Fast Log Replication in Highly Available Data Store

Donghui Wang[1], Peng Cai*[1], Weining Qian, Aoying Zhou[1],
Tianze Pang[2], and Jing Jiang[2]

[1] Institute for Data Science and Engineering at East China Normal University,
Shanghai 200062, People's Republic of China
[2] Software Development Center at Bank of Communications,
Shanghai 201201, People's Republic of China
{donghuiwang}@stu.ecnu.edu.cn,
{pcai,wnqian,ayzhou}@sei.ecnu.edu.cn,
{pangtz,jiangj_5}@bankcomm.com

Abstract. Modern large-scale data stores widely adopt consensus protocols to achieve high availability and throughput. The recently proposed Raft algorithm has better understandability and widely implemented in large amount of open source projects. In these consensus algorithms including Raft, log replication is a common and frequently used operation which has significant impact on the system performance. Especially, since the commit latency is capped by the slowest follower out of the majority followers responded to the leader, it's important to design a fast scheme to process the replicated logs by follower nodes. Based on the analysis on how the follower node handles the received log entries in Raft algorithm, we figure out the main factors influencing the duration time from when the follower receives the log and to when it acknowledges the leader this log was received. In terms of these factors we propose an effective log replication scheme to optimize the process of flushing logs to disk and replaying them, referred to as Raft with Fast Followers (FRaft). Finally, we compare the performance of Raft and FRaft using YCSB benchmark and Sysbench test tools, and experimental results demonstrate FRaft has lower latency and higher throughput than the Raft only using straightforward pipeline and batch optimization for log replication.

Keywords: Log replication; High availability; Consensus system; Raft

1 Introduction

Today's modern applications often require the back-end data store not only to provide the acceptable system performance but also to meet the high availability requirements. State machine replication is regarded as the most general approach to implementing a highly available data store where the data is replicated across a set of servers and consensus protocols are used to guarantee the consistency

* Corresponding author.

among different copies. Consensus protocols, including Paxos or its variants [11, 17, 18], Viewstamped Replication[19] and Zab[15], reach an agreement on each operation and ensure all replicas execute the operation in the same order. Consensus is the fundamental problem in distributed systems and these protocols have became the key component of large-scale and fault-tolerant data store[10, 21, 6].

In contrast to the famous Paxos protocol, the recently proposed Raft algorithm has better understandability and widely implemented in large amount of open source projects [20, 2]. During the execution of these consensus protocols including the recently proposed Raft, log replication is a common and frequently used operation which has significant impact on the system performance. In Raft, a transaction can be committed if its log has been replicated on the majority of followers. However, log replication algorithm also comes with inevitable performance problems because of the latency caused by network and processing time in followers (mainly from disk latency).

Raft achieves consensus among a group of members via an elected leader. Only leader can accept new request from clients, and then replicates log entries to followers. When the leader accept the acknowledgment from the majority of followers, it commits the transaction and both leader and followers can safely apply log entries to their replicas. In the naive implementation of Raft, the leader propagates one request at a time. In general, this is highly ineffective because multiple network transmissions increase the delay of each request. There are two optimizations widely used in the implementation of consensus protocols [13, 7, 5, 12]: batching and pipeline. Batching is to pack several requests into a single *AppendEntries* RPC, which spread the overhead on a set of requests. Pipeline allows the leader to propagate a new *AppendEntries* RPC to followers before the previous ones are acknowledged [16]. Pipeline can effectively improve the throughput especially in the WAN network with high latency.

Although batching and pipeline can improve the performance of Raft consensus protocol, the follower still needs to wait for flushing a batch to disk before processing the next batch in the task queue which holds the many batches sent by the leader. On the other hand, the strategy of replaying logs after they are committed incurs a large amount of expensive memory copy operation (see the details in the problem analysis section). To address these challenges, we redesign the log replication scheme for Raft protocol. The basic idea is to separate flushing a batch log from the log processing flow. Instead of directly writing the received batch logs to disk by followers, the batch is immediately moved from the task queue to a batch buffer. By this way, the next batch can be handled without any blocking. A single thread is used to monitor the batch buffer, and asynchronously flush a group of batch to disk in order to reduce disk IO overhead. Furthermore, in order to decrease the operations of memory copy, the received logs are also replayed immediately but the applied results are invisible until the corresponding transaction are committed.

