Low-Latency Multi-Datacenter Databases using Replicated Commit

Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, Amr El Abbadi
University of California
Santa Barbara, CA, USA
{hatem,nawab,pucher,agrawal,amr}@cs.ucsb.edu

ABSTRACT

Web service providers have been using NoSQL datastores to provide scalability and availability for globally distributed data at the cost of sacrificing transactional guarantees. Recently, major web service providers like Google have moved towards building storage systems that provide ACID transactional guarantees for globally distributed data. For example, the newly published system, Spanner, uses Two-Phase Commit and Two-Phase Locking to provide atomicity and isolation for globally distributed data, running on top of Paxos to provide fault-tolerant log replication. We show in this paper that it is possible to provide the same ACID transactional guarantees for multi-datacenter databases with fewer cross-datacenter communication trips, compared to replicated logging. Instead of replicating the transactional log, we replicate the commit operation itself, by running Two-Phase Commit multiple times in different datacenters and using Paxos to reach consensus among datacenters as to whether the transaction should commit. Doing so not only replaces several inter-datacenter communication trips with intra-datacenter communication trips, but also allows us to integrate atomic commitment and isolation protocols with consistent replication protocols to further reduce the number of cross-datacenter communication trips needed for consistent replication; for example, by eliminating the need for an election phase in Paxos. We analyze our approach in terms of communication trips to compare it against the log replication approach, then we conduct an extensive experimental study to compare the performance and scalability of both approaches under various multi-datacenter setups.

1. INTRODUCTION

The rapid increase in the amount of data that is handled by web services as well as the globally-distributed client base of those web services have driven many web service providers towards adopting NoSQL datastores that do not provide transactional guarantees but provide more scalability and availability via transparent sharding and replication of large amounts of data. For example, systems like Google's Bigtable [10], Apache Cassandra [24], and Amazon's Dynamo [15] do not guarantee isolation or atomicity for multi-row transactional updates. Other systems like Google's Megastore [3], Microsoft's SQL Azure [8], and Oracle's NoSQL Database [31] provide these guarantees to transactions whose data accesses are confined to subsets of the database (e.g., a single shard). Recently, however, major web service providers have moved towards building storage systems that provide unrestricted ACID transactional guarantees. Google's Spanner [13] is a prominent example of this new trend. Spanner uses Two-Phase Commit and Two-Phase Locking to provide atomicity and isolation, running on top of a Paxos-replicated log to provide fault-tolerant synchronous replication across datacenters. The same architecture is also used in Scatter [17], a distributed hash-table datastore that provides ACID transactional guarantees for sharded, globally replicated data, through a key-value interface. Such layered architecture, in which the protocols that guarantee transactional atomicity and isolation are separated from the protocol that guarantees fault-tolerant replication, has many advantages from an engineering perspective, such as modularity, and clarity of semantics.

We show in this paper that it is possible to provide the same strong ACID transactional guarantees for cross-datacenter databases but with fewer cross-datacenter communication trips, compared to a system that uses log replication, such as Spanner, by using a more efficient architecture. We still use a layered architecture that separates transactional atomicity and isolation from fault tolerant replication; however, instead of running Two-Phase Commit and Two-Phase Locking on top of Paxos to replicate the transactional log, we run Paxos on top of Two-Phase Commit and Two-Phase Locking to replicate the commit operation itself. That is, we execute Two-Phase Commit operation multiple times, once per datacenter, with each datacenter executing Two-Phase Commit and Two-Phase Locking internally, and only use Paxos to reach consensus among datacenters about the fate of the transaction for the commit and abort decision. We refer to this approach as Replicated Commit, in contrast to the replicated log approach.

Replicated Commit has the advantage of replacing several inter-datacenter communication trips with intra-datacenter communication, when implementing ACID transactions on top of globally-replicated data. Moreover, replicating the Two-Phase Commit operation rather than replicating log entries allows us to integrate the atomic commitment and isolation protocols with the consistent replication protocol in a manner that further reduces the number of cross-datacenter communication trips needed for consistent replication; for example, by eliminating the need for an election phase in Paxos. Reducing the number of cross-datacenter communication trips is crucial in order to reduce transaction response time as perceived by the user. Studies performed by different web service providers [19] already demonstrate that even small increases in latency result in significant losses for service providers; for example, Google observes that an extra 0.5 seconds in search page generation

Proceedings of the VLDB Endowment, Vol. 6, No. 9

time causes traffic to drop by 20%, while Amazon reports that every 100ms increase in latency results in 1% loss in sales [19]. When replicating a database across multiple datacenters in different continents, each cross-datacenter communication trip consumes tens or even hundreds of milliseconds, depending on the locations of the datacenters. In fact, as revealed by our experiments on the Amazon EC2 platform, cross-datacenter communication over the Internet typically requires much more time than the theoretical lower bound on packet transmission time (i.e., the speed of light). For example, a packet sent from the East Coast to the West Coast takes about 45ms, which is nearly three times the time it takes a light pulse to travel that same distance.

By reducing the number of cross-datacenter communication trips, Replicated Commit not only reduces the response times of individual transactions as perceived by the users of the database, but also significantly reduces the amount of time a transaction holds exclusive locks on data items. Thus, if the database serves a workload that is skewed towards certain *hot* data items, Replicated Commit reduces lock contention, and thus avoids thrashing. Since skewed data access is fairly common in practice, reducing lock contention is expected to result in significant performance improvements.

We summarize our contributions in this paper as follows.

- We propose an architecture for multi-datacenter databases, namely Replicated Commit, that is designed to reduce cross-datacenter communication trips by replicating the Two-Phase Commit operation among datacenters, and by using Paxos to reach consensus on the commit decision.
- We compare Replicated Commit against the replicated log architecture, that is currently used in production systems such as Google's Spanner, in order to analyze Replicated Commit's savings in terms of cross-datacenter communication trips.
- We conduct an extensive experimental study to evaluate the performance and scalability of Replicated Commit, compared to the replicated log approach, under various multi-datacenter setups.

The rest of this paper is organized as follows. Section 2 presents a motivating example that demonstrates the number of cross-datacenter communication trips required to perform Two-Phase Commit in a typical replicated log system. In Section 3 we propose our new architecture, namely Replicated Commit. In Section 4 we compare our architecture against the replicated log architecture to assess the reduction in the number of cross-datacenter communication trips analytically, then in Section 5 we conduct an extensive experimental study to compare the performance and scalability of both approaches, Replicated Commit and replicated logs, under various multi-datacenter setups. Finally, we discuss related work in Section 6, and conclude in Section 7.

2. MOTIVATING EXAMPLE

In this section, we present a motivating example that demonstrates the overhead of running distributed transactions on top of globally replicated data, while guaranteeing strong consistency using log replication. In a replicated log system that runs Two-Phase Commit and Two-Phase Locking on top of Paxos, as is the case with Spanner and Scatter, typically the client program executing the transaction begins by reading from the database, and acquiring shared locks, while buffering all updates locally; then after all reading and processing is done, the client submits all updates to the database in Two-Phase Commit. Consider the case when a transaction updates three data items X, Y, and Z in three different shards

