MDCC: Multi-Data Center Consistency

Tim Kraska Gene Pang Michael J. Franklin Samuel Madden[♠] Alan Fekete[†]

University of California, Berkeley [♠]MIT [†]University of Sydney

{kraska, gpang, franklin}@cs.berkeley.edu madden@csail.mit.edu alan.fekete@sydney.edu.au

Abstract

Replicating data across multiple data centers allows using data closer to the client, reducing latency for applications, and increases the availability in the event of a data center failure. MDCC (Multi-Data Center Consistency) is an optimistic commit protocol for geo-replicated transactions, that does not require a master or static partitioning, and is strongly consistent at a cost similar to eventually consistent protocols. MDCC takes advantage of Generalized Paxos for transaction processing and exploits commutative updates with value constraints in a quorum-based system. Our experiments show that MDCC outperforms existing synchronous transactional replication protocols, such as Megastore, by requiring only a single message round-trip in the normal operational case independent of the master-location and by scaling linearly with the number of machines as long as transaction conflict rates permit.

1. Introduction

Tolerance to the outage of a single data center is now considered essential for many online services. Achieving this for a database-backed application requires replicating data across multiple data centers, and making efforts to keep those replicas reasonably synchronized and consistent. For example, Google's e-mail service Gmail is reported to use Megastore [2], synchronously replicating across five data centers to tolerate two data center outages: one planned, one unplanned.

Replication across geographically diverse data centers (called geo-replication) is qualitatively different from replication within a cluster, data center or region, because inter-data center network delays are in the hundreds of milliseconds and vary significantly (differing between pairs of locations, and also over time). These delays are close enough to the limit on total latency that users will tolerate, so it becomes crucial to reduce the number of message round-trips taken between data centers, and desirable to avoid waiting for the slowest data center to respond.

For database-backed applications, it is a very valuable feature when the system supports transactions: multiple operations (such as individual reads and writes) grouped together, with the system ensuring at least atomicity so that all changes made within the transaction are eventually persisted or none. The traditional mechanism for transactions that are distributed across databases is two-phase commit (2PC), but this has serious drawbacks in a geo-replicated system. 2PC depends on a reliable coordinator to determine the outcome of a transaction, so it will block for the duration of a coordinator failure, and (even worse) the blocked transaction will be holding locks that prevent other transactions from making progress until the recovery is completed.[1]

In deployed highly-available databases, asynchronous replication is often used where all update transactions must be sent to a single master site, and then the updates are propagated asynchronously to other sites which can be used for reading (somewhat stale) data. Other common approaches give up some of the usual guarantees or generality of transactions. Some systems achieve only eventual consistency by allowing updates to be run first at any site (preferably local to the client) and then propagate asynchronously with some form of conflict resolution so replicas will converge later to a common state. Others restrict each transaction so it can be decided at one site, by only allowing updates to co-located data such as a single record or partition. In the event of a failure, these diverse approaches may lose committed transactions, become unavailable, or violate consistency.

Various projects [2, 8, 11, 18] proposed to coordinate transaction outcome based on Paxos [14]. The oldest design, *Consensus on Transaction Commit* [11], shows how to use Paxos to reliably store the abort or commit decision of a resource manager for recovery. However, it treats data replication as an orthogonal issue. Newer proposals focus on using Paxos to agree on a log-position similar to state-machine replication. For example, Google's Megastore [2] uses Paxos to agree on a log-position for every

[1] We are referring to the standard 2PC algorithm for transaction processing, which requires a durable abort/commit log entry stored at the coordinator. Of course, this log entry could be replicated at the cost of an additional message round-trip (as in 3-phase commit).

Eurosys'13 April 15-17, 2013, Prague, Czech Republic
Copyright © 2013 ACM 978-1-4503-1994-2/13/04. . . $15.00

commit in a data shard called *entity group* imposing a total order of transactions per shard. Unfortunately, this design makes the system inherently unscalable as it only allows executing one transaction at a time per shard; this was observed [13] in Google's App Engine, which uses Megastore. Google's new system Spanner [8] enhances the Megastore approach, automatically resharding the data and adding snapshot isolation, but does not remove the scalability bottleneck as Paxos is still used to agree on a commit log position per shard (i.e., tablet). Paxos-CP [20] improves Megastore's replication protocol by combining non-conflicting transactions into one log-position, significantly increasing the fraction of committed transactions. However, the same system bottleneck remains, and the paper's experimental evaluation is not encouraging with only four transactions per second.

Surprisingly, all these new protocols still rely on two-phase commit, with all its disadvantages, to coordinate any transactions that access data across shards. They also all rely on a single master, requiring two round-trips from any client that is not local to the master, which can often result in several hundred milliseconds of additional latency. Such additional latency can negatively impact the usability of websites; for example, an additional 200 milliseconds of latency, the typical time of one message round-trip between geographically remote locations, can result in a significant drop in user satisfaction and "abandonment" of websites [23].

In this paper, we describe MDCC (short for "Multi-Data Center Consistency"), an optimistic commit protocol for transactions with a cost similar to eventually consistent protocols. MDCC requires only a single wide-area message round-trip to commit a transaction in the common case, and is "master-bypassing", meaning it can read or update from any node in any data center. Like 2PC, the MDCC commit protocol can be combined with different isolation levels that ensure varying properties for the recency and mutual consistency of read operations. In its default configuration, it guarantees "read-committed isolation" without lost updates [4] by detecting all write-write conflicts. That is, either all updates inside a transaction eventually persist or none *(we refer to this property as atomic durability)*, updates from uncommitted transactions are never visible to other transactions *(read-committed)*, concurrent updates to the same record are either resolved if commutative or prevented (no lost updates), but some updates from successful committed transactions might be visible before all updates become visible *(no atomic visibility)*. It should be noted, that this isolation level is stronger than the default, read-committed isolation, in most commercial and open-source database platforms. On the TPC-W benchmark deployed across five Amazon data centers, MDCC reduces per transaction latencies by at least 50% (to 234 ms) as compared to 2PC or Megastore, with orders of magnitude higher transaction throughput compared to Megastore.

MDCC is not the only system that addresses wide-area replication, but it is the only one that provides the combination of low latency (through one round-trip commits) and strong consistency (up to serializability) for transactions, without requiring a master or a significant limit on the application design (e.g., static data partitions with minimization of cross-partition transactions). MDCC is the first protocol to use Generalized Paxos [15] as a commit protocol on a per record basis, combining it with techniques from the database community (escrow transactions [19] and demarcation [3]). The key idea is to achieve single round-trip commits by 1) executing parallel Generalized Paxos on each record, 2) ensuring every prepare has been received by a *fast* quorum of replicas, 3) disallowing aborts for successfully prepared records, and 4) piggybacking notification of commit state on subsequent transactions. A number of subtleties need to be addressed to create a "master-bypassing" approach, including support for commutative updates with value constraints, and for handling conflicts that occur between concurrent transactions.

In summary, the key contributions of MDCC are:

- A new optimistic commit protocol, which achieves wide-area transactional consistency while requiring only one network round trip in the common case.
- A new approach to ensure value constraints with quorum protocols.
- Performance results of the TPC-W benchmark showing that MDCC provides strong consistency with costs similar to eventually consistent protocols, and lower than other strongly-consistent designs. We also explore the contribution of MDCC's various optimizations, the sensitivity of performance to workload characteristics, and the performance impact during a simulated data center failure.

In section 2 we show the overall architecture of MDCC. Section 3 presents MDCC's new optimistic commit protocol for the wide area network. Section 4 discusses the MDCC's read consistency guarantees. Our experiments using MDCC across 5 data centers are in section 5. In section 6 we relate MDCC to other work.

2. Architecture Overview

MDCC uses a library-centric approach similar to the architectures of DBS3 [5], Megastore [2] or Spanner [8] (as shown in Figure 1). This architecture separates the stateful component of a database system as a distributed record manager. All higher-level functionality (such as query processing and transaction management) is provided through a stateless DB library, which can be deployed at the application server.

As a result, the only stateful component of the architecture, the storage node, is significantly simplified and scalable through standard techniques such as range partitioning, whereas all higher layers of the database can be replicated freely with the application tier because they are stateless. MDCC places storage nodes in geographically distributed data centers, with every node being responsible for one or

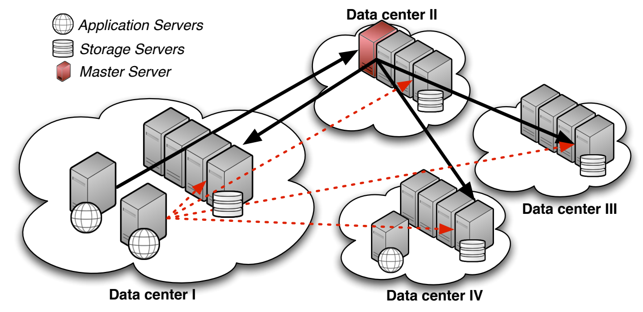

Figure 1. MDCC architecture

more horizontal partitions. Although not required, we assume for the remainder of the paper that every data center contains a full replica of the data, and the data within a single data center is partitioned across machines.