The time consuming on processing the logs by follower has significant impact on the throughput and the end-to-end transaction response time as perceived by

the user. In this paper, we optimize the log flushing and replay in the follower, referred to as Raft with Fast Followers (FRaft). Since FRaft has the advantage of reducing the follower servers' processing time, it has lower latency and higher throughput than the Raft only using straightforward pipeline and batch optimization for log replication.

We summarize our contributions of this paper as follows:

- According to the analysis of processing log by followers from engineering perspective, we claim that follower servers' processing time has became one of obstacles for performance promotion in Raft protocol.
- We propose a new log replication approach based on Raft, namely FRaft, that the follower is designed to more effectively handle the received logs and pre-replay the logs to avoid additional memory copy. Accordingly, we also present the corresponding recovery strategy.
- We implemented Raft and FRaft based on the open source database OceanBase developed by Alibaba.
- We conduct extensive experiments to evaluate Raft and FRaft under different benchmarks with YCSB and Sysbench test tools.

The paper is organized as follows. In Section 2, we present the analysis of log replication in Raft and introduce a motivation example that demonstrates the bottleneck of log processing in the follower. Section 3 present our proposed log replication approach FRaft and its strategy of relaying logs. In section 4, we list the recovery approach under different anomalies. In section 5, we conduct an experimental study to compare the performance of different approaches. In section 6, we introduce the related works, and conclude this paper in section 7.

2 Problem Analysis

2.1 Log replication in Raft

Figure 1 demonstrates the basic log replication scheme without any optimizations in Raft [20]. Both leader and follower maintain a *commitIndex* which means log entries before it have been applied. Firstly, when receiving a new request from a client, the leader appends corresponding log entries to disk and replicates it to all followers by broadcasting *AppendEntries* RPCs encapsulated log entries and *commitIndex*. Secondly, each follower appends these log entries to disk and pushes it into *commitqueue* waiting for being applied to memtable(which is often implemented with B+ Tree or SkipList by in-memory DBMS or Key-Value data store) in the next round, and then sends acknowledgment to leader. When the majority of followers return success, the leader updates its *commitIndex* and applies log entries to leader's replicas. Finally, the leader will broadcast new *commitIndex* in the next *AppendEntries* RPC. Once follower learns *commitIndex* of leader, it applies log entries whose index is between its own *commintIndex* to leader's *commitIndex* to memtable. This means the log entries received and flushed in this round will be reread for applying in the next round, which will incur additional memory copy and thus increase

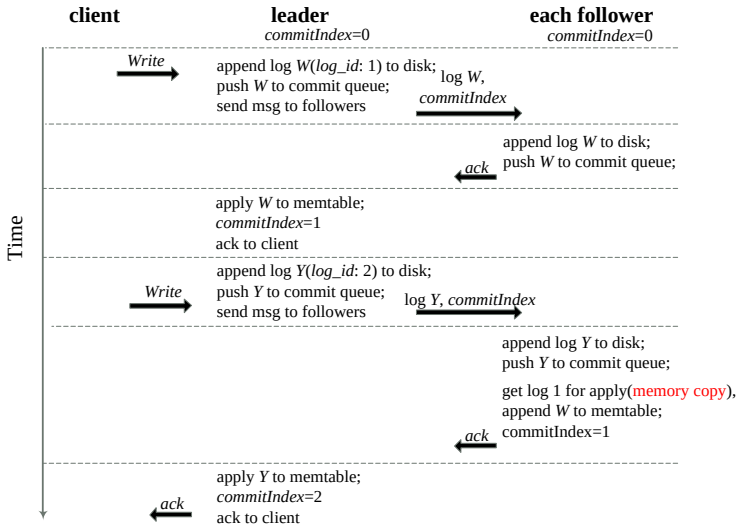

Fig. 1. The basic log replication scheme in Raft without any optimizations.

transaction latency. In addition, the approach of writing log to disk directly in followers causes high disk I/O and increases the waiting time of next log being processed.

2.2 A Motivation Example

Figure 2 presents the Raft with batch and pipeline optimization. Similar to standard Raft, follower appends the received log entries from leader to the disk sequentially for durability while these uncommitted log entries cannot be directly applied to memtable. Both leader and follower keep track of the $commitIndex$. Once a follower knows $commitIndex$ of leader, it will apply the log entries whose log index between $commitIndex$ of its own to $commitIndex$ of leader to memtable.