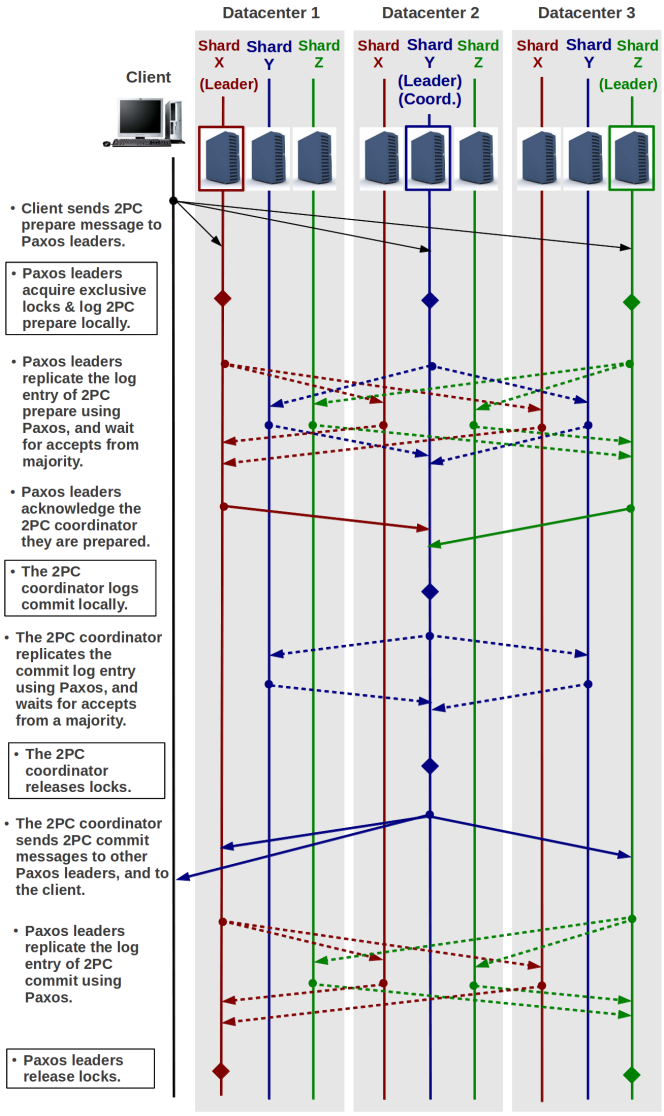


Figure 1: Typical sequence of messages and operations when running Two-Phase Commit on top of a Paxos-replicated log.

of the database. Figure 1 shows the messages exchanged during Two-Phase Commit on a system where logs are replicated across datacenters using Paxos. We use solid lines to illustrate Two-Phase Commit communication, and use dashed lines to illustrate Paxos communication. The setup consists of three datacenters. Each datacenter contains three data servers, where each data server holds a replica of a shard of the database. We label the shards X, Y, and Z. The transactional log of each shard is replicated across datacenters using a variant of Paxos called multi-Paxos, which is the Paxos variant that is used by most production systems that implement Paxos-replicated logs [9, 13]. In multi-Paxos, once a node is elected as a leader during a Paxos instance, that node remains the leader for subsequent Paxos instances until the node crashes; this avoids the cost of re-electing a leader each time the system logs an entry. To account for the case when the Paxos leader crashes, re-election of the Paxos leader in multi-Paxos is done periodically, rather than at each Paxos instance. In our example we consider the general case when Paxos leaders are in different datacenters.

The special case when all Paxos leaders are in the same datacenter saves only one cross-datacenter communication trip, as we show when counting communication trips.

The sequence of operations and messages shown in Figure 1 can be summarized as follows: (i) The client picks the Paxos leader of one of the shards involved in the transaction, say the shard that contains Y, to be the *coordinator* of Two-Phase Commit, while the other Paxos leaders of X and Z are *cohorts*; (ii) The client sends Two-Phase Commit prepare messages to the Paxos leaders of X, Y, and Z, and informs them that Y is their coordinator. This is a **cross-datacenter one-way trip** because the Paxos leaders can be in any datacenter arbitrarily far from the client; (iii) The Paxos leaders acquire exclusive locks on the target data items, X, Y, and Z, then log the Two-Phase Commit prepare message to their Paxos logs. Paxos logging requires a **cross-datacenter roundtrip** because Paxos leaders have to forward the prepare log entry to, and receive accepts from, a majority of replicas in other datacenters; (iv) The Two-Phase Commit cohorts (i.e., the Paxos leaders of X and Z) acknowledge the Two-Phase Commit coordinator (i.e., the Paxos leader of Y) that they have logged the Two-Phase Commit prepare message successfully. If the Paxos leaders are in different datacenters, this requires a **cross-datacenter one-way trip**; (v) The Two-Phase Commit coordinator (i.e., the Paxos leader of Y) logs a Two-Phase Commit commit entry in its own Paxos log. This requires another **cross-datacenter roundtrip** because the Paxos leader of Y has to forward the commit log entry to, and receive accepts from, a majority of replicas in other datacenters; (vi) The Two-Phase Commit coordinator (i.e., the Paxos leader of Y) forwards the commit message to the cohorts (i.e., the Paxos leaders of X and Z) and to the client. This requires a **cross-datacenter one-way trip**; (vii) Once the client receives the commit message, the client deems the transaction committed. Any further communication does not affect latency as perceived by the client, but delays the release of locks; (viii) The Two-Phase Commit cohorts (i.e., the Paxos leaders of X and Z) log the commit entry in their Paxos logs. This is a **cross-datacenter roundtrip** because the Paxos leaders of Y and Z have to forward the commit log entry to, and receive accepts from, a majority of replicas in other datacenters; (ix) The Two-Phase Commit cohorts (i.e., the Paxos leaders of X and Z) release theirs locks after receiving accepts from a majority of datacenters that the commit entry has been logged successfully.

If the Paxos leaders of X, Y, and Z are in the same datacenter, the number of cross-datacenter one-way trips that take place during the period starting after the client sends the Two-Phase Commit prepare message until the client receives an acknowledgment from the Two-Phase Commit coordinator, equals six one-way trips. If the Paxos leaders of X, Y, and Z are in different datacenters, an additional cross-datacenter one-way trip is incurred. The number of cross-datacenter trips that take place during the period starting after the Paxos leaders acquire exclusive locks on X, Y, and Z, until these locks are released, equals eight one-way trips if the Paxos leaders of X, Y, and Z are in different datacenters, or seven one-way trips if the Paxos leaders of X, Y, and Z are in the same datacenter. Locking data for long periods of time affects concurrency negatively, specially when there is contention on certain data items. In summary, this example demonstrates that the design decision of running Two-Phase Commit and Two-Phase Locking on top of Paxos is not very efficient in terms of latency and concurrency. Experiments in Section 5 also confirm this observation.

3. REPLICATED COMMIT

We begin by presenting the data model and infrastructure that Replicated Commit runs on, then we explain the Replicated Commit protocol stack. A discussion and analysis of the correctness of the protocol and other special considerations is also presented.

3.1 Data Model and Infrastructure

Our implementation of Replicated Commit runs on a key-value store, however, Replicated Commit is agnostic to whether the database is relational, or is a key-value store. We target databases that are replicated across multiple databases across the globe. Typically, data is fully-replicated across datacenters for availability and fault-tolerance. Within each datacenter the database is sharded across multiple servers to achieve high scalability. All replicas of a data item are peers in the sense that there are no leader replicas. Each replica of each data item is guarded by its own lock, thus each server in each datacenter has its own lock table. By maintaining the locks in a distributed manner Replicated Commit avoids a single point of failure, as is the case in Spanner where a single lock per data-item is maintained at the Paxos leader copy of that data-item. In Replicated Commit a transaction does not have to restart as long as it maintains locks on a majority of replicas.

3.2 Transaction Execution

In Replicated Commit, as the transaction execution proceeds, read operations are processed at the replicas stored at the datacenters. However, the client program that executes the transaction buffers all updates locally while reading from the database. Once the client program is done with reading and processing, the client submits all updates to the database using Two-Phase Commit.

3.2.1 Transactional Reads

In Replicated Commit, a transactional read is performed by sending a read request to all replicas. Whenever a data server receives a request to read a data item, the data server places a shared (read) lock on that data item and sends the most recent version of the data item back to the client. We have chosen a very simple strategy to avoid deadlocks in the system in that if the lock being requested by a transaction cannot be granted due to an existing lock on the data-item then the lock request is denied. The only exception to this is that an exclusive (write) lock can take over an existing shared lock. This ensures that there is no unbounded waiting albeit at the expense of potentially unnecessary aborts. The client for every read request waits until it receives responses from a majority of replicas before reading the data item. The client reads from a majority to ensure that shared locks are placed on a majority of replicas so that a transaction that writes on one of the items in the read-set will be detected as we will show later when the Paxos accept phase is discussed. Once the client receives responses to its read request from a majority of replicas, the client uses the most recent version to process the read operation.

3.2.2 Transaction Termination

Once a transaction has finished all readings and processing, the client program that executes the transaction submits all buffered updates to the database as part of Two-Phase Commit. Each transaction starts a new Paxos instance for its own Two-Phase Commit.

