Towards High-Performance Replicated Transactional Data Store

Zhongmiao Li
Université catholique de Louvain
Instituto Superior Técnico, Lisboa & INESC-ID

***Abstract*—State-machine replication (SMR) is a well-known approach to implement fault-tolerant services. SMR totally orders operations and requires every replica to execute operations in the same order, which limits system scalability. To adopt the SMR approach to build large-scale fault-tolerant data stores, recent works have proposed to partition data store states and execute commands only at involved partitions to enhance scalability. However, existing systems suffer from the following problems: 1) some systems require knowing the read-set/write-set of transactions before execution, which may not be possible for all applications; 2) within each partition, transactions are serialized by a single thread, which is a waste of resource if these partitioned are hosted by today's prevalent multi-core servers, 3) existing systems rely on non-genuine multicast protocols to send transactions to partitions, which inherently limits the scalability of systems.**

In this paper, we propose ParScale, a parallel, scalable replicated transactional data store that is designed for today's large-scale, multi-core clusters. The key novelty of ParScale lies in its massively parallel concurrency control that leverages speculative execution, and its scalable, genuine atomic multicast protocol to dispatch operations. Our preliminary results show that in certain workloads, ParScale outperforms the state-of-art systems in throughput by 3×.

I. Introduction

State-machine replication (SMR) [1] is a well-known technique to build fault-tolerant replicated services. SMR relies on consensus protocols [2] to ensure that non-faulty replicas receive operations in the same order and execute operations (which are supposed to be deterministic) by the agreed total order. As this approach naturally provides linearizability, it has been leveraged to build replicated transactional data stores. However, a major limitation of this approach is its scalability: since all replicas have to execute all operations, the maximal achievable throughput is bounded by the computation capacity of a single server. Nevertheless, today's large-scale online services hosted by data stores strive to serve millions of clients simultaneously, which is far beyond the capability of single server.

Recent works [3], [4] circumvent the scalability drawback of SMR-based transactional data stores by state partitioning [5]. Data is still fully replicated among replicas, while each replica partitions its state to multiple servers. More importantly, these systems strive to only execute transactions on servers that contain data required by these transactions, in order to achieve (near) linear scalability by adding servers. As such, different to aforementioned classic SMR approach, both systems require to dispatch transactions to their involved servers (in contrast to sending operations always to all replicas/servers), and execute transactions in different as parallel as possible (in contrast to executing operations sequentially). For example, in S-SMR [4], single-partition transactions are only sent to their corresponding partitions, while multi-partition transactions are totally ordered by a 'global paxos' instance and broadcast to all partitions. Within each partition, a single thread executes transactions sequentially. Another prominent example is Calvin [3], which further leverages the assumption of knowing transactions' read and write set in advance to allow higher execution parallelism [1]. Within each partition, Calvin relies on a single locking thread to serialize the lock requests of transactions, and transactions that have been acquired locked are guaranteed to be conflict-free with the other concurrent transactions; then a pool of work threads execute these transactions without further concurrency control. Though, Calvin's dispatching phase is simplistic: after replication, all servers within a replica (typically a datacenter) send each other messages that contain their received multi-partition transactions, then received messages are merged by each partition in a deterministic order.

Nonetheless, existing works have the following drawbacks: 1) these systems only utilize a single thread to execute or serialize transactions in each partition. While multi-core machines are prevalent nowadays, this design can result in low computation utilization, and 2) these systems employs non-scalable, non-genuine multi protocols to dispatch transactions among partitions, and 3) Calvin's assumption of knowing transaction's read and write set requires extra developer's effort, and may not be possible for certain applications.

In this paper, we present ParScale, a parallel, scalable transactional data store designed for today's large-scale, mutli-core clusters. The novelty of ParScale are as follows: 1) its massively-parallel concurrency control scheme that exploits speculative execution to allow fully utilizing available computation capacity, 2) it employs a genuine, scalable multicast protocol to dispatch operations. Last but not least, ParScale does not require knowing the read-set or write-set of transactions in advance.

[1]if this assumption is not met, Calvin performs a 'reconnaissance phase' of a transaction before its execution to estimate its read and write set.

II. System model and assumption

We consider our datastore consists of N replicas, and each replica is partitioned to P servers, each containing a disjoint set of the whole data set D. Without loss of generality, we assume each replica has the same partitioning topology.