However, in practice, follower should reread the log entries from disk and apply them to memtable which causes highly disk I/O usage during log replication. An optimized way is caching these uncertain log entries into buffer to avoid disk I/O, but still invokes additional memory copy. As shown in Figure 2, follower appends a batch log entries to the $commitQueue$ which holds the uncertain log entries. Then, reading log entries for applying from $commitQueue$ will produce a memory copy. Moreover, as follower appends the received log entries to disk directly which requires a disk I/O for each batch log entries, which increase the overall I/O cost of processing logs. Even worse, the next batch log entries is blocked in the $processQueue$ and can not be processed until the prior one finished.

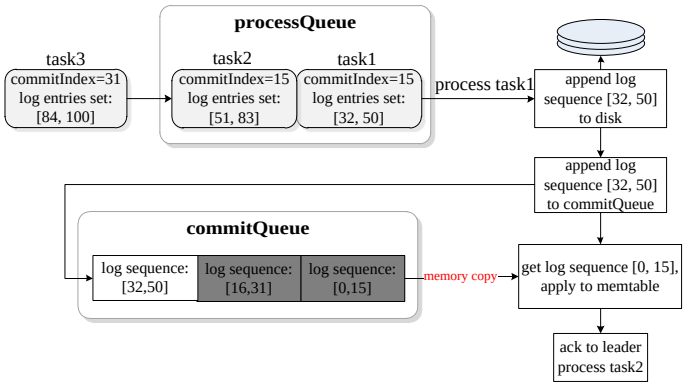

Fig. 2. A motivation example of follower's processing on Raft, with batch and pipeline optimization.

3 Fast Log Replication Approach

FRaft optimizes the process of appending log entries to disk and applying log entries to memtable. We append a batch of log entries to a in-memory buffer at first, and use a designated thread to flushing several batches via one disk I/O; Also, log entries are applied ahead of time to avoid additional memory copy.

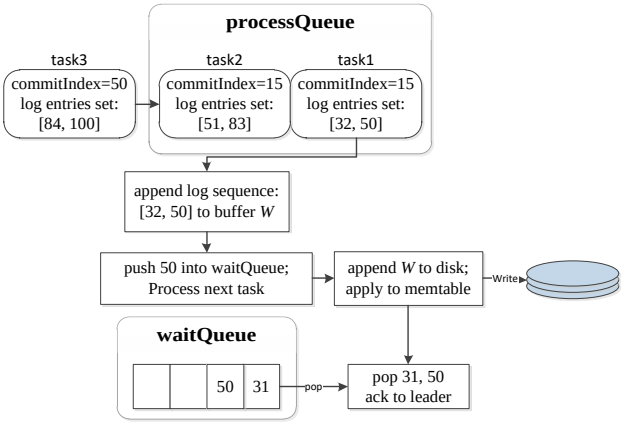

Fig. 3. Processing logs by the follower in FRaft

Algorithm 1 Follower processing algorithm on FRaft

```
1: procedure FollowerProcess(processQueue)
2:     while TRUE do
3:         task ← processQueue.pop()
4:         resolove task
5:         log_sequence ← task.logSequence
6:         log_end_index ← task.log_end_index
7:         append log_sequence to W
8:         waitQueue.push(log_end_index)
9:     end while
10: end procedure
1: procedure Replay(W)
2:     while TRUE do
3:         append W to disk
4:         for all log_sequence ∈ W do
5:             log_end_index ← log_sequence.log_end_index
6:             apply log_sequence to memtable
7:             if waitQueue.head <= log_end_index then
8:                 waitQueue.pop()
9:             end if
10:            send response to client
11:        end for
12:    end while
13: end procedure
```

3.1 Process of the Follower

Figure 3 shows the fast scheme to process the replicated log entries by follower nodes in FRaft. The *processQueue* is used for receiving tasks sent by leader. Firstly, we resolve the log entries of the head task of *processQueue* and append them to a specified buffer W. This approach decrease the stall time of each batch in the *processQueue*. Secondly, this task is pushed into *waitQueue* to wait for responding to client when log entries of this task has been appended to disk. Since more batches are flushed to disks only once by a single thread, the wait time of a task in *waitQueue* is also reduced. In FRaft, the next task in *processQueue* will be handled immediately.

Algorithm 1 provides a high level description of the FRaft log replication approach. The procedure *FollowerProcess* depicts the tasks sent by leader processing of followers. A single thread's processing method used to append log sequence to disk and apply to memtable are shown in the procedure *Replay*. The leader in FRaft, like it in Raft, takes charge of sending log entries set to followers with batching and pipelining optimizations. This approach brings a new problem that log entries have not been applied to memtable but have been applied in the followers so that we may read an updated state which have not been applied in the leader from the follower. We solve this problem by a special apply strategy.

