Designing an Efficient Replicated Log Store with Consensus Protocol

Jung-Sang Ahn Woon-Hak Kang Kun Ren Guogen Zhang Sami Ben-Romdhane
eBay Inc.

Abstract

Highly available and high-performance message logging system is critical building block for various use cases that require global ordering, especially for deterministic distributed transactions. To achieve availability, we maintain multiple replicas that have the same payloads in exactly the same order. This introduces various challenging issues such as consistency between replicas after failure, while minimizing performance degradation. Replicated state machine-based consensus protocols are the most suitable candidates to fulfill those requirements, but double-write problem and different logging granularity make it hard to keep the system efficient. This paper suggests a novel way to build a replicated log store on top of Raft consensus protocol, aiming at providing the same level of consistency as well as fault-tolerance without sacrificing the throughput of the system.

1 Introduction

Nowadays logging system that guarantees strict ordering of incoming payloads plays an important role in distributed systems, for crash recovery or determining the order of distributed transactions [7, 8, 12, 17, 20]. There are multiple producers putting payloads to the logging system at the same time, and multiple consumers subscribing to the system to retrieve the payloads. Since a number of clients are working on it in parallel, it is required to be efficient. In addition to that, usually this kind of logging system is a single point of failure of its subscribers, thus it has to be highly available against the failure of multiple nodes.

To achieve high availability, incoming payloads should be replicated to multiple nodes (i.e., replicas). All replicas should have exactly the same payloads in exactly the same order, so that subscribers can be distributed over different replicas for reducing the burden of the master replica. Once the master replica fails, one of the other replicas should be able to take it over quickly and then continue to serve operations. Even after failure and recovery, the original order of logs should be preserved.

Well-known consensus schemes such as Paxos [13] and Raft [15] might be a good solution to address such issues. Today's implementations are using state machine-based replication [19], where state machine is usually defined as a file or a back-end database that we want to make identical among all replicas. They maintain a sort of write-ahead log (WAL) which keeps a sequence of state changes, usually database mutation commands. Once a log entry (i.e., a state change) is agreed by a majority of replicas, it is committed to the state machine which results in the execution of the given database command.

However, applying state machine-based replication to logging system brings various inefficiencies that have significant impact on performance:

Double-write problem The state machine of logging system will be defined as another append-only log. Every payload will be written in WAL first, and then written again in the state machine after commit. It will end up with cutting the disk bandwidth in half.

Different granularity Clients want to commit a set of payloads at once atomically. Moreover, server also should be able to do group commit of multiple payloads to maximize network or storage throughput. Unfortunately, the basic unit of replication, consensus, and commit is a single log entry of WAL. This granularity gap either degrades the performance or makes the system complicated, as there are more things to do beyond the consensus protocol.

This paper proposes a new scheme that addresses aforementioned issues on top of Raft consensus protocol. We define two log stores: *Raft log store* for WAL of Raft and *data log store* as a state machine, and then introduce a log sharing scheme between them to avoid double-write problem, thus user payloads are written to disk only once. However, having only one copy of data may spoil the consistency of the system upon node failures. We also show that our log sharing scheme will not break the original consensus protocol by help of the

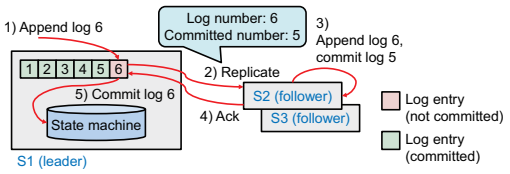


Figure 1: Raft protocol overview.

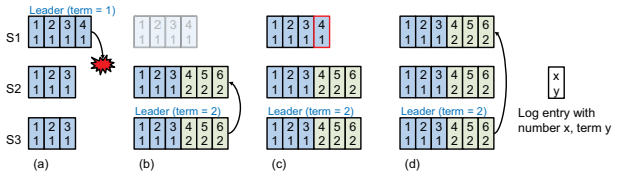


Figure 2: Conflict resolution based on term. (a) Leader S1 fails to replicate log 4 and then crashes. (b) S3 is elected as a new leader with term 2, and replicates new logs 4, 5, and 6. (c) Previous leader S1 is recovered from the fault but log 4 has a conflict. (d) Since log 4 in S3 has bigger term, log 4 in S1 is overwritten by the new value.

characteristics of our state machine definition, which is also append-only data structure unlike normal databases.