The DB library provides a programming model for transactions and is mainly responsible for coordinating the replication and consistency of the data using MDCC's commit protocol. The DB library also acts as a transaction manager and is responsible to determine the outcome of a transaction. In contrast to many other systems, MDCC supports an individual master per record, which can either be storage nodes or app-server and is responsible to coordinate the updates to a record. This allows the transaction manager to either take over the mastership for a single record and to coordinate the update directly, or to choose a storage node (e.g., the current master) to act on its behalf (black arrows in Figure 1). Furthermore, often it is possible to avoid the master all together, allowing the transaction manager to coordinate the update, without acquiring any mastership (red arrows in Figure 1). This leads to a very flexible architecture in which storage nodes or application servers can act as coordinators, depending on the situation.

In the remaining sections, we concentrate on the MDCC protocol. Other parts of the system, such as load balancing or storage node design are beyond the scope of this paper.

3. The MDCC Protocol

In this section, we describe our new optimistic commit protocol for transactions operating on cross-partition replicated data in the wide-area network. Intra-data center latencies are largely ignored because they are only a few milliseconds compared to hundreds of milliseconds for inter-data center latencies. Our target is a fault-tolerant atomic commit protocol with reduced latency from fewer message rounds by avoiding contacting a master, and high parallelism. We trade-off reducing latency by using more CPU cycles to make sophisticated decisions at each site. We exploit a key observation of real workloads; either conflicts are rare, or many updates commute up to a limit (e.g., add/subtract with a value constraint that the stock should be at least 0).

At its core, the protocol is based on known extensions of Paxos, such as Multi-Paxos [14] and Generalized Paxos [15]. Innovations we introduce enhance these consensus algorithms in order to support transactions on multiple data items without requiring partitioning. In this section, we present a sequence of optimizations, refining from an initial design to the full MDCC protocol. Subsection 3.2 allows multi-record transactions with read committed isolation and no lost updates (see Section 4.1) using Multi-Paxos, with two round-trips of messaging except when the masters for all items are local. Section 3.3 incorporates Fast Paxos, so one round-trip is often possible even without a local master. Then Section 3.4 uses Generalized Paxos to combine commit decisions for transactions that are known to be commutative, and this relies on database techniques that determine state-based commutativity for operations like decrement-subject-to-a-limit. While the component ideas for consensus and for deciding transaction commutativity were known, how we use them for transaction and the combination of them is novel.

3.1 Background: Paxos

In the following we provide some background on the principles of Paxos and how we use it to update a single record.

3.1.1 Classic Paxos

Paxos is a family of quorum-based protocols for achieving consensus on a single value among a group of replicas. It tolerates a variety of failures including lost, duplicated or reordered messages, as well as failure and recovery of nodes. Paxos distinguishes between *clients*, *proposers*, *acceptors* and *learners*. These can be directly mapped to our scenario, where clients are app-servers, proposers are masters, acceptors are storage nodes and all nodes are learners. In the remainder of this paper we use the database terminology of clients, masters and storage nodes. In our implementation, we place masters on storage nodes, but that is not required.

The basic idea in Classic Paxos [14], as applied for replicating a transaction's updates to data, is as follows: Every record has a master responsible for coordinating updates to the record. At the end of a transaction, the app-server sends the update requests to the masters of each the record, as shown by the solid lines in Figure 1. The master informs all storage nodes responsible for the record that it is the master for the next update. It is possible that multiple masters exist for a record, but to make progress, eventually only one master is allowed. The master processes the client request by attempting to convince the storage nodes to *agree* on it. A storage node accepts an update if and only if it comes from the most recent master the node knows of, and it has not already accepted a more recent update for the record.

In more detail, the Classic Paxos algorithm operates in two phases. **Phase 1** tries to establish the mastership for an update for a specific record r. A master P, selects a proposal number m, also referred to as a ballot number or round, higher than any known proposal number and sends a *Phase1a* request with m, to at least a majority of storage nodes responsible for r. The proposal numbers must be unique for each master because they are used to determine the latest request.[2] If a storage node receives a *Phase1a* request greater than any proposal number it has already responded to, it responds with a *Phase1b* message containing m, the highest-numbered update (if any) including its proposal number n, and promises not to accept any future requests less than or equal to m. If P receives responses containing its proposal number m from a majority Q_C of storage nodes, it has been chosen as a master. Now, only P will be able to commit a value for proposal number m.

Phase 2 tries to write a value. P sends an accept request *Phase2a* to all the storage nodes of Phase 1 with the ballot number m and value v. v is either the update of the highest-numbered proposal among the *Phase1b* responses, or the requested update from the client if no *Phase1b* responses contained a value. P must re-send the previously accepted update to avoid losing the possibly saved value. If a storage node receives a *Phase2a* request for a proposal numbered m, it accepts the proposal, unless it has already responded to a *Phase1a* request having a number greater than m, and sends a *Phase2b* message containing m and the value back to P. If the master receives a *Phase2b* message from the majority Q_C of storage nodes for the same ballot number, consensus is reached and the value is learned. Afterwards, the master informs all other components, app-servers and responsible storage nodes, about the success of the update [3].

Note, that Classic Paxos is only able to learn a single value per single instance, which may consist of multiple ballots or rounds. Thus we use one separate Paxos instance per version of the record with the requirement that the previous version has already been chosen successfully.

3.1.2 Multi-Paxos

The Classic Paxos algorithm requires two message rounds to agree on a value, one in Phase 1 and one in Phase 2. If the master is reasonably stable, using Multi-Paxos (multi-decree Synod protocol) makes it possible to avoid Phase 1 by reserving the mastership for several instances [14]. Multi-Paxos is an optimization for Classic Paxos, and in practice, Multi-Paxos is implemented instead of Classic Paxos, to take advantage of fewer message rounds.

We explore this by allowing the proposers to suggest the following meta-data *[StartInstance, EndInstance, Ballot]*. Thus, the storage nodes can vote on the mastership for all instances from *StartInstance* to *EndInstance* with a single ballot number at once. The meta-data also allows for different masters for different instances. This supports custom master policies like round-robin, where *serverA* is the master for instance 1, *serverB* is the master for instance 2, and so on. Storage nodes react to these requests by applying the same semantics for each individual instance as defined in *Phase1b*, but they answer in a single message. The database stores this meta-data including the current version number as part of the record, which enables a separate Paxos instance per record. To support meta-data for inserts, each table stores a default meta-data value for any non-existent records.

Therefore, the default configuration assigns a single master per table to coordinate inserts of new records. Although a potential bottleneck, the master is normally not in the critical path and is bypassed, as explained in section 3.3.

3.2 Transaction Support

The first contribution of MDCC is the extension of Multi-Paxos to support multi-record transactions with read-committed isolation and without the lost-update problem. That is, we ensure atomic durability (all or no updates will persist), detect all write-write conflicts (if two transactions try to update the same record concurrently at most one will succeed), and guarantee that updates only from successful transactions are visible. Guaranteeing higher read consistencies, such as atomic visibility and snapshot isolation, is an orthogonal issue and discussed in Section 4.

We guarantee this consistency level by using a Paxos instance per record to accept an *option* to execute the update, instead of writing the value directly. After the app-server learns the options for all the records in a transaction, it commits the transaction and asynchronously notifies the storage nodes to execute the options. If an option is not yet executed, it is called an *outstanding option*.

3.2.1 The Protocol

As in all optimistic concurrency control techniques, we assume that transactions collect a write-set of records at the end of the transaction, which the protocol then tries to commit. Updates to records create new versions, and are represented in the form $v_{read} \rightarrow v_{write}$, where v_{read} is the version of the record read by the transaction and v_{write} is the new version of the record. This allows MDCC to detect write-write conflicts by comparing the current version of a record with v_{read}. If they are not equal, the record was modified between the read and write and a write-write conflict was encountered. For inserts, the update has a missing v_{read}, indicating that an insert should only succeed if the record doesn't already exist. Deletes work by marking the item as deleted and are handled as normal updates. We further allow there to be only one outstanding option per record and that the update is not visible until the option is executed.