3.2 New Apply Strategy

In order to achieve higher speed of applying log entries, we adopt multiple threads(apply workers) for applying. When apply workers are running, log entries are applied to memtable in parallel so that the update states are chained to the update list. However, these updated states which should agree on sequentiality is invisible at this moment. We push the applied log index into a sorted *publishPool* to ensure that the update states are published in sequences. Consequently, these undetermined update states cannot be accessed by clients although we apply them to local memtable. We leverage the *publishPool* to publish the update states. When the follower receives *commitIndex* from the leader, it will get relevant tasks which $task.logIndex <= commitIndex$ from *publishPool* to publish them. Figure 4 shows the processing in detail.

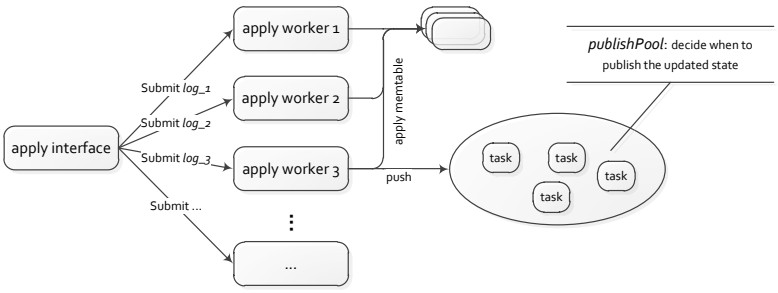

Fig. 4. Apply Strategy in the Followers.

4 Recovery

The most common types of failure are network anomaly and server failure. In FRaft, the recovery strategy is a bit different from Raft as we actually apply the undetermined log entries to memtable ahead of time in followers.

Network anomaly. Network anomaly is mainly caused by the network jitter, delay or network packet that may lead to some abnormal phenomena for the leader that it cannot receive the responses from followers, and cannot apply the transaction; for the follower, network anomaly may lead receiving a discontinuous log sequence. The leader won't apply the transaction or retry to send *AppendEntries* RPCs when it doesn't receive the most majority of members' responses, because there is no way to know whether the followers receive the log entries or not. When the leader sends the next RPC combined with *commintIndex*, the followers check if the log entries is consecutive. If it is discontinuous, the followers will ask for the leader to obtain the missing log entries. The follower won't response to leader until the log is consecutive.

Server failure.Recovery strategy should ensure that the system is restored to the latest state before fault and there is no serious problem of data loss when

a random server fails. If a server restarts as the leader, it will apply the log before *commitIndex* firstly, and then wait for log entries from *commitIndex* to *lastestIndex* are obtained by all followers before providing service. Otherwise, if a server restarts as a follower, it will obtain the log entries between its own *commitIndex* and the leader's *commitIndex* firstly.

Special case. In FRaft, there exists a bit difference from Raft as the log entries applied to memtable ahead of time in followers so that the update states are chained to the update list, although it is invisible. We need to revoke update states that have not been replicated on the majority of followers from the update list in some specific cases. There are some other strategies for recovery in FRaft which is not elaborated here due to length limitations.

5 Experiment

This section evaluates the performance of FRaft and Raft which are both implemented in UpdateServer in-memory database engine of OceanBase [1]. There are many UpdateServers in OceanBase and each UpdateServer represents a replica where transactions are dealt with by the leader UpdateServer, log entries are replicated on all follower UpdateServers. When the leader fails, new leader can be elected from the remaining servers to takeover the system that guarantees consistency and high availability.

The experiment is divided into two parts. Comparison of FRaft and Raft from several aspects is evaluated in part 1 by running on the public benchmark YCSB 0.7.0 [9]. In part 2, primary-copy log replication and FRaft implementation based on OceanBase are compared in case of banking business by using Sysbench test tools [3], which aims to illustrate that FRaft ensures strong data consistency without sacrificing too much performance in action production.

In part 1, there are 3 UpdateServers(3 replicas) in the system. The configuration of each server is shown in the Table 1. We conduct an experiment in part

Table 1. Server Setup

Type	Description
CPU	Intel(R) Xeon(R) E5-2640*2 2.3GHZ 20cores/CPU
Memory	504GB
Network	Intel Corporation I350 Gigabit Ethernet
Operating System	Red Hat Enterprise Linux Server release 6.2 (Santiago).

