Managing Geo-replicated Data in Multi-datacenters

Divyakant Agrawal[1], Amr El Abbadi[1], Hatem A. Mahmoud[1], Faisal Nawab[1], and Kenneth Salem[2]

[1] Department of Computer Science, University of California at Santa Barbara
{agrawal,amr,hatem,nawab}@cs.ucsb.edu
[2] School of Computer Science, University of Waterloo
kmsalem@uwaterloo.ca

Abstract. Over the past few years, cloud computing and the growth of global large scale computing systems have led to applications which require data management across multiple datacenters. Initially the models provided single row level transactions with eventual consistency. Although protocols based on these models provide high availability, they are not ideal for applications needing a consistent view of the data. There has been now a gradual shift to provide transactions with strong consistency with Google's Megastore and Spanner. We propose protocols for providing full transactional support while replicating data in multi-datacenter environments. First, an extension of Megastore is presented, which uses optimistic concurrency control. Second, a contrasting method is put forward, which uses gossip-based protocol for providing distributed transactions across datacenters. Our aim is to propose and evaluate different approaches for geo-replication which may be beneficial for diverse applications.

1 Introduction

During the past decade, cloud computing and large-scale datacenters have emerged as a dominant model for the future of computing and information technology infrastructures. User and enterprise applications are increasingly being hosted in the cloud and as a consequence much of user data is now stored and managed in remote datacenters whose locations remain completely transparent to the users. For most users, the main concern is the guarantee and confidence that they can access and quickly retrieve their data on demand from anywhere and at anytime. Much progress has been made in the successful realization of this model especially by Internet companies, such as Google, Amazon, Yahoo!, and others, who were confronted with the problem of supporting their respective Internet-scale applications designed to serve hundreds of millions of users dispersed around the world. Initial design considerations were primarily driven by the scalability and interactive response-time concerns. In fact these concerns were so paramount that the first generation of cloud computing solutions abandoned the traditional (and proven) data management principles and instead proposed and developed a radically different data management paradigm that is now commonly referred to as the NoSQL or key-value data stores [1, 2]. The prevailing argument was that the traditional data management approach takes a holistic view of data, which makes it almost impossible to scale commercial database management solutions (DBMSs) on a large number of commodity servers. Key-value stores, instead, constrain the atomicity to a single key-value pair and hence can be scaled to thousands or even tens of thousands of servers.

As our reliance on the cloud computing model has become prevalent as well as the initial cynicism[1] with this model has subsided, there is a renewed interest both within the academic community and in the industrial arena to address some of the formidable research and development challenges in this context. The first and foremost challenge is the issues of data management and data consistency of the distributed applications hosted in the cloud. Initial designs in the context of these applications took the radical approach that consistency of distributed data need not be addressed at the system level; rather, relegated to the applications. With this design consideration, cloud datastores provided only *eventually consistent* update operations, guaranteeing that updates would eventually propagate to all replicas. While these cloud datastores were highly scalable, developers found it difficult to create applications within the eventual consistency model [3, 4]. Many cloud providers then introduced support for atomic access to individual data items, in essence, providing *atomicity* guarantees for a single key-value pair. Atomic access of single data items is sufficient for many applications. However, if several data items must be updated atomically, the burden to implement this atomic action in a scalable, fault tolerant manner lies with the software developer. Several recent works have addressed the problem of implementing ACID transactions in cloud datastores [5, 6, 7], and, while full transaction support remains a scalability challenge, these efforts demonstrate that transactions are feasible so long as the number of tuples that are transactionally related is not "too big".

While many solutions have been developed to provide consistency and fault tolerance in cloud datastores that are hosted within a single data center, these solutions are of no help if the entire datacenter becomes unavailable. For example, in April 2011, a software error brought down one of Amazon's EC2 availability zones and caused service disruption in the entire Eastern Region of United States [8]. As a result, major web sites like Reddit, Foursquare, and Quora were unavailable for hours to days [9]. In August 2011, lightning caused Microsoft and Amazon clouds in Dublin [10] to go offline for hours. Similar outages have been reported by other service providers such as Google, Facebook, and others. In many of these instances, failures resulted in data losses.