The app-server coordinates the transaction by trying to get the options accepted for all updates. It proposes the options to the Paxos instances running for each record, with the participants being the replicas of the record. Every storage node responds to the app-server with an accept or reject of the option, depending on if v_{read} is valid, similar to validated Byzantine agreement [6]. Hence, the storage nodes make an active decision to *accept* or *reject* the option. This is fundamentally different than existing uses of Paxos (e.g., Consensus on Transaction Commit [11] or Megastore), which re-

[2] To ensure uniqueness we concatenate the requester's ip-address.

[3] It is possible to avoid this delay by sending *Phase2b* messages directly to all involved nodes. As this significantly increases the number of messages, we do not use this optimization.

quire to send a fixed value (e.g., the "final" accept or commit decision) and only decide based on the ballot number if the value should be accepted. The reason why this change does not violate the Paxos assumptions is that we defined at the end of Section 3.1.1 that a new record version can only be chosen if the previous version was successfully determined. Thus, all storage nodes will always make the same abort or commit decision. This prevents concurrent updates, but only per record and not for the entire shard as in Megastore. We will relax this requirement in Section 3.4.

Just as in 2PC, the app-server commits a transaction when it learns all options as accepted, and aborts a transaction when it learns any option as rejected. The app-server learns an option if and only if a majority of storage nodes agrees on the option. In contrast to 2PC we made another important change: MDCC does not allow clients or app-servers to abort a transaction once it has been proposed. Decisions are determined and stored by the distributed storage nodes with MDCC, instead of being decided by a single coordinator with 2PC. This ensures that the commit status of a transaction depends only on the status of the learned options and hence is always deterministic even with failures. Otherwise, the decision of the app-server/client after the prepare has to be reliably stored, which either influences the availability (the reason why 2PC is blocking) or requires an additional round as done by three-phase commit or Consensus on Transaction Commit [11].

If the app-server determines that the transaction is aborted or committed, it informs involved storage nodes through a *Learned* message about the decision. The storage nodes in turn execute the option (make visible) or mark it as rejected. Learning an option is the result of each Paxos instance and thus generates new version-id of the record, whether the option is learned as accepted or rejected. Note, that so far only one option per record can be outstanding at a time as we require the previous instance (version) to be decided.

As a result, it is possible to commit the transaction (commit or abort) in a single round-trip across the data centers if all record masters are local. This is possible because the commit/abort decision of a transaction depends entirely on the learned values and the application server is not allowed to prematurely abort a transaction (in contrast to 2PC or Consensus on Transaction Commit). The *Learned* message to notify the storage nodes about the commit/abort can be asynchronous, but does not influence the correctness, and only affects the possibility of aborts caused by stale reads. By adding transaction support, this design is able to achieve 1 round-trip commits if the master is local, but when the master is not local it requires 2 round-trips, due to the additional communication with the remote master. Communication with a local master is ignored because the latency is negligible (few milliseconds) compared to geographically remote master communication (hundreds of milliseconds).

3.2.2 Avoiding Deadlocks

The described protocol is able to atomically commit multi-record transactions. Without further effort, transactions might cause a deadlock by waiting on each other's options. For example, if two transactions t_1 and t_2 try to learn an option for the same two records r_1 and r_2, t_1 might successfully learn the option for r_1, and t_2 for r_2. Since transactions do not abort without learning at least one of the options as aborted, both transactions are now deadlocked because each transaction waits for the other to finish. We apply a simple pessimistic strategy to avoid deadlocks. The core idea is to relax the requirement that we can only learn a new version if the previous instance is committed. For example, if t_1 learns the option $v_0 \rightarrow v_1$ for record r_1 in one instance as accepted, and t_2 tries to acquire an option $v_0 \rightarrow v_2$ for r_1, t_1 learns the option $v_0 \rightarrow v_1$ as accepted and t_2 learns the option $v_0 \rightarrow v_2$ as rejected in the next Paxos instance. This simple trick causes transaction t_1 to commit and t_2 to abort or in the case of the deadlock as described before, both transactions to abort. The Paxos safety property is still maintained because all storage nodes will make the same decision based on the policy, and the master totally orders the record versions.

3.2.3 Failure Scenarios

Multi-Paxos allows our commit protocol to recover from various failures. For example, a failure of a storage node can be masked by the use of quorums. A master failure can be recovered from by selecting a new master (after some timeout) and triggering Phase 1 and 2 as described previously. Handling app-server failures is trickier, because an app-server failure can cause a transaction to be pending forever as a "dangling transaction". We avoid dangling transactions by including in all of its options a unique transaction-id (e.g., UUIDs) as well as all primary keys of the write-set, and by additionally keeping a log of all learned options at the storage node. Therefore, every option includes all necessary information to reconstruct the state of the corresponding transactions. Whenever an app-server failure is detected by simple timeouts, the state is reconstructed by reading from a quorum of storage nodes for every key in the transaction, so any node can recover the transaction. A quorum is required to determine what was decided by the Paxos instance. Finally, a data center failure is treated simply as each of the nodes in the data center failing. In the future, we might adapt bulk-copy techniques to bring the data up-to-date more efficiently without involving the Paxos protocol (also see [2]).

3.3 Transactions Bypassing the Master

The previous subsection showed how we achieve transactions with multiple updates in one single round-trip, if the masters for all transaction records are in the same data center as the app-server. However, 2 round-trips are required when the masters are remote, or mastership needs to be acquired.

3.3.1 Protocol

Fast Paxos [16] avoids the master by distinguishing between *classic* and *fast* ballots. Classic ballots operate like the classic Paxos algorithm described above and are always the fall-back option. Fast ballots normally use a bigger quorum than classic ballots, but allow bypassing the master. This saves one message round to the master, which may be in a different data center. However, since updates are not serialized by the master, collisions may occur, which can only be resolved by a master using classic ballots.

We use this approach of fast ballots for MDCC. All versions start as an implicitly *fast* ballot number, unless a master changed the ballot number through a *Phase1a* message. This default ballot number informs the storage nodes to accept the next options from any proposer.

Afterwards, any app-server can propose an option directly to the storage nodes, which in turn promise only to accept the first proposed option. Simple majority quorums, however, are no longer sufficient to learn a value and ensure safeness of the protocol. Instead, learning an option without the master requires a *fast* quorum [16]. Fast and classic quorums, are defined by the following requirements: (i) any two quorums must have a non-empty intersection, and (ii) there is non-empty intersection of any three quorums consisting of two fast quorums Q_F^1 and Q_F^2 and a classic quorum Q_C. A typical setting for a replication factor of 5 is a classic quorum size of 3 and a fast quorum size of 4. If a proposer receives an acknowledgment from a fast quorum, the value is safe and guaranteed to be committed. However, if a fast quorum cannot be achieved, collision recovery is necessary. Note, that a Paxos collision is different from a transaction conflict; collisions occur when nodes cannot agree on an option, conflicts are caused by conflicting updates.

To resolve the collision, a new classic ballot must be started with *Phase 1*. After receiving responses from a classic quorum, all potential intersections with a fast quorum must be computed from the responses. If the intersection consists of all the members having the highest ballot number, and all agree with some option v, then v must be proposed next. Otherwise, no option was previously agreed upon, so any new option can be proposed. For example, assume the following messages were received as part of a collision resolution from 4 out of 5 servers with the previously mentioned quorums (notation: *(server-id, ballot number, update)*): (1,3,$v_0 \rightarrow v_1$), (2,4,$v_1 \rightarrow v_2$), (3,4,$v_1 \rightarrow v_3$), (5,4, $v_1 \rightarrow v_2$). Here, the intersection size is 2 and the highest ballot number is 4, so the protocol compares the following intersections:

$$[(2,4,v_1 \rightarrow v_2),(3,4,v_1 \rightarrow v_3)]$$
$$[(3,4,v_1 \rightarrow v_3),(5,4,v_1 \rightarrow v_2)]$$
$$[(2,4,v_1 \rightarrow v_2),(5,4,v_1 \rightarrow v_2)]$$

Only the last intersection has an option in common and all other intersections are empty. Hence, the option $v_1 \rightarrow v_2$ has to be proposed next. More details and the correctness proofs of Fast Paxos can be found in [16].

MDCC uses Fast Paxos to bypass the master for accepting an option, which reduces the number of required message rounds. Per fast ballot, only one option can be learned. However, by combining the idea of Fast Paxos with Multi-Paxos and using the following adjusted ballot-range definitions from Section 3.1.2, *[StartInstance, EndInstance, Fast, Ballot]*, it is possible to pre-set several instances as fast. Whenever a collision is detected, the instance is changed to classic, the collision is resolved and the protocol moves on to the next instance, which can start as either classic or fast. It is important that classic ballot numbers are always higher ranked than fast ballot numbers to resolve collisions and save the correct value. Combined with our earlier observation that a new Paxos instance is started only if the previous instance is stable and learned, this allows the protocol to execute several consecutive fast instances without involving a master.