In addition to the log sharing scheme, Raft log store and data log store will have different granularity; a single Raft log entry will be associated to a set of data log entries, to replicate and commit multiple payloads at once. That process will be pipelined over multiple replicas, to maximize the throughput of the system. Our experimental results show that proposed log sharing scheme is working well along with group commit and pipelining, regardless of the number of replicas, the size of atomic batch, or the size of each payload.

2 Raft Consensus Protocol

Raft [15] has been suggested for enhancing understandability over Paxos [13], while keeping equivalent consistency level without sacrificing efficiency. Recent distributed systems such as Apache Kudu [2, 14] and Yugabyte [4] are using it for their consensus protocol over multiple replicas.

Raft adopts a strong leader policy; there is only one leader at a time. Only the leader accepts incoming write requests, and then replicates it to other non-leader replicas, called *followers*. Each incoming write request corresponds to a single log entry, which is stored in a separate log section. Each log entry has a unique index number (hereafter *Raft log number*) in a monotonically increasing order.

Figure 1 illustrates the overview of the protocol. Once the leader gets a write request, it appends a log entry to its log section, and then replicates the request to all other followers. When a follower receives the log entry, it also appends the log to its log section, and then returns a response back to the leader. Raft is using quorum write policy, thus the leader will commit the log entry and apply it to the state machine once it gets responses from a majority of replicas including itself. After commit, the leader replicates the committed log number to followers, and followers finally commit the log corresponding to the given log number and then apply it to their state machines. Note that a log replication request contains the committed log number of previous log replication, so that they are always pipelined and there is no specific protocol for two-phase commit.

The leader periodically sends heartbeats to all followers, and each follower has a randomized timer whose lower bound is bigger than the heartbeat interval. Whenever a follower receives heartbeat, it resets its timer with new random expiry. If the timer is expired due to no heartbeat for a while, the follower regards the situation as the death of the leader, and then requests a vote on the election of the next leader to all the other replicas. Once a majority of replicas vote for the replica who initiated the election, the replica becomes a leader and starts to serve write requests.

To resolve conflict that can happen during the absence of leader or network disconnection, Raft maintains a special counter called *term*, which is increased whenever a new leader election is initiated. All followers should have the same term value as that of the current leader. Each replication request or heartbeat contains the term value of the current leader, and also each log entry keeps the term value of the time when it was generated. If the current leader receives a message with higher term from other replica, that means a new leader has been elected by a majority of replicas so that it immediately gives up the leader role and becomes follower.

Conflict usually happens when the leader succeeds to append a new log entry to its log section, but fails to replicate it to followers due to various reasons such as server crash or network isolation, and then new leader is elected and serves new logs. If the previous leader is recovered from the fault and then re-joins as a follower, the previous leader and the current leader will see different log entry whose log number is the same, as shown in Figure 2. In such case, the previous leader will find the last log entry whose term is the same as that of corresponding log entry in the current leader. And then it starts to overwrite log entries starting from that point using the current leader's log entries. In other words, the previous leader discards its local log entries that failed to reach a consensus.

3 Log Store Design

3.1 Requirements and Problem Statement

To guarantee strict ordering, the requirements for log store are as follows:

1. Each payload should have a unique log sequence number (LSN), which is a non-zero positive integer.
2. All replicas should have the same data in the same LSN order.
3. LSN should be continuous; no empty number in the middle is allowed.

4. Clients may send a set of payloads in batch. They should be committed atomically, and partial commit is not allowed.
5. Clients get the response of a payload only after it is committed—i.e., x replicas have the payload, where $x > \lfloor \frac{N}{2} \rfloor$ and N is the number of replicas.

