SHAFT: Serializable, Highly Available and Fault Tolerant Concurrency Control in the Cloud

By

Yuqing Zhu

Technical Report No. ICT-ACS-SG-201301

July 2013

Institute of Computing Technology

Chinese Academy of Science

Beijing, China 100190

SHAFT: Serializable, Highly Available and Fault Tolerant Concurrency Control in the Cloud

Yuqing Zhu
Advanced Computer System Research Center
Institute of Computing Technology, Chinese Academy of Sciences

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. SHAFT is a pessimistic concurrency control protocol for partitioned and replicated data, which can be distributed across multiple datacenters. Laying its basis on the Paxos algorithm, the SHAFT protocol guarantees Serializability, High Availability and Fault Tolerance simultaneously for transactions. The distributed concurrency control process of SHAFT meets the strict two-phase locking requirements. SHAFT can restart a transaction aborted due to the inability to lock its data. High availability is guaranteed as read-only transactions are processed with shorter procedures. Concurrent transactions can be merged together and processed in one SHAFT instance to reduce costs and latency, while increasing concurrency. Different from other existing transactional replication protocols, 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 a recent related work MDCC, which outperforms other synchronous transactional replication protocols, e.g. Megastore.

KEY WORDS: transaction; concurrency control; serializability; availability; fault tolerance

SHAFT: Serializable, Highly Available and Fault Tolerant Concurrency Control in the Cloud

Yuqing Zhu
State Key Laboratory of Computer Architecture
Advanced Computer System Research Center
Institute for Computing Technology, Chinese Academy of Sciences, Beijing 100190, China
zhuyuqing@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. SHAFT is a pessimistic concurrency control protocol for partitioned and replicated data, which can be distributed across multiple datacenters. Laying its basis on the Paxos algorithm, the SHAFT protocol guarantees *S*erializability, *H*igh *A*vailability and *F*ault *T*olerance simultaneously for transactions. The distributed concurrency control process of SHAFT meets the *strict two-phase locking* requirements. SHAFT can restart a transaction aborted due to the inability to lock its data. High availability is guaranteed as read-only transactions are processed with shorter procedures. Concurrent transactions can be merged together and processed in one SHAFT instance to reduce costs and latency, while increasing concurrency. Different from other existing transactional replication protocols, 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 a recent related work MDCC, which outperforms other synchronous transactional replication protocols, e.g. Megastore.

1. INTRODUCTION

High availability and fault-tolerance are two important properties of systems in the cloud. Nodes of a system can fail; and, an entire datacenter can also become unavailable. For example, in June 2012, Amazon's Elastic Compute Cloud in North Virginia went down due to thunder storms[6]. Similar outages have been reported by other service providers such as Google, Facebook, and others. In many of these instances, failures 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 [5, 4]. Therefore, it is important that the software layer of a system implements measures to guarantee high availability and fault-tolerance.

Transaction support is also a very valuable feature for database-backed applications. Consistency was once relaxed to improve availability and fault-tolerance, as consistency, availability and network partition tolerance are in a trade-off relation [14]. However, the lack of strong consistency semantics, e.g. transaction support, leads to great difficulty of application development [8]. In recent years, there are increasing attentions on transaction support in the cloud. Among all isolation levels, serializability is a much desired one by applications [32] and guarantees the strongest consistency.

To support scalability, high availability and fault tolerance in the cloud, there are three fundamental measures, i.e. data partition, distribution and replication. But these measures increase the difficulties of supporting transactions. Two phase commit (2PC) is the most accepted protocol for distributed transaction commit [35], while two phase locking (2PL) is the most widely used technique to guarantee serializability [12]. On the other hand, the Paxos algorithm [24] is the most widely known protocol in guaranteeing high availability and fault-tolerance based on replication. Various projects are thus proposed to support transactions exploiting the above three techniques [10, 29, 17]. Nevertheless, there are few complete proposals supporting transaction and replication over partitioned and distributed data with high availability, fault-tolerance, and serializability guarantees simultaneously.

In fact, transactions over replicated data are inherently different from those over non-replicated data. With replication, each localized transaction in distributed databases now becomes a distributed one. We might treat each replica as a different shard and run 2PC as if in classic distributed databases. But fault-tolerance and availability will be impaired, as 2PC is a blocking and non-fault-tolerant protocol [20]. While there exists some fault-tolerant distributed commit protocol [20], one would think that we can use such protocol over the basic Paxos algorithm and classic local concurrency control methods, e.g. 2PL or timestamp ordering [12], to integrate a solution for the serializable, highly available and fault tolerant transactional support. In such case, *replication*, *local concurrency control* and *distributed commit decision* are handled independently. When there are two concurrent transactions with intersecting data partitions, *the replication component* can align one transaction's writes before another transaction's writes, but *the distributed commit decision component* can decide the two transactions to commit in a reverse order. This results in inconsistency and thus non-serializability!

As important as high availability and fault-tolerance is, most recent proposals for supporting transactions over partitioned, distributed and replicated data place their basis on the Paxos algorithm nonetheless. In these proposals, Paxos-based components are mainly for replication or *atomic commitment*. Megastore [10] is based on the Paxos algorithm and 2PC, but supporting transactions only within single entity group, while Spanner [17] with the same basis supports global transactions. Unfortunately, 2PC is block-

ing and not fault-tolerant. MDCC [22] supports transactions across multiple data partitions with replication based on the Paxos algorithm, but it guarantees some isolation level unknown before but weaker than serializability. Besides, applications cannot actively abort a transaction once it starts, which deviates the transaction definition. The most recent proposal is replicated commit [29], which layers replication over the distributed transaction support. As the full replica in each data center is assumed and each datacenter processes transactions independently, there is high possibility that different datacenters make different decisions on transaction scheduling, especially when users are submitting transactions from various geographical locations, such that conflicts among replicas occur. Although Paxos-CP [24] guarantees serializable transactions based on the Paxos algorithm, it also assumes a full replica and a transaction service per datacenter such that all transactions in a datacenter are coordinated by the single transaction service, thus leading to low concurrency level and scalability.

Despite the challenges of the problem, the work [20] that adapts the original Paxos algorithm for the transactional commit purpose motivates and encourages us. In this paper, we also take an adaptation approach towards the problem of supporting transactions with serializability, high availability/concurrency, and fault tolerance guarantees simultaneously. The Paxos algorithm is the basis to achieve high availability and fault tolerance. We *incarnate the Paxos algorithm differently* from other proposals, including the instance, the consensus, the configuration and the leadership determination. We also *update the semantics of* ***majority***, such that a Paxos-based distributed two phase locking procedure is implemented to guarantee serializability. 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. not full replica in a datacenter or datacenters with different partition sets, our proposal is also feasible. In this paper, we make the following key contributions by SHAFT:

- A new distributed concurrency control protocol, which guarantees serializability, high availability and fault tolerance simultaneously for transactions over the wide-area network.
- A simulation framework that exempts the trouble of protocol implementation details and that fairly compares concurrency control protocols, under various configurable conditions.
- Performance results from extensive simulations showing that SHAFT guarantees stronger consistency with costs similar to the best counterpart, and providing higher concurrency.