2 to compare the performance of FRaft and primary-copy running the realistic banking production workload with 3 UpdateServers. There are two transactions in the realistic banking production: *contract* and *pay* where including one update statement, two query statements in *contract* and 5 query statements and 3 update statements in *pay*.

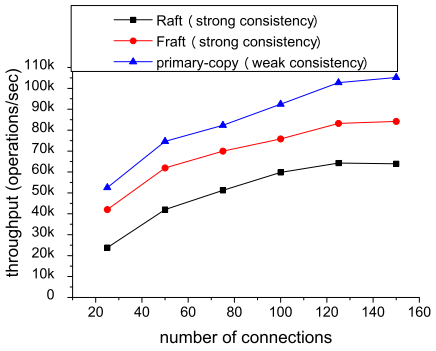

Fig. 5. Write transaction throughput.

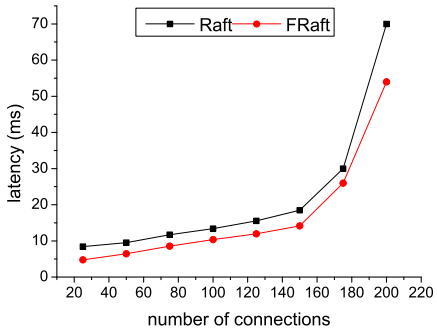

Fig. 6. Transaction latency.

Throughput

The first set of experiment measures the amount of throughput that the system can support. Throughput here is the number of successfully committed operations per second. Clients send replace auto-commit transactions to the leader UpdateServer. Results are shown in Figure 5 and Figure 6, FRaft's peak throughput is 84172 operations per second(ops) for 150 connections and performance tends to close this value. Conversely, the peak throughput of Raft in this set of experiment is 64300 ops for 125 connections. Generally, performance of FRaft has an improvement of 1.3x than Raft, thanks to a change in a way FRaft replicated log on followers. On the other hand, we compared the throughput with primary-copy replication. In primary-copy replication, the leader does not have to wait for the majority of followers' responses before applying transactions. Although this approach benefits a lot because of low latency, it cannot provide high availability when there is a crash in the leader. Results show that FRaft sacrifices 20% performance to provide strong data consistency and high availability in such a case.

Figure 6 shows the latency of two log replication approaches with the increases of number of connections. Raft and FRaft's performance has an upward trend while Raft spends more than 4 microseconds than FRaft in average. This is because the leader on Raft waits for the follower longer than that on FRaft, leading to increase the whole transaction latency. We will discuss the amount of time Raft and FRaft spend on the follower in the next set of experiment. There has been a sharp rise in latency when the number of connections reaches to 180 connections, this is because the system is congested.

Follower's criteria. FRaft and Raft have different log replication strategies on the followers, thus we evaluate the performance of the followers by observing the following criteria. (1)FRaft appends log entries to buffer firstly and then writes a batch to disks, instead of appending log to disk directly in Raft. Thus it is important to observe the disk I/O when running high workloads. We show the relevant criteria for disk of Raft and FRaft in Figure 7 and Figure 8. The duration of this experiment last 100 seconds. We monitor the run state of disk by nmon,

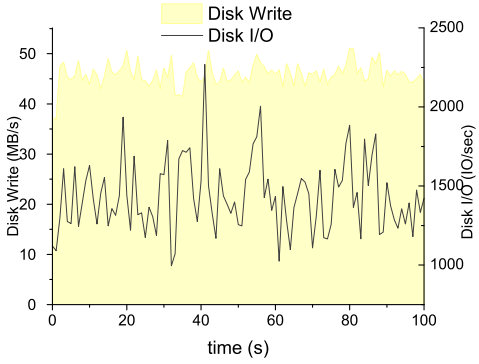

Fig. 7. The behavior of disk in Raft.

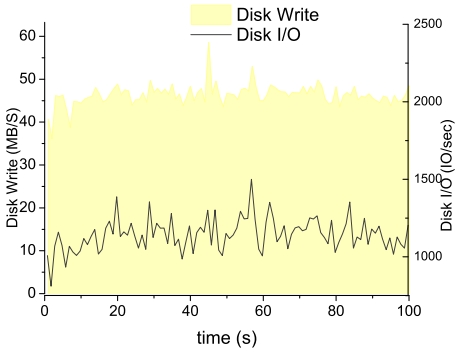

Fig. 8. The behavior of disk in FRaft.

