SHAFT: Supporting Transactions with Serializability and Fault-Tolerance in Highly-Available Datastores

Yuqing Zhu, Yilei Wang
Institute for Computing Technology, Chinese Academy of Sciences, Beijing 100190, China
{*zhuyuqing,wangyl*}*@ict.ac.cn*

***Abstract*—Guaranteeing transaction semantics in a highly available and fault tolerant manner is desirable to application developers. Besides, it is a very valuable feature for database-backed applications. In this paper, we propose SHAFT to support transactions with serializability in highly-available datastores, which partition, distribute and replicate data across datacenters. SHAFT is a transactional replication protocol guaranteeing *S*erializability, *H*igh *A*vailability and *F*ault *T*olerance simultaneously for transactions. Laying its basis on the Paxos algorithm, SHAFT guarantees serializability by a two-phase locking procedure in a fault-tolerant manner. Different from other transactional replication protocols like MDCC, SHAFT allows a client to actively abort a transaction. SHAFT also allows flexible data partition, replication and distribution, a proper combination of which can reduce costs and improve performance. SHAFT performs well even under failures. Our experiments show that SHAFT outperforms MDCC, which outperforms other synchronous transactional replication protocols, e.g. Megastore.**

***Keywords*-concurrency control; transaction; high availability; consistency; isolation;**

I. Introduction

High availability and fault-tolerance are two important properties of large-scale systems. Nodes of a system can fail; even an entire datacenter can become unavailable as well. For example, in June 2012, Amazons Elastic Compute Cloud in North Virginia went down due to thunder storms [1]. Similar outages have also been reported in large-scale storage systems by Google, Facebook, etc. In many of these events, failures have resulted in data losses. Besides, even if a system can recover and restart from failures and losses, the downtime when the system recovers can cause great economic losses [2], [3]. Therefore, applications are widely deployed over highly-available datastores that partition, distribute and replicate data across datacenters.

Transaction support is a very valuable feature for database-backed applications. Consistency was once relaxed to improve availability and fault-tolerance in highly-available datastores, as consistency, availability and network partition tolerance are in a trade-off relation [4]. However, the lack of strong consistency semantics, e.g. transactional support, leads to great difficulty of application development [5]. In recent years, there are increasing attentions to transactional support in highly-available datastores. Besides, the strong transactional support with serializability is much desired by applications [6].

There are three fundamental measures, i.e. data partition, distribution and replication, that are key to guarantee availability, scalability and fault tolerance in storage systems. But these measures increase the difficulties of supporting transactions. While various methods are proposed to support transactions exploiting the three measures, fault-tolerant approaches proposed recently usually base on the Paxos algorithm [7]. Paxos is the most widely known protocol in guaranteeing high availability and fault-tolerance based on replication, while two phase commit (2PC) is the most accepted protocol for distributed transaction commit, and two phase locking (2PL) is the most widely used technique to guarantee serializability. Paxos, 2PC and 2PL are exploited in various proposals [8], [9], [10], [11], [12]. Among these proposals, few supports transaction and replication over partitioned and distributed data can guarantee high availability, fault-tolerance, and serializability simultaneously.

In fact, transactions over replicated data are inherently different from those operating non-replicated data, even both are distributed transactions. With replication, we might treat each replica as a different shard and run 2PC as if in a distributed database without replication. However, fault-tolerance and availability will be impaired, as 2PC is a blocking protocol. Furthermore, 2PC mainly focuses on whether a participating node can commit or not. The actual execution order at each node is disregarded, thus different replicas can have different execution orders for transactional operations on the same data item. That is, there would be conflicting replicas problem even if every node votes commit in 2PC. From the aspect of correctness, the consensus on replica state change has a higher priority than the commit consensus of transactions. Not only commit consensus is important, so is commit ordering.

In this paper, we address the problem of supporting transactions with serializability, high availability, and fault tolerance guarantees simultaneously. To achieve high availability and fault tolerance, we exploit the Paxos algorithm as the basis as well. However, we *incarnate the roles of the Paxos algorithm differently* from other proposals, including the leader election and the configuration determination. We also *update the operation semantics of the Paxos algorithm*, such that a Paxos-based distributed two phase locking and commit procedure is implemented to guarantee

serializability. The updated operation semantics include *consensus* and *majority*. SHAFT allows a client to actively abort its submitted transaction by *using two Paxos instances*. The SHAFT proposal can be implemented in any system that partitions, distributes and replicates data in a large scale. Even if the partition, the distribution and the replication is not uniform, e.g. no full replica in a datacenter or datacenters with different partition sets, our proposal is also feasible. In this paper, we make the following contributions:

- An integrated transactional support approach, which guarantees serializability, high availability and fault tolerance simultaneously for transactions in highly-available datastores.
- A new fault-tolerant approach to implementing strict two-phase locking in the distributed environment.
- Performance results from extensive experiments showing that SHAFT guarantees stronger consistency with a higher concurrency and a cost similar to other transactional replication protocol.