In Section 2 we first introduce some preliminary knowledge that we heavily rely upon in the presentation of SHAFT. Section 3 describes SHAFT's new concurrency control protocol, along with the correctness proof. Section 4 presents a few mechanisms to reduce the cost and improve the performance of SHAFT. We evaluate SHAFT and summarize a few application indications based on experiments in Section 5. In Section 6 we relate SHAFT to other works. We finally draw the conclusion and point out some interesting future work in Section 7.

2. PRELIMINARY

We first briefly describe the Basic Paxos algorithm. The key concepts, as well as what can be changed and what must be followed in an incarnation, are pointed out. The terms used in the description will be adopted in the presentation of SHAFT to facilitate the discussion. Besides, the description also serves as a basis to demonstrate the different incarnation and the updated term meanings of the Paxos algorithm in SHAFT. Then, we introduce the data model and the infrastructure that SHAFT is based on. The assumptions made upon the data model are important as to the scalability and high availability guarantees.

2.1 Basic Paxos Algorithm

2.1.1 Overview and Key Concepts

Paxos algorithm is for reaching a single consensus among the set of acceptors. A run of the Paxos algorithm is called ***an instance***. Each instance can only reach a single consensus, disregard of failures. Four roles exist in the algorithm. They are a proposer, a leader, acceptors, and learners. The set of acceptors and that of learners, which together are called a *configuration*. With $2F+1$ acceptors, an instance can tolerate F failures [24].

Safety and liveness properties constitute the fundamental correctness criteria of a distributed algorithm. The safety property of the Paxos algorithm can be proved even in the face of failures [24]. If the uniqueness of leadership is guaranteed, the algorithm also has the property of liveness. The famous FLP impossibility result [19] implies that a reliable algorithm for electing a leader must use either randomness or real time−for example, by using timeouts. Anyhow, safety is ensured by Paxos algorithm regardless of the success or failure of the leader election, providing fault tolerance. In actual implementation, we can devise various algorithms to elect a leader, e.g. [7].

Paxos algorithm can be employed for different scenarios requiring a distributed consensus through different incarnations, e.g. replication [13], coordination service [15] and transaction commit [20]. The four roles of the Paxos algorithm can be flexibly appointed in an incarnation. To exploit and incarnate the Paxos algorithm for a new scenario, the concepts of ***instance***, ***consensus***, ***configuration***, and ***leadership*** can be flexibly appointed with any concrete meanings, as long as the following conditions are met. First, an instance can be uniquely differentiated. Second, each instance can only reach a single consensus, which can be equally interpreted by all acceptors and learners. Third, the 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[1]. Fourth, the uniqueness of leadership is key to the liveness property of the algorithm. Fifth, the basic procedure that constitutes the main part of the Paxos algorithm must be followed.

2.1.2 Basic 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. ***Given that the instance is start anew, i.e. that no consensus value is ever chosen, and that the leadership is sure to be unique, the first phase can be skipped and the initiator can directly proceed to send a phase 2a message***; 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, when an acceptor receives the phase 1a message for *bal*, if it has not already 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.

[1]A recent improved version of Paxos [26] permits reconfiguration with the help of an auxiliary configuration master, but it transforms the problem of a single leader into one of a non-failable configuration master.

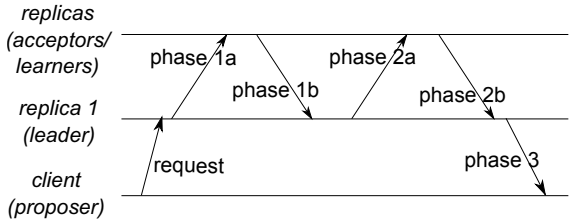

Figure 1: Roles and phases in Paxos algorithm for replicated state machines.

The acceptor ignores the phase 1a message if it has performed an action for a ballot numbered *bal* or greater. Now comes phase 2a.

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.

2.2 Data Model and Infrastructure

Our implementation of SHAFT runs on a key-value store, but SHAFT is a distributed concurrency control protocol that is agnostic to whether the database is relational, or is a key-value store. We target databases that are partitioned and distributed. Distribution is based on partitions, which are called *shard* and replicated. The size of the shard can be as large as a whole database, or it can be as small as a data unit for operation, e.g. record. Replication can be uniform, i.e. each partition with the same replica number, or non-uniform. All replicas of a shard are peers in the sense that they can be equally accessed by users. Each shard is uniquely identified by its ID. The IDs of all shards can be ordered monotonically, i.e. there exists a least ID among a set of IDs. System nodes are also uniquely identified by their IDs and located in one or more datacenters. Each system node hosts a subset of shards. Note that, the above assumptions are necessary and common, as the three techniques of data partition, distribution, and replication, all of which are implemented in almost all NoSQL databases [1, 3, 2], are the fundamental techniques to guarantee ***scalability and high availability***.

We assume the replicated state machine (RSM) [23] model for each replica. Each replica of a shard is an RSM, whose state is deterministic by its corresponding sequence of operations. The system node hosting a shard executes the sequence of operations for the shard to change its state. Usually, RSM model is implemented as a log as in Megastore, though this is not necessary. The Paxos algorithm can be exploited for fault-tolerant replication based on the RSM model. In such exploitation, the leader is chosen among all replicas of a replica. All replicas of the shard constitute the acceptors and the learners. At any moment, only one Paxos instance is allowed for one shard. The proposer is the client. Figure 1 illustrates the execution process of the basic Paxos algorithm for RSM. Notice that, phase 3b message is not sent to learners, i.e. replicas, as they already learn them as acceptors.

3. THE SHAFT PROTOCOL

SHAFT integrates the replication and the concurrency control in transactional databases, providing a complete concurrency control protocol for transactions over partitioned, distributed and replicated data. Different from other existent proposals, SHAFT treats a singular read/write operation over replicated data as a transaction. The integration of replication with concurrency control makes a complete alignment of consistency levels possible, which will be discussed in the future work section.

Figure 2 illustrates the typical sequence of messages and operations when using SHAFT. SHAFT exploits the Paxos algorithm as the basis. Each transaction corresponds to *two Paxos instances*. One of the instance is to support users' request of abort after initiating a transaction. We call this as the *decision* instance and the other the *processing* instance. ***If the transaction commit decision can be made locally by each shard or the transaction is read-only, SHAFT requires only the processing Paxos instance.*** Transactions can be uniquely identified by their IDs, which can be generated distributively by hashing functions. Thus, the Paxos instances of a transaction can be uniquely identified as well.

3.1 The Processing Instance

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. In the following, we describe the whole process of the processing instance, pointing out the adaptations where they first occur.