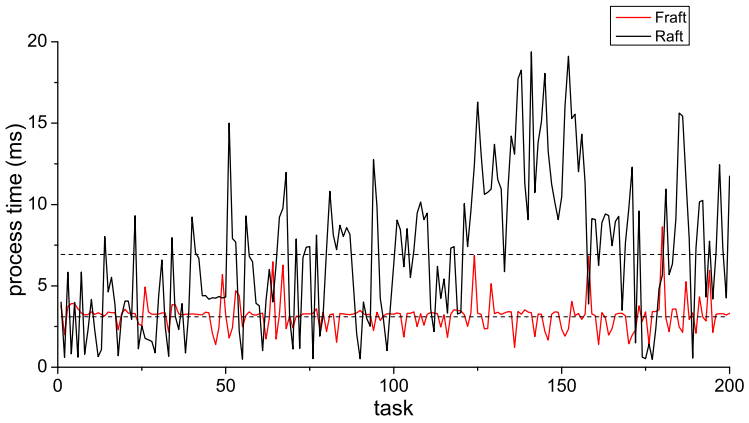

Fig. 9. Process time of follower.

which is a computer performance system monitor tool for the AIX and Linux operating systems developed by IBM. Results show that the amount of data written to disk reach to 50MB per seconds which are similar in Raft and FRaft. However, Raft has a disk IOPS close to 1500 and FRaft has a IOPS just below 1200. This is because FRaft writing more log entries to disk once, compared to Raft that writes less log entries once, has a less disk IOPS. (2)Another important criteria as evaluation standard of Raft and FRaft is the average follower's process time per task. We gathered the processing time of the task for a period of time, result is shown in Figure 9. It is obvious that the process time is unstable with Raft, but smooth with FRaft. Write disk directly and *processQueue* blocking in Raft that makes such abnormal phenomena. We statistic the average process time in the follower under 50 connections and 120 connections. Figure 10 shows that the total process time per task under 120 connections with FRaft

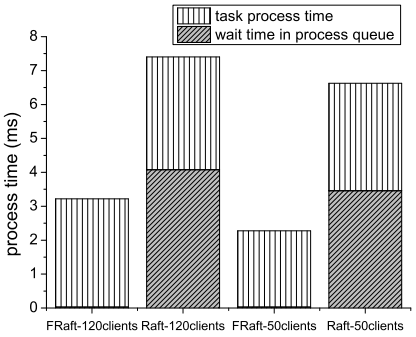

Fig. 10. Average process time of follower.

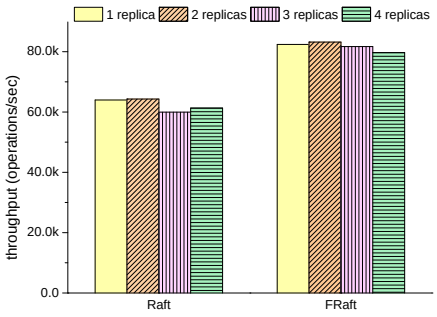

Fig. 11. The effect of increasing the number of replicas on the throughput.

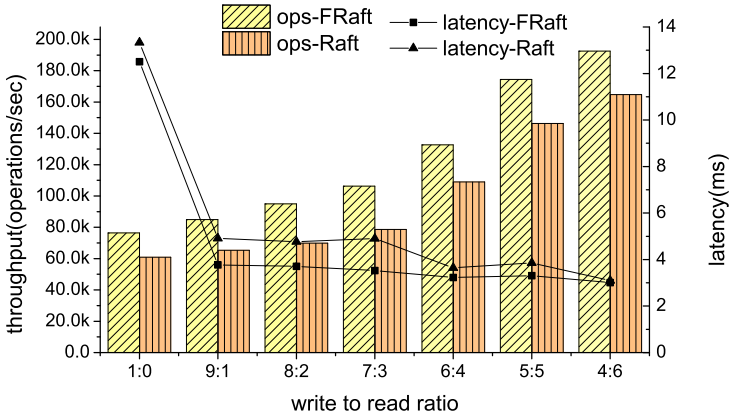

Fig. 12. The effect of the ratio of writes to reads on performance.

is approximately 3.2 microseconds, compared 7.4 microseconds with Raft. When the system is running under 50 connections, the value is 2.2 microseconds and 6.6 microseconds. FRaft spent less than 4 microseconds on average, and the results show good agreement with Figure 6. The wait time in the *processQueue* of FRaft is no more than 0.03 microseconds which reaches to 4 microseconds in Raft, this is because tasks do not have to wait for appending the log sequence to disk in FRaft.