By adopting Raft, we can easily satisfy those requirements. Once a write request comes in, we first put it into the log section of Raft and then replicate. After we get acknowledges from a majority of replicas, the request is committed and applied to the state machine of Raft, which is the actual log store that we want to implement. Since Raft guarantees the same order of commit, state machines in all replicas will get the data in the same order so that we can just append them to log store sequentially.

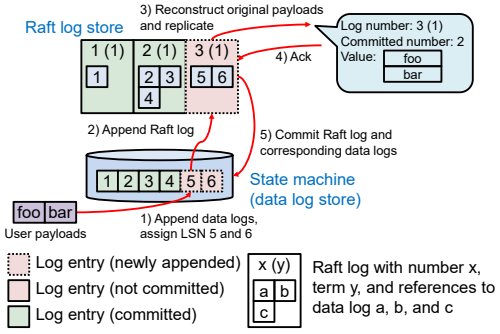
This is a typical journaling approach for traditional file systems or databases. However, it will be problematic if the back-end store is the same append-only log. We will end up having duplicate logs: Raft log and state machine, which will double the space usage, and also will halve the write throughput. Since all disk write operations are expected to be fully sequential in log store, the relative amount of degradation by double-writing will be significant, compared to other normal database systems.

We can simply avoid double-write problem by having Raft logs only, directly using a Raft log number as an LSN. However, this approach introduces a couple of challenging issues. 1) We cannot use group commit; both replication and commit will be executed on a per-Raft log basis. Partial commit is inevitable and eventually breaks the fourth requirement. 2) Raft itself generates a special log for membership or configuration change that needs to be globally consistent. Since that special log also occupies a Raft log number, it ends up with an empty LSN from the user payload's perspective, which breaks the third requirement. If we allow LSN gaps, log consumers are not able to know in advance whether the missing LSN is skipped by the system or lost in the middle, without having extra communication. It makes the overall protocol expensive.

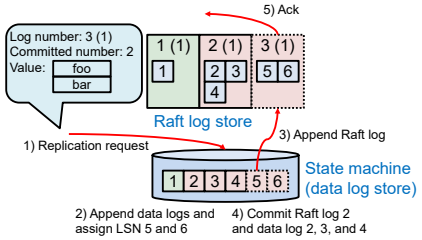
We need more clever approach to handle such issues.

3.2 Log Sharing and Tweaked State Machine

To address aforementioned problems together, we propose a log sharing scheme. We define two local log stores: *Raft log store* for Raft log section and *data log store* for the state machine that is the actual log store to serve user payloads. Once a write request which consists of one or more payloads comes in, it is directly written to the state machine (i.e., data log store) first, skipping Raft log store. At the time we append payloads to the data log store, the data log store sequentially assigns a unique LSN to each payload. After that, we append a log entry to Raft log store, which contains a set of LSNs that need to be committed atomically. Note that the LSNs in a Raft log entry should be consecutive, and also adjacent Raft



(a) Leader's point of view



(b) Follower's point of view

Figure 3: Log sharing and replication.

log entries should have consecutive LSNs. Figure 3(a) depicts the overview of log sharing and its replication process.

For each replication task, it reads a Raft log entry to be replicated, and fetches the LSNs belonging to the Raft log. Then it re-constructs the original batch of payloads by reading the data logs corresponding to the LSNs sequentially, and the request is sent to followers. The state machines of followers will do the same thing: appending payloads to data log store first, assigning LSNs for each payload sequentially, and then appending a Raft log entry which consists of a set of LSNs it assigned, as illustrated in Figure 3(b).

Since the data log store in each replica is locally assigning LSNs in advance before waiting for replication and Raft commit, it may cause an inconsistency of LSNs between replicas if failure happens after assigning LSNs but before getting consensus. In such case, the original Raft can simply overwrite existing Raft log entries as uncommitted logs are not applied to the state machine yet. However, we cannot use the same approach in here as payloads are already applied to the state machine before commit.