In the following, Section II relates SHAFT to existent works. Section III describes SHAFT's new concurrency control protocol. Section IV illustrates the evaluation of SHAFT. Section V concludes the paper.

II. Related Work

Transactional support for distributed storage systems can be divided into three categories. The first category considers data partition but no replication. G-store [13] provides on-demand transactional access over partitioned data through group communication protocol. However, replication is left to the underlying storage's consideration and not considered in the implementation.

The second category considers a whole replication of database, but no partition. Walter [14] supports parallel snapshot isolation (PSI), precluding write-write conflicts of concurrent transactions by timestamps at different sites, each of which is a complete copy of the whole database. But its dependence on precise timestamps is impractical as time coordination can hardly be implemented with precision in large-scale systems. Megastore [8] supports serial transaction execution within each entity group, exploiting Paxos algorithm for fault-tolerance. Each entity group acts as an individual database in effect. Paxos-CP [12] is another improvement of Megastore. It supports serializable schedules of transactions through combination and promotion enhancements.

The third category considers both replication and partition. Spanner [9] extends Megastore. While two phase locking and Paxos algorithm are exploited within a shard and their replicas, a global transaction commit layer with True Time support guarantees the snapshot isolation of transactions, as well as external consistency. Calvin [15] implements a middle layer for replication and transaction scheduling functions. Its transaction scheduling is serializable and deterministic over strongly consistent replicas. Eiger [16] enables causal consistency by adding dependency metadata to each write. These dependencies are checked before applying any write. Unsatisfying a dependency check causes a write to block till all writes it depending on have been applied. But fault-tolerant algorithms are only proposed for read-only and write-only transactions based on the causal replica consistency.

Recent transactional proposals target at high availability and fall into the third category. MDCC [11] supports transactions across multiple data partitions with replication based on Paxos algorithm, but it guarantees only the isolation level of read committed without lost update. Besides, applications cannot abort a transaction after it starts, unless the master of any data partition requests an abort. Replicated commit [10] layers replication over the distributed transaction support. It is blocking in data center with any failure as it exploits the blocking two-phase commit under fault-tolerant replication. There is also a recent work [17] on providing stronger consistency by implementing a middle layer over eventually consistent datastores such that availability and relatively stronger consistency are simultaneously guaranteed. However, transaction-level consistency is not provided.

In comparison to other transactional proposals for highly-available datastores that guarantee only weak consistency levels, SHAFT can guarantee the strong consistency of serializability even on failures. Moreover, while other Paxos-based protocols only deal with replication or commitment, SHAFT is a complete protocol integrating the replication, concurrency control and commitment procedures for transactions over partitioned, distributed and replicated data.

III. The SHAFT Protocol

SHAFT exploits Paxos as the basis. Each transaction corresponds to two Paxos instances, i.e. the *processing* instance and the *decision* instance. The processing instance is for operation processing and concurrency control in the transaction, while the decision instance supports users' request of abort after initiating a transaction. If the user never aborts a transaction after initiation, SHAFT requires only the processing Paxos instance. Transactions can be uniquely identified by their IDs, which can be distributively generated by hashing functions. Thus, the Paxos instances of a transaction can be uniquely identified as well.

A. Preliminary: Basic Paxos Algorithm

Paxos algorithm is for reaching a single consensus among a set of acceptors. A run of the Paxos algorithm is called an instance. Each instance can only reach a single consensus, disregard of failures. Three roles exist in the algorithm. They are a proposer, acceptors, and learners. The proposer is also

called the leader. With $2F+1$ acceptors, an instance can tolerate F failures [7].

Paxos algorithm can be employed for different scenarios requiring a distributed consensus through different incarnations, e.g. replication [18], coordination service [19] and transaction commit [20]. Although the four roles can be flexibly appointed in an incarnation, there is an important requirement to be satisfied in any incarnation. The set of acceptors and that of learners, which together are called a *configuration*, for an instance stay unchanged and known to any leader until a consensus is reached, even though some acceptors or learners can fail in the process.

Procedure. Each Paxos instance has three phases. Phase 1 and phase 2 have two sub-phases *a* and *b* respectively. A leader initiates a Paxos instance on receiving a proposal from the proposer. The initiator, being certain of its first and unique leadership in the instance, can proceed to send a phase 2a message directly; otherwise, the leader starts from phase 1a. In phase 1a, the leader chooses a ballot number *bal* that it believes to be larger than any ballot number seen in the instance. The leader sends a phase 1a message with *bal* to every acceptor. Phase 1b follows.