Without the option concept of Section 3.2 fast ballots would be impossible to use. Without options it would be impossible to make an abort/commit decision without requiring a lock first in a separate message round on the storage servers or some master (e.g., as done by Spanner). This is also the main reason why other existing Paxos commit protocols cannot leverage fast ballots. It can be shown that the correctness of Paxos with options and the deadlock avoidance policy still holds with fast instances, as long as every outstanding version is checked in order. That is because fast instances still determine a total order of operations. Finally, whenever a fast quorum of nodes are unavailable, classic ballots can be used, ensuring the same availability as before.

3.3.2 Fast-Policy

There exists a non-trivial trade-off between fast and classic instances. With fast instances, two concurrent updates might cause a collision requiring another two message rounds for the resolution, whereas classic instances usually require two message rounds, one to either contact the master or acquire the mastership, and one for Phase 2. Hence, fast instances should only be used if conflicts and collisions are rare.

Currently, we use a very simple strategy. The default meta-data for all instances and all records are pre-set to fast with *[0,∞,fast=true,ballot=0]*. As the default meta-data for all records is the same, it does not need to be stored per record. A record's meta-data is managed separately, only when collision resolution is triggered. If we detect a collision, we set the next γ instances (default 100) to classic. After γ transactions, fast instances are automatically tried again.

This simple strategy stays in fast if possible (recall, fast has no upper instance-limit) and in classic when necessary, while probing to go back to fast every γ instances. More advanced models could explicitly calculate the conflict rate and remain as future work.

3.4 Commutative Updates

The design based on Fast Paxos allows many transactions to commit with a single round-trip between data centers. However, whenever there are concurrent updates to a given data item, conflicts will arise and extra messages are needed. MDCC efficiently exploits cases where the updates are commutative, to avoid extra messages by using Generalized

Paxos [15], which is an extension of Fast Paxos. In this section, we show how our novel option concept and the idea to use Paxos on a record instead of a database log-level as described in the previous sections enable us to use Generalized Paxos. Furthermore, in order to support the quite common case of operations on data that are subject to value constraints (e.g., stock should be at least 0), we developed a new demarcation technique for quorums.

3.4.1 The Protocol

Generalized Paxos [15] uses the same ideas as Fast Paxos but relaxes the constraint that every acceptor must agree on the same exact sequence of values/commands. Since some commands may commute with each other, the acceptors only need to agree on sets of commands which are compatible with each other. MDCC utilizes the notion of compatibility to support commutative updates.

Fast commutative ballots are always started by a message from the master. The master sets the record *base value*, which is the latest committed value. Afterwards, any client can propose commutative updates to all storage nodes directly using the same option model as before. In contrast to the previous section, an option now contains commutative updates, which consist of one or more attributes and their respective delta changes (e.g., $decrement(stock, 1)$). If a fast quorum Q_F out of N storage nodes accepts the option, the update is committed. When the updates involved commute, the acceptors can accept multiple proposals in the same ballot and the orderings do not have to be identical on all storage nodes. This allows MDCC to stay in the fast ballot for longer periods of time, bypassing the master and allowing the commit to happen in one message round. More details on Generalized Paxos are given in [15].

3.4.2 Global Constraints

Generalized Paxos is based on commutative operations like increment and decrement. However, many database applications must enforce integrity constraints, e.g., that the stock of an item must be greater than zero. Under a constraint like this, decrements do not commute in general. However, they do have state-based commutativity when the system contains sufficient stock. Thus we allow concurrent processing of decrements while ensuring domain integrity constraints, by requiring storage nodes to only accept an option if the option would not violate the constraint under all permutations of commit/abort outcomes for pending options. For example, given 5 transactions $t_{1...5}$ (arriving in order), each generating an option $[stock = stock - 1]$ with the constraint that $stock \geq 0$ and current $stock$ level of 4, a storage node s will reject t_5 even though the first four options may abort. This definition is analogous to Escrow [19] and guarantees correctness even in the presence of aborts and failures.

Unfortunately, this still does not guarantee integrity constraints, as storage nodes base decisions on local, not global, knowledge. Figure 2 shows a possible message ordering for the above example with 5 storage nodes. Here, clients wait

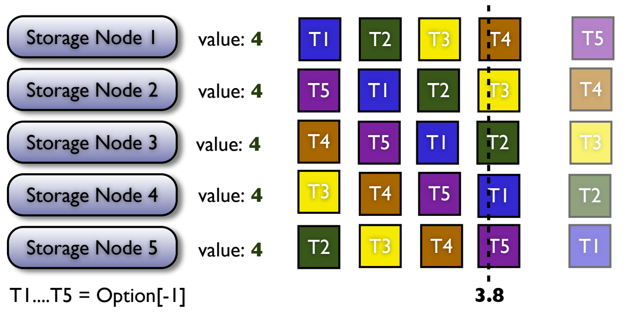

Figure 2. Message order

for Q_F responses (4), and each storage node makes a decision based on its local state. Through different message arrival orders it is possible for all 5 transactions to commit, even though committing them all violates the constraint.

We therefore developed a new *demarcation* based strategy for quorum systems. Our demarcation technique is similar to the earlier technique [3] in that they both use local limits, but is used in different scenarios. Original demarcation uses limits to safely update distributed values, where MDCC uses limits for quorum replicated values.

Without loss of generality, we assume a constraint of value at least 0 and that all updates are decrements. Let N be the replication factor (number storage nodes), X be the base value for some attribute and δ_i be the decrement amount of transaction t_i for the attribute. If we consider every replicated base value X as a *resource*, the total number of *resources* in the system is $N \cdot X$. In order to commit an update, Q_F storage nodes must accept the update, so every successful transaction t_i reduces the resources in the system by at least $Q_F \cdot \delta_i$. If we assume m successful transactions where $\sum_{i=1}^{m} \delta_i = X$, this means the attribute value reached 0, and the total amount of resources would reduce by at least $Q_F \cdot \sum_{i=1}^{m} \delta_i = Q_F \cdot X$. Even though the integrity constraint forbids any more transactions, it is still possible that the system still has $(N - Q_F) \cdot X$ resources remaining due to failures, lost, or out-of-order messages.

The worst case is where the remaining resources are equally distributed across all the storage nodes (otherwise, at least one of the storage nodes would start to reject options earlier). The remaining resources $(N - Q_F) \cdot X$ are divided evenly among the N storage nodes to derive a lower limit to guarantee the value constraint. Storage nodes must reject an option if it would cause the value to fall below:

$$L = \frac{N - Q_F}{N} \cdot X$$

This limit L is calculated with every new base value. When options in fast ballots are rejected because of this limit, the protocol handles it as a collision, resolves it by switching to classic ballots, and writes a new base value and limit L.

3.4.3 MDCC Pseudocode

The complete MDCC protocol as pseudocode is listed in algorithms 1, 2, and 3, while table 1 defines the used symbols

Symbols	Definitions
a	an acceptor
l	a leader
up	an update
$\omega(up, \_)$	an option for an update, with ✓or ✗
✓/ ✗	acceptance / rejection
m	ballot number
$val_a[i]$	cstruct at ballot i at acceptor a
bal_a	$\max\{k \mid val_a[k] \neq none\}$
val_a	cstruct at bal_a at acceptor a
$mbal_a$	current ballot number at acceptor a
$ldrBal_l$	ballot number at leader l
$maxTried_l$	cstructs proposed by leader l
Q	a quorum of acceptors
$Quorum(k)$	all possible quorums for ballot k
$learned$	cstruct learned by a learner
$\sqcap$	greatest lower bound operator
$\sqcup$	least upper bound operator
$\sqsubseteq$	partial order operator for cstructs
$val\bullet\, \omega(up, \_)$	appends option $\omega(up, \_)$ to cstruct val

Table 1. Definition of symbols for MDCC pseudocode.

