Low-Latency Multi-Datacenter Databases using Replicated Commits

Hatem A. Mahmoud, Alexander Pucher, Faisal Nawab,
Divyakant Agrawal, Amr El Abbadi
Universoty of California
Santa Barbara, CA, USA
{hatem,pucher,nawab,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, by using a more efficient architecutre. Instead of replicating the transactional log, we replicate the commit operation itself, by running Two-Phase Commit multiple times in different datacenters, and use 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 so as 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.

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 has driven many web service providers towards adopting NoSQL datastores that do not provide transactional guarantees but provide more scalablility and availability via transparent sharding and replication of large amounts of data. For example, systems like Google's Bigtable [6], Apache Cassandra [14], and Amazon's Dynamo [9] do not guarantee isolation or atomicity for multi-row transactional updates. Other systems like Google's Megastore [3], Microsoft's SQL Azure [4], and Oracle's NoSQL Database [18] 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 [8] is a prominent example of such 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 [11], a distributed hashtable 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 ACID transactional guarantees for cross-datacenter databases with a smaller number of cross-datacenter roundtrips, compared to a system that uses log replication, such as Spanner, by using a more efficient architecutre. 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 the Two-Phase commit multiple time, once per datacenter, with each datacenter executing Two-Phase Commit and Two-Phase Locking internally, and we use Paxos to reach a consensus among datacenters as to whether the transaction should eventually commit. 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 trips, when implementing ACID transactions on top of globally-replicated data. Moreover, replicating the Two-Phase Commit operation, rather than replicating log entries, also allows us to integrate atomic commitment and isolation protocols with consistent replication protocols so as 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. Reducing the number of cross-data center communication trips is crucial in order to reduce transaction response time as perceived by the user. Studies performed by different web service providers [1] 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 time causes traffic to drop by 20%, while Amazon reports that every 100ms increase in latency results in 1% loss in sales. When replicating a database across datacenters in different continents, each cross-datacenter communication trip consumes tens of milliseconds, or even hunderds 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 require much more time that the theoretical lower bound on packet transmission speed; that is, the speed of light. For example, a packet sent from the East Coast to the West Coast takes about 90ms, which is nearly three times the time it takes a light pulse to transfer that 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 data items (i.e., a small subset of data items receives much more traffic than the rest of the database), Replicated Commit reduces lock contention considerably, 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 arhcitecture, that is currently used in production systems such as Google's Spanner, in order to analyze Replicated Commit's savings in cross-datacenter communications.

The rest of the paper is organized as follows. Section 2 presents a motivating example by analyzing cross-datacenter communication in a typical replicated log system. In Section 3 we propose our new architecture and discuss special cases, such as partially-replicated databases. 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. We present related work in Section 5, and conclude in Section 6.

2. MOTIVATING EXAMPLE

In this section, we present an example that demonstrates the overhead of running distributed transactions on top of globally replicated data, with strong consistency guarantees, using log replication. In a replicated log system that runs Two-Phase Commit and Two-Phase Locking on top of Paxos, as the case with Spanner and Scatter, typically the client program that executes 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 of the database. Figure 1 shows the messages exchanged during Two-Phase Commit on a system where logs are replicated across data centers using Paxos. Solid lines are used to illustrate Two-Phase Commit communication, while dashed lines are used 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. Each shard has a Paxos leader at one of the datacenters. In this example each Paxos leader is in a different datacenter, but in general this may or may not be the case. We explain the sequence of operations shown in Figure 1 as follows.