These recent outages demonstrate the need for replication of application data at multiple datacenters as well as the importance of using provably correct protocols for performing this replication. In a recent work, Baker et al. [5] proposed Megastore which enabled applications within Google with transactional support in the cloud with full replication at multiple datacenters. Recently, Google has just announced a completely revamped cloud computing architecture called Spanner [11] where cross-datacenter geo-replication is supported as a first-class notion to all its hosted applications. While these papers present an overview of the respective systems, they lack the formality and details required to verify the correctness of the underlying protocols. We assert that such formal analysis is needed for cloud datastores, especially in light of the recent outages

[1] In its early stages, cloud computing received a lot of criticism from both academic and industrial communities that it is just a *marketing* tool redefining earlier notions such as distributed systems and grid computing.

described above and the widely acknowledged difficulties associated with the implementation of complex distributed synchronization protocols [12, 13, 14]. The other concern that arises is the performance and latency issues that must be addressed in the design of such protocols. Protocols to manage replication face significant challenges with large latencies between datacenters. These concerns mandate that systematic investigation is needed to develop robust and reliable approaches for managing geo-replicated data in multi-datacenters especially since data management over multi-datacenters will be extremely critical for national technology and information infrastructures.

Geo-replication of data across multiple datacenters offers numerous advantages. First, due to the geographic availability of data in multiple parts of the world, services and applications can be structured in such a way that user accesses from different parts of the world can be directed to the nearest datacenter. However, this operational mode requires that data replication is based on *peered* replication model in that all replicas are treated in the same manner. Second, peered replication across multiple datacenters provides the necessary fault-tolerance in case of datacenter outages. Replicating data within a datacenter, a common practice in all cloud computing architectures, ensures data availability in the case of host-failures but fails when the entire datacenter is lost due to an outage. Finally, geo-replication of data across multiple datacenters also facilitates disaster recovery in case of catastrophic failures when an entire facility is lost due to natural disasters (e.g., a earthquake) or a human-induced activity (e.g., a terrorist attack).

Data replication has been an active area of research for more than two decades [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37] and it is only recently that wide-area replication solutions are being adapted in practice [5, 11, 38]. A natural question arises as to what has changed in the intervening years to cause this transformation? Prior approaches for wide-area replication solutions were all in a context of wide-area networks where data was maintained on individual host machines geographically dispersed in the network. Given the unreliability of individual machines and the network as well as high-latency and low-bandwidth links connecting these machines rendered many of the proposed wide-area data replication protocols impractical. During the past decade, with the advances in cloud computing and datacenter technology the infrastructure landscape has undergone a significant transformation. In particular, even though each individual datacenter is a large distributed system comprising thousands to tens of thousands of machines connected by a high-speed network, software techniques such as Google's Chubby [39] and Apache's Zookeeper [40] allow us to model an individual datacenter as a unified centralized architecture[2]. Furthermore, given the level of investment that is involved to create such facilities, these datacenters are interconnected with dedicated network connections that are very high-bandwidth and extremely low-latency. With these transformations in the infrastructure space, new generation of replica management protocols need to be designed that can leverage from a small number of datacenters (in single digits) with high-speed and high-bandwidth interconnections.

[2] Essentially these software techniques maintain a globally consistent view of the system state at all times.

2 Design Elements for Multi-datacenter Data Management Architecture

In our quest to build storage infrastructures for geo-replicated data that are global, reliable, and arbitrarily large in scale, we have to start from hardware building blocks, i.e., individual datacenters, that are geographically confined and failure-prone. In order to bind these individual datacenters into a unified data infrastructure, data needs to be replicated across multiple datacenters. In this section, we start by identifying the key design components that are critical in designing cross-datacenter replica management protocols. We note that all these components are well-studied and well-researched. The novel aspect of our research is to conduct an investigation in combining these components in different ways so that they take advantage of the properties of the underlying infrastructure substrate, i.e., multiple datacenters. As will become clear in our exposition, some of the design choices are indeed influenced by the requirement that these systems remain highly scalable.

2.1 Data-Model for Datacenters

Application data in the cloud is stored using a *key-value store* [1, 41, 2, 5, 11, 42] where each data item consists of a unique *key* and its corresponding *value*, an arbitrary number of *attributes* (also called columns). The original data model for key-value stores was in terms of a single table, e.g., Google's BigTable [1] and Apache's HBase [42], where each key was considered an independent entity. Furthermore, the atomicity of accesses was constrained to a single key which enabled the key-value stores to be highly scalable since the rows in a single logical table could be arbitrarily partitioned across a large number of servers.