In Phase 1b, on receiving the phase 1a message for *bal*, if an acceptor has not yet performed any action for messages with a ballot number *bal* or higher, it responds with a phase 1b message consisting of the largest ballot number that it has ever seen, the largest ballot number that it has sent a phase 2b message with, and the accepted consensus in the corresponding phase 2b message. The acceptor ignores the phase 1a message if it has performed an action for a ballot numbered *bal* or greater.

Then in phase 2a, if the leader has received a phase 1b message with *bal* from a majority of the acceptors, it can choose the consensus value for this instance based on the following logic. If none of the majority of acceptors reports accepting any consensus before, the leader decides the consensus value, but usually picks the first value proposed by the proposer. Otherwise, let μ be the maximum ballot number of all the reported phase 2b messages, and let M_μ be the set of all those phase 2b messages that have ballot number μ. All the messages in M_μ have the same consensus *v*, which might already have been chosen. The leader has to set the consensus to *v*. Finally, the leader sends a phase 2a message with the consensus and *bal* to every acceptor. Phase 2b comes next.

In phase 2b, when an acceptor receives a phase 2a message for a consensus *v* and *bal*, if it has not seen a larger ballot number, it accepts *v* as the consensus, and sends a phase 2b message for *v* and *bal* to the leader. The acceptor ignores the message if it has already seen a higher ballot number. The following phase 3 ends the process.

In phase 3, when the leader has received phase 2b messages for *v* and *bal* from a majority of the acceptors, it knows that *v* has been accepted as the consensus and communicates that fact to all interested processes, usually learners and the proposer, with a phase 3 message. In most cases, acceptors also take the learn roles, thus phase 3 is generally not needed.

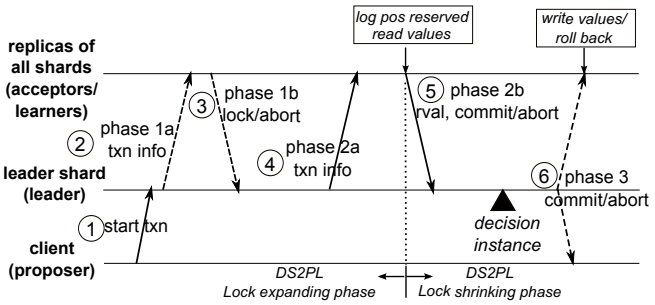

Figure 1. Typical sequence of messages and operations when using SHAFT. Steps ② and ③ are needed only when transaction leader failures happen.

B. The Processing Instance

Figure 1 illustrates the typical sequence of messages and operations when using SHAFT. The main part of the SHAFT protocol is the processing Paxos instance, which defines the major processing flow. In each processing instance, the client that submits a transaction takes the proposer role. All replicas of all shards accessed by a transaction takes the roles of acceptor and learner. Among all acceptors and learners, one of them is chosen as the leader. The choice of leader is through some predefined rule as illustrated in the following leader election part. With the predefined rule, even the client can find out the leader of a transaction. As long as the leader is alive, the leader is unique throughout a transaction. In sum, the configuration is static and known to the leader throughout the processing instance.

Leader election. On submission of a transaction, the client first collects all shards accessed by the transaction. An access can be a read or a write. Among all shards, there exists a shard *s* with the least ID. The client chooses the system node with the least ID among all nodes hosting replicas of *s*. This chosen system node becomes the leader in the processing instance.

A client submits the transaction to the chosen leader, which starts the processing instance. The leader thus enters phase 1a. The transaction information is included in the phase 1a message sent by the leader. Receiving a phase 1a message, an acceptor locks the *current* position of its log and enters phase 1b. If the current log position is already locked by other transactions, the acceptor adds a *reject* vote in its phase 1b message; otherwise, the current log position and an *accept* vote are included.

A different consensus. Note the changes as to the basic Paxos algorithm here. The consensus of the processing instance is whether to commit the transaction at the *current* position of each replica's log. Here, *current* position is interpreted into different numbers by different shards.

Thus, when incarnating the Paxos algorithm into the processing instance, the message sent in each phase changes. The replicas also need to process transactional operations accordingly.

In phase 2a, the leader waits until all acceptors respond or it times out. Then it proceeds to check if the *majority* condition is satisfied. If the majority condition is not satisfied, the leader sends out a phase 2a message with the *abort* consensus; otherwise, the leader adds the log positions voted by the *majority* and the *commit* consensus to its phase 2a message.

A different majority. Another change in the incarnation of the Paxos algorithm is the interpretation of *majority*. The majority condition now represents the following conditions: (1) there are a quorum of replicas voting for each shard involved in the transaction; (2) for each shard, a quorum of replicas vote the same position with the same accept or reject decisions.

On receiving a phase 2a message with the commit decision, an acceptor reads values requested by the transaction and sends a phase 2b message with the read values, the log position and the commit consensus. If the phase 2a message indicates to abort, the acceptor releases its lock to the current log position and replies to the leader with a phase 2b message indicating the abort.