Terminology. We use the standard terminology of Paxos [25, 26]; that is, each Paxos instance has proposers, acceptors, and learners. A proposer is an entity that advocates a client request, trying to convince acceptors to accept a value. Eventually, learners need to learn the value that the acceptors accepted. In the basic Paxos algorithm, each Paxos instance consists of two phases. During the first phase, acceptors vote for a leader from among all potential proposers, then in the second phase, acceptors accept the value proposed by the leader that they elected in the first phase. Learn-

ers need to learn the value that has been accepted by a majority of acceptors which varies with different Paxos implementations [25].

The Commitment Protocol. At a high level, the way Replicated Commit performs Two-Phase Commit is that each transaction executes a new Paxos instance to replicate Two-Phase Commit across datacenters. The client itself acts as the (sole) proposer for this Paxos instance. Each datacenter acts as both an acceptor and a learner. There is no need for an election phase since there is only one proposer for transaction commitment. In Replicated Commit, the value to be agreed on at the end of a Paxos instance is whether to commit a transaction or not. The default value is not to commit, so a majority of datacenters need to accept the prepare request of Two-Phase Commit in order for a transaction to commit. Algorithm 1 provides a high level description of the Replicated Commit protocol.

Algorithm 1 Replicated Commit

1: The client appoints a shard to be the coordinator at each datacenter DC.
2: The client sends Paxos accept request to the coordinator in each DC
3: **for all** DC **do**
4: The coordinator sends 2PC prepare request to all cohorts within the same DC, including the coordinator itself
5: All cohorts acquire locks and log the 2PC prepare operation
6: The coordinator waits for acknowledgments from all cohorts within the same DC that they are prepared
7: **end for**
8: Coordinators inform each others and the client that they accept the Paxos request
9: **for all** DC **do**
10: The coordinator sends 2PC commit request to all cohorts within the same DC
11: All cohorts log the 2PC commit operation and release locks
12: **end for**

Paxos Accept Phase & 2PC Prepare Phase. A datacenter accepts a Paxos request from a transaction only after (1) acquiring all the exclusive locks needed by that transaction, as specified in the request, (2) verifying that shared locks that have been acquired by that transaction are still being held by the same transaction, thus no shared locks have been released by an exclusive lock of another transaction, and (3) logging the prepare operation to the transactional log. At each datacenter, acquiring exclusive locks, checking shared locks, and logging commit requests are all done *locally* on each of the data servers that hold data items accessed by the transaction that issued the prepare request. Note that each data server at each datacenter maintains a lock table and a transactional log locally, to manage the data replica that is stored locally on the server. Furthermore, once the cohort has locked the prepare operation, shared locks held locally can be released without violating the Two-phase locking rule [36]. The three operations of acquiring exclusive locks, checking shared locks, and logging the commit request all constitute the first phase (i.e., the prepare phase) of Two-Phase Commit. Therefore the prepare phase of Two-Phase Commit can be thought of as a subroutine that is nested under the accept phase of Paxos, and for each datacenter the accept phase of Paxos has to wait for the prepare phase of Two-Phase Commit to finish before completing.

Whenever a client sends a Paxos accept request to a datacenter, piggybacked with information about acquired and required locks, the client also appoints one of the shards involved in Two-Phase Commit as the coordinator of Two-Phase Commit. The coordinator data server in each datacenter waits until all other data servers involved in Two-Phase Commit within the same datacenter respond to that coordinator indicating that the prepare operations of Two-Phase Commit has been done successfully on each of those data servers. Once the coordinator in any datacenter receives acknowledgments from all Two-Phase Commit cohorts within the same datacenter, the coordinator sends a message back to the client indicating that it has accepted the Paxos request. If a datacenter cannot perform one or more of the operation(s) needed during the prepare phase of Two-Phase Commit, the datacenter does not accept the Paxos request of the client, and acts as a faulty acceptor. Once the client receives acceptances from a majority of datacenters, the client considers the transaction committed.

2PC Commit Phase. As a result of the outcome of the Paxos accept phase, the client and all the coordinators will reach consensus with regard to the outcome of the transaction in question. When a coordinator at a datacenter learns of the consensus, it initiates the second phase of the commit. Thus, during this phase, the coordinator in each datacenter sends commit messages to other cohorts within the same datacenter. Once the cohorts receive this message, they perform the actual updates required by the transaction, log the commit operation, and release all the locks held by this transaction. These three operations of performing updates, logging the commit operation, and releasing locks constitute the second phase (i.e., the commit phase) of Two-Phase Commit.

3.3 Discussion and Correctness

Deadlocks. As discussed above, we have chosen a simple approach to avoid deadlocks in the system. In particular, we do not allow unbounded waiting for locks by a transaction to occur by denying a conflicting shared and exclusive lock requests, except that an exclusive lock request can take over an existing shared lock. Effectively, we transfer the responsibility of waiting to the transaction program itself. For example, a transaction requesting shared locks can choose to either abort the transaction when its lock request is denied at a majority of datacenters or retry the request after some randomized amount of back-off. The rationale for our approach is that in a large database conflicts will be rare and a simpler mechanism for deadlock avoidance is preferable to a complex deadlock resolution mechanism which will result in a significant increase in the system-level overhead for all transactions. Furthermore, if indeed it becomes necessary, a more elaborate deadlock resolution technique can be deployed.

Read Optimization. In the base Replicated Commit protocol, we require that each read operation waits until the transaction receives replies from a majority of replicas. This of course has the consequence that the read latency incurred by the transaction is a function of the slowest datacenter in the majority. An alternative would be to allow optimistic processing of read operations by assuming that the first response received is up-to-date [1] thus overlapping the read operation while the responses are collected in parallel. This requires a more complex execution logic in the client programs in that the application must be designed such that it can be rolled-back if the optimistic assumption of the first response being up-to-date turns out to be incorrect.

Correctness. In order to establish the correctness of Replicated Commit, we need to show that transaction executions in Replicated Commit are one-copy serializable [5]. One-copy serializability of a set of transaction executions, denoted H, is established in two steps. First, we need to ensure that the serialization graph induced by H, denoted $SG(H)$ on the physical replicas of data-items in the database does not have any cycles. However, acyclicity of $SG(H)$ alone is not enough to ensure one-copy serializability. As a second

step, we also need to establish that $SG(H)$ satisfies the *replicated data serialization graph*, referred to as $RDSG$, in that it induces a write order and read order on the logical data-items. Acyclicity of $SG(H)$ for the transaction execution in Replicated Commit follows from the fact that we use two-phase commit and two-phase locking on the copies of each data-item read or written by the transactions. Furthermore, $SG(H)$ is $RDSG$ follows from the fact that we are using majority quorum protocol for replica synchronization i.e., reading and writing replicated data (see pages 298-301 in [5]). This ensures that the execution of transactions committed in Replicated Commit are one-copy serializable.

4. REPLICATED COMMIT COMPARED TO REPLICATED LOG

In the following subsections, we analyze the number of cross-datacenter communication trips needed by Replicated Commit and compare it to systems that are based on replicated logs.

4.1 Transactional reads

Replicated Commit performs transactional reads by reading from a majority of replicas and picking the value with the highest timestamp. Although reading from a majority of replicas adds more messaging, it may impact latency marginally since the requests are processed in parallel. In the case of replicated logs, for each data item there is one replica that acts as a long living Paxos leader of that data item and maintains shared and exclusive locks on the data item, thus the client needs to read from only that Paxos leader in order to hold a shared lock on the data item. Although reading from the Paxos leader only results in a single read request message, the Paxos leader of each data item may be in any datacenter that is arbitrarily far from the client. For example, in Spanner [13], which is a system that uses long-living Paxos leaders, Paxos leaders are distributed arbitrarily among datacenters. Thus, many data reads end up being answered from remote datacenters. This is particularly the case when each transaction reads multiple data items, then those read requests end up directed to multiple datacenters, similar to Replicated Commit.

Reading a majority of replicas, as the case with Replicated Commit, has advantages related to fault-tolerance. In the case of replicated logging systems, the lock table of any shard is maintained at a single node, that is the Paxos leader. Thus whenever a Paxos leader crashes, all clients that are trying to access the lock table at that Paxos leader need to wait until a new Paxos leader gets elected, which may take multiple seconds. For example, the time between two consecutive elections in Spanner is 10 seconds [13]; all clients that try to access the lock table at a failed Paxos leader have to wait for the next elections to take place first. Replicated Commit does not have the same issue; that is, as long as a majority of replicas at different datacenters are up and running, transactions can continue reading data without interruption.