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 execute the transaction at the *current* position of each replica's log, such that reads of the transaction return values corresponding to the *current* position and writes, if committed, are applied at the *current* position. Here, *current* position is interpreted

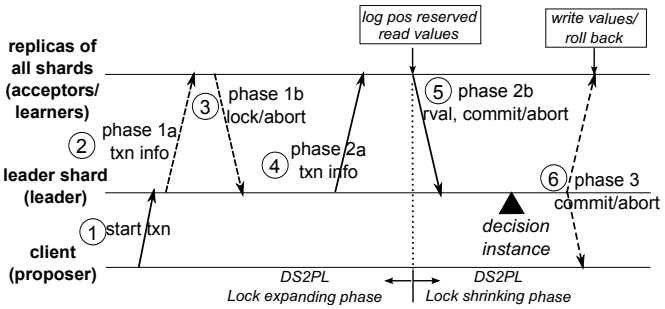

Figure 2: Typical sequence of messages and operations when using SHAFT. Steps ② and ③ are needed only when transaction leader failures happen.

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 take transaction processing 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.

3.2 The Decision Instance

Compared to the inability of supporting active transaction abort by user in other Paxos-based concurrency control approaches, e.g. MDCC, SHAFT supports the **active transaction abort by client**. This is achieved by including a *decision instance* in SHAFT.

As demonstrated in Figure 2, the decision instance is activated before phase 3. The consensus for the decision instance is *whether to commit the transaction and what order to apply the committed writes*. Note that, the actual values for the consensus are decided by the leader. The decision instance 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. Besides, if no failures even happen and the leadership is unique, which is the common case, the decision instance can skip phase 1 and directly move to phase 2.

The main function of the decision instance is to 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.

3.3 Lock Acquisition Failure and Restart

The concurrent processing of cross-partition transactions with intersecting shard sets can compete for the same shard, but both abort due to the competition. For example, transaction T_1 accesses shards X,Y,Z and T_2 accesses Y,Z,A. If T_1 locks X,Y and T_2 locks A,Z, then the acceptors for Z have to vote reject on T_1 and those for Y also vote reject on T_2. Such leads to the abort decisions of both T_1 and T_2, according to the current SHAFT protocol.

To restart the transaction after deadlock abort, we must first distinguish the fail-to-lock from the fail-on-consistency-check in the reject vote by the acceptor. If the acceptor cannot lock a requested shard, it votes reject; or, if it finds the transaction does not meet the consistency conditions, e.g. predefined constraints, it also votes reject. Therefore, we improve the SHAFT protocol by giving acceptors three choices instead of two. The three choices are *lockReject*, *checkReject*, and *accept*. Both *lockReject* and *checkReject* are called *reject*. When sending a phase 1b or phase 2b message, an acceptor votes *lockReject* on failing to lock and *checkReject* on consistency check failures.

The processing logic of the leader must also be adapted. In phase 2a and phase 3, if $\overline{Q} = |total - majority|$ votes from replicas of any shard include reject votes and all reject votes are *lockReject*, the leader marks the instance as *toRestart*. After phase 3, the *toRestart* instance does not end at the leader. Rather, the leader waits for a random time and restarts as if receiving a proposal from the same transaction client again. Note that, the leader will only restart a transaction when it receive no other reject votes, i.e. *checkReject*, except for *lockReject*. Besides, this *toRestart* must be proposed together with the transaction outcome decision as the consensus for the *decision* instance, in order to preserve fault tolerance.

However, on the transaction restart due to lock acquisition failures, the transaction gets a new unique transaction ID. The reason is that the consensus, i.e. *abort*, has already been reached in the previous instance. With the same transaction ID, it means the same instance, thus the consensus will never change. Thus, the leader restarts a fail-to-lock transaction with a new unique ID and in new processing/decision instances.

3.4 Fault Tolerance and Protocol Correctness

To prove that SHAFT is fault tolerant, we need only first demonstrate that it preserves the ***safety*** and the ***liveness*** properties of the Paxos algorithm. Secondly, we need to present measures for leader failure handling.

Protocol Correctness. We briefly sketch an intuitive proof of the correctness of SHAFT. A formal presentation of the proof is left for future work. Recall that, in Section 2, we stated that a valid incarnation of the Paxos algorithm can give any concrete meanings to the instance, the consensus, the configuration, and the leadership concepts without hurting the safety and the liveness properties, as long as the five conditions are met. Examining the SHAFT protocol, we can easily see all those conditions are satisfied.

First, the processing and the decision instances of SHAFT can be uniquely identified by the corresponding transaction ID. Second, the processing instance only reaches the single consensus of *whether to execute the transaction at the current position of each replica's log* and the decision instance only reaches the single consensus of *whether to commit the transaction and what order to apply the committed writes*. Third, all shards and their replicas involved in the transaction constitute both instances' configuration, which remains unchanged and known to any leader that is chosen among the acceptors. Fourth, the strict ordering of shard IDs establishes the unique leadership, which is guaranteed even on failures as described in the following. Fifth, the basic procedure is followed, except for the *majority* condition.

Now, we need only prove that SHAFT's interpretation of *majority* does not impair the protocol correctness. In the correctness proof [24] of the Paxos algorithm, the *majority* condition is only to guarantee that there still exist acceptors/learners to remember and tell the progress of the algorithm execution. The first statement of our *majority* interpretation for the processing instance can make the above guarantee. The second statement is to strengthen the condition that there must be enough acceptors voting the same position and the same decision. This cannot violate the guarantee made by the first statement. Besides, as long as there are more than a quorum of correct replicas for a shard, the second statement can naturally follow from the first one.

However, the *majority* interpretation can affect the tolerable failures. Although all replicas of all shards accessed by a transaction act as acceptors and learners in the processing instance of SHAFT, the number of tolerable failures remains to be F, where the least replica number among all shards' replica numbers is $2F + 1$, according to SHAFT's *majority* interpretation. The decision instance also tolerates the same number of failures, i.e. F.

Leader Failure Handling. We take a timeout plus random backoff approach towards leader failure handling. Each acceptor in the transactional instance times on how long it receives responses from the current leader. If it times out, it first probes the leader. On receiving no response, it requests the next leader by the leader selection rule to resume the transaction processing. The new leader can then resume the transaction processing from the first phase of the processing instance or the decision instance.

If the original leader becomes active again, there can be a competition of leadership. The non-uniqueness of leaders in a Paxos instance can impair its liveness property. Therefore, we exploit the *random backoff* technique in such failure recoveries. That is, on not acquiring the leadership, any competing substitute leader waits for a random time before its next attempt to obtain the leadership.

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.

In comparison to MDCC, we are employing a leader for each transaction, thus the failure of the client cannot impact the liveness and safety properties of the SHAFT protocol. Besides, as the leader is uniquely chosen among all acceptors, the liveness of the SHAFT protocol is guaranteed, as long as the number of failures is no more than what SHAFT can tolerate.

3.5 The Proof of Serializability – DS2PL

At present, we have been certain that SHAFT can execute transactions in a fault-tolerant way. In the following, we briefly sketch an intuitive proof of the serializability of SHAFT in transaction processing.

Recall the processing procedure of SHAFT. It locks all shards to read or write before transaction processing in the processing instance. 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 after the decision instance. The locking of SHAFT is in fact ***strong strict two-phase locking (SS2PL)*** [33]. No concurrent transactions can be processed over any of the shards accessed by a transaction.