The leader again waits until receiving all acceptors' phase 2b messages or it times out in phase 3. Then, it proceeds to check if there are a majority (or quorum) of replicas voting for each shard involved in the transaction. If so, it executes the transaction logic, and decides the transaction outcome by checking the majority votes for each shard. If all majority votes of all shards are to commit, then the transaction outcome is commit; otherwise, abort. A phase 3 message is sent out accordingly.

If an acceptor receives a phase 3 message with commit decision, it applies all writes and releases the lock of the current position. On an abort decision, the acceptor just releases the lock.

C. The Decision Instance

SHAFT supports the active transaction abort by client. This is achieved by using a **decision instance**. In comparison, transactional replication protocols like MDCC cannot support the same function. The decision instance can survive the user decision of the transaction outcome over leader failures. On leader recovery, the new leader can always reach consistent transaction decisions by restarting the processing and the decision Paxos instances.

As demonstrated in Figure 1, the decision instance is activated before phase 3. The decision instance follows the basic Paxos algorithm process with the transaction decision as the consensus value. Its leader is still the leader of the processing instance. The acceptors and the learners are different. Only replicas of the leader shard are the acceptors. The learners are those to recover a transaction. The acceptors of the decision instance store the accepted value next to that of the processing instance. Note that, the decision instance does not need a phase 3. After the leader collects a majority of phase 2b messages in the decision instance, it sends out the phase 3 message of the processing instance.

D. Fault Tolerance

The leader failure of a transaction's processing instance can be detected by other concurrent transactions accessing an intersecting data shards. When a transaction T_1 accesses a shard locked by another transaction T_0, T_1 can locate the leader replica of T_0 and probe to see if the hosting node is alive. If a leader failure is detected, the leader of T_1 can act for T_0's leader to drive T_0 to finish. As there might be multiple transactions trying to act for T_0's leader, there can be multiple leaders competing in T_0's processing and decision instances. The non-uniqueness of leaders in a Paxos instance can impair its liveness property. Therefore, we exploit *random backoff* techniques in such failure recoveries. That is, a substitute leader will randomly wait for sometime after it detects competition and before it restarts another round of recovery.

If a decision has already been decided by the last leader, the decision must have been accepted by the *decision* instance. The new leader will get to know the chosen decision after re-running the decision instance. Then the new leader must make the same decision as indicated by the consensus of the decision instance. Otherwise, the new leader can choose a transaction outcome freely.

E. Serializability – DS2PL

At present, SHAFT guarantees the serial processing of transactions. It locks all shards to read or write before processing the transaction. The locking forbids any concurrent operations by other transactions on the same shards. That is, the locks are exclusive. Besides all locks are not released until the transaction commits. The locking of SHAFT is in fact ***strong strict two-phase locking (SS2PL)*** [21]. No other transactions can be concurrently processed over any of the shards that a transaction accesses.

To increase concurrency, we can relax the locking procedure. We have a transaction release its locks on all shards that the transaction only reads and not write, once all read values are returned, i.e. immediately after sending the phase 2b message. This turns SHAFT's locking into the *strict two-phase locking (S2PL)*. The *expanding* phase of S2PL ends immediately after an acceptor sends the phase 2b message. This is also when the *shrinking* phase of S2PL starts. All locks on shards to be written are not released until the transaction commits. As SHAFT is a distributed protocol, we call the locking of SHAFT as ***distributed strict two-phase locking (DS2PL)***.

SS2PL, S2PL, and thus DS2PL are all proper subclasses of the *two-phase locking (2PL)*. It is proved and well known that 2PL can guarantee serializability [22]. Notice that, replicas of the same shard never evaluate to different states *before applying a transaction's writes*. Although failures can lead to the divergence of replica states, the recovered replicas can catch up by copying and processing virtual logs from the correct replicas, as illustrated by Megastore [8]. That is, SHAFT guarantees *single-copy* transaction histories. We can thus accordingly deduce that SHAFT guarantees serializability.

In fact, some processes in SHAFT can be further improved. For example, read transactions can lock no data to fasten the process, while they are guaranteed at lest the read-committed isolation level; or, transaction instances can merged as done in PaxosCP [12] to reduce communications; or, schemes for data distribution, replication and partition can be adjusted to suit the workloads and improve access latency [23]. The pseudocode of SHAFT can be found in our technical report [24].

IV. Evaluation

We evaluate SHAFT and compare it to the transactional replication protocol MDCC based on extensive experiments. We choose MDCC because it is fault-tolerant and outperforms a few widely known counterparts. Except for results reported in this section, we also summarize our observations on the usage of transactions in highly-available datastores. The observations can be found in our technical report [24].