Algorithm 1 Pseudocode for MDCC

```
 1: procedure TRANSACTIONSTART                                ▷ Client
 2:     for all $up \in tx$ do
 3:         run SENDPROPOSAL($up$)
 4:     wait to learn all update options
 5:     if $\forall up \in tx$ : learned $\omega(up, ✓)$ then
 6:         send $Visibility[up, ✓]$ to $Acceptors$
 7:     else
 8:         send $Visibility[up, ✗]$ to $Acceptors$
 9: procedure SENDPROPOSAL($up$)                              ▷ Proposer
10:     if classic ballot then
11:         send $Propose[up]$ to $Leader$
12:     else
13:         send $Propose[up]$ to $Acceptors$
14: procedure LEARN($up$)                                     ▷ Learner
15:     collect $Phase2b[m, val_a]$ messages from $Q$
16:     if $\forall a \in Q : v \sqsubseteq val_a$ then
17:         $learned \leftarrow learned \sqcup v$
18:     if $\omega(up, \_) \notin learned$ then
19:         send $StartRecovery[]$ to $Leader$
20:         return
21:     if classic ballot then
22:         move on to next instance
23:     else
24:         $isComm \leftarrow up$ is $CommutativeUpdate[delta]$
25:         if $\omega(up, ✗) \in learned \wedge isComm$ then
26:             send $StartRecovery[]$ to $Leader$
```

and variables. For simplicity, we do not show how liveness is guaranteed. The remainder of this section sketches the algorithm by focusing on how the different pieces from the previous subsections work together.

The app-server or client starts the transaction by sending proposals for every update on line 3. After learning the status of options of all the updates (lines 14-26), the app-server sends visibility messages to "execute" the options on lines 5-8, as described in Section 3.2.1. While attempting to learn options, if the app-server does not learn the status of an option (line 18), it will initiate a recovery. Also, if the app-server learns a commutative option as rejected during a fast ballot (line 25), it will notify the master to start recovery.

Algorithm 2 Pseudocode for MDCC - Leader l

```
27: procedure RECEIVELEADERMESSAGE($msg$)
28:     switch msg do
29:         case $Propose[up]$ :
30:             run PHASE2ACLASSIC($up$)
31:         case $Phase1b[m, bal, val]$ :
32:             if received messages from $Q$ then
33:                 run PHASE2START($m, Q$)
34:         case $StartRecovery[]$ :
35:             $m \leftarrow$ new unique ballot number greater than $m$
36:             run PHASE1A($m$)
37: procedure PHASE1A($m$)
38:     if $m > ldrBal_l$ then
39:         $ldrBal_l \leftarrow m$
40:         $maxTried_l \leftarrow none$
41:         send $Phase1a[m]$ to $Acceptors$
42: procedure PHASE2START($m, Q$)
43:     $maxTried_l \leftarrow$ PROVEDSAFE($Q, m$)
44:     if new update to propose exists then
45:         run PHASE2ACLASSIC($up$)
46: procedure PHASE2ACLASSIC($up$)
47:     $maxTried_l \leftarrow maxTried_l \bullet\, \omega(up, \_)$
48:     send $Phase2a[ldrBal_l, maxTried_l]$ to $Acceptors$
49: procedure PROVEDSAFE($Q, m$)
50:     $k \equiv \max\{i \mid (i < m) \wedge (\exists a \in Q : val_a[i] \neq none)\}$
51:     $\mathcal{R} \equiv \{R \in Quorum(k) \mid$
            $\forall a \in Q \cap R : val_a[k] \neq none\}$
52:     $\gamma(R) \equiv \sqcap\{val_a[k] \mid a \in Q \cap R\}$, for all $R \in \mathcal{R}$
53:     $\Gamma \equiv \{\gamma(R) \mid R \in \mathcal{R}\}$
54:     if $R = \emptyset$ then
55:         return $\{val_a[k] \mid (a \in Q) \wedge (val_a]k \ \neq none)\}$
56:     else
57:         return $\{\sqcup\Gamma\}$
```

Learning a rejected option for commutative updates during fast ballots is an indicator of violating the quorum demarcation limit, so a classic ballot is required to update the limit.

When accepting new options, the storage nodes must evaluate the compatibility of the options and then accept or reject it. The compatibility validation is shown in lines 83-99. If the new update is not commutative, the storage node compares the read version of the update to the current value to determine the compatibility, as shown in lines 86-92. For new commutative updates, the storage node computes the quorum demarcation limits as described in section 3.4.2, and determines if any combination of the pending commutative options violate the limits (lines 93-99). When a storage node receives a visibility message for an option, it executes the option in order to make the update visible, on line 103.

4. Consistency Guarantees

MDCC ensures atomicity (i.e., either all updates in a transaction persist or none) and that two concurrent write-conflicting update transactions do not both commit. By combining the protocol with different read-strategies it is possible to guarantee various degrees of consistency.

4.1 Read Committed without Lost Updates

MDCC's default consistency level is *read committed*, but without the lost update problem [4]. Read committed isolation prevents dirty reads, so no transactions will read any

Algorithm 3 Pseudocode for MDCC - Acceptor a

```
58: procedure ReceiveAcceptorMessage($msg$)
59:     switch $msg$ do
60:         case $Phase1a[m]$ :
61:             run Phase1b($m$)
62:         case $Phase2a[m, v]$ :
63:             run Phase2bClassic($m, v$)
64:         case $Propose[up]$ :
65:             run Phase2bFast($up$)
66:         case $Visibility[up, status]$ :
67:             run ApplyVisibility($up, status$)
68: procedure Phase1b($m$)
69:     if $mbal_a < m$ then
70:         $mbal_a \leftarrow m$
71:         send $Phase1b[m, bal_a, val_a]$ to $Leader$
72: procedure Phase2bClassic($m, v$)
73:     if $bal_a \leq m$ then
74:         $bal_a \leftarrow m$
75:         $val_a \leftarrow v$
76:         SetCompatible($val_a$)
77:         send $Phase2b[m, val_a]$ to $Learners$
78: procedure Phase2bFast($up$)
79:     if $bal_a = mbal_a$ then
80:         $val_a \leftarrow val_a \bullet \omega(up, \_)$
81:         SetCompatible($val_a$)
82:         send $Phase2b[m, val_a]$ to $Learners$
83: procedure SetCompatible($v$)
84:     for all new options $\omega(up, \_)$ in $v$ do
85:         switch $up$ do
86:             case $PhysicalUpdate[v_{read}, v_{write}]$ :
87:                 $validRead \leftarrow v_{read}$ matches current value
88:                 $validSingle \leftarrow$ no other pending options exist
89:                 if $validRead \wedge validSingle$ then
90:                     set option to $\omega(up, ✓)$
91:                 else
92:                     set option to $\omega(up, ✗)$
93:             case $CommutativeUpdate[delta]$ :
94:                 $\mathcal{U} \leftarrow$ upper quorum demarcation limit
95:                 $\mathcal{L} \leftarrow$ lower quorum demarcation limit
96:                 if any option combinations violate $\mathcal{U}$ or $\mathcal{L}$ then
97:                     set option to $\omega(up, ✗)$
98:                 else
99:                     set option to $\omega(up, ✓)$
100: procedure ApplyVisibility($up, status$)
101:     update $\omega(up, \_)$ in $val_a$ to $\omega(up, status)$
102:     if $status = ✓$ then
103:         apply $up$ to make update visible
```

other transaction's uncommitted changes. The lost update problem occurs when transaction t_1 first reads a data item X, then one or more other transactions write to the same data item X, and finally t_1 writes to data item X. The updates between the read and write of item X by t_1 are "lost" because the write by t_1 overwrites the value and loses the previous updates. MDCC guarantees read committed isolation by only reading committed values and not returning the value of uncommitted options. Lost updates are prevented by detecting every write-write conflict between transactions.

Currently, Microsoft SQL Server, Oracle Database, IBM DB2 and PostgreSQL all use read committed isolation by default. We therefore believe that MDCC's default consistency level is sufficient for a wide range of applications.

4.2 Staleness & Monotonicity

Reads can be done from any storage node and are guaranteed to return only committed data. However, by just reading from a single node, the read might be stale. For example, if a storage node missed updates due to a network problem, reads might return older data. Reading the latest value requires reading a majority of storage nodes to determine the latest stable version, making it an expensive operation.

In order to allow up-to-date reads with classic rounds, we can leverage techniques from Megastore [2]. A simple strategy for up-to-date reads with fast rounds is to ensure that a special pseudo-master storage node is always part of the quorum of Phases 1 and 2 and to switch to classic whenever the pseudo-master cannot be contacted. The techniques from Megastore can apply for the pseudo-master to guarantee up-to-date reads in all data centers. The same strategy can guarantee monotonic reads such as *repeatable reads* or *read your writes*, but can be further relaxed by requiring only the local storage node to always participate in the quorum.

4.3 Atomic Visibility

MDCC provides atomic durability, meaning either all or none of the operations of the transaction are durable, but it does not support atomic visibility. That is, some of the updates of a committed transaction might be visible whereas other are not. Two-phase commit also only provides atomic durability, not visibility unless it is combined with other techniques such as two-phase locking or snapshot isolation. The same is true for MDCC. For example, MDCC could use a read/write locking service per data center or snapshot isolation as done in Spanner [8] to achieve atomic visibility.