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 writes, once all read values are returned, i.e. immediately after sending the phase 2b message in the processing instance. 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 in the processing instance. This is also when the *shrinking* phase of S2PL starts. All locks on shards to be written are not released until the transaction commits in the decision instance. 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 guarantees serializability [12]. Note that, all replicas of a 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 [10]. Thus, SHAFT guarantees *single-copy* transaction histories. Then we accordingly deduce that SHAFT guarantees serializability.

3.6 SHAFT Pseudocode

Up to now, we have the complete SHAFT protocol in its basic version. We list the pseudocode of the SHAFT processing instance in algorithms 1, 2 and 3, with the used symbols and variables defined in Table 1. The pseudocode of the decision instance is not listed because we think its pseudocode can be easily obtained through a slight modification to that of the processing instance.

For the processing instance, the codes for the client, the acceptors and the learners are in Algorithm 1, while those for the leader are in Algorithm 2. The pseudocode in Algorithm 3 defines the important *majority* condition and the *randomBackoff* process for the leader.

A client starts a transaction by calling the `INITTRANSACTION(tx)` function on line 1. Leader election rule is executed on line 3. The leader processes the transaction request from line 53. The beginning of each phase is signified by receiving messages from the previous phase. For example, an acceptor receiving a *Phase2a* message starts the Phase 2b (lines 10-11, 30-45). On leader failures and local timeouts, an acceptor requests the resume of the transaction execution (lines 28-29, 44-45).

The leader decides whether a consensus has been reached for the processing instance in the `MAJORITY` function (lines 106-122). The first statement of the consensus for the processing instance is checked on line 109, while the second statement is checked on lines 111-121. If the majority condition is not satisfied, the leader will retry after a random backoff (lines 65, 73).

The transaction commit decision is made by the leader upon all collected votes from acceptors (lines 70-80). If the transaction is aborted due to lock contention, the leader restarts the transaction (lines 75-76). The leader further makes the consensus decision for the decision instance on lines 96-100. It then triggers the decision instance on line 101. The final decision is sent to all acceptors and the client on line 103. An acceptor processes the decision and releases the locks on lines 46-50.

Table 1: Definition of symbols for SHAFT pseudocode.

Symbols	Definitions
tx	the transaction
S_{tx}	the set of all shards accessed by tx
s	a shard
H	all nodes hosting replicas of shards in S_{tx}
l	a leader
m	ballot number
S_{msg}	set of all received messages
S_p	set of (vt, pos) pairs by majority replicas of all shards
Q	a quorum number of a set
$\overline{Q}$	the size of a set minus the quorum Q
vt	a vote of acceptance vt_a of rejection vt_r
vt_{rl}/vt_{rc}	reject vote due to fail-to-lock/fail-on-consistency-check
ups	writes of the transaction
$ldrBal$	the leader's current ballot number
$mbal$	the largest bal an acceptor ever sees
m_a	the accepted bal in an acceptor's message
bal_p, vt_p	the bal and the vote an acceptor accepts
pos_c	the current least unused log position
$rval$	values returned by reads
$restart$	whether a leader restarts tx on a *fail-to-lock* abort

4. IMPROVEMENTS

Paxos-based protocols are said to be costly due to multiple communication round trips. Thus, many efforts [25, 27] have been devoted to reducing the costs. In this section, we discuss a few improvements which can be made upon SHAFT to increase availability and concurrency, but which are not limited to Paxos communication reductions.

4.1 Read-Only Transactions - Availability

A read-only transaction, even one with only a single read operation, is treated as a complete transaction in SHAFT. Each transaction must go through the whole process of SHAFT. This greatly reduces the availability of the database system, especially when read-only transactions form the major part of user requests. We can improve availability through the following measures.

Review the processing procedure of SHAFT in Figure 2. All read values are returned in step ⑤. Locks are released at all replicas of read-only shards. If a leader receives the transaction request from a client, the leader can be certain that the following processing is not a recovery, such that the SHAFT's process can skip steps ② and ③. That is, in a correct process, only steps ④ and ⑤ are required for a read-only transaction. The leader can respond to the client immediately after step ⑤, skipping the decision instance and step ⑥. Be aware that, the leader is necessary here in case of the fail-to-lock abort. This involves merely two communication round trips in all. Serializability is nevertheless guaranteed for all transactions.

We can further let the client send its read-only transaction to any active replica of a shard it accesses. The replica reads the current value and responds to the client. At the same time, transactions with writes are processed in normal SHAFT procedures. Since writes are not applied until a transaction commits, we can still guarantee read-committed consistency level for all read-only transactions at least.

Among these read-committed transactions, if a transaction has only one read operation, it can read outdated data committed by an early transaction; but when a transaction consists of multiple read operations, it can exhibit the read-skew anomaly [11]. Consider transaction T_0 that reads X, Y and T_1 that writes X, Y. Due to communication delay, T_0 reads the value of X before T_1 commits and the value of Y written by the committed T_1. The phantom anomaly is also possible. However, the fuzzy read anomaly will never occur, since read operations are always executed only once on the data. By this, we can deduce a new transaction consistency level ***Read-Committed***+, which is stronger than the read-committed consistency level but weaker than the repeatable-read consistency level.

Notice that, the changes to how we process read-only transactions do not have any influence on transactions with writes. Thus, all other non-read-only transactions are still guaranteed with serializability.

Algorithm 1: Pseudocode for SHAFT - client, acceptor, learner

```
1  Procedure InitTransaction(tx)                                   // client
2      s ← the shard with the least ID in S_tx;
3      l ← the live node hosting a replica of s and with the least ID;
4      send Propose[tx, S_tx, H] to l;
5      wait for response from l;
6  Procedure ReceiveAcceptorMessage(msg)                           // acceptor
7      switch msg do
8          case Phase1a[m, tx]
9              Phase1b(m, tx, l); break;
10         case Phase2a[m, tx, vt]
11             Phase2b(m, tx, vt, l); break;
12         case commit[tx, up]
13             process(tx, vt, ups); break;
14         case abort[tx]
15             process(tx, vt, ∅); break;
16 Procedure Phase1b(m, tx, l)                                      // acceptor,learner
17     if tx locks pos_c ∨ ¬lock(pos_c) then // not locked by others
18         if instance(tx) never seen then
19             mbal ← −1, bal_p ← null, vt_p ← null;
20         if mbal < m then
21             mbal ← m;
22             if bal_p ≠ null then
23                 send Phase1b[mbal, bal_p, vt_p, pos_c] to l;
24             else
25                 send Phase1b[mbal, bal_p, vt_a, pos_c] to l;
26     else
27         send Phase1b[m, −1, vt_rl, −1] to l;
28         if tx_p locks pos_c ∧ timesOut(tx_p) then
29             send Recovery[tx_p, m] to the new leader;
30 Procedure Phase2b(m, tx, vt, l)                                  // acceptor,learner
31     if tx locks pos_c ∨ ¬lock(pos_c) then // not locked by others
32         if mbal ≤ m then
33             mbal, bal_p ← m, vt_p ← vt;
34             if vt == commit then
35                 read rval;
36                 if consistencyCheck() then vt_p ← vt_a;
37                 else vt_p ← vt_rc;
38             else unlock(pos_c);
39             send Phase2b[bal_p, vt_p, pos_c, rval] to l;
40             if tx is readOnly on pos_c then
41                 unlock(pos_c)                                   // release read locks
42     else
43         send Phase2b[m, vt_rl, −1, null] to l;
44         if tx_p locks pos_c ∧ timesOut(tx_p) then
45             send Recovery[tx_p, m] to the new leader;
46 Procedure process(tx, vt, ups)                                   // learner
47     if tx locks pos_c then
48         if vt == commit ∧ ups ≠ ∅ then
49             apply ups;
50         unlock(pos_c)                                           // release write locks
```