4.2 Two-Phase Commit

Figure 2 shows how Replicated Commit works for the same example illustrated in Figure 1. That is, given three datacenters, with three data servers in each datacenter. Each data server holds a shard of the database, and shards are labeled X, Y, and Z. Solid lines indicate Two-Phase Commit communication, while dashed lines indicate Paxos communication.

The following sequence of messages takes place in Replicated Commit: (i) The client picks a shard, say the shard of Y, as the Two-Phase Commit coordinator; (ii) The client sends a Paxos accept request to the coordinator in each datacenter. This requires a

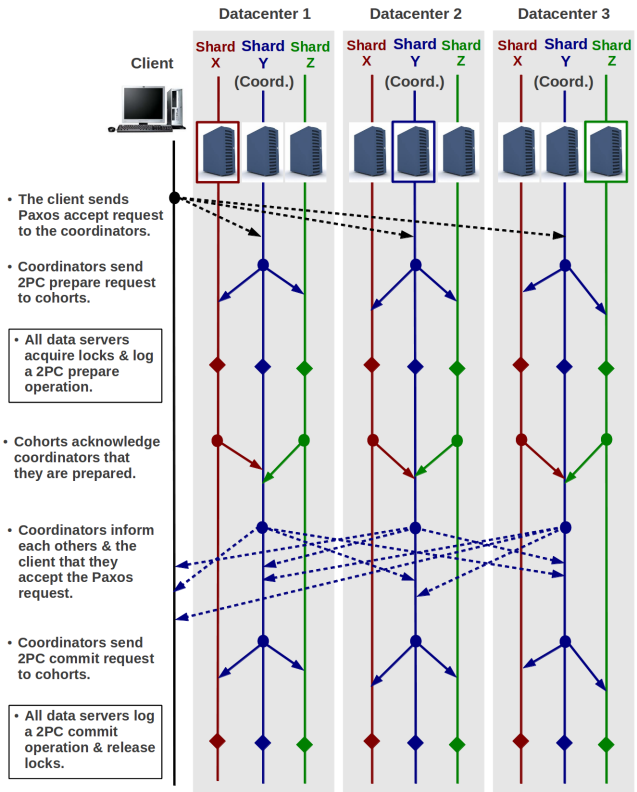


Figure 2: Operations and messages involved in Two-Phase Commit operations when using Replicated Commit.

cross-datacenter one-way trip; (iii) The coordinator in each datacenter sends a Two-Phase Commit prepare message to all Two-Phase Commit cohorts in the same datacenter, and to itself, e.g., the cohorts are the servers that host X, and Z. The coordinator and the cohorts acquire locks, and log the Two-Phase Commit prepare message in their local logs, then respond back to their coordinators; (iv) After receiving from all Two-Phase Commit cohorts in the same datacenter, the coordinator in each datacenter responds back to the client, confirming that it has finished the prepare phase of Two-Phase Commit, and thus has accepted the Paxos request. Coordinators also send that message to each other so that all datacenters learn about this acceptance. This phase requires a **cross-datacenter one-way trip**; (V) Once the client receives responses from a majority of datacenters, the client deems the transaction committed. Any further communication does not affect latency as perceived by the user; (Vi) In each datacenter, the coordinator waits until it learns that a majority of datacenters have accepted the commit request, then the coordinator commits its own datacenter by sending commit messages to all Two-Phase Commit cohorts within its datacenter, including the coordinator itself. All cohorts log the commit message, then release their locks.

Here we compare the number of cross-datacenter communication trips incurred by Replicated Commit during Two-Phase Commit against those required by replicated logs. The number of cross-datacenter communication trips that take place starting after the client sends the accept request to datacenters until the client receives a commit acknowledgment equals two one-way communication trips. The number of cross-datacenter communication trips that take place starting after data servers acquire exclusive locks until these locks are released equals only one-way communica-

tion trip. Thus Replicated Commit eliminates five cross-datacenter communication trips from total response time, compared to replicated logging when Paxos leaders are in different datacenters, or four communication trips when Paxos leaders are in the same datacenter. Moreover, Replicated Commit eliminates seven communication trips from total locking time, compared to replicated logging when Paxos leaders are in different datacenters, or six communication trips when Paxos leaders are in the same datacenter. Taking Spanner as an example of a production system that uses Paxos-replicated logs, the experimental setup of Spanner states that Paxos leaders are randomly scattered over zones, so it is more realistic not to assume that the leaders are in the same datacenter. Moreover, the amount of time a lock is retained by a given transaction affects the performance of other transactions, specially when there is contention over some data items.

4.3 Latency Analysis

Replicated Commit and Replicated Log use different read and commit strategies. A trade-off between read latency and commit latency can be inferred by examining the overall behavior of those strategies. Here, a formulation to calculate the time required to commit a given number of transactions is presented for both protocols. The analysis corresponds to the sum of the transactions latencies, where the transaction latency is the time from the beginning of the transaction until it receives a commit decision. We denote the transaction latency by T_t. T_t is equal to the sum of the latency required to serve all reads plus the commit latency, hence $T_t = \sum^{N_r}(T_r) + T_c$, where N_r is the number of read operations, T_r is the latency of one read, and T_c is the commit latency.

Using the aforementioned values, the duration of completing N_t transactions by a client is $T_e = \sum^{N_t} T_t$. To simplify the presentation, we take the average of the read and commit latencies so that T_e is equal to $(T_r N_r + T_c)N_t$. To differentiate between Replicated Commit and Replicated Log, any value corresponding to Replicated Log will have a superscript rl and rc for Replicated Commit, *e.g.*, T_r^{rl} for Replicated Log. A fair comparison between the two protocols will be for identical values of N_r and N_t. Given N_r and N_t we would like to find which protocol performs better. Consider subtracting the duration of Replicated Commit from the duration of Replicated Log, $T_e^{rc} - T_e^{rl}$. If the value is positive it means Replicated Log performs better and vice versa. Substituting from the aforementioned formulas it turns out that there is a unique value of N_r so that Replicated Commit and Replicated Log have identical performance. This value, denote it as $\bar{N}_r$, is equal to $\frac{-(T_c^{rl} - T_c^{rc})}{T_r^{rl} - T_r^{rc}}$. It follows that if N_r is greater than $\bar{N}_r$ then Replicated Log is better, otherwise Replicated Commit is better. This shows that the number of read operations decides which protocol is better and that Replicated Commit performs better for scenarios with a smaller number of reads per transaction.

Consider a simple scenario of N datacenters and k shards. Assume for Replicated Log that shard leaders are uniformly distributed across datacenters, hence each datacenter contains $\frac{k}{N}$ leaders. Also, assume that communication latency between any two replicas is identical. The RTT will be denoted as RTT. The read and commit latencies of Replicated Log and Replicated Commit will be derived now to aid in getting more insight on $\bar{N}_r$. Reads for Replicated Log are either local or remote. Local reads are immediate and remote reads takes RTT time. Thus, the average read latency T_r^{rl} equals to $(1 - \frac{1}{N})RTT$. The commit latency equals 7 one-way communication messages, hence $T_c^{rl} = 3.5RTT$. The read and commit latencies of Replicated Commit are identical and equal to RTT. By plugging these values to the formula for $\bar{N}_r$, its value becomes equal to $2.5N$. Note from this that the size of the transaction that decides which protocol is better only depends on the number of datacenters in this simple scenario. Thus, if the number of average reads per transaction is less than the product $2.5N$ then Replicated Commit will perform better, otherwise as the number of reads per transaction increases to larger values Replicated Log performs better. This result is verified in the evaluation section (Figure 5).

	O	V	I	S
C	21	86	159	173
O	-	101	169	205
V	-	-	99	260
I	-	-	-	341

Table 1: RTT latencies between different datacenters.