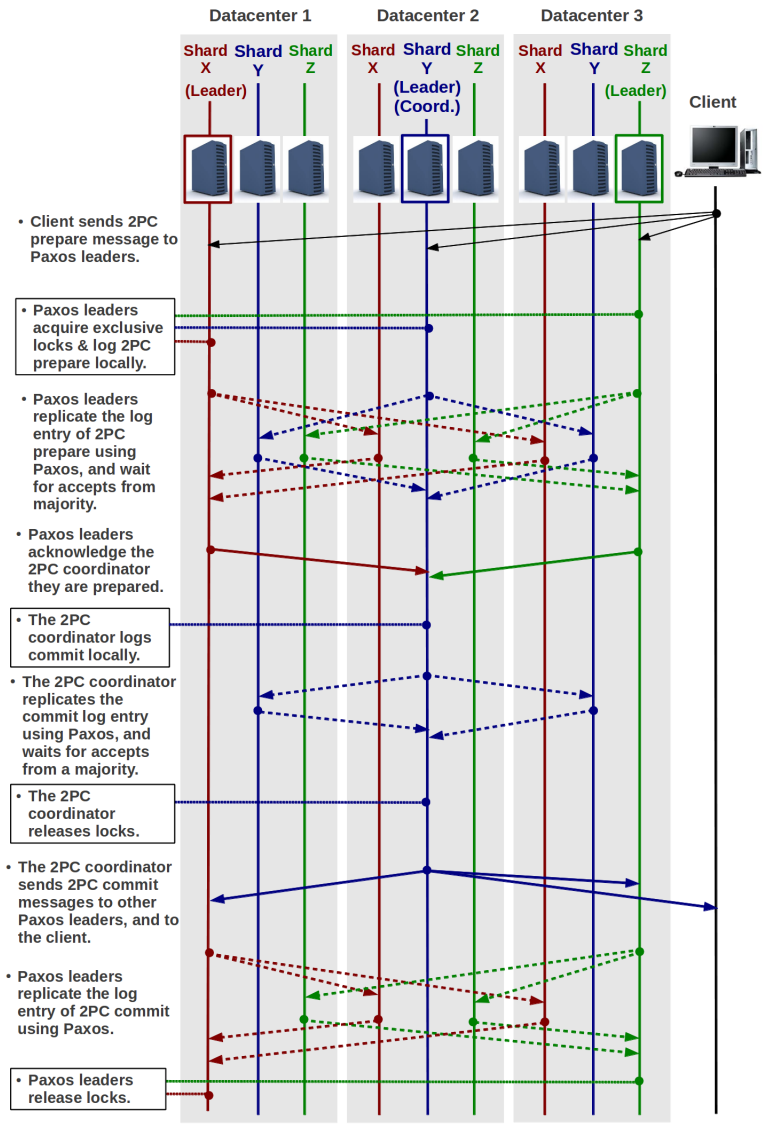

Figure 1: Typical sequence of messages and operations when running Two-Phase Commit on top of a Paxos-replicated log.

1. 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 other Paxos leaders of X and Z are *cohorts*.

2. The client sends Two-Phase Commit prepare messages to the Paxos *leaders*[1] 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.

3. The Paxos leaders acquire exclusive locks on the target data

[1]To avoid the cost of re-electing a leader each time the system logs an entry, production systems (e.g., [5, 8]) use a variation of Paxos called multi-Paxos. In multi-Paxos, once a node is elected a leader during a Paxos instance, it remains the leader for subsequent Paxos instances until the leader crashes; re-elections are done periodically, rather than in each Paxos instance, in order to account for the case of a crashing leader.

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 log entry to, and receive acknowledgements from, a majority of replicas in other datacenters.

4. 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 data centers, this a cross-datacenter one-way trip.

5. 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.

6. 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.

7. 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.

8. The Two-Phase Commit cohorts (i.e., the Paxos leaders of Y and Z) log the commit entry in their Paxos logs. This is a cross-datacenter roundtrip.

9. The Two-Phase Commit cohorts (i.e., the Paxos leaders of Y and Z) release theirs locks after receiving acknowledgements 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 acknowledgement from the Two-Phase Commit coordinator, is 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 seven one-way trips, if the Paxos leaders of X, Y, and Z are in the same datacenter, or eight one-way trips if the Paxos leaders of X, Y, and Z are in different datacenters. 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.

3. REPLICATED COMMIT

We begin by presenting the data model and infrastructure that Replicated Commit runs upon, then we explain the Replicated Commit protocol stack, and show its correctness.