4.2 Merge of Instances

Now, we consider the situation when the processing/decision instances of multiple transactions can be merged into one. Such merge is possible only because we have chosen the RSM model and SHAFT locks the virtual log position instead of the real data.

According to the leader selection criteria of SHAFT, the same system node will be chosen as the leader for transactions with the

Algorithm 2: Pseudocode for SHAFT - leader

```
47 Procedure RECEIVELEADERMESSAGE(msg)                          // leader
48     S_msg1, S_msg2 ← ∅, restart ← false;
49     switch msg do
50         case Recovery[tx, m]
51             ldrBal ← m + rand();
52             PHASE1A(ldrBal, tx, H); break;
53         case Propose[tx, S_tx, H]
54             ldrBal ← 0, S_p ← ∅;
55             foreach s ∈ S_tx do
56                 S_p ← S_p ∪ (vt_a, −1);
57             PHASE2A(tx, S_p, H); break;
58         case Phase1b[m, m_a, vt, pos]
59             S_msg1 ← S_msg1 ∪ {msg};
60             if timeout ∨ |S_msg1| == |H| then
61                 S_p ← ∅;
62                 if MAJORITY(S_msg1, S_tx, S_p) then
63                     PHASE2A(tx, S_p, H);
64                 else
65                     RANDOMBACKOFF(tx, ldrBal, H);
66             break;
67         case Phase2b[m, vt, pos, rval]
68             S_msg2 ← S_msg2 ∪ {msg}, vt_final ← commit;
69             if timeout ∨ |S_msg2| == |H| then
70                 foreach s ∈ S_tx do
71                     S^R_msg ← {msg_r | msg_r ∈ S_msg ∧ msg_r sent by
                       h_r ∧ r ∈ R_s};
72                     if |S^R_msg| < Q then
73                         RANDOMBACKOFF(tx, ldrBal, H);
74                         return
75                     if |{msg | vt_msg = vt_rl, ∀msg ∈ S^R_msg}| ≥ Q̄ then
76                         restart ← true;
77                     else if vt_msg == vt_rc, ∃msg ∈ S^R_msg then
78                         vt_final ← abort;
79                 if vt_final == abort then restart ← false;
80                 else if restart then vt_final ← abort;
81                 PHASE3(tx, vt_final, H, S_rval);
82             break;
83 Procedure PHASE1A(m, tx, S_A)                                  // leader
84     send Phase1a[m, tx] to all h ∈ S_A;
85 Procedure PHASE2A(tx, S_p, S_A)                                // leader
86     if S_p == ∅ then vt_final ← vt_r;
87     else
88         vt_final ← vt_a;
89         foreach (vt_s, pos_s) ∈ S_p do
90             if vt_s == vt_rc then vt_final ← vt_r;
91     send Phase2a[ldrBal, tx, vt_final] to S_A;
92 Procedure PHASE3(tx, vt, S_A, S_d)                             // leader
93     if ups == ∅ then                                         // read-only
94         send S_d to client;
95         return
96     if vt == commit then
97         vt_final ← commit[tx, ups];
           /* process tx over S_d; vt_final can change                */
98     else
99         vt_final ← abort[tx];
100        if restart == true then
               /* l restarts tx in rand() time                        */
101    DecisionInstance_Paxos(txn, ldrBal, vt_final, restart);
102    send vt_final to S_A and client;
```

Algorithm 3: Pseudocode for SHAFT - Key Steps

```
106 Procedure MAJORITY(S_msg, S_tx, S_p)
107     foreach s ∈ S_tx do
108         S^R_msg ← {msg_r | msg_r ∈ S_msg ∧ msg_r by h_r ∧ r ∈ R_s} ;
109         if |S^R_msg| < Q then return false; // not enough replies
110         m_a^max ← −1;
111         foreach msg ∈ S^R_msg do
112             if m_a ≠ null ∧ m_a^max < m_a then
113                 m_a^max ← m_a;
114                 (vt_s, pos_s) ← (vt_msg, pos_msg);
115         if m_a^max < 0 then
116             (vt_s, pos_s) ← the (vote,position) pair by the most
                  messages in S^R_msg ;
117             count_s ← number of messages with (vt_s, pos_s);
118             if count_s < Q then
119                 S_p ← ∅;
120                 break;
121         S_p ← S_p ∪ (vt_s, pos_s);
122     return true;
123 Procedure RANDOMBACKOFF(tx, bal, S_A)
124     bal ← bal + rand();
125     sleep(rand());
126     PHASE1A(bal, tx, S_A);
```

same shard s_{min} as their smallest shard. For example, transaction T_1 accesses D, X, Y and T_2 accesses D, W, Z. D is the smallest shard among all shards of T_1 and T_2 respectively. Thus, the system node with the least ID among all nodes hosting replicas of D is the leader for the processing instances of both T_1 and T_2. When a leader simultaneously receives *propose* messages for multiple transactions from multiple clients, the leader merges the processing instances of these transactions and processes them in one processing instance and one decision instance, based on the following rules.

The leader merges shards of the concurrent transactions $T = T_1, ..., T_i$ into S_T. Now, a shard $s_r \in S_T$ is read-only if no transaction in T writes s_r. Except for the read-only shards, the lock on any other shard in S_T is held till the end of the processing instance. For acceptors, operations from all transactions of T are now treated as if they were from a single transaction.

However, the leader always processes each transaction separately. That is, if a shard is involved in multiple transactions, the leader processes the message sent by each replica of the shard once for each transaction. An *abort* decision is sent to a shard if only all transactions accessing the shard abort. The *restart* is separately marked for each transaction. When the leader executes transactions of T together in phase 3, it guarantees serializability in the execution. Writes on the same record by multiple transactions of T can be merged into one before inserting to the write set *ups*. In the basic version of SHAFT, a vector $(txn, ldrBal, vt_{final}, restart)$ is recorded for each transaction decision (Algorithm line 101). Now, a matrix is needed. Each row of the matrix is the vector for a transaction. The decision is translated into multiple decisions for T's transactions respectively. Each transaction's decision is sent back to its submitting client.