5. EXPERIMENTS

A performance evaluation study of Replicated Commit is done in this section. In our study, we are interested in observing the behavior of Replicated Commit. For this we designed variations of size-up, scale-up, and speed-up experiments [11]. Machines in multiple Amazon EC2 datacenters are leveraged as an infrastructure for our experiments. Used machines are High-CPU Medium (c1.medium) which have two CPU cores and 1.7GB of cache memory. Machines are distributed in five datacenters: California (C), Virginia (V), Oregon (O), Ireland (I), and Singapore (S). In what follows we used the capitalized first initial of each datacenter as its label. RTT latencies are shown in Table 1. In any datacenter, three different servers run instances of Replicated Commit. Each server is responsible for an independent shard of the data. There are three shards in our systems, denoted X, Y, and Z. Each server will be named by the pair of its host datacenter and shard of data assigned to it. For example, server $I.Y$ is the server responsible for shard Y in the datacenter in Ireland. For our evaluation, YCSB [11] is leveraged to generate workloads and evaluate the system. Since YCSB was not designed to support transactions, we use an extended version [14] that supports transactions and generates transactional workloads. The extended YCSB issues transactions constituting of reads, writes, and a commit operation. Each read or write accesses a different key in the database. Thus, the extended YCSB generates workloads consisting of multi-record transactions.

The workload consists of 2500 transactions divided equally at each datacenter. An additional machine, called *Workload Generator*, in each datacenter is deployed to generate the corresponding workload. Each Workload Generator forks five traffic generating threads (*TGTs*) to issue transactions either with a target throughput or back to back. Each TGT waits to receive the commit request reply before proceeding with the next transaction; no two transactions can be active at the same time in a single TGT. Each transaction contains five operations, either reads or writes. The ratio of reads to writes is 1:1 unless mentioned otherwise. An operation operates on an item from a pool of 3000 items divided equally between the three shards, X, Y, and Z. This small number of items will enable us to observe the performance of the system under contention. We will show results demonstrating the effect of increasing and decreasing the database size. Operations are served consecutively, meaning that no two operations are served in any one time for one TGT. The target throughput default value is 50 operations per second in each datacenter. For five datacenters the throughput is 250 operations per second.

A reference implementation of a Replicated Log protocol is used

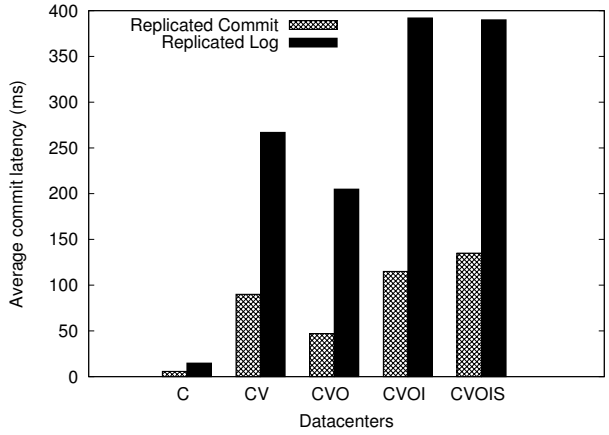


Figure 3: Transactions average commit latency for scenarios with different numbers of datacenters

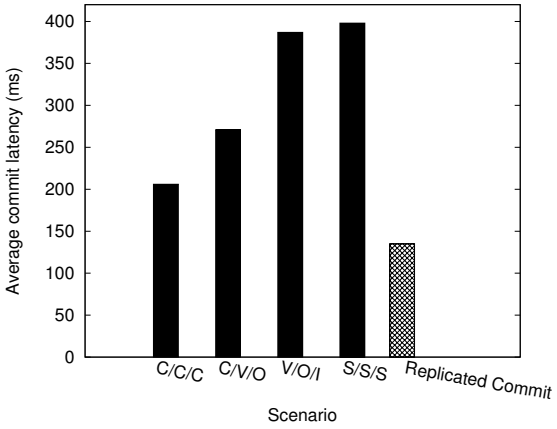


Figure 4: Average commit latency for scenarios with different replica leader placements of replicated log compared with replicated commit

to compare against Replicated Commit. We base our Replicated Log implementation on Google Spanner. We use the same setup as Replicated Commit, where five data centers have three machines each to hold three different shards. In Replicated Log one machine for each shard is assigned as a replica leader. Reads are served by the replica leader of the corresponding shard. Two-Phase commit is used to commit transactions over involved replica leaders. Transactions that are detected to be involved in a deadlock are immediately aborted. Replication from the replica leader to other replicas is done using Paxos. In Spanner, the authors used leases of 10 seconds for a replica leader. Here we assume infinite leases for a leader, hence a leader maintains its status throughout the experiment. This is done to enable us to observe the effect of the non-failure case of Replicated Log. Clients are distributed equally among datacenters in Replicated Commit experiments, since there is a coordinator in each datacenter. In Replicated Log, there are only three replica leaders in our setup. To avoid the additional cost of Replicated Log when a transaction starts at a datacenter with no replica leader, we distribute the clients equally over datacenters that contain replica leaders. Overall target throughput, number of transactions, and number of TGTs are identical for Replicated Commit and Replicated Log. We denote the placement of replica leaders by $R/R/R$ where each initial is the datacenter hosting the replica leader. The first initial is for shard X, the second is for Y, and the third is for Z. For example $V/O/S$ denotes an experiment with shard X's replica leader in datacenter V, shard Y's replica leader in datacenter O, and shard Z's replica leader in datacenter S.

Number of datacenters. The first set of experiments measures the effect of scaling up the number of datacenters while fixing the size of the database. Figure 3 shows the results of five configurations. The scenario is denoted by the initials of involved datacenters. For example, experiment $CVOI$ is an experiment performed on four datacenters, namely California, Virginia, Oregon, and Ireland. Replicated Log's replica leader placement of the scenarios are $C/C/C$, $C/C/V$, $C/V/O$, $V/O/I$, and $V/O/I$, respectively. A run with only one datacenter, C, is shown to give a sense of the overhead caused by intra-datacenter Two-Phase commit in addition to internal processing for Replicated Commit and internal processing and intra-datacenter communication between the client and machines for Replicated Log. This overhead is measured to be 5.8ms for Replicated Commit and 14.8ms for Replicated Log. This difference is due to different patterns of message exchange for both protocols. For the case of one datacenter, Replicated Commit sends one message to a designated coordinator that will perform Two-Phase Commit and return with the result to the client, whereas Replicated Log's client sends three messages to all machines and then a coordinator completes the transaction and returns to the client. As we increase the number of datacenters the effect of communication latency becomes more pronounced. In experiment CV, for example, the average incurred latency is 90ms for Replicated Commit, which agrees with our expectation to be the RTT value (86ms in this case) plus the overhead caused by intra-datacenter communication and processing. This is compared to a commit latency of 267ms, a 196.67% increase, for Replicated Log. Consider the scenario CVO to observe that a transaction only needs a majority to commit; the average latency of this scenario is lower than CV for both cases. This is because datacenters C and O are closer than C and V, thus a majority can be acquired in a shorter time. For Replicated Commit, transactions issued in C and O can commit with a latency slightly above their RTT value, *i.e.*, 21ms. The average latency is higher than this value because transactions originating from datacenter V still incur higher latency. Replicated Log benefits from having a closer majority by shortening the time required for replication, though the time required for Two-Phase commit is not affected. The largest two scenarios show the expected result which is the average of required latency to get a reply by a majority for all datacenters for Replicated Commit and the average time of Two-Phase Commit exchange and replication for Replicated Log. For the inter-continental cases, $CVOI$ and $CVOIS$, Replicated Log incur a commit latency higher than Replicated Commit by 241% and 189%, respectively.

Replica leaders placement. The performance of Replicated Log depends on the placement of replica leaders. To investigate the effect of different placements on performance we conducted a set of experiments with the following replica leader placements: $C/C/C$, $C/V/O$, $V/O/I$, and $S/S/S$. Results are shown in Figure 4 compared to the commit latency observed by Replicated Commit. The first placement, $C/C/C$, groups all leaders in C. This setting yields the best results since C have the lowest latency to acquire the vote of the majority and Two-Phase Commit's message exchange is all inside the datacenter. The commit latency is 206ms, which is to be expected since two rounds of replication are necessary and each replication round require 86ms. The second

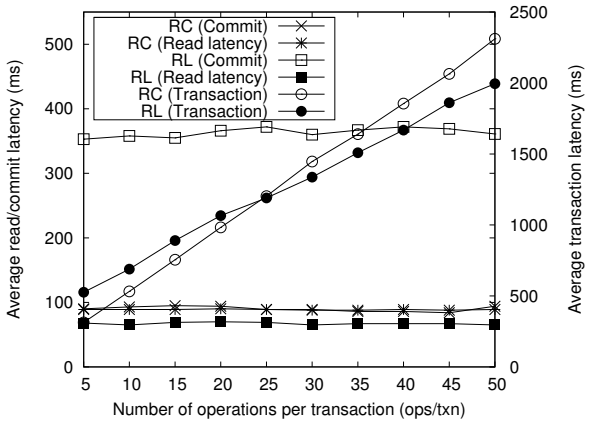


Figure 5: Increasing the number of read operations per transaction.