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; that is, each datacenter has a full copy of the database. A variant of Replicated Commit also works for partially replicated data; that is, the case when some or all datacenters have subsets of the database rather than full copies. Within each datacenter the database is sharded across multiple servers to achieve high throughput. Replicated Commit supports transaction-time databases as the case with Spanner; that is, all versions of a data item are maintained in the database, each with its own timestamp that indicates the realtime clock at the time when that version was committed into the database.

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. This is different from the case when running Two-Phase Locking on top of a Paxos-replicated log, where a single replica is elected a Paxos leader (and gets periodically re-elected until it crashes); as we show in Section 2, the Paxos leader in that case is the only replica that maintains locks on its data items. When it is only the Paxos leader that maintains a lock table, if a Paxos leader crashes, all transactions that hold locks at the lock table of that Paxos leader have to abort and restart then re-acquire locks from a newly-elected Paxos leader. Replicated Commit eliminates this single point of failure because a transaction does not have to restart as long as it maintains locks on a majority of replicas, and as long as that majority does not fail.

3.2 Transactions

In Replicated Commit, each transaction is assigned a timetamp at the beginning. We use the begining timestamp to avoid deadlocks using wound-wait, and to ensure data consistency. As the transaction proceeds, 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 at the end of the transaction as part of Two-Phase Commit. In the following subsections we explain transctional reads and Two-Phase Commit for the general case of multi-shard read-write transactions, as well as special cases such as single-shard transactions, and partially-replicated data.

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. The client 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 other transctions that are trying to update the same data item get blocked until this transction commits; an exception to that is when an older transaction (i.e., a transaction with a lower timestamp) tries to update the same data item, then read locks are released immediately to avoid deadlocks. Once the client receives responses to its read request from a majority of replicas, the client is done with the read operation and may proceed to process the data item that has been read. If different datacenters respond with different values for the same data item the client picks the value that is associated with the highest timestamp; this is to ensure that the client processes the most up-to-date version of the data item that it reads. Since an update operation releases exclusive locks only after updating a majority of replica, thus reading from a majority of replicas guarantees that the most up-to-date value is retrieved from at least one of the replicas.

3.2.2 Two-Phase Commit

Once a transaction has finished all reading 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. In this section we consider the general case of multi-shard transactions on fully-replicated data; then in later sections we discuss the special cases of single-shard transactions and partial replication.

We use the standard terminology of Paxos from [15, 16]; 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, while in the second phase, acceptors accept the value proposed by the leader that they elected in the first phase. Learners need to learn the value that has been accepted by a majority of acceptors. Learners can learn the accepted value in different ways; for example, by having each acceptor broadcasts the value that it has accepted to all learners, or by appointing one (or more) distinguished learner(s) to learn the accepted values from acceptors, then forward the learnt value (i.e., the value accepted by a majority) to other learners.

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. There is only one proposer for this Paxos instance that is the client program itself, and each datacenter acts as both an acceptor and a learner. Thus there is no need for an election phase because the Paxos leader is always the client; however, that undisputed leader should not be thought of as a dictator (as the case in asynchronous master-slave replication for example) because a Paxos leader still needs acceptances from a majority of acceptors to pass its proposal. Following a similar political analogy to that used in the original Paxos paper [15], single-proposer Paxos can be thought of as a constitutional monarchy, in constrast with representative democracy in the case of multi-proposer Paxos. 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. If the database is required to support transaction-time queries, the commit time of each transaction is also a value that needs to be agreed on; in that case, if a transaction commits, the same Paxos instance that is used to reach consensus on the commit decision is also used to reach consensus on a commit timestamp.