4.3 Data Granularity and Distribution

We assume a shard as the unit for transaction operations. A virtual log is correlated with each shard. To further improve concurrency and availability, we can reduce the data granularity by reducing the size of data partitions. However, the number of virtual logs to be maintained by a system node increases as the data granularity decreases. When the data granule is record-level, a node will have to maintain an astonishing number of virtual logs. This is obviously too costly. Besides, the reduction of data granularity also reduces the possibility of merging instances (Section 4.2). Anyhow, there is a trade-off here to be considered.

Customized replication and distribution can significantly improve bandwidth usage and user experience [21]. SHAFT allows the flex-

ible replication, distribution and partition of data globally. If the majority of shard replicas accessed by a transaction locate in the same datacenter, SHAFT can reduce the processing time by setting an adequate *timeout* duration (Algorithm lines 60 and 69). As long as messages from a majority are received, SHAFT can progress and decide. The lagging replicas in other datacenters will finally learn the progress and the decision later.

5. EVALUATION

We evaluate SHAFT and compare it to the most related solution MDCC by using a simulator for large-scale distributed protocols and based on extensive simulations. We choose MDCC because it outperforms other contemporary transactional replication protocols and becomes widely known recently. Both SHAFT and MDCC are based on the Paxos algorithm. Although the pseudocode of many Paxos-based protocols has limited lines of codes, the actual implementation usually demands non-trivial engineering efforts [16]. Besides, how well these protocols are implemented by codes greatly affects the actual system performance. Therefore, we choose the simulation method for evaluation. In the following, we first elaborate the simulation settings that enable the fair comparison. Then we proceed to present various evaluation results based on the simulations.

5.1 The Simulation Settings

In the simulations, we exploit PeerSim [30], a well-known network simulator capable of verifying the correctness and comparing the performance of network protocols. Distributed concurrency control protocols like SHAFT and MDCC are one type of network protocols. This fact validates our choice and exploitation of PeerSim in the evaluation.

We take the event-based mode of PeerSim. In each experiment, PeerSim keeps a virtual clock that ticks by cycles. The throughput and the message delay is computed against the clock time. Each message delivery is considered an event. Each operation processing takes a unit of virtual clock time, i.e. cycle. Each experiment will continue for a given length of virtual clock time. Our implementations are based on a previous Paxos simulator implentation[2].

In simulations with PeerSim, we can implement not only the targeted protocol, but also many aspects of conditions under test. For example, we can specify in the *control* implementations about 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 can simulate systems with multi-datacenters, and observe system behaviors under different workloads. Furthermore, PeerSim also supports simulation on node failures and communication channel breaks. Thus, we can also watch how a system is influenced by different failures. In fact, the simulator can truly simulate the actual situation if given precise specification.

SHAFT and MDCC [22] are both implemented in the simulator. To enable a fair comparison, we comply tightly with and borrow the core logic of the open-source implementation of MDCC[3]. In each unit of virtual clock time, we let the simulator run both SHAFT and MDCC over the same simulated network of nodes. The same workloads and the same failure conditions are applied to both SHAFT and MDCC. However, the two protocols run independently, and the results are collected separately. Our implementations and the source codes for the above mentioned simulators, together with the simulation results, can be downloaded[4].

[2]A fault-tolerant FSM. http://peersim.sourceforge.net/#code
[3]https://github.com/hiranya911/mdcc
[4]http://prof.ict.ac.cn/∼yuqing/SHAFT

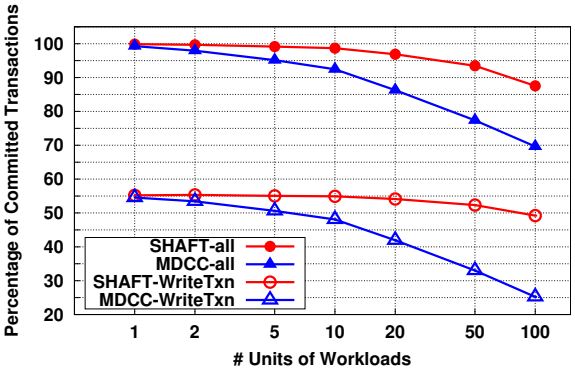

Figure 3: The percentage of committed transactions as workloads increase. SHAFT outperforms MDCC as workloads increase.

In the evaluation, we simulate 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 communication, which we set to one virtual unit of time in the experiments. We set the number of nodes in each datacenter to 50. The total number of unique shards is 1000, which must be multiplied by the replication factor to get the total number of shards. 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.

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%. Note that, setting 20% cross-shard transactions is enough to prove that SHAFT can deal with intra- & cross-shard transactions. The percentage only affects performance. We choose the percentage based on a general understanding of the cloud transaction pattern and the 20/80 rule. 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 can be different. Each operation processing, e.g. read or write, takes as long as sending an intra-datacenter message.

We carry out three groups of experiments. We will further explain the detailed configurations of each experiment. The metrics that we focus on are mainly the throughput rate in percentage and the response time in virtual time units.

5.2 Workload Influence

To study the concurrency allowed by SHAFT, we test SHAFT and MDCC under different workloads. We set the workloads to be 1 unit to 100 units of workloads. One workload unit is one transaction per hundred units of virtual clock time, and 100 workload units means one transaction per unit of virtual clock time. We randomly choose among all nodes to initiate a transaction.

According to the protocols of SHAFT and MDCC, the two protocols should have similar performance when the workload is low. As the workload increases, SHAFT can outperform MDCC. The reason is that SHAFT is a pessimistic concurrency control protocol and MDCC is an optimistic. High concurrency can lead to

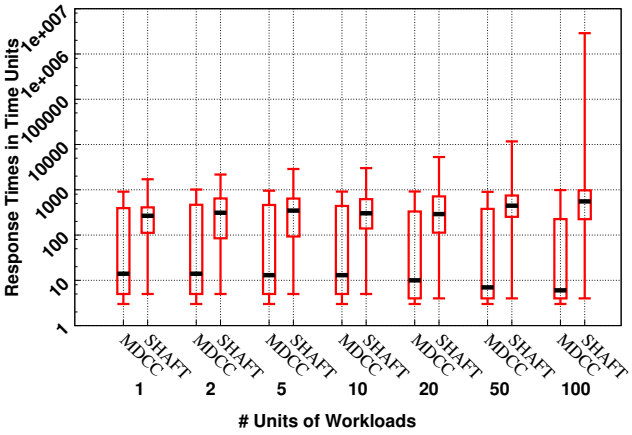

Figure 4: Response times as workloads increase. 75% transaction response times of SHAFT are comparable to those of MDCC, although SHAFT has a longer tail in response times.