4.4 Other Isolation Levels

Finally, MDCC can support higher levels of isolation. In particular, Non-monotonic Snapshot Isolation (NMSI) [22] or Spanner's [8] snapshot isolation through synchronized clocks are natural fits for MDCC. Both would still allow fast commits while providing consistent snapshots. Furthermore, as we already check the write-set for transactions, the protocol could easily be extended to also consider read-sets, allowing us to leverage optimistic concurrency control techniques and ultimately provide full serializability.

5. Evaluation

We implemented a prototype of MDCC on top of a distributed key/value store across five different data centers using the Amazon EC2 cloud. To demonstrate the benefits of MDCC, we use TPC-W and micro-benchmarks to compare the performance characteristics of MDCC to other transactional and other non-transactional, eventually consistent protocols. This section describes the benchmarks, experimental setup, and our findings.

5.1 Experimental Setup

We implemented the MDCC protocol in Scala, on top of a distributed key/value store, which used Oracle BDB Java

Edition as a persistent storage engine. We deployed the system across five geographically diverse data centers on Amazon EC2: US West (N. California), US East (Virginia), EU (Ireland), Asia Pacific (Singapore), and Asia Pacific (Tokyo). Each data center has a full replica of the data, and within a data center, each table is range partitioned by key, and distributed across several storage nodes as m1.large instances (4 cores, 7.5GB memory). Therefore, every horizontal partition, or shard, of the data is replicated five times, with one copy in each data center. Unless noted otherwise, all clients issuing transactions are evenly distributed across all five data centers, on separate m1.large instances.

5.2 Comparison with other Protocols

To compare the overall performance of MDCC with alternative designs we used TPC-W, a transactional benchmark that simulates the workload experienced by an e-commerce web server. TPC-W defines a total of 14 web interactions (WI), each of which are web page requests that issue several database queries. In TPC-W, the only transaction which is able to benefit from commutative operations is the product-buy request, which decreases the stock for each item in the shopping cart while ensuring that the stock never drops below 0 (otherwise, the transaction should abort). We implemented all the web interactions using our own SQL-like language but forego the HTML rendering part of the benchmark to focus on the database part. TPC-W defines that these WI are requested by emulated browsers, or clients, with a wait-time between requests and varying browse-to-buy ratios. In our experiments, we forego the wait-time between requests and only use the most write-heavy profile to stress the system. It should also be noted that read-committed is sufficient for TPC-W to never violate its data consistency.

In these experiments, the MDCC prototype uses fast ballots with commutativity where possible (reverting to classic after too many collisions have occurred as described in Section 3.3.2). For comparison, we also implemented forms of some other replica management protocols in Scala, using the same distributed store, and accessed by the same clients.

Quorum Writes (QW). The quorum writes protocol (QW) is the standard for most eventually consistent systems and is implemented by simply sending all updates to all involved storage nodes then waiting for responses from *quorum* nodes. We used two different configurations for the write quorum: quorum of size 3 out of 5 replicas for each record (we call this QW-3), and quorum of size 4 out of 5 (QW-4). We use a read-quorum of 1 to access only the local replica (the fastest read configuration). It is important to note that the quorum writes protocol provides no isolation, atomicity, or transactional guarantees.

Two-Phase Commit (2PC). Two-phase commit (2PC) is still considered the standard protocol for distributed transactions. 2PC operates in two phases. In the first phase, a transaction manager tries to prepare all involved storage nodes to commit the updates. If all relevant nodes prepare successfully, then in the second phase the transaction manager sends a commit to all storage nodes involved; otherwise it sends an abort. Note, that 2PC requires all involved storage nodes to respond and is not resilient to single node failures.

Megastore*. We were not able to compare MDCC directly against the Megastore system because it was not publicly available. Google App Engine uses Megastore, but the data centers and configuration are unknown and out of our control. Instead, we simulated the underlying protocol as described in [2] to compare it with MDCC; we do this as a special configuration of our system, referred to as Megastore*. In [2], the protocol is described mainly for transactions within a partition. The paper states that 2PC is used across partitions with looser consistency semantics but omits details on the implementation and the authors discourage of using the feature because of its high latency. Therefore, for experiments with Megastore*, we placed all data into a single entity group to avoid transactions which span multiple entity groups. Furthermore, Megastore only allows that one write transaction is executed at any time (all other competing transactions will abort). As this results in unusable throughput for TPC-W, we include an improvement from [20] and allow non-conflicting transactions to commit using a subsequent Paxos instance. We also relaxed the read consistency to read-committed enabling a fair comparison between Megastore* and MDCC. Finally, we play in favor of Megastore* placing all clients and masters in one data center (US-West), to allow all transactions to commit with a single round-trip.

5.2.1 TPC-W Write Response Times

To evaluate MDCC's main goal, reducing the latency, we ran the TPC-W workload with each protocol. We used a TPC-W scale factor of 10,000 items, with the data being evenly ranged partitioned and replicated to four storage nodes per data center. 100 evenly geo-distributed clients (on separate machines) each ran the TPC-W benchmark for 2 minutes, after a 1 minute warm-up period.[4]

Figure 3 shows the cumulative distribution functions (CDF) of the response times of committed write transactions for the different protocols. Note that the horizontal (time) axis is a log-scale. We only report the response times for write transactions as read transactions were always local for all configurations and protocols. The two dashed lines (QW-3, QW-4) are non-transactional, eventually consistent protocols, and the three solid lines (MDCC, 2PC, Megastore*) are transactional, strongly consistent protocols.

Figure 3 shows that the non-transactional protocol QW-3 has the fastest response times, followed by QW-4, then the transactional systems, of which MDCC is fastest, then 2PC,

[4] In a separate experiment, we studied for each protocol the effect of using different numbers of clients and storage nodes, and found that 4 storage nodes per data center for 100 clients has the best utilization before the latency starts increasing because of queuing/network effects. For brevity we omit this experiment.

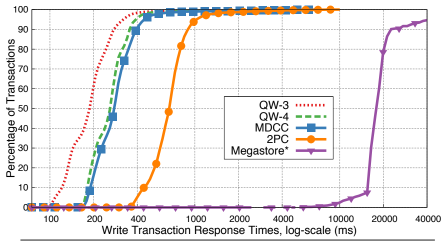

Figure 3. TPC-W write transaction response times CDF

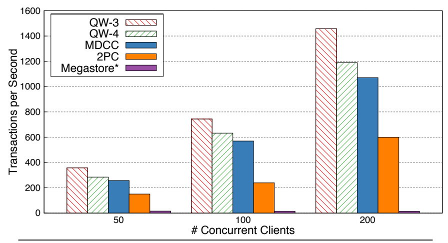

Figure 4. TPC-W transactions per second scalability

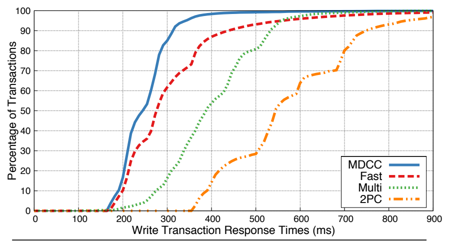

Figure 5. Micro-benchmark response times CDF

and finally Megastore* with slowest times. The median response times are: 188ms for QW-3, 260ms for QW-4, 278ms for MDCC, 668ms for 2PC, and 17,810ms for Megastore*.

Since MDCC uses fast ballots whenever possible, MDCC often commits transactions from any data center with a single round-trip to a quorum size of 4. This explains why the performance of MDCC is similar to QW-4. The difference between QW-3 and QW-4 arises from the need to wait longer for the 4th response, instead of returning after the 3rd response. There is an impact from the non-uniform latencies between different data centers, so the 4th is on average farther away than the 3rd, and there is more variance when waiting for more responses. Hence, an administrator might choose to configure a MDCC-like system to use classic instances with a local master, if it is known that the workload has most requests being issued from the same data center (see Section 5.3.3 for an evaluation).

MDCC reduces per transaction latencies by 50% compared to 2PC because it commits in one round-trip instead of two. Most surprisingly, however, is the orders of magnitude improvement over Megastore*. This can be explained since Megastore* must serialize all transactions with Paxos (it executes one transaction at a time) and heavy queuing effects occur. This queuing effect happens because of the moderate load, but it is possible to avoid the effect by reducing the load or allowing multiple transactions to commit for one commit log record. If so, performance would be similar to our classical Paxos configuration discussed in Section 5.3.1. Even without queuing effects, Megastore* would require an additional round-trip to the master for non-local transactions. Since Google's Spanner [8] uses 2PC across Paxos groups, and each Paxos group requires a master, we expect Spanner to behave similarly to the 2PC data in figure 3.