Each datacenter accepts any Two-Phase Commit prepare request from any transaction only after (1) acquiring all the exclusive locks needed by that transaction, as specified in the prepare request, (2) checking that shared locks (i.e., read locks) that have been acquired by that transaction are still being held by the same transaction, thus no shared locks have been released because of wound-wait or because of timeouts for example, 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. 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 Two-Phase Commit prepare request (through 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, and piggybacks this appointment to the prepare request that the client sends to all data servers involved in 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 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 Two-Phase Commit prepare request. If a datacenter can not perform one or more of the operation(s) needed during the prepare phase of Two-Phase Commit, the datacenter does not accept the prepare 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.

Each datacenter, whether accepted the commit request or not, needs to learn whether a majority of datacenters has accepted that commit request or not. To achieve this, each coordinator in each datacenter whenever it responds back to the client with an accept, forwards that accept message to the corresponding coordinators of the same transaction in other datacenters as well. Whenever a coordinator in a datacenter learns that a majority of datacenters has accepted the Two-Phase Commit prepare request, the coordinator proceeds to the second phase of Two-Phase Commit; that is, the commit phase. During the commit 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 all locks constitute the second phase (i.e., the commit phase) of Two-Phase Commit.

3.2.3 Single-Shard Transactions

In Replicated Commit, the number of cross-datacenter communication trips required to commit a transaction that accesses only one shard is the same as that required by a transaction that acesses multiple shards. This is due to the fact that Replicated Commit performs all the communication required by Two-Phase Commit locally inside a datacenter. The sequence of operations that take place during single-shard transactions can be derived directly from that of multi-shard transactions (as explained in Section 3.2.2) by considering it a special case in which the Two-Phase Commit coordinator in each datacenter is the only participant in Two-Phase Commit; thus the prepare and commit phases occur on that coordinator only.

3.2.4 Partial Replication

If the database is partially replicated, Replicated Commit performs Two-Phase Commit in a way different from that explained in Section 3.2.2. At a high level, Replicated Commit initiates one Paxos instance for each shard that is involved in Two-Phase Commit. The client acts as the sole proposer of each of these Paxos instances, thus it is always the leader. Each replica of each shard

performs the two phases of Two-Phase Commit locally, and the commit phase depends on the decision agreed on using Paxos. For the client to initiate Two-Phase Commit, the client sends a Paxos accept request to each replica of every shard that is involved in Two-Phase Commit; however, the client does not appoint a particular coordinator among replicas as the case with full replication. For each shard involved in Two-Phase Commit (i.e., for each Paxos instance that the client leads) the client waits for accept responses from a majority of replicas. Whenever a data server receives an accept request from a client, the data server acquires exclusive locks that are required by the transaction, checks that shared locks are still being held by the transaction, and logs the Two-Phase Commit prepare operation; if these three operations get executed successfully, the data server responds back to the client with an accept response. Once the client receives accept responses from a majority of replicas of each shard that is involved in Two-Phase Commit, the client proceeds to the learning phase. The value to be learnt by the replicas of each Paxos instance is whether the transaction should commit or not. The default value is not to commit, so each data server that is involved in Two-Phase Commit does not proceed to the commit phase until it learns from the client that the transaction should commit. If the client crashes, data servers that are involved in that transaction timeout and exchange messages with each others in one round of communication to learn whether a majority of replicas of each shard has accepted the Paxos proposal (i.e., has finished the prepare phase of Two-Phase Commit successfully). Once a data server learns about the commit decision, the data server performs the actual commit operations; that is, updates the data items that the transaction requests to update, logs the commit operation, then releases all locks.

3.3 Correctness

The consistency of Replicated Commit follows directly from formulating Replicated Commit in terms of Paxos running on top of Two-Phase Commit. Since all Two-Phase Commit operations occur inside datacenters, Replicated Commit treats all datacenters as replicas of a single data item, and executes a Paxos instance on top of those replicas. Paxos is already proven to guarantee consistency, thus all replicas in all datacenters are guaranteed to be consistent. Within each datacenter, two Paxos instances of two conflicting transactions are isolated by means of Two-Phase Locking that takes place inside each datacenter. To see how the transactional hisotry is always serializable, consider any two conflicting transactions, T1 and T2. If the conflict between T1 and T2 is a write-write conflict, or a transitivity thereof, only one of the two transactions could acquire exclusive locks on the data items that they are trying to update, but not both transaction, because each of them requires locks from a majority of datacenters. Similarly, for read-write conflicts, or transitives thereof, a transction can not start the update phase before all reads are done, and since each transaction has to acquire read locks, for each data item that it reads, from a majority of datacenters, only one of the two conflicting transactions could acquire the required locks. In case of parital-replicated, notice that each shard has its own Paxos instance, thus for each shard to be updated the transaction still needs exclusive locks from a majority of a datacenter to get the Paxos proposal accepted.