As shown in Figure 4, once a conflict happens so that one or more Raft logs have to be overwritten, we first find the smallest LSN in the conflicting Raft logs: LSN 2 in Raft log 2. Then do rollback of the data log store to the log whose LSN is right ahead 2, before overwriting Raft logs with newer term. Since the state machine is defined as a log-structured format, rolling back can be easily implemented as a simple truncation, unlike normal databases that require separate undo logging. After rollback, we can append incoming payloads from the new leader, and assign new LSNs that will be identical to

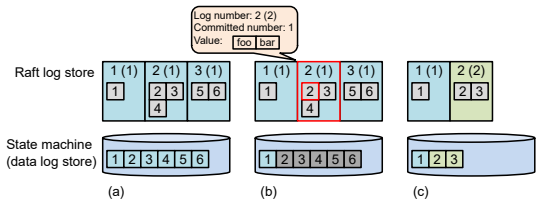


Figure 4: Rollback of state machine in conjunction with Raft conflict resolution. (a) Initially data log store has 6 logs. (b) Raft log 2 conflicts with that in new leader, who has higher term 2. Find the smallest LSN from the Raft log and truncate all data logs whose LSN is equal to or greater than the identified LSN. (c) Append new payloads with new term. Now assigned LSNs will be consistent with those in the new leader.

what the leader has.

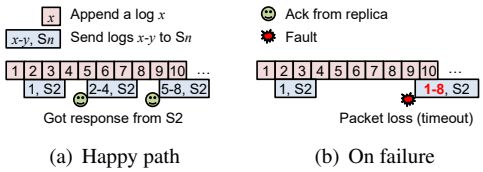
Since payloads are already appended to the data log store prior than Raft commit, the commit process of state machine is straightforward. The state machine of each replica maintains a cursor that indicates the last committed LSN of its data log store. All payloads whose LSN is greater than the last committed LSN should not be visible to clients, even though they already exist in the data log store. Once Raft gets consensus from quorum, it executes a commit on the agreed Raft log number. The state machine finds the last LSN in the Raft log entry, and then moves its cursor to that LSN.

There are a few more corner cases that we need to consider. If a node crashes after updating data log store but before Raft log store, we should discard the stale data in data log store on server restart, to avoid LSN inconsistency. Even without crash, both data and Raft log stores should be updated atomically; another replication request from a new leader should not interfere in the middle, and be handled sequentially after the previous one is done. Note that the safety of Raft log store should be guaranteed by Raft itself, as that part remains unchanged.

3.3 Pipelining and Group Commit

To maximize the throughput, leader can keep accepting new payloads from clients and appending them to its data log store, while previous replication requests are still in flight. Following replication request contains the logs kept in the meantime and replicates them at once as a group commit. However, such a pipelined replication should be very careful about LSN order inversion; it can happen when previous replication request is lost in the middle while following replication request is successfully delivered to replicas.

To avoid such an issue, we only allow 2-stage pipeline for each follower as illustrated in Figure 5. The basic assumption is based on the third requirement mentioned in Section 3.1; if a log whose LSN k is successfully replicated, it implies all previous logs whose LSN x where $x < k$ already have been replicated. Thus, each replication request should contain all consecutive logs starting from the log right next to last



(a) Happy path (b) On failure

Figure 5: 2-stage pipeline for follower S2.

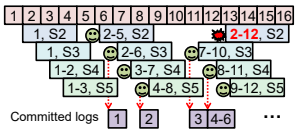


Figure 6: Combined N-stage pipeline.

successful log, without skipping anything in between.

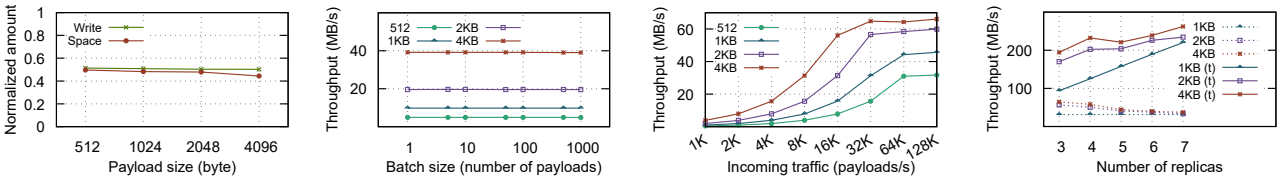
Figure 5(a) presents an example of 2-stage pipeline for follower S2. Suppose that we append only one payload for each Raft log entry for simplicity. Once a new payload comes in, it is appended to the data log store with LSN 1, and the corresponding Raft log entry is also appended to the Raft log store. Then the log 1 is sent to S2. Before getting response from S2, the leader receives more payloads from clients and appends corresponding logs. At the time the response from S2 arrives to the leader, we have 3 more logs: from 2 to 4. We set the last successfully replicated LSN of S2 to 1, and accordingly the next replicate request will contain logs from 2 to 4.