We conclude from this experiment that MDCC achieves our main goal: it supports strongly consistent transactions with latencies similar to non-transactional protocols which provide weaker consistency, and is significantly faster than other strongly consistent protocols (2PC, Megastore*).

5.2.2 TPC-W Throughput and Transaction Scalability

One of the intended advantages of cloud-based storage systems is the ability to scale out without affecting performance. We performed a scale-out experiment using the same setting as in the previous section, except that we varied the scale to (50 clients, 5,000 items), (100 clients, 10,000 items), and (200 clients, 20,000 items). For each configuration, we fixed the amount of data per storage node to a TPC-W scale-factor of 2,500 items and scaled the number of nodes accordingly (keeping the ratio of clients to storage nodes constant). For the same arguments as before, we used a single partition for Megastore* to avoid cross-partition transactions.

Figure 4 shows the results of the throughput measurements of the various protocols. We see that the QW protocols have the lowest message and CPU overhead and therefore the highest throughput, with the MDCC throughput not far behind. For 200 concurrent clients, the MDCC throughput was within 10% of the throughput of QW-4. The experiments also demonstrate that MDCC has higher throughput compared to the other strongly consistent protocols, 2PC and Megastore*. The throughput for 2PC is significantly lower, mainly due to the additional waiting for the second round.

As expected, the QW protocols scale almost linearly; we see similar scaling for MDCC and 2PC. The Megastore* throughput is very low and does not scale out well, because all transactions are serialized for the single partition. This low throughput and poor scaling matches the results in [13] for Google App Engine, a public service using Megastore. In summary, figure 4 shows that MDCC provides strongly consistent cross data center transactions with throughput and scalability similar to eventually consistent protocols.

5.3 Exploring the Design Space

We use our own micro-benchmark to independently study the different optimizations within the MDCC protocol, and how it is sensitive to workload features. The data for the micro-benchmark is a single table of items, with randomly chosen stock values and a constraint on the stock attribute that it has to be at least 0. The benchmark defines a simple buy transaction, that chooses 3 random items uniformly, and for each item, decrements the stock value by an amount between 1 and 3 (a commutative operation). Unless stated otherwise, we use 100 geo-distributed clients, and a pre-populated product table with 10,000 items sharded on 2 storage nodes per data center.

5.3.1 Response Times

To study the effects of the different design choices in MDCC, we ran the micro-benchmark with 2PC and various MDCC configurations: *MDCC:* our full featured protocol, *Fast:* without the commutative update support, and *Multi:* all instances being Multi-Paxos (a stable master can skip

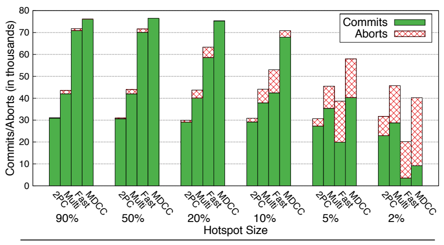

Figure 6. Commits/aborts for varying conflict rates

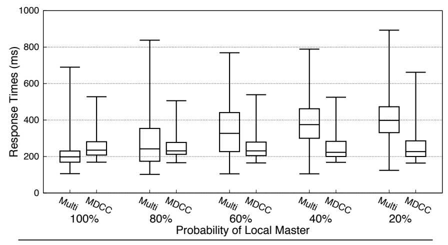

Figure 7. Response times for varying master locality

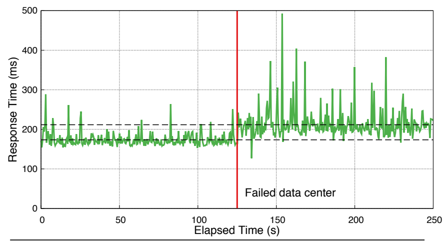

Figure 8. Time-series of response times during failure

Phase 1). The experiment ran for 3 minutes after a 1 minute warm-up. Figure 5 shows the cumulative distribution functions (CDF) of response times of the successful transactions.

The median response times are: 245ms for MDCC, 276ms for Fast, 388ms for Multi, and 543ms for 2PC. 2PC is the slowest because it must use two round-trips across data centers and has to wait for responses from all 5 data centers. For Multi, with the masters being uniformly distributed across all the data centers, most of the transactions (about $4/5$ of them) require a message round-trip to contact the master, so two round-trips across data centers are required, similar to 2PC. In contrast to 2PC, Multi needs responses from only 3 of 5 data centers, so the response times are improved. The response times for Multi would also be observed for Megastore* if no queuing effects are experienced. Megastore* experiences heavier queuing effects because all transactions in the single entity group are serialized, but with Multi, only updates per record are serialized.

The main reason for the low latencies for MDCC is the use of fast ballots. Both MDCC and Fast return earlier than the other protocols, because they often require one round-trip across data centers and do not need to contact a master, like Multi. The improvement from Fast to MDCC is because commutative updates reduce conflicts and thus collisions, so MDCC can continue to use fast ballots and avoid resolving collisions as described in Section 3.3.2.

5.3.2 Varying Conflict Rates

MDCC attempts to take advantage of situations when conflicts are rare, so we study how the MDCC commit performance is affected by different conflict rates. We therefore defined a *hot-spot* area and modified the micro-benchmark to access items in the *hot-spot* area with 90% probability, and accesses the *cold-spot* portion of the data with the remaining 10% probability. By adjusting the size of the *hot-spot* as a percentage of the data, we alter the conflict rates in the access patterns. For example, when the *hot-spot* is 90% of the data, the access pattern is essentially uniformly at random, since 90% of the accesses will go to 90% of the data.

The Multi system uses masters to serialize transactions, so Paxos conflicts occur when there are multiple potential masters, which should be rare. Multi will simply abort transactions when the read version is not the same as the version in the storage node (indicating a write-write transaction conflict), to keep the data consistent, making Paxos collisions independent of transaction conflicts. On the other hand, for Fast Paxos, collisions are related to transaction conflicts, as a collision/conflict occurs whenever a quorum size of 4 does not agree on the same decision. When this happens, collision resolution must be triggered, which eventually switches to a classic ballot, which will take at least 2 more message rounds. MDCC is able to improve on it by exploring commutativity, but still might cause an expensive collision resolution whenever the quorum demarcation integrity constraint is reached, as described in Section 3.4.2.

Figure 6 shows the number of commits and aborts for different designs, for various *hot-spot* sizes. When the *hot-spot* size is large, the conflict rate is low, so all configurations do not experience many aborts. MDCC commits the most transactions because it does not abort any transactions. Fast commits slightly fewer, because it has to resolve the collisions which occur when different storage nodes see updates in different orders. Multi commits far fewer transactions because most updates have to be sent to a remote master, which increases the response times and decreases the throughput.

As the *hot-spot* decreases in size, the conflict rate increases because more of the transactions access smaller portions of the data. Therefore, more transactions abort as the *hot-spot* size decreases. When the *hot-spot* is at 5%, the Fast commits fewer transactions than Multi. This can be explained by the fact that Fast needs 3 round-trips to ultimately resolve conflicting transactions, whereas Multi usually uses 2 rounds. When the *hot-spot* is at 2%, the conflict rate is very high, so both Fast and MDCC perform very poorly compared to Multi. The costly collision resolution for fast ballots is triggered so often, that many transactions are not able to commit. We conclude that fast ballots can take advantage of master-less operation as long as the conflict rate is not very high. When the conflict rate is too high, a master-based approach is more beneficial and MDCC should be configured as Multi. Exploring policies to automatically determine the best strategy remains as future work.

5.3.3 Data access locality

Classic ballots can save message trips in the situation when the client requests have affinity for data with a local master. To explore the tradeoff between fast and classic ballots, we modified the benchmark to vary the choice of data items within each transaction, so a given percentage will access records with local masters. At one extreme, 100% of transactions choose their items only from those with a local master; at the other, 20% of the transactions choose items with a local master (items are chosen uniformly at random).

Figure 7 shows the boxplots of the latencies of Multi and MDCC for different master localities. When all the masters are local to the clients, then Multi will have lower response times than MDCC, as shown in the graph for 100%. However, as updates access more remote masters, response times for Multi get slower and also increase in variance, but MDCC still maintains the same profile. Even when 80% of the updates are local, the median Multi response time (242ms) is slower than the median MDCC response time (231ms). Our MDCC design is targeted at situations without particular access locality, and Multi only out-performs MDCC when the locality is near 100%. Interesting to note is, that the max latency of the Multi configuration is higher than for full MDCC. This can be explained by the fact that some transactions have to queue until the previous transaction finishes, whereas MDCC normally operates in fast ballots and everything is done in parallel.