A. Experimental Setup

We implement not only the protocols of SHAFT and MDCC, but also various experimental conditions of the underlying system for thorough test. For example, we implement conditions on how nodes connect to each other, how messages are sent to and arrive at each node, how workloads are applied to the system, etc. By such, we test system under different workloads in multi-datacenter scenarios. Furthermore, we test to find out how a system is influenced by different failures. To enable a fair comparison, the same workloads and the same failure conditions are applied to both protocols, while the two protocols run independently with results collected separately.

In the evaluation, we experiment with a multi-datacenter scenario. The number of datacenters is set to three or five in different experiments. The communication latencies between datacenters are randomly chosen from 20 to 200 times of the intra-datacenter round-trip time, which we denote as *one unit of time* in the experiments. We set the number of nodes in each datacenter to 50. Data shards are distributed to nodes. With regard to data distribution, we consider three cases: (1) replicas are placed across datacenters randomly by uniform distribution; (2) a majority of replicas are placed in a datacenter closest to the transaction clients; (3) a complete copy of data shards is placed at each datacenter.

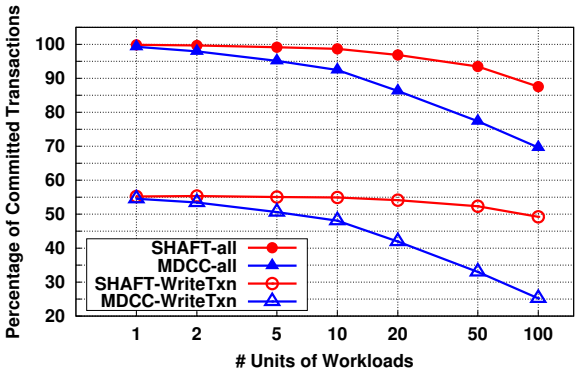

Figure 2. The percentage of committed transactions as workloads increase. SHAFT outperforms MDCC as workloads increase.

We generate transactions for different workloads. Transactions randomly access data shards, following either the uniform distribution or the Zipfian distribution. The percentage of cross-shard transactions is set to 20%. The number of shards accessed by a transaction is randomly chosen from less than 10. The number of operations in a transaction is randomly chosen from less than 50, but which is no smaller than the number of shards. To generate read-only transactions, we set 80% operations of all transactions to be reads and the others writes; but for a single transaction, the percentage varies randomly. Each operation processing, e.g. read or write, takes as long as sending an intra-datacenter message. We carry out three groups of experiments. The metrics that we concerns are the throughput rate in percentage and the response time in time units.

B. Workload Influence

We first evaluate the concurrency level. Different workload levels are considered, varying from 1 unit to 100 units of workloads. One workload unit is one transaction per hundred units of time, and 100 workload units means one transaction per unit of time. We randomly choose among all nodes to initiate a transaction.

SHAFT and MDCC have similar performance when the workload is low. As the workload increases, SHAFT outperforms MDCC. The reason is that SHAFT is a pessimistic concurrency control protocol and MDCC is an optimistic. High concurrency can lead to high abort rates of transactions using MDCC. On the other hand, the throughput of read-only transactions will keep at a high level using MDCC, although the read-only transactions can return values belong to different versions. As SHAFT guarantees serializability, read-only transactions always return consistent values. The throughput of read-only transactions will drop as the workload increases using SHAFT.

Figure 2 shows the throughputs of all committed transactions and committed write transactions for SHAFT and

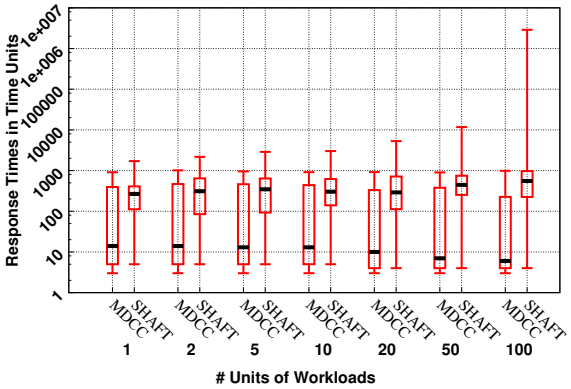

Figure 3. Response times as workloads increase. 75% transaction response times of SHAFT are comparable to those of MDCC, although SHAFT has a longer tail in response times.

MDCC as the workloads increase. The throughputs of both SHAFT and MDCC drop as workloads increase. MDCC's throughputs of committed write transactions drop fiercely as workloads increase. The reason is due to concurrent data access by transactions with intersecting data sets. More and more write transactions are aborted as workloads increase when using MDCC. SHAFT outperforms MDCC in throughputs of both committed transactions and committed write transactions, as workloads increase. Since MDCC guarantees only a low isolation level for read-only transactions, the throughput of committed read-only transactions is not influenced by the increasing workloads in MDCC. SHAFT guarantees serializability for all transactions, thus its throughput of committed read-only transactions is affected by the increasing workloads.