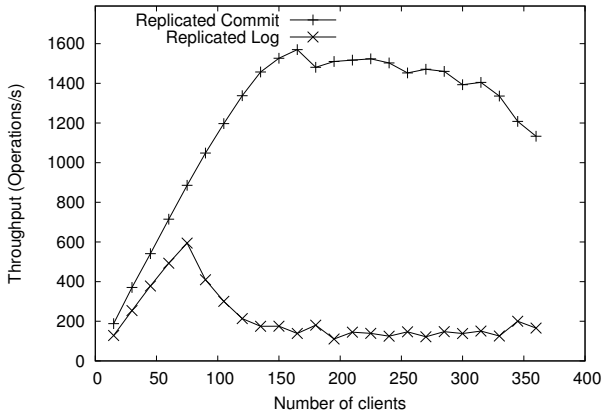


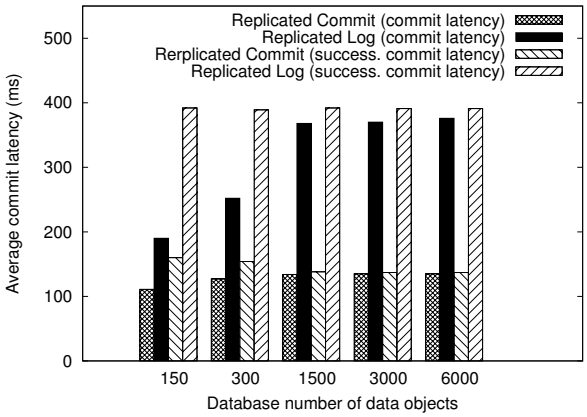
Figure 6: The effect of increasing the number of clients on the throughput

placement, $C/V/O$, assigns all replica leaders in the United States. Observe that the commit latency is higher than for the $C/C/C$ placement. An inter-continental replica distribution is studied in the third placement, $V/O/I$, where the commit latency jumps to 387ms due to the wide-area communication required by Two-Phase Commit and replication. The fourth placement, $S/S/S$, assigns all replica leaders to S. Note that the commit latency is much higher than the similar $C/C/C$ due to the higher cost of replication caused by a farther majority. The results show that Replicated Commit outperforms Replicated Log even for $C/C/C$, the best possible placement. In the remainder of this section we will use $V/O/I$ as a replica leader placement strategy for Replicated Log.

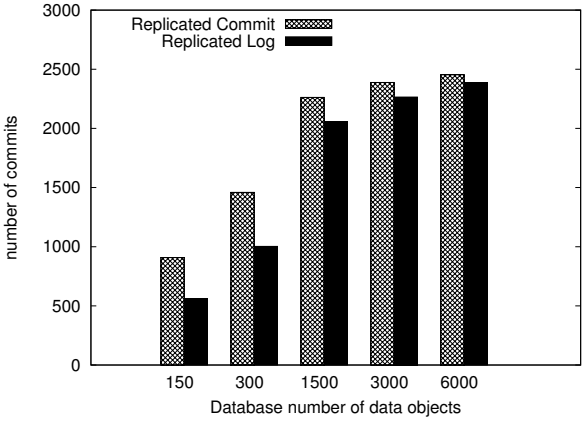
Number of operations per transaction. Increasing the number of reads per transaction causes more time to be spent servicing those reads before the transaction can request to commit. In Figure 5 the effect of increasing the number of operations per transaction is shown for results collected from clients at V accessing items from a pool of 50000 data items. The size is larger for this experiment to isolate the effect of read operations from any contention caused by the larger number of operations. Read operations are independent from each other, which causes the read latency to maintain the same average latency even while increasing the number of operations per transactions. Similarly, the commit latency maintains its value for both protocols. The effect of the increase of the number of reads is witnessed on the transaction latency, which is the latency spent on a transaction, which includes the time to read the read-set in addition to the time to commit. As the number of operations increase, the transaction latency increases too for both protocols. We showed in Section 4.3 that the number of reads per transaction is the sole determinant of whether Replicated Log or Replicated Commit performs better. Recall that Replicated Log is better for scenarios with a large number of reads and that there is a value that represents a critical point between number of reads that causes Replicated Commit to perform better and number of reads that causes Replicated Log to perform better. This value is 2.5 times the number of datacenters, *i.e.*, 12.5 in this case. Since half the operations are writes, the number of operations per transaction that represents the critical point in the figure is 25 operations per transaction. Note that the crossing of the two plots of the transaction latency crosses at the expected point, *i.e.*, 25 operations per transaction. This validates our analysis. Note that as the number of operations per transaction increases, the gap of the transaction latency between the two protocols increases too.

Throughput. Now, we discuss the results of an experiment to study the amount of throughput that the system can support. Throughput here is the *number of successfully committed operations per second*. We set the database size to be 50000 data objects to be able to accommodate the large number of requests. We do this by increasing the number of clients, which are TGTs in our case and observe how much throughput can be achieved. Each client issues transactions back-to-back, *i.e.*, maximum throughput it is able to generate. Each run from this experiment has a duration of 10 minutes. Results are shown in Figure 6. We increase the number of clients from 15 to 360. Replicated Commit's peak throughput is 1570 operations per second (ops/s) for 165 clients. Replicated Commit maintains a close throughput to this value for a larger number of clients and then starts decreasing for more than 300 clients. Replicated Log scales poorly with increasing demand when compared to Replicated Commit. The peak throughput of Replicated Log in this set of experiments is 594 ops/s for 75 clients. Replicated Log then experiences a drop of throughput as the number of clients increases. This drop of throughput is due to the contention caused at the leaders. This causes Replicated Log to thrash and be not able to commit transactions.

Contention. Replicated Commit's reaction to contention is quantified by varying the number of data objects in the database. Results are shown in Figure 7. The number of data objects is varied from 150 to 6000 data objects. Commit latency and number of commits are shown in Figures 7(a) and 7(b), respectively. The average commit latency of *successfully committed transactions only* is also reported. Consider the scenario with 150 data objects, hence highest contention. Commit latency is lower than those observed in cases with lower contention for Replicated Log. This is because the number of aborted transactions is higher; an aborted transaction exhibits lower latency due to its early termination compared to a committed transaction. This is noticed by examining that the commit latency of successfully committed transactions does not vary as much as the overall commit latency. Replicated Commit maintains its average commit latency even with high contention. This is because the client waits for the majority response for both aborts and commits. However, there is a slight decrease due to the failed reads that cause the transaction to abort early. Even for a small number of data objects, 150, Replicated Commit recorded 36% successful commits compared to 22.32% successful commits for Replicated Log. The behavior converges to deliver the expected commit latency and number of commits as contention decreases for



(a) Commit latency.



(b) Number of commits out of 2500 transactions.

Figure 7: Measuring the effect of contention on commit latency and number of commits by varying the size of the database.

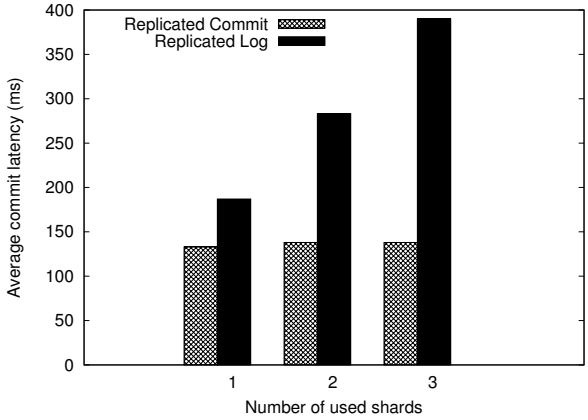


Figure 8: The effect of number of used shards on commit latency

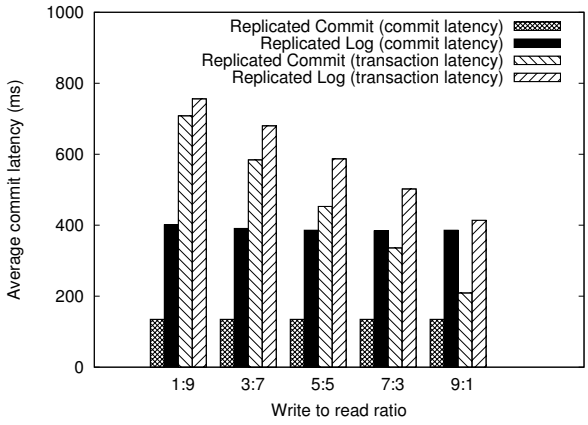


Figure 9: The effect of the ratio of writes to reads on commit latency

both cases.