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 3 shows the throughputs of all committed transactions and committed write transactions for SHAFT and 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

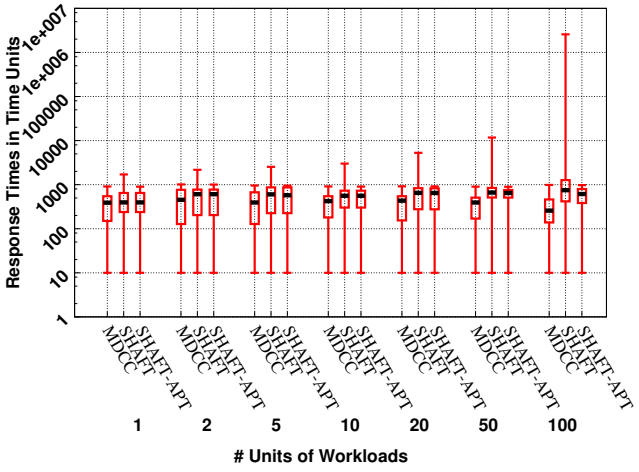

Figure 5: Response times of write transactions as workloads increase. The actual processing times of committed write transactions using SHAFT (SHAFT-APT) is very close to those of MDCC. The long tail of response times when using SHAFT is due to the waiting of locks.

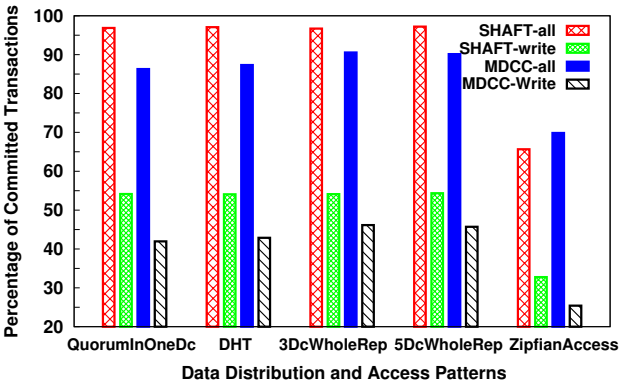

Figure 6: The percentage of committed transactions on different data distribution and access patterns. (a) SHAFT outperforms MDCC in throughput in all scenarios except for ZipfianAccess; (b) SHAFT has a higher throughput of committed write transactions (SHAFT-wrtie vs. MDCC-write) under ZipfianAccess; (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 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 copies.

transactions, thus its throughput of committed read-only transactions is affected by the increasing workloads.

Figure 4 shows the boxplots of the responses times of committed transactions for SHAFT and MDCC. A boxplot illustrates the min, the 25%, the median, the 75%, and the max response times in an experiment. 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 5, 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.

5.3 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 6 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

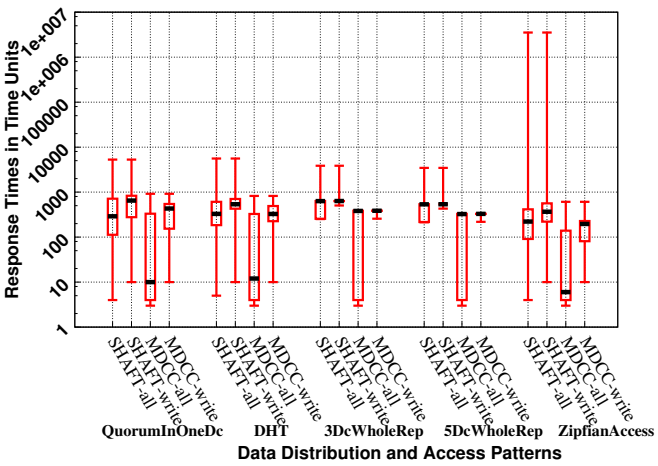

Figure 7: Response times on different data distribution and access patterns. (a) Write transactions take longer time to response than read-only transactions (-All vs. -Write); (b) Zipfian access pattern (ZipfianAccess) has a great impact on response times of SHAFT; (c) Whether a quorum of replicas is 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).

the throughput of SHAFT, but MDCC has higher throughputs of committed write transactions when datacenters have complete data copies. The reason for MDCC's higher throughputs on complete data copies in datacenters is smaller abort rates and fewer competing transactions. This can be related to the workload generator of the simulations. In such scenarios, the workload generator tends to generate fewer transactions with intersecting data sets.

Figure 7 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 greater impacts on response times than the number of copies, when placing complete copies in datacenters.

5.4 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 simultaneous. 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 virtual clock 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 Paxos algorithm.

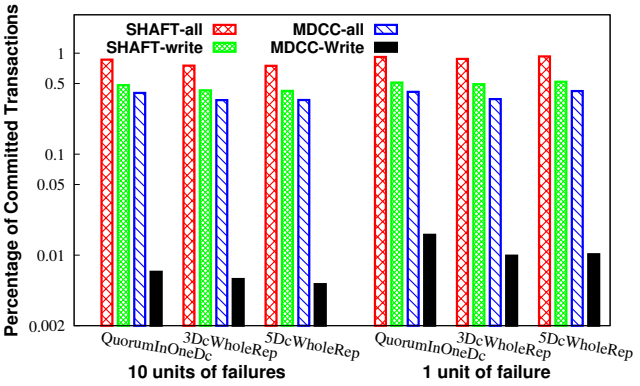

Figure 8: The percentage of committed transactions on failures. Failures have little influence on the throughputs of SHAFT, but have great impacts on MDCC's throughput of committed write transactions.

Figure 8 shows the throughputs of committed transactions on failures. The *y* axis of Figure 8 is in log scale and the values are multiplied by 500 before applying logarithm. Observing the result, we can see that failures have little influence on the throughputs of SHAFT, but have great impacts on MDCC's throughput 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 9 shows the transaction response times on failures. Comparing to the response times on 10 workload units in Figure 3, 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

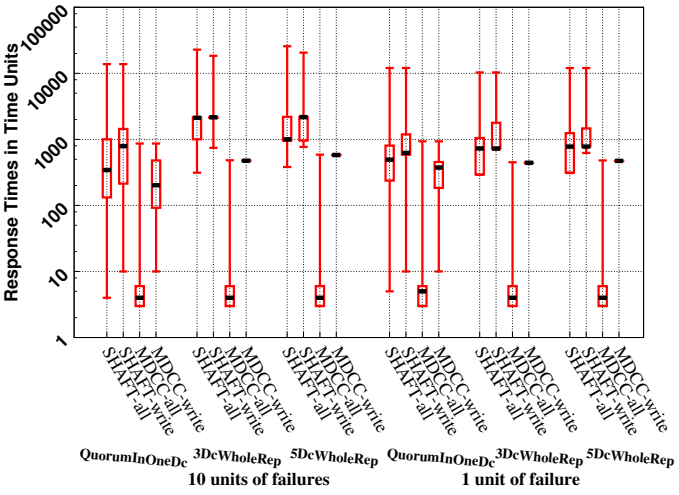

Figure 9: Response times on failures. The response times of SHAFT increase on failures as compared to the normal case, while the response times of MDCC's committed transactions remain stable.

number of copies does not have much influence on transaction throughput and response times, even on failures.

5.5 Application Indications for SHAFT

First, restart transactions only when the load is certain to be very low. SHAFT has the mechanism to automatically restart a transaction that fails to lock the necessary data shards. This mechanism should be used carefully. In our experiments, restart does not lead to improved throughputs. In fact, restarted transactions can exacerbate the contention of locks, the bandwidth consumption, and the processing workloads. The throughput might thus be reduced and the response latency is prolonged. If the transaction workload is inevitably high, we can reduce the transaction consistency level such that transactions are processed without serializability guarantees. We have succeeded in finding a new series of consistency levels suitable for transactions with fault-tolerance and high availability demands. We leave the report of this result to a future paper.