Number of replicas. We measures the performance of FRaft and Raft by setting different number of replicas to study FRaft on different configurations in this set of experiments. Results are shown in Figure 11. FRaft and Raft behave similarly and maintain a close throughput under several settings. This is because log replication on replicas is in parallel, the number of replicas will not cause too much impact on the performance. However, there is still a slight downward

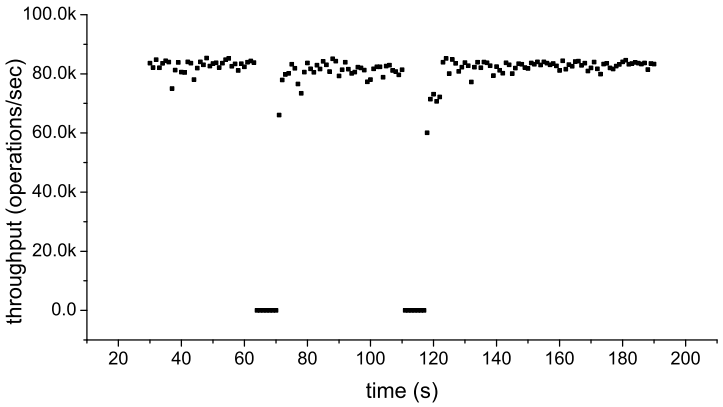

Fig. 13. The effect of server fault on the system.

trend of the operations per second with the number of replicas increases in both approaches. This is because the leader should wait for more followers' responses to apply transactions when more servers with different processing time participant in the system. It is obvious that the commit latency is capped by the slowest follower out of the majority followers responded to the leader.

Write to read ratio. As different applications have different read and write behavior, this set of experiment aims to evaluate whether FRaft is suitable for read- or write-intensive workloads, and the results of different write to read ratio are shown in Figure 12. With the increase of the ratio of read transactions, throughput of FRaft is always higher than Raft.

Fault-tolerance. We observe the behavior of the system in the case of server failures in this set of experiment through throughput to judge whether the system can provide service. The experiments last 3 minutes. We kill UpdaterServer process of the leader with the *kill* command after 64 seconds passed by. Then 6 seconds later, new leader is elected and the system becomes alive. Results of this set of experiment are shown in Figure 13. Another exception occurred in 111 seconds, the system spends 6 seconds to be alive again. The results prove that FRaft is able to detect anomalies and recover quickly under high workload that makes the system can serve normally.

Workloads. In this set of experiments, we evaluate the performance of FRaft under the realistic banking production workload, compared with primary-copy replication. Results are shown in Figure 14 and Figure 15. The peak transactions per second(tps) of *contract* in primary-copy is 30177, which reaches to 26701 tps, a 11.52% decrease, for FRaft. Consider the second transactions *pay*, the performance are 8447 tps in primary-copy, 8158 in FRaft with 650 connections, declined 3.42%. The results indicate that FRaft sacrifices not too much performance on high consistency and availability basis.

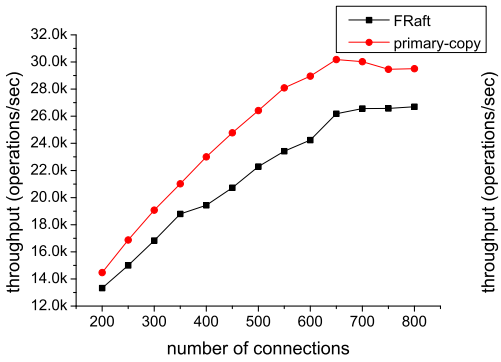

Fig. 14. Throughput of *contract*.

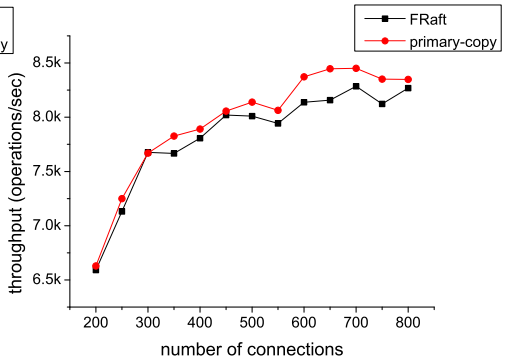

Fig. 15. Throughput of *pay*.