Intra-datacenter overhead. It is necessary to quantify intra-datacenter overhead on the overall behavior of Replicated Commit. We showed in Figure 3 that the commit latency in a single datacenter with intra-datacenter communication only to be 5.8ms for Replicated Commit. Another aspect that needs to be studied is whether the interaction between datacenters can create additional complexities and overheads in intra-datacenter communication and logic. For this we design an experiment where the number of shards is varied in each datacenter to observe whether having more shards affects the overall behavior of the system. We compare the results with Replicated Log. Note that the effect of decreasing the number of involved shards is different for Replicated Log; Inter-datacenter communication is still necessary for replication. Results of this experiment are shown in Figure 8 where we vary the number of shards from one to three shards. A slight increase is observed from 133 to 138ms for Replicated Commit. However, this change is affected by the smaller number of commits for the case of one shard to other cases. Varying the number of shards plays a larger role in deciding Replicated Log's commit latency. The commit latency with one shard is almost half the commit latency of three shards. This highlights the significance of access locality for Replicated Log protocols. The figure shows that Replicated Commit outperforms Replicated Log even in the case of transactions accessing one shard by 28.8%.

Write to read ratio. In our previous experiments we set the ratio of writes to reads is 1:1. Different applications have different read and write behavior. To test whether Replicated Commit is suitable for read- or write-intensive workloads we show results of varying write to read ratio in Figure 9. We observe a slight effect on commit latency with high write to read ratio for both Replicated Commit and Replicated Log. However, the main effect is on the overall transaction latency, which includes the latency of read operations. Write operations are buffered and thus do not affect the transaction latency whereas read operations are served at the time of the operation. The figure shows that Replicated Commit is impacted more from the increase in the number of reads when compared to Replicated Log. This is due to the difference in read strategies; Replicated Commit reads from a majority and Replicated Log reads from a leader. The number of commits is affected by workloads with more writes than reads. The effect of comparing a write-intensive workload (9:1) to a read-intensive workload (1:9) is more visible for Replicated Log (16.2% decrease of number of commits) than Replicated Commit (6.2% decrease). This effect of write to read ratio is slight compared to the other factors we discussed earlier. This is because the design of Replicated Log and

Replicated Commit make write locks held for a short duration, *i.e.*, write locks are held only for the duration of processing the transaction, unlike read locks for Replicated Log.

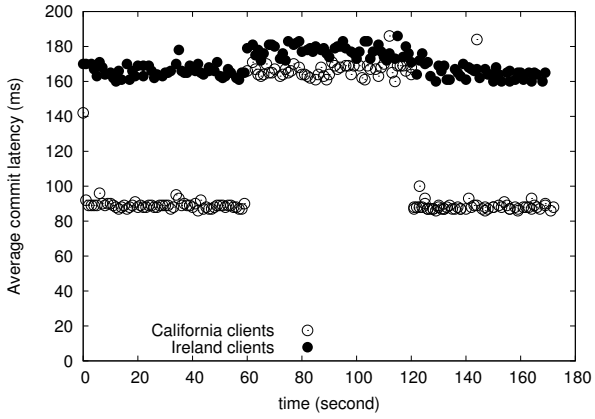


Figure 10: Fault-tolerance of the system by introducing a sudden outage of datacenter California.

Fault-tolerance. In the next experiment we demonstrate the ability of Replicated Commit to operate in the case of failures. We consider a scenario with all five datacenters and ten TGTs in C and I. The experiments take three minutes. After the first minute, an outage of datacenter C is simulated. Then in the second minute, the datacenter becomes alive. Results of this experiment are shown in Figure 10. In the figure, every transaction commit is represented with a point in the figure. The position of the point is determined by its commit time (x-axis) and commit latency (y-axis). Note that we plot every tenth transaction only to make the figure less cluttered. Observe that commit latency before the outage is around 90ms for C's clients and 165ms for I's clients as expected for the case of normal operation. When C outage occurs, the commit latency of C's clients immediately jumps to be around 160ms. This is because clients can no longer get a majority including C, thus they need to wait for messages that take longer from either I or S to form a majority with the closer datacenters O and V. I's clients, on the other hand are not affected as drastically as C's clients. This is because although C was part of the majority of I's clients, the next best datacenter, O, is of comparable latency to C. This sudden adaptation of Replicated Commit is because committing a transaction is possible with any majority of operating datacenters. This is in contrast to Replicated Log, where replica leaders hold leases, 10 seconds for the case of Spanner, that need to be expired before proceeding with a different replica leader.

Read and commit latencies. Replicated Log and Replicated Commit have different read and commit strategies. It is important to observe the trade-off between the latency of a read and a commit. In Figure 11 we plot the read and commit latencies for transactions. To simplify the presentation we only display a subset of the transactions that committed at V and I. Each point represents a read operation where the x-axis is the commit latency of the corresponding transaction and the y-axis is the latency of that read operation. Replicated Commit's points are clustered in regions that have similar read and commit latencies, *i.e.*, V transactions have a read and commit latency close to 100ms and I transactions have a latency just below 200ms. This is because Replicated Commit waits for a majority for both the read and the commit operations. Replicated Log on the other hand displays a different behavior. Read latencies depend on the target leader. Thus, transactions for I for example have read values ranging from 0 to 200ms. Note that the read values have three possible levels for I: 0 read latency is for reads served by the local replica, 100ms latency is for reads served by the next closest leader, and larger latencies are for reads of the farthest replica. V has two levels for read latencies because the second nearest (O) and farthest (I) leaders almost have the same RTT from V, *i.e.* around 100ms. The same applies for commit latencies because a commit reaches leaders that are involved in the transaction only. For example, transactions for I that have a commit latency around 350ms are transactions that access two shards only. Transactions that access all three shards from I have a commit latency of more than 400ms.

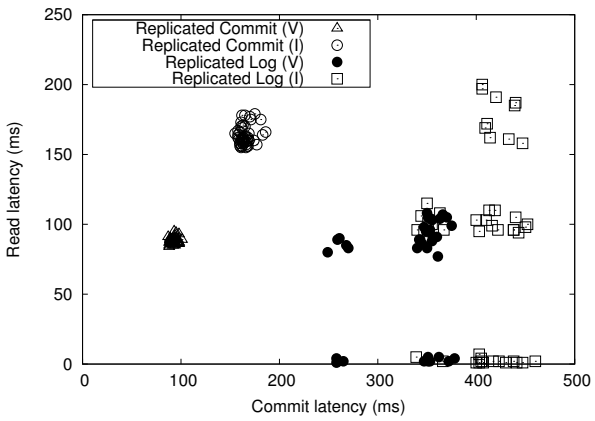


Figure 11: Commit and read latencies of transactions.

6. RELATED WORK

Running transactions on top of replicated storage was first proposed in Gifford's Ph.D. thesis [16], and has been recently used in various systems like Spanner [13] and Scatter [17], by making use of Paxos [25] for log replication. Replicated Commit, in comparison, runs Paxos on top of Two-Phase Commit and Two-Phase Locking to replicate the commit operation instead of the log. In other words, Replicated Commit considers each datacenter as a stand-alone data storage system that provides transactional guarantees (using 2PC); Replicated Commit replicates the execution of transactions on those datacenters. Thus, although Replicated Commit still uses a layered architecture that separates atomic commitment, concurrency control, and consistent replication into different layers, Replicated Commit inverts the layered architecture to achieve lower latency.