Second, do not use transactions extensively in distributed environments. As transaction workloads increase, the percentage of committed transactions will be reduced and the response times are prolonged. Whenever possible, a lower consistency level should be considered for applications. In SHAFT, we treat all operations on replicated data as transactions. Though strong consistency level is provided, the performance is harmed. If all reads are processed in non-transactional semantics, as in MDCC, the contention can be reduced and the performance improved.

Third, consider the data access pattern of applications and scatter *hot data* to different nodes, when partitioning and distributing data in the system. As reflected by the experiments, the concentrated access of data, e.g. Zipfian access pattern, has a great impact on the performance of transaction processing. Furthermore, complete copies in all datacenters will lead to monotonous response times. This fact can be exploited whenever necessary. Putting a majority of replicas in a datacenter closest to clients can improve performance only if the timeout is set to an appropriate value, the setting of which might need actual engineering experiences.

Finally, failures can have occurred if the distribution of transaction response times moves up to larger values and the workload has been kept stable. Although such failures do not affect the transaction processing of SHAFT as long as the failure number is under a certain level, removing failures from the system can improve the performance of transaction processing.

6. RELATED WORK

Transaction proposals for the cloud can be divided into three categories. The first category considers data partition but no replication, e.g., G-store [18]. It provides on-demand transactional access over partitioned data through group communication protocol. However, replication is left to the underlying storage's consideration, i.e., not considered.

The second category considers a whole replication of database, but no partition, e.g., Walter [34]. It supports parallel snapshot isolation (PSI), which precludes write-write conflicts of concurrent transactions by timestamps at different sites, each of which is a complete copy of the whole database. Time coordination is known to be a imprecise between sites, thus remains a problem to the PSI implementation. Megastore [10] only provides serial transaction support within each entity group. It exploits Paxos protocol for fault-tolerance. A leader is selected among a group of replicas for high availability, such that the system can recover automatically from leader failures. Paxos-CP [31] is another improvement of Megastore. Serializable schedules of transactions are supported through combination and promotion enhancements. But it mainly deals with transactions across datacenters, assuming one complete replica and one transaction service per datacenter.

The third category considers both replication and partition. Spanner [17] extends Megastore. While two phase locking and Paxos protocol 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 [36] implements a middle layer for replication and transaction scheduling functions. Its transaction scheduling is serializable and deterministic over strongly consistent replicas. Distributed transactions are composed of local transactions. Locking is exploited. Eiger [28] 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. Based on the causal replica consistency, non-blocking algorithms for both read-only and write-only transactions are proposed. These algorithms are variant of the Paxos algorithm.

MDCC [22] is the closest counterpart of SHAFT. It supports the isolation level of read-committed without lost updates. MDCC is based on the Paxos algorithm. Multiple individual Paxos instances are involved in each cross-shard transaction. The transaction initiator collects results from each Paxos instance to decide the final commit/abort decision for the transaction. Note that, MDCC does not allow the transaction to be aborted actively once started, unless some Paxos instance votes to abort. In MDCC, the abort of a transaction means all processing work of the transaction is wasted. On a restart, the transaction must read again all data. If the consistent reads are required, the abort rates of MDCC become stunning.

Replicated commit [29] is the most recent related work on cloud transactions. It separates replication from distributed transaction commit support. It layers the replication layer over the transaction commit layer. Although the involved cross-datacenter communications are reduced, the possibility of different transaction decisions by different datacenters for the same transaction is high such that there can be a low throughput problem.

There is also a recent work [9] 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, SHAFT can guarantee serializability consistency for transactions even on failures, as long as the number of failures is smaller than the tolerable failures of Paxos algorithm. Different from other Paxos-based protocols that are only for atomic commitment, SHAFT is a complete concurrency control protocol in distributed environment.

7. CONCLUSION AND FUTURE WORK

Targeting at providing serializable transaction processing over partitioned, distributed and replicated data, we propose in the paper SHAFT, a Paxos-based pessimistic concurrency control protocol. SHAFT has a distributed strict two-phase locking procedure. Thus, SHAFT 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. As SHAFT mainly guarantees strong transaction consistency level, users can sought for other existing methods to provide weaker consistency degrees or isolation levels when such is needed. However, SHAFT is as non-blocking a protocol as others guaranteeing weaker consistency levels.

We also propose a simulation-based evaluation framework. Our usage of a comprehensive simulator in our evaluation with SHAFT guarantees a fair comparison towards other counterparts. The framework exempts protocol developers from engineering details but focusing on the key features. Implementations under the framework can also help to guide the real implementation of the protocols. Furthermore, the analysis can guide applications in their choices of protocols. Experiments under the framework show that, while guaranteeing the strongest consistency level, SHAFT has a comparable performance to the recent related work MDCC, which outperforms other synchronous transactional replication protocols, e.g. Megastore.

The transaction processing procedure of SHAFT integrates replication with concurrency control. It treats each operation over replicated data as a transaction. Such integration makes a complete alignment of consistency levels possible. We are expecting that SHAFT, together with the related work mentioned in the paper, will stimulate new definitions of transaction consistency levels to emerge in the near future.

8. REFERENCES

[1] Cassandra website. http://cassandra.apache.org/.

[2] Hbase website. http://hbase.apache.org/.

[3] Mongodb website. http://www.mongodb.org/.

[4] 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.

[5] 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.

[6] 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/.

[7] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *Distributed Computing*, pages 108–122. Springer, 2001.

[8] P. Bailis and A. Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63, May 2013.

[9] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772. ACM, 2013.

[10] 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*, pages 223–234, 2011.

[11] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

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

[13] 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 conference on Networked systems design and implementation*, pages 11–11. USENIX Association, 2011.

[14] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–. ACM, 2000.

[15] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

[16] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live-an engineering perspective (2006 invited talk). In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing-PODC*, volume 7, 2007.

[17] 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*, page 1, 2012.

[18] 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*, pages 163–174. ACM, 2010.

[19] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[20] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, Mar. 2006.

[21] 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*, 4(11), 2011.

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

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[24] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[25] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[26] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 312–313. ACM, 2009.

[27] L. Lamport and M. Massa. Cheap paxos. In *Dependable Systems and Networks, 2004 International Conference on*, pages 307–314. IEEE, 2004.

[28] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of USENIX NSDI*, nsdi'13, pages 313–328, 2013.

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

[30] A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, sep 2009.

[31] 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*, 5(11):1459–1470, 2012.

[32] D. R. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *Proceedings of the VLDB Endowment*, 5(12):1850–1861, 2012.

[33] 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*, pages 292–312, 1992.

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

[35] M. Tamer èOzsu and P. Valduriez. *Principles of distributed database systems*. Springer, 2011.

[36] 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*, pages 1–12, 2012.