5.3.4 Data Center Fault Tolerance

We also experimented with various failure scenarios. Here, we only report on a simulated full data center outage while running the micro-benchmark, as other failures, such as a failure of a transaction coordinator mainly depend on set time-outs. We started 100 clients issuing write transactions from the US-West data center. About two minutes into the experiment, we simulated a failed US-East data center, which is the data center closest to US-West. We simulated the failed data center by preventing the data center from receiving any messages. Since US-East is closest to US-West, "killing" US-East forces the protocol to tolerate the failure. We recorded all the committed transaction response times and plotted the time series graph, in figure 8.

Figure 8 shows the transaction response times before and after failing the data center, which occurred at around 125 seconds into the experiment (solid vertical line). The average response time of transactions before the data center failure was 173.5 ms and the average response time of transactions after the data center failure was 211.7 ms (dashed horizontal lines). The MDCC system clearly continues to commit transactions seamlessly across the data center failure. The average transaction latencies increase after the data center failure, but that is expected behavior, since the MDCC commit protocol uses quorums and must wait for responses from another data center, potentially farther away. The same argument also explains the increase in variance. If the data center comes up again (not shown in the figure), only records which have been updated during the failure, would still be impacted by the increased latency until the next update or a background process brought them up-to-date. These results show MDCC's resilience against data center failures.

6. Related Work

There has been recent interest in scalable geo-replicated datastores. Several recent proposals use Paxos to agree on log-positions similar to state-machine replication. For example, Megastore [2] uses Multi-Paxos to agree on log-positions to synchronously replicate data across multiple data centers (typically five data centers). Google Spanner [8] is similar, but uses synchronized timestamps to provide snapshot isolation. Furthermore, other state-machine techniques for WANs such as Mencius [18] or HP/CoreFP [10] could also be used for wide-area database log replication. All these systems have in common that they significantly limit the throughput by serializing all commit log records and thus, implicitly executing only one transaction at a time. As a consequence, they must partition the data in small shards to get reasonable performance. Furthermore, these protocols rely on an additional protocol (usually 2PC, with all its disadvantages) to coordinate any transactions that access data across shards. Spanner and Megastore are both master-based approaches, and introduce an additional message delay for remote clients. Mencius [18] uses a clever token passing scheme to not rely on a single static master. This scheme however is only useful on large partitions and is not easy applicable on finer grained replication (i.e., on a record level). Like MDCC, HP/CoreFP avoids a master, and improves on the cost of Paxos collisions by executing classic and fast rounds concurrently. Their hybrid approach could easily be integrated into MDCC but requires significant more messages, which is worrisome for real world applications. In summary, MDCC improves over these approaches, by not requiring partitioning, natively supporting transactions as a single protocol, and/or avoiding a master when possible.

Paxos-CP [20] improves Megastore's replication protocol by allowing non-conflicting transactions to move on to subsequent commit log-positions and combining commits into one log-position, significantly increasing the fraction of committed transactions. The ideas are very interesting but their performance evaluation does not show that it removes the log-position bottleneck (they only execute 4 transactions per second). Compared to MDCC, they require an additional master-based single node transaction conflict detection, but are able to provide stronger serializability guarantees.

A more fine-grained use of Paxos was explored in Consensus on Commit [11], to reliably store the resource manager decision (commit/abort) to make it resilient to failures. In theory, there could be a resource manager per record. However, they treat data replication as an orthogonal issue and require that a single resource manager makes the decision (commit/abort), whereas MDCC assumes this decision is made by a quorum of storage nodes. Scalaris [24] applied consensus on commit to DHTs, but cannot leverage Fast Paxos as MDCC does. Our use of record versioning with Paxos has some commonalities with multi-OHS [1], a protocol to construct Byzantine fault-tolerant services, which also supports atomic updates to objects. However, multi-OHS only guarantees atomic durability for a single server (not across shards) and it is not obvious how to use the protocol for distributed transactions or commutative operations. The authors of Spanner describe that they tried a more fine-grained use of Paxos by running multiple instances per shard, but that they eventually gave up because of the complexity. In this paper, we showed it is possible and presented a system that uses multiple Paxos instances to execute transactions without requiring partitioning.

Other geo-replicated datastores include PNUTS [7], Amazon's Dynamo [9], Walter [25] and COPS [17]. These use asynchronous replication, with the risk of violating consistency and losing data in the event of major data center failures. Walter [25] also supports a second mode with stronger consistency guarantees between data centers, but this relies on 2PC and always requires two round-trip times.

Use of optimistic atomic broadcast protocols for transaction commit were proposed in [12, 21]. That technique does not explore commutativity and often has considerably longer response-times in the wide-area network because of the wait-time for a second verify message before the commit is final.

Finally, our demarcation strategy for quorums was inspired by [3], which proposed for the first time to use extra limits to ensure value constraints.

7. Conclusion

The long and fluctuating latencies between data centers make it hard to support highly available applications that can survive data center failures. Reducing the latency for transactional commit protocols is the goal of this paper.

We proposed MDCC as a new approach for synchronous replication in the wide-area network. MDCC's commit protocol is able to tolerate data center failures without compromising consistency, at a similar cost to eventually consistent protocols. It requires only one message round-trip across data centers in the common case. In contrast to 2PC, MDCC is an optimistic commit protocol and takes advantage of situations when conflicts are rare and/or when updates commute. It is the first protocol applying the ideas of Generalized Paxos to transactions that may access records spanning partitions. We also presented the first technique to guarantee domain integrity constraints in a quorum-based system.

In the future, we plan to explore more optimizations of the protocol, such as determining the best strategy (fast or classic) based on client locality, or batching techniques that reduce the message overhead. Supporting other levels of read isolation, like PSI, is an interesting future avenue.

MDCC provides a transactional commit protocol for the wide-area network which achieves strong consistency at a similar cost to eventually consistent protocols.

8. Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Robbert van Renesse, for all the helpful comments and suggestions. This research is supported in part by NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, Blue Goji, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Samsung, Splunk, VMware and Yahoo!.

References

[1] M. Abd-El-Malek et al. Fault-Scalable Byzantine Fault-Tolerant Services. In *Proc. of SOSP*, 2005.

[2] J. Baker et al. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, 2011.

[3] D. Barbará and H. Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems. *VLDB J.*, 3(3), 1994.

[4] H. Berenson et al. A Critique of ANSI SQL Isolation Levels. In *Proc. of SIGMOD*, 1995.

[5] M. Brantner et al. Building a database on S3. In *Proc. of SIGMOD*, 2008.

[6] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and Efficient Asynchronous Broadcast Protocols. In *Advances in Cryptology-Crypto 2001*, 2001.

[7] B. F. Cooper et al. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1, 2008.

[8] J. C. Corbett et al. Spanner: Google's Globally-Distributed Database. In *Proc. of OSDI*, 2012.

[9] G. DeCandia et al. Dynamo: Amazon's Highly Available Key-Value Store. In *Proc. of SOSP*, 2007.

[10] D. Dobre, M. Majuntke, M. Serafini, and N. Suri. HP: Hybrid paxos for WANs. In *Dependable Computing Conference*, 2010.

[11] J. Gray and L. Lamport. Consensus on Transaction Commit. *TODS*, 31, 2006.

[12] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proc. of ICDCS*, 1999.

[13] D. Kossmann, T. Kraska, and S. Loesing. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In *Proc. of SIGMOD*, 2010.

[14] L. Lamport. The Part-Time Parliament. *TOCS*, 16(2), 1998.

[15] L. Lamport. Generalized Consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[16] L. Lamport. Fast Paxos. *Distributed Computing*, 19, 2006.

[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 *Proc. of SOSP*, 2011.

[18] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *Proc. of OSDI*, 2008.

[19] P. E. O'Neil. The Escrow Transactional Method. *TODS*, 11, 1986.

[20] S. Patterson et al. Serializability, not Serial: Concurrency Control and Availability in Multi-Datacenter Datastores. *Proc. VLDB Endow.*, 5(11), 2012.

[21] F. Pedone. Boosting System Performance with Optimistic Distributed Protocols. *IEEE Computer*, 34(12), 2001.

[22] M. Saeida Ardekani, P. Sutra, N. Preguiça, and M. Shapiro. Non-Monotonic Snapshot Isolation. Research Report RR-7805, INRIA, 2011.

[23] E. Schurman and J. Brutlag. Performance Related Changes and their User Impact. Presented at Velocity Web Performance and Operations Conference, 2009.

[24] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: Reliable Transactional P2P Key/Value Store. In *Erlang Workshop*, 2008.

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