Gray and Lamport [18] proposed Paxos Commit as an atomic commitment protocol to solve the blocking issue of Two-Phase Commit. Eliminating blocking during atomic commitment is an orthogonal issue; in other words, Replicated Commit can still use Paxos Commit as an alternative to Two-Phase Commit for atomic commitment inside each datacenter, while maintaining the layered architecture of running an inter-datacenter replication layer on top of an intra-datacenter atomic commit layer.

Asynchronous primary-copy replication [34] is a different approach to database replication that has been the most commonly-used replica management protocol for a long time [6]. Unlike Paxos replication, in primary-copy replication, replica leaders do not have to wait for a majority of replicas to receive an update before acknowledging the client (or the 2PC coordinator); instead, updates are considered done once they are applied to the replica leader, then these updates propagate lazily to the rest of the replicas. Although

primary copy replication reduces transaction latencies, it may result in data loss in case of server failures, thus it is no longer practical for cloud databases running on cheap commodity hardware that crashes frequently.

During the past few years, several consistency models weaker than one-copy serializability have been proposed. However, in general, it has been now broadly acknowledged even by the proponents of weaker data consistency models that weak consistency introduces costs and overheads that more than offset the performance and scalability advantages. One particular proposal that has gained some degree of acceptance for replicated data is referred to as *one-copy snapshot isolation* (SI) [22, 27]. New replica management protocols [28, 32] have been proposed for wide-area networks (WANs) that realize one-copy snapshot isolation. However, as was pointed out in [20]: "1-copy SI can lead to data corruption and violation of integrity constraints [4]. 1-copy serializability is the global correctness condition that prevents data corruption." Furthermore, we note that these protocols were not developed in the context of geo-replicated data over multiple datacenters. Rather, the underlying architecture leverages data replication for database computing in the edges of a network. Furthermore, these protocols are based on the assumption of atomic multicast [7] over WAN which enables consistent commitment of update transactions with a single WAN message to a centralized certifier for synchronizing transactional read and write accesses. In essence, the complexity of distributed atomic commitment and synchronized execution of transactions is incorporated in the implementation of the atomic multicast which in itself is not scalable in the WANs.

Message Futures [30] is another recent proposal that aims at reducing the number of cross-datacenter communication trips required to achieve strong consistency for multi-datacenter databases. Message Futures indeed reduces the number of cross-datacenter communication trips required by each transaction to only one cross-datacenter roundtrip; however, Message Futures does not tolerate datacenter failures. In other words, if one of the datacenters goes down the system does not make progress, thus geo-replication in that case serves to improve read availability but not fault-tolerance.

MDCC [23] is a recently proposed approach to multi-datacenter databases that uses variants of Paxos to provide both consistent replication and atomic commitment. In comparison to MDCC, the layered architecture that is used in Replicated Commit, as well as in replicated logging, separates the atomic commitment protocol from the consistent replication protocol. Separating the two protocols reduces the number of acceptors and learners in a Paxos instance to the number of datacenters, rather than the number of data servers accessed by the transaction; the latter equals the number of datacenters times the number of shards accessed by the transaction. Besides, the layered architecture has several engineering advantages such as modularity and clarity of semantics. Unifying different protocols via semantically-rich messages has been investigated before in other contexts; for example, integrating concurrency control with transactional recovery [2].

Other examples of multi-datacenter datastores include Cassandra [24] and PNUTS [12]; however, those system do not support transactions. COPS [29] delivers a weaker type of consistency; that is, causal consistency with convergent conflict handling, which is referred to as *causal+*. Other systems have been also developed with focus on distributed transactions. For example, H-Store [21] and Calvin [35] are recently-proposed distributed datastores that eliminate concurrency by executing transactions serially, when the set of locks to be acquired during a transaction are known in advance, but they do not provide external consistency. Walter [33] is another recently-proposed datastore that extends Snapshot Isolation to a variant called Parallel Snapshot Isolation (PSI).

7. CONCLUSION

We present an architecture for multi-datacenter databases that we refer to as Replicated Commit, to provide ACID transactional guarantees with much fewer cross-datacenter communication trips compared to the replicated log approach. Instead of replicating the transactional log, Replicated Commit replicates the commit operation itself by running Two-Phase Commit multiple times in different datacenters, and uses Paxos to reach consensus among datacenters as to whether the transaction should commit. Doing so not only replaces several inter-datacenter communication trips with intra-datacenter communication trips, but also allows us to eliminate the election phase before consensus so as to further reduce the number of cross-datacenter communication trips needed for consistent replication. Our proposed approach also improves fault-tolerance for reads. We analyze Replicated Commit by comparing the number of cross-datacenter communication trips that it requires versus those required by the replicated log approach, then we conduct an extensive experimental study to evaluate the performance and scalability of Replicated Commit under various multi-datacenter setups.

8. ACKNOWLEDGMENTS

This work is partially supported by NSF Grant 1053594. Faisal Nawab is partially funded by a fellowship from King Fahd University of Petroleum and Minerals. We Also would like to thank Amazon for access to Amazon EC2.

9. REFERENCES

[1] D. Agrawal and A. J. Bernstein. A nonblocking quorum consensus protocol for replicated data. *Parallel and Distributed Systems, IEEE Transactions on*, 2(2):171–179, 1991.

[2] G. Alonso, R. Vingralek, D. Agrawal, Y. Breitbart, A. El Abbadi, H.-J. Schek, and G. Weikum. Unifying concurrency control and recovery of transactions. *Information Systems*, 19(1):101–115, 1994.

[3] 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.

[4] 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.

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

[6] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. The Morgan Kaufmann Series in Data Management Systems, 2009.

[7] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76, 1987.

[8] D. G. Campbell, G. Kakivaya, and N. Ellis. Extreme scale with full sql language support in microsoft sql azure. In *Proceedings of the 2010 international conference on Management of data*, pages 1021–1024. ACM, 2010.

[9] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live - an engineering perspective (2006 invited talk). In *PODC*, 2007.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proc. 7th USENIX Symp. Operating Systems Design and Implementation*, pages 15–28, 2006.

[11] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. 1st ACM Symp. Cloud Computing*, pages 143–154. ACM, 2010.

[12] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[13] J. 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. pages 251–264, 2012.

[14] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. VLDB Endow.*, 4(8):494–505, May 2011.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symp. Operating Systems Principles*, pages 205–220, 2007.

[16] D. K. Gifford. *Information storage in a decentralized computer system*. PhD thesis, Stanford, CA, USA, 1981. AAI8124072.

[17] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 15–28, New York, NY, USA, 2011. ACM.

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

[19] T. Hoff. Latency is everywhere and it costs you sales - how to crush it. *High Scalability, July*, 25, 2009.

[20] H. Jung, H. Han, A. Fekete, and U. Röhm. Serializable snapshot isolation for replicated databases in high-update scenarios. *Proceedings of the VLDB Endowment*, 4(11):783–794, 2011.

[21] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, Aug. 2008.

[22] B. Kemme, R. Jimenez-Peris, and M. Patino-Martinez. Database replication. *Synthesis Lectures on Data Management*, 2(1):1–153, 2010.

[23] T. Kraska, G. Pang, M. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *EuroSys*, pages 113–126, 2013.

[24] A. Lakshman and P. Malik. Cassandra: A structured storage system on a p2p network. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 47–47. ACM, 2009.

[25] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[26] L. Lamport. Paxos Made Simple. *SIGACT News*, 32(4):51–58, Dec. 2001.

[27] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 419–430. ACM, 2005.

[28] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris. Enhancing edge computing with database replication. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 45–54. IEEE, 2007.

[29] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[30] F. Nawab, D. Agrawal, and A. El Abbadi. Message futures: Fast commitment of transactions in multi-datacenter environments. In *CIDR*, 2013.

[31] M. Seltzer. Oracle nosql database. In *Oracle White Paper*, 2011.

[32] D. Serrano, M. Patiño-Martínez, R. Jiménez-Peris, and B. Kemme. An autonomic approach for replication of internet-based services. In *Reliable Distributed Systems, 2008. SRDS'08. IEEE Symposium on*, pages 127–136. IEEE, 2008.

[33] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400. ACM, 2011.

[34] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Trans. Softw. Eng.*, 5(3):188–194, May 1979.

[35] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 international conference on Management of Data*, pages 1–12. ACM, 2012.

[36] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann, 2001.