4. REPLICATED COMMIT VERSUS REPLICATED LOG

Figure 2 compares Replicated Commit against replicated logs at an architectural level. 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.

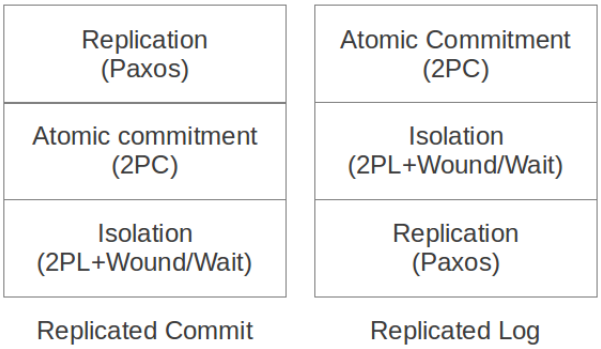

Figure 2: Architectural differences between Replicated Commit and Replicated Log.

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 does not add significant latency compared to systems that are based on replicated logging and long-living Paxos leaders. In the case of long-living Paxos leaders, for each data item there is one replica that acts as the Paxos leader of that data item and maintains shared and exclusive locks on the data item, thus the client needs to read from that Paxos leader only 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 [8], which is a system that implements Two-Phase Commit and Two-Phase Locking on top of a Paxos-replicated log, Paxos leaders are distributed arbitrarily among datacenters. Thus, many data reads end up answered from remote datacenters. This is particulary 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 [8]; 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. As long as a majority of nodes are up and running, transactions can continue reading data without interruption.

4.2 Two-Phase Commit

Figure 3 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 client program begins by picking a shard to act as the coordinator; that is, data servers that hold replicas of the selected shard act as coordinators of Two-Phase Commit

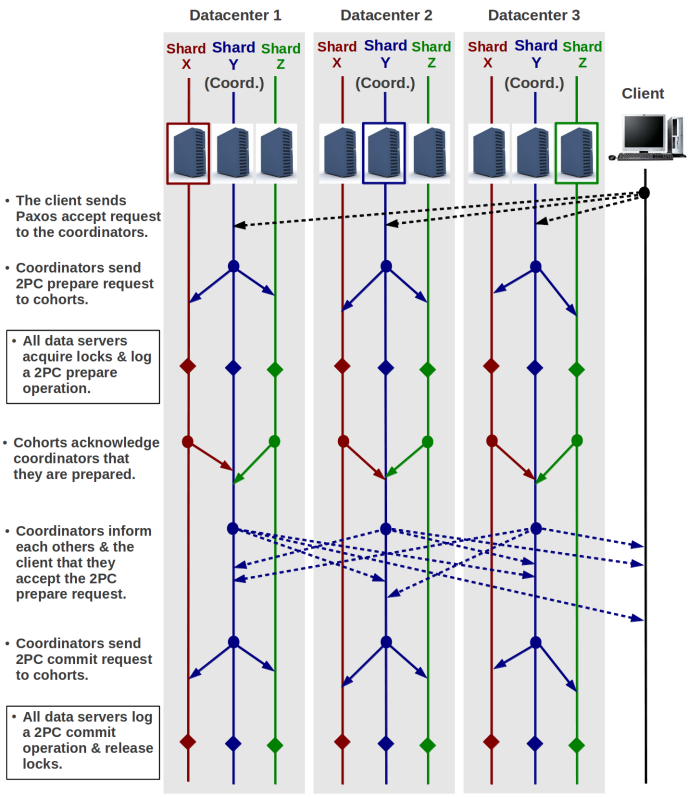

Figure 3: Typical Two-Phase Commit operations when using Replicated Commit.

in each datacenter. The following sequence of messages takes place in Replicated Commit.