6 Related Works

The architecture of running transactions on top of replicated data store was proposed by Gifford [14] in 1981 which has been widely used in various systems like Google Spanner, by making use of Paxos for log replication. Besides Paxos replication, primary-copy replication [23] proposed by Michael Stonebraker in 1979 also has been frequently used in systems, i.e. MySQL. The leader does not have to wait for the followers' responses in primary-copy replication to commit transactions. Primary-copy replication has a low transaction latency but it cannot guarantee high availability and strong data consistency when there is a crash on the system. Paxos as a consensus protocol have been widely used in many systems [21, 10, 6] in which log replication is sponsored by leader node. The leader appends log entries to disk and applies them to memtable until it receives responses indicating log entries are persistent from the majority of followers in the first phase. In the next round, the leader send asynchronous commit message which contains last commit log index(LSN) to followers, followers apply the log before LSN to memtable.

Raft is a consensus algorithm designed as an alternative to Paxos, which was meant to be more understate than Paxos by means of separation logic while also guarantee data consistency and high availability. Comparing with Paxos, the leader will send a *AppendEntries* RPC which packs log sequence and last commit log index($commitIndex$) to the followers to reduce a network transmission.

There are many works about improving performance of Paxos in engineering. SMARTER [6] was implemented by Microsoft based on SMART with batching and pipelining client requests where pipelining means that it allows multiple proposal run simultaneously, which almost likes slide window mechanism of TCP/IP; batching which was adopted by many systems [6, 4, 8] means that packaging several requests into one task, the disk and network overhead distributed on multiple requests. Nuno Santos et al. [22] made some efforts on tanning Paxos

for high-throughput with batching and pipelining. There are many parameters like the network latency, bandwidth, the speed of the nodes and the properties of applications that affect the performance of these two optimizations in Paxos, thus they present an analytical model that can be used for gathering the parameters for tanning Paxos to achieve higher performance.

7 Conclusion

In this work, we present an efficient log replication approach FRaft to optimize the process of flushing log entries to disk and applying them to replicas. Instead of writing log to disk directly in the standard Raft, FRaft writes log entries to memory buffer firstly as much as possible. This enables more aggressive batch disk I/O and can handle the received tasks in a non-blocking manner. Moreover, FRaft avoids additional memory copy through applying log entries ahead of time. Experimental results show that the throughput of RFaft is 1.3 times that of the Raft and has a lower transaction latency.

Acknowledgements

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. OceanBase. `https://github.com/alibaba/oceanbase/`.
2. Raft. `https://raft.github.io/`.
3. Sysbench. `https://launchpad.net/sysbench/`.
4. R. Ananthanarayanan, V. Basker, S. Das, and A. G. et al. Photon: fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the ACM SIGMOD*, pages 577–588, (2013).
5. A. Bartoli, C. Calabrese, M. Prica, E. A. D. Muro, and A. Montresor. Adaptive message packing for group communication systems. In *On The Move to Meaningful Internet Systems*, pages 912–925, (2003).
6. W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, (2011).
7. B. Carmeli, G. Gershinsky, A. Harpaz, N. Naaman, H. Nelken, J. Satran, and P. Vortman. High throughput reliable message dissemination. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC)*, pages 322–327, (2004).
8. T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 398–407, (2007).

9. B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010*, pages 143–154, (2010).
10. J. C. Corbett, J. Dean, M. Epstein, and A. F. et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, pages 8:1–8:22, (2013).
11. C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, pages 288–323, (1988).
12. R. Friedman and E. Hadad. Adaptive batching for replicated servers. In *25th IEEE Symposium on Reliable Distributed Systems*, pages 311–320, (2006).
13. R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *Proceedings of the 6th International Symposium on High Performance Distributed Computing*, pages 233–242, (1997).
14. D. K. Gifford. *Information storage in a decentralized computer system.* Univ. Microfilms, (1982).
15. F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, pages 245–256. IEEE Computer Society, (2011).
16. L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, (1998).
17. Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 369–384. USENIX Association, (2008).
18. I. Moraru, D. G. Andersen, and M. Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 22:1–22:13. ACM, (2014).
19. B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17. ACM, (1988).
20. D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical ConferenceATC*, pages 305–319, (2014).
21. J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *PVLDB*, pages 243–254, (2011).
22. N. Santos and A. Schiper. Tuning paxos for high-throughput with batching and pipelining. In *13th International Conference Distributed Computing and Networking*, pages 153–167, (2012).
23. M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on Software Engineering*, (3):188–194, (1979).