Figure 3 shows the boxplots of the responses times of committed transactions for SHAFT and MDCC. 75% transaction response times of SHAFT are comparable to those of MDCC, although SHAFT has a longer tail in response times. However, taking a closer look and observing

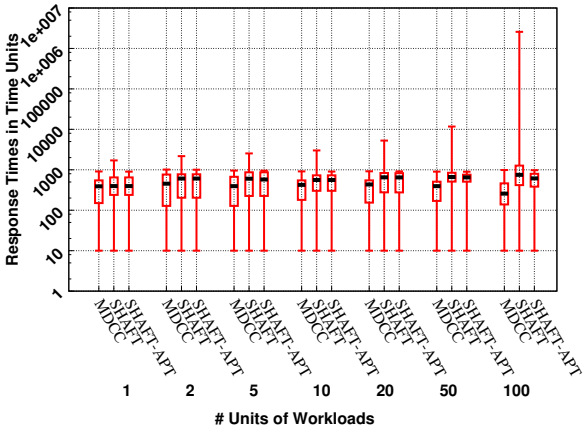

Figure 4. Response times of write transactions as workloads increase. The actual processing times of committed write transactions using SHAFT (SHAFT-APT) is very close to those of MDCC. The long tail of response times when using SHAFT is due to the waiting of locks.

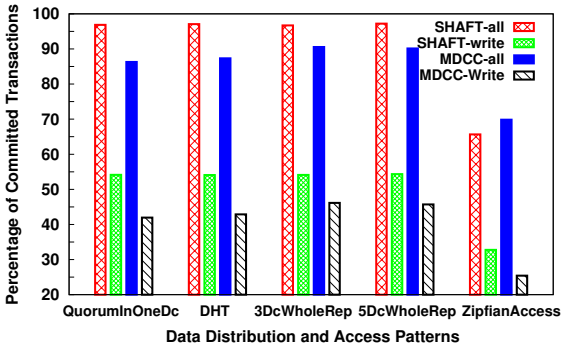

Figure 5. The percentage of committed transactions on different data distribution and access patterns. (a) SHAFT outperforms MDCC in throughputs of all scenarios except for ZipfianAccess; (b) SHAFT has a higher throughput of committed write transactions (SHAFT-wrtie vs. MDCC-write) under ZipfianAccess; (c) Whether a quorum of replicas are placed in the same datacenter, whether datacenters have complete copies of data, and the number of replicas when each datacenter has a full replica do not have impact on the throughput of SHAFT, but MDCC has higher throughputs of committed write transactions when datacenters have full replicas.

Figure 4, we can see that the actual processing times of committed write transactions using SHAFT (SHAFT-APT) is very close to those of MDCC. The long tail of response times when using SHAFT is due to the waiting of locks.

C. Data Distribution and Access Patterns

In actual deployments, data can be distributed to datacenters according to the application patterns. In the following, we study how data distribution and access patterns can influence the throughputs and response times of transactions. We apply 20 units of workloads in all scenarios for this subsection. We study five scenarios including placing a quorum of replicas in one datacenter (QuorumInOneDc), uniformly distributing data to nodes accross datacenters (DHT), placing a complete copy of data in each of the *three* datacenters (3DcWholeRep), placing a complete copy of data in each of the *five* datacenters (5DcWholeRep), and accessing data following Zipfian distribution under QuorumInOneDc condition (ZipfianAccess).

Figure 5 shows the throughputs of transactions on different data distribution and access patterns. We can easily observe that SHAFT outperforms MDCC in throughput in all scenarios except for ZipfianAccess. Under the Zipfian access pattern, SHAFT has a higher throughput of committed write transactions than MDCC. The reason is that many concurrent transactions accessing intersecting data sets have to be aborted in MDCC.

On the one hand, Zipfian access pattern greatly impacts the resulting throughput and concurrency. On the other hand, whether a quorum of replicas are placed in the same datacenter, whether datacenters have complete copies of data, and the number of replicas when datacenters have complete data copies do not have impact on the throughput of SHAFT, but MDCC has higher throughputs of committed write transactions when datacenters have complete data