1. The client picks a shard, say the shard of Y, as the Two-Phase Commit coordinator.
2. The client sends a Paxos accept request to the coordinator in each datacenter. This requires a cross-datacenter one-way trip.
3. The coordinator in each data center sends a Two-Phase Commit prepare message to all Two-Phase Commit cohorts in the same datacenter, including itself. In this example, the cohorts are the servers that host X, Y, and Z. All cohorts acquire locks, and log the Two-Phase Commit prepare message in their local logs, then respond back to their coordinators.
4. 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 others so that all datacenters learn about this acceptance. This phase requires a cross-datacenter one-way trip.
5. 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.
6. 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 acknowledgement 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 communication 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 communcation 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 data center. Moreover, the amount of time a lock is retained by a given transaction affects the performance of other transactions, specially when there is contention on some data items.

4.3 Single-Shard Transactions

Consider the case of single-shard transactions. In a typical replicated log system, such as Spanner or Scatter, the client sends the transaction to the Paxos leader of the single shard that is accessed by the transaction. The leader can be in any datacenter, arbitrarily far from the client, thus for many transactions, this communication counts as a cross-datacenter communication trip. The Paxos leader logs the commit message in its Paxos log; this requires a cross-datacenter roundtrip. Then the leader acknowledges the client; this requires a cross-datacenter one-way trip. In total, a replicated log system requires four cross-datacenter communication trips to commit a single-shard transaction. When using Replicated Commit, a single-shard transaction has the same number of cross-datacenter trips as a multi-shard transaction (that is, three cross-datacenter trips), because Two-Phase Commit takes place internally inside a datacenter, and does not affect cross-datacenter communication.

5. RELATED WORK

Running transactions on top of replicated storage was first proposed in [10], and has been recently used in various systems like Spanner [8] and Scatter [11], by making use of Paxos [15] 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. 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. MDCC [21] is another widely-circulated 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 repli-

cation protocol. Separating the two protocols reduces the number of acceptors and learners in a Paxos instance to the number of datacenters instead of the number of data servers accessed by the transaction; the latter equals the number of datacenters times the number of shards involved in the transaction. Besides, the layered architecuter has some engineering advantages such as modularily and clarity of semantics. Integrating different protocols via semantically-rich messages has been investigated before in other contexts; for example, integrating concurrency control with transactional recovery [2].

Gray and Lamport [12] propose 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.

Other examples of multi-datacenter datastores include Cassandra [14] and PNUTS [7]; however, those system do not support transactions. COPS [17] 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 distirbuted transactions. For example, H-Store [13] and Calvin [20] are recently-proposed distributed datastores that eleminiate 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 [19] is another recently-proposed datastore that extends Snapshot Isolation to a variant called Parallel Snapshot Isolation (PSI).

6. 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 elimiante 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 comapring 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.

7. REFERENCES

[1] http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

[2] G. Alonso, R. Vingralek, D. Agrawal, Y. Breitbart, A. E. 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. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[4] D. G. Campbell, G. Kakivaya, and N. Ellis. Extreme scale with full sql language support in microsoft sql azure. In *SIGMOD*, 2010.

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

[6] 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 *OSDI*, 2006.

[7] 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.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[9] 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 *SOSP*, 2007.

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

[11] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *SOSP*, 2011.

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

[13] 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.

[14] A. Lakshman and P. Malik. Cassandra: structured storage system on a p2p network. In *PODC*, 2009.

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

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

[17] 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 *SOSP*, 2011.

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

[19] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[20] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[21] M. J. F. S. M. Tim Kraska, Gene Pang. Mdcc: Multi-data center consistency. arXiv:1203.6049 [cs.DB].