We do not assume knowing the read-set and write-set of a transaction before its execution. However, we assume that the partitions accessed by a transaction is known before its execution. In case the exact set of partitions to be access can not be known in advance, one can take an over-approximation, such as assuming all partitions will be accessed by this transaction.

III. ParScale overview

This section presents a high-level description of the ParScale protocol.

A. Batching and dispatching

ParScale progresses in batches: each partition buffers operations received by clients for a specified period, then executes an atomic multicast [?] protocol to dispatch its received multi-partition transactions to corresponding partitions. Our protocol is inspired by Skeen's algorithm [?], a genuine atomic multicast protocol. Intuitively, a partition sends its received multi-partition transactions to all involved partitions, and guarantees that all involved partitions include these transactions in the same batch. The detailed execution is as follows:

1) When a partition finishes batching, it multicasts all multi-partition transactions in this batch to corresponding involved partitions. Then it checks if the current batch has any pending multicast messages. If there is pending messages, the partition waits until there is no pending message. When a partition is about to send the current batch to its consensus instance, it checks a 'to send' dictionary for any multi-partition transactions to send for this batch (as will be clear later, this was inserted by previous multicast instances). Then it sends these multi-partition transactions (if there is any) along with its single-partition transaction to the consensus instance of the current batch. Then, the partition's batch number increments by one.
2) If a partition receives a multicast message from another partition, it proposes a batch number that it wishes to include these transactions. The batch number can be picked freely as long as it has not already been proposed by this partition. Then the partitions increments the pending multicast message count for the proposed batch, thus promising that it will not send that batch until it receives reply for this multicast message.
3) When a partition has received all replies for its sent multicast message, it calculates the batch for transactions of this multicast message as the maximal between the proposed batch, and the next batch to propose in this partition. After calculating the batch to include for these transactions, the partition inserts these transactions into the 'to send' dictionary of the corresponding batch.
4) After a transaction has received the result, which indicates the batch number to include these transactions, for a multicast message it has proposed, it decrements the pending multicast message count for the batch it proposed to this message. Then it inserts the transactions of this multicast message to the 'to send' dictionary, according to this given batch number.

B. Replication

After the batching and dispatching phase, replicas of the same partition initiates a consensus instance for the current batch, and each of them proposes its received transaction. We rely on the consensus instance to totally order the proposed operations for non-faulty replicas. We do not discuss the detailed execution of the consensus protocol, as it is orthogonal to ParScale.

C. Execution

After the replication phase, each replica of a partition receives a sequencer of transactions. A conservative approach to guarantee that all replicas reach the same state is to execute transactions sequentially according to the given order. In order to maximize throughput, ParScale allows multiple worker threads to execute transactions concurrently, without enforcing the given sequence order. Though, transactions still commit according to the given order, i.e. a transaction can only commit if all transactions ordered before it has committed.

In detail, each partition has multiple worker threads that execute transactions. Whenever a worker thread is idle, it obtains a transaction from the sequencer and initiates the transaction. When the transaction reads any key, it leaves a read dependency on this key. This dependency triggers the abort of this transaction, if later on any transaction ordered before it tries to insert a new version for this key, i.e., the reader transaction missed a version it should read. On the other hand, when writing any key, the transaction locks the corresponding key, and aborts any transaction that are ordered after it but have left a read dependency on this key, likewise. Upon finishing all read and write operations, the transaction is speculatively-committed by unlocking all locked keys and inserting all its updates to corresponding version chains. A transaction can only be turned from speculative-commit state to commit state, if all transactions ordered before it has committed. While if a transaction is aborted after speculative-commit, it removes all its applied updates and also aborts all transactions that have read from it.

Multiple-partition transactions are executed in following way: when a partition tries to execute multi-partition transactions, if it encounters keys it does not replicate, it suspends execution and waits until it receives the read results for these keys; on the other hand, if a partition encounters read to its locally-replicated keys, it sends these read results to all other involved partitions.

IV. Preliminary results

We are developing ParScale on top of the code base of Calvin [2]. Our preliminary results show that in low-contention workloads with low percentage of distributed transactions, ParScale outperforms the throughput Calvins by $3\times$.

References

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[2] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[3] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 1–12.

[4] C. E. Bezerra, F. Pedone, and R. Van Renesse, "Scalable state-machine replication," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 2014, pp. 331–342.

[5] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li *et al.*, "Tao: Facebook's distributed data store for the social graph."

[2]https://github.com/yaledb/calvin