if the operations are executed in the same order on all replicas. In reality, database systems utilize lazy master replication protocol[5] to realize SMR.

Paxos replication is widely used to implement a highly available system. [6] describes some algorithmic and engineering challenges encountered in moving Paxos from theory to practice and gives the corresponding solutions. To further increase understandability and practicability, some multi-Paxos variants adopting strong leadership are proposed. In Spinnaker [2], the replication protocol is based on this idea. Nevertheless, its Leader election relies an external coordination service ZooKeeper [7]. Raft [1] is a consensus algorithm designed for educational purposes and ease of implementation, which is adopted in many open source database systems, e.g., CockroachDB [8] and TiDB [9]. Although these protocols can guarantee the correctness of Follower recovery, they neglect to locate the *divergent point* directly.

There are many other consensus algorithms which are similar to multi-Paxos. The Viewstamped Replication (VR) protocol [10] is a technique that handles failures in which nodes crash. Zab [11] (ZooKeeper Atomic Broadcast) is a replication protocol used for the ZooKeeper [7] configuration service, which is an Apache open source software implemented in Java. In these protocols, When a new Leader is elected, it replicates its entire log to Followers. [12] presents the differences and the similarities between Paxos, VR and Zab.

7 Conclusion

Follower recovery is not non-trivial in Paxos replication systems, which not only guarantees the correctness of data, but also should make the recovering replica back to normal fast. In this work, we introduced the Follower recovery using special mark log entry (FRS) algorithm, which does not rely on the *commit point* and transmits the least number of log entries by using only one network round trip. The experimental results demonstrate the effectiveness of our method in terms of recovery time.

Acknowledgments. This work is partially supported by National High-tech R&D Program (863 Program) under grant number 2015AA015307, National Science Foundation of China under grant numbers 61432006 and 61672232, and Guangxi Key Laboratory of Trusted Software (kx201602). The corresponding author is Peng Cai.

References

1. Ongaro, D., Ousterhout J.: In search of an understandable consensus algorithm. In: ATC, pp. 305–320 (2014)
2. Rao, J., Shekita, E.J., Tata, S.: Using Paxos to build a scalable, consistent, and highly available datastore. In: VLDB, pp. 243–254 (2011)
3. Cooper, B.F., Silberstein, A., Tam, E., et al: Benchmarking cloud serving systems with YCSB. In: Socc, 143–154 (2010)
4. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Comput. Surv. 22(4), 299–319 (1990)
5. Gray, J., Helland, P., O'Neil, P., et al: The Dangers of Replication and a Solution. SIGMOD Rec. 25(2), 173–182 (1996)
6. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos Made Live: An Engineering Perspective. In: PODC, pp. 398–407 (2007)
7. ZooKeeper website. `http://zookeeper.apache.org/`.
8. CockroachDB website. `https://www.cockroachlabs.com/`.
9. TiDB website. `https://github.com/pingcap/tidb`.
10. Oki, B.M., Liskov, B.H.: Viewstamped replication: A new primary copy method to support highly-available distributed systems. In: PODC, pp. 8–17 (1988)
11. Junqueira, F.P., Reed, B.C., Serafini, M.: Zab: High-performance Broadcast for Primary-backup Systems. In: DSN, pp. 245–256 (2011)
12. Van Renesse, R., Schiper, N., Schneider, F.B.: IEEE TDSC 12(4), 472–484 (2015)