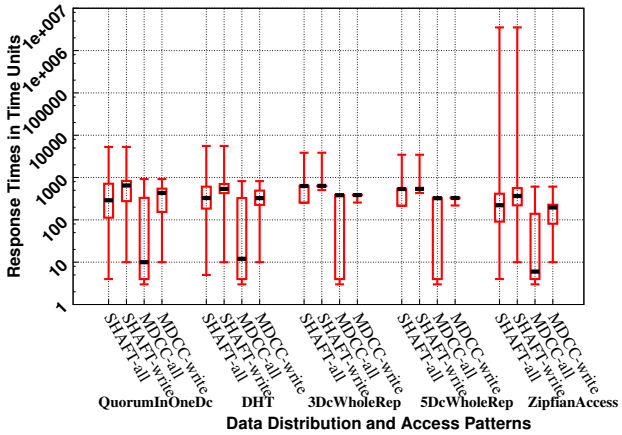

Figure 6. Response times on different data distribution and access patterns. (a) Write transactions take longer time to response than read-only transactions (-All vs. -Write); (b) Zipfian access pattern (ZipfianAccess) has a great impact on response times of SHAFT; (c) Whether a quorum of replicas is in one datacenter (QuorumInOneDc vs. DHT) does not lead to obvious differences in response times; (d) Complete copies in datacenters (3DcWholeRep & 5DcWholeRep) lead to monotonous response times; (e) Cross-datacenter communication latency has greater impacts on response times than the number of copies, when placing complete copies in datacenters (3DcWholeRep vs. 5DcWholeRep).

copies. The reason for MDCC's higher throughputs on complete data copies in datacenters is smaller abort rates and fewer competing transactions. In such scenarios, the workload generator tends to generate fewer transactions with intersecting data sets.

Figure 6 demonstrates the transaction response times on different data distribution and access patterns. We can easily observer that write transactions take longer time to response than read-only transactions. While Zipfian access pattern has greater impact on MDCC's throughput of committed write transactions, it has a great impact on response times of SHAFT. Whether a quorum of replicas is in one datacenter does not lead to obvious differences in response times, but complete copies in datacenters lead to a different distribution of response times. In fact, complete copies in datacenters lead to monotonous response times. Observing closely, we can see that cross-datacenter communication latency has

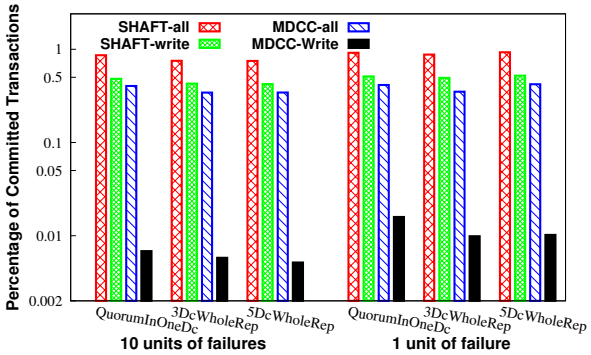

Figure 7. The percentage of committed transactions on failures. Failures have little influence on the throughputs of SHAFT, but have great impacts on MDCC's throughput of committed write transactions.

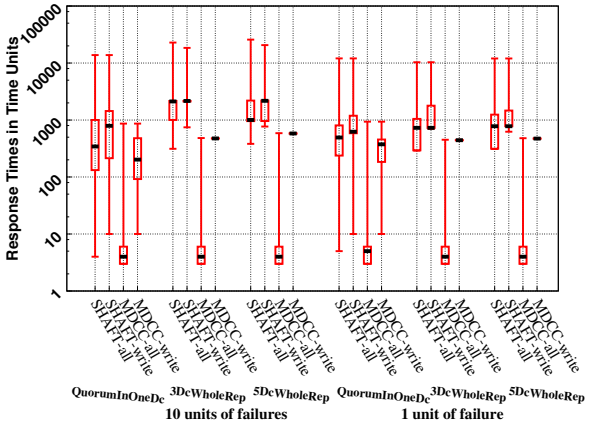

Figure 8. Response times on failures. The response times of SHAFT increase on failures as compared to the normal case, while the response times of MDCC's committed transactions remain stable.

greater impacts on response times than the number of copies, when placing complete copies in datacenters.

D. Fault Tolerance

In the case with failures, we find out how failures can affect the throughput and response times. We apply 10 units of workloads. Two kinds of failures are considered, i.e. datacenter blackout and failed nodes, but we will guarantee that no more than F among 2F+1 replicas failed simultaneously. When *1 unit of failure* is applied, some nodes randomly fail. With *10 units of failures*, a whole datacenter of nodes fail simultaneously. Here, *1 unit of failure* means that there is a failed node every hundred thousand units of time. Though the units of failures are different, all failable nodes will fail before every experiment ends, i.e. only a necessary quorum of replicas are left. The quorum is computed based on the tolerable failures of the Paxos algorithm.

Figure 7 shows the throughputs of committed transactions on failures. The *y* axis of Figure 7 is in log scale and the values are multiplied by 500 before applying the logarithm. Observing the result, we can see that failures have little influence on the throughputs of SHAFT, but have great impacts on MDCC's throughputs of committed write transactions. When failures happen, SHAFT exhibits a higher concurrency level than MDCC. When the application server of MDCC fails, no other nodes can take up the job and push the transaction to finish, thus leaving the transaction in the blocking state. In comparison, SHAFT permits failures of any role and continues to work as long as no more than tolerable number of acceptor/learner failures happen.