However, if any failures such as packet loss happen as shown in Figure 5(b), the leader should be able to catch it quickly and then re-send the replication request. But at this time, since the last successfully replicated LSN has not been changed, the request should contain all new logs accumulated in the meantime including the previously failed log: from 1 to 8. In this manner, we can guarantee that any logs cannot be replicated prior than previously failed logs, so that logs in all replicas should be always sequential.

Now we can combine pipelines for all followers and then organize N-stage pipeline, where N denotes the number of replicas including the leader. Figure 6 depicts an example when there are 4 followers: S2 to S5. If there are enough number of replicas, we can fully utilize either disk or network bandwidth. Note that a log x is committed only after the last successfully replicated LSNs of a majority of replicas become equal to or greater than x. Clients will get the response of the log if and only if it is committed by the leader.

4 Evaluation

We implement log store as a service for eBay's home-grown distributed database systems, based on our own Raft implementation. The communication between client and server is done by gRPC [3] streaming, where client keeps sending requests asynchronously without waiting for the responses of previous requests. Each request contains one or more payloads



(a) Disk write and space usage (b) Batch size (c) Incoming traffic (d) Number of replicas

Figure 7: Evaluation results. (a) denotes the amount of disk write and space usage normalized to those of the naive approach. (b), (c), and (d) indicate the write throughput of log store according to batch size, incoming traffic, and the number of replicas, respectively.

in batch that need to be committed atomically. We deploy log store service in eBay datacenter's VMs equipped with gigabit ethernet and local SSDs. The maximum throughput of a single network stream varies with VM deployment details such as the latency between racks and physical machines, which is totally random. The average throughput between nodes that we use for evaluation is measured as 85–90 MB/s. We use three replicas including the leader, unless otherwise noted. Client is located in a separate node so that incoming traffic also consumes network bandwidth from the leader's point of view. All replicas including client are in the same datacenter.

First of all, we measure the amount of disk writes and space occupied by logs, compared to the naive approach which does not use log sharing scheme. Figure 7(a) shows the results of log sharing normalized to those of naive approach, varying the size of each payload. Both disk write and space usage become almost half by using log sharing scheme.

Next we evaluate the write throughput of log store according to the batch size that clients want to commit atomically. We generate 10,000 payloads/s traffic with the various sizes of payloads, and then measure the throughput of the amount of payloads committed by log store. Figure 7(b) depicts the results. By help of group commit, the throughput remains constant regardless of the batch size; when a batch is small, a replication request packs more batches so that it does not affect the overall performance.

After that, we explore the maximum write throughput that log store is able to handle. The same as the previous evaluation, we measure the throughput, but at this time we vary the incoming traffic from 1,000 to 128,000 payloads/s. The throughput increases proportionally according to the incoming traffic, but it is saturated at some point, as illustrated in Figure 7(c). The saturation point gets higher as payload size increases, and log store can achieve 67 MB/s with 4 KB payloads. It will be close to the maximum network throughput with enlarged payloads, since bigger payload may help increasing network utilization as well as reducing the overhead of Raft protocol due to fewer number of logs.

And then we assess the write throughput with the different number of replicas, to see if N-stage pipeline is working as expected. The incoming traffic is fixed to 32,000 payloads/s. Figure 7(d) presents the results. The dotted line plots the write throughput of log store from the client's perspective, while the solid line means the total throughput that the leader node is processing. If the incoming traffic is T, and the number of replicas including the leader is N, the leader node should handle $N \cdot T$ traffic: T from client and $(N-1) \cdot T$ for replication. With 1 KB payloads, the throughput of log store is constant regardless of the number of replicas, as the total throughput is linearly increasing by help of pipelining. With bigger payloads, the total throughput is still increasing but not exactly proportional to the number of replicas, as the leader node and its Raft logic are burdened with processing payloads.