Figure 8 shows the transaction response times on failures. Comparing to the response times on 10 workload units in Figure 2, SHAFT's range of response times widens on failure conditions, while MDCC's remain stable. The reason is that the unsuccessful transactions are aborted or remain blocked in MDCC. On failures, both the number of retried

transactions and the number of lock-waiting transactions increase in SHAFT, thus leading to a wider range of response times. The more failures, the wider the range. Note that, when each datacenter has a complete copy of data, the number of copies does not have much influence on transaction throughput and response times, even on failures.

V. Conclusion

Targeting at providing serializable transaction processing for highly-available datastores, we propose in the paper SHAFT, a Paxos-based protocol integrating the replication, concurrency control and commitment procedures. SHAFT uses pessimistic concurrency control based on distributed strict two-phase locking. It guarantees serializability, high availability and fault tolerance simultaneously. Different from other synchronous transactional replication protocol, SHAFT allows clients to actively abort a transaction. In comparison to other Paxos-based proposals, the key techniques that SHAFT exploits are the different incarnation and the updated operation semantics of the Paxos algorithm. SHAFT guarantees strong transaction consistency with fault tolerance. SHAFT is non-blocking; it tolerates failures of both clients and servers. Experiments show that SHAFT outperforms the synchronous transactional replication protocol MDCC, which outperforms other protocols such as Megastore.

Acknowledgment

This work is supported in part by the State Key Development Program for Basic Research of China (Grant No. 2014CB340402) and the National Natural Science Foundation of China (Grant No. 61303054 and No. 61432006).

References

[1] "Amazon cloud goes down friday night, taking netflix, instagram and pinterest with it," October 2012, http://www.forbes.com/sites/anthonykosner/2012/06/30/amazon-cloud-goes-down-friday-night-taking-netflix-instagram-and-pinterest-with-it/.

[2] "Reddit, quora, foursquare, hootsuite go down due to amazon ec2 cloud service troubles," April 2011, http://www.huffingtonpost.com/2011/04/21/amazon-ec2-takes-down-reddit-quora-foursquare-hootsuite_n_851964.html.

[3] "Lightning causes amazon, microsoft cloud outages in europe," August 2011, http://www.crn.com/news/cloud/231300384/lightning-causes-amazon-microsoft-cloud-outages-in-europe.htm.

[4] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the 19th PODC*. ACM, 2000, pp. 7–.

[5] P. Bailis and A. Ghodsi, "Eventual consistency today: limitations, extensions, and beyond," *Commun. ACM*, vol. 56, no. 5, pp. 55–63, May 2013.

[6] D. R. Ports and K. Grittner, "Serializable snapshot isolation in postgresql," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1850–1861, 2012.

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

[8] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proc. of CIDR*, 2011, pp. 223–234.

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally-distributed database," *Proceedings of OSDI*, p. 1, 2012.

[10] H. A. Mahmoud, A. Pucher, F. Nawab, D. Agrawal, and A. E. Abbadi, "Low latency multi-datacenter databases using replicated commits," in *Proceedings of VLDB 2013*.

[11] T. Kraska, G. Pang, M. J. Franklin, and S. Madden, "Mdcc: Multi-data center consistency," in *Eurosys*, 2013.

[12] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. El Abbadi, "Serializability, not serial: Concurrency control and availability in multi-datacenter datastores," *Proceedings of the VLDB Endowment*, 2012.

[13] S. Das, D. Agrawal, and A. El Abbadi, "G-store: a scalable data store for transactional multi key access in the cloud," in *Proceedings of ACM SoCC*. ACM, 2010, pp. 163–174.

[14] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proceedings of ACM SOSP*, 2011, pp. 385–400.

[15] 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 ACM SIGMOD*, 2012, pp. 1–12.

[16] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Proceedings of USENIX NSDI*, 2013.

[17] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proceedings of SIGMOD 2013*. ACM, 2013, pp. 761–772.

[18] 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 NSDI*. USENIX Association, 2011, pp. 11–11.

[19] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th OSDI*. USENIX Association, 2006, pp. 335–350.

[20] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 133–160, Mar. 2006.

[21] Y. Raz, "The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource mangers using atomic commitment," in *Proceedings of the VLDB Endowment*, 1992.

[22] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-wesley New York, 1987, vol. 370.

[23] S. Kadambi, J. Chen, B. Cooper, D. Lomax, R. Ramakrishnan, A. Silberstein, E. Tam, and H. Garcia-Molina, "Where in the world is my data," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, 2011.

[24] Y. Zhu, "Shaft: Serializable, highly available and fault tolerant concurrency control in the cloud," Institute of Computing Technology, Chinese Academy of Sciences, No. ICT-ACS-SG-201301, Tech. Rep., 2013, http://prof.ict.ac.cn/~yuqing/SHAFT/shaft.pdf.