5 Related Work

Apache Kafka [1] is a popular message streaming platform, which also can be used for building a replicated logging system [21]. However, Kafka cluster replication does not allow group commit for a single topic, which results in significant performance degradation when there are many producers generating traffic in parallel.

Balakrishnan et al. have suggested CORFU [7], which implements a log store whose data is shared among network-attached SSDs. A global sequencer node assigns the next log index so that clients can directly access the destination SSDs where the log will be written or read. However, the sequencer may introduce a hole in the middle after failure of it, and they do not provide solid logic for conflict resolution.

Other Paxos-based approaches such as S-Paxos [9], SMARTER [10], and Spinnaker [16] attempt to achieve similar goals: group commit and eliminating the extra copy of data on each node.

6 Conclusion

This paper suggests an efficient way to build a logging system based on Raft protocol. Raft provides an intuitive way to replicate data and to resolve conflict upon node failure, but introduces double-write and commit granularity problems if we are going to build a log store on top of it. We propose a log sharing scheme that addresses aforementioned issues. In addition to that, we show a practical way to implement group commit as well as pipelining for log replication.

Discussion Topics

Distributed log store In this paper, we only propose a single log store cluster. If we need much higher throughput to support large-scale distributed systems, a considerable option is to make the log store distributed for scaling out. However, it brings a new challenging issue: how can we partition logs while keeping their order globally consistent? We will continue to study this topic.

Extending log sharing scheme We can easily achieve log sharing on top of consensus protocol, since the state machine of log store is based on a log-structured format so that easy to execute rollback. It will be good to extend this scheme for general purpose databases to reduce the write amplification of the system. However, the rollback operation will be much more complicated than that of log store. We can simply think two options for now: 1) maintaining separate undo logs, but it will cancel the benefit of log sharing. Or, 2) using log-structured key-value stores [5, 6] or file systems [11, 18] for back-end database, but those approaches have a couple of disadvantages such as compaction overhead or degraded range query performance. We will explore this subject more.

References

[1] *Apache Kafka: A distributed streaming platform*. https://kafka.apache.org/.

[2] *Apache Kudu*. https://kudu.apache.org/.

[3] *gRPC: A high-performance, open-source universal RPC framework*. https://grpc.io/.

[4] *YugaByte DB*. https://www.yugabyte.com/.

[5] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. Forestdb: A fast key-value storage system for variable-length string keys. *IEEE Transactions on Computers (TC)*, 65(3):902–915, 2016.

[6] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide: Time to Relax*. "O'Reilly Media, Inc.", 2010.

[7] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. Corfu: A shared log design for flash clusters. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2012.

[8] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–340. ACM, 2013.

[9] Martin Biely, Zarko Milosevic, Nuno Santos, and André Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Proceedings of the 31st Symposium on Reliable Distributed Systems (SRDS)*, pages 111–120, Oct 2012.

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

[11] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. Technical report, 2003.

[12] Matt Freels. Faunadb: An architectural overview. https://fauna.com/.

[13] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[14] Todd Lipcon, David Alves, Dan Burkert, Jean-Daniel Cryans, Adar Dembo, Mike Percy, Silvius Rus, Dave Wang, Matteo Bertozzi, Colin Patrick McCabe, et al. Kudu: storage for fast analytics on fast data. 2015. https://kudu.apache.org/kudu.pdf.

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

[16] Jun Rao, Eugene J Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the VLDB Endowment*, 4(4):243–254, 2011.

[17] Kun Ren, Alexander Thomson, and Daniel J Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *Proceedings of the VLDB Endowment*, 7(10):821–832, 2014.

[18] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[19] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[20] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1–12. ACM, 2012.

[21] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with apache kafka. *Proceedings of the VLDB Endowment*, 8(12):1654–1655, 2015.