In order to reduce the impedance mismatch between the traditional relational model and the key-value store model, a new data-model has emerged that can be considered as being *loosely relational*. In this model, application data is governed by a schema which comprise a collection of tables. The collection of tables is partitioned based on a *primary key* of the governing table. For example, consider a photo application that consists of *user accounts* and *photographs* tables. The data can be partitioned in such a way that an individual user account and all photographs of that particular user are co-located. This hierarchical schema-based partitioning has become a prevalent data-model which is used both in the context of relational cloud systems [7, 43, 44] and scalable cloud systems [5, 11]. In Megastore [5], this notion is referred to as *entity groups* whereas in Spanner [11], it is referred to as *shards*. Ensuring the locality of a shard to a single machine enables the use of light-weight techniques for atomic access to data in the presence of concurrent accesses and failures. Also, the system still remains scalable and elastic since the shards of an application can be easily distributed and migrated freely in a datacenter. An added advantage of the shard-based data model is that database operations such as select, project, and join can be fully supported within a shard [7, 45]. We will use the notion of *shard* as the underlying data-model since it is compatible with the cloud computing database architecture that are relational [43, 45, 7, 44] and that are based on key-value stores [5, 11]. Thus, a cloud computing data model comprises a large number of *shards* which are distributed over a collection of storage servers in the datacenter.

Single-shard Consistency. The first-generation key-value stores [1, 42] in which a single key represented a shard, the most common consistency model was to guarantee *read-modify-write* operations[3]. Given that there is only a single data-element involved, this amounted to *mutually-exclusive* or *serial* execution of operations. The extension of this model to a shard which is a collection of multiple data-elements, a serial execution can still guarantee correctness of operation executions on the shard in the presence of concurrent accesses and failures. However, this model can be extended easily by using a concurrency control mechanism (e.g., Two-Phase Locking or optimistic concurrency control) to ensure *serializable* executions of operations on a shard. Although a dominant trend in the cloud has been to use optimistic protocols, recent proposals have emerged that use Two-Phase Locking [11].

Multiple Shard Consistency. Scalability and elasticity concerns in the cloud led to initial designs of data management solutions that chose not to support any notions of consistency across multiple shards. Borrowing from these designs, the relational database management solutions in the cloud [43, 45, 7] also chose not to support transactions over multiple shards[4]. The rationale of this design decision was that given the fact that shards are distributed over multiple servers in a datacenter, ensuring transactional execution over multiple shards will require expensive distributed coordination and distributed synchronization for atomic commitment of transactions which is implemented using the Two-Phase commit protocol. However, there is a new realization that absence of distributed transactions leads to application complexity and hence transactions should be supported at the system level [11]. The multi-shard consistency model is based on atomic execution of transactions over multiple shards which may be stored on different machines.

2.2 Replica Synchronization in Multi-datacenters

Cloud data management and storage solution have historically integrated data replication for ensuring availability of data. However, in the past such replication has been confined to a single datacenter to deal with host failures within a datacenter. With the availability of multiple datacenters as well as the experience with large-scale outages of datacenter facility, proposals are emerging to support geo-replication of data across multiple datacenters [46, 5, 11, 47, 48, 49]. Commercial DBMSs, in general, support *asynchronous* replication using a master-slave configuration where a master node replicates write-ahead log entries to at least one slave. The master can support fast ACID transactions but risks down-time or data loss during fail-over to a slave. In the case of

[3] We note that some key-value stores support weaker notions relegating the issue of consistency to the application tier. However given the current consensus that weaker notions of consistency results in application complexity, we require that this is the minimal consistency model that is warranted for data management in the cloud.

[4] Relational Cloud from MIT is an exception [44].

data losses, manual intervention becomes necessary to bring the database to a consistent snapshot. Given the highly autonomic nature of the cloud, *asynchronous replication* therefore is not a feasible solution for data-intensive environments.

Table 1. Design Elements for Multi-datacenter Replication

Design Features	Underlying Approaches
Sharded Data Model	Key-value stores: *Entity Groups* RDBMS: *Schema-level Partitioning*
Single-shard Atomicity	Serial Execution *Mutual-exclusion* Serializable Execution *Pessimistic (2-phase locking)* *Optimistic CC (Read/Write Set validation)*
Multi-shard Atomicity	Atomic Commmitment *Two-phase Commit*
Replica Synchronization	Synchronous Replication *Distributed Consensus (Paxos)* *Majority Quorums (two-way handshake)* *Gossip protocol (causal communication)*

All existing cross-datacenter geo-replicated data management protocols rely on synchronous replication. In synchronous replication, a transaction's updates on a shard (or multiple shards) is performed *synchronously* on all copies of the shard as a part of the transaction execution. For cross-datacenter geo-replication, existing protocols [5, 39, 49] have all chosen to use Paxos as the underlying protocol for ensuring mutual consistency of all replicas. We however note that Paxos is not the only protocol that must be used in this context. In general, from a design perspective, synchronous replication can be implemented using at least three different types of protocols: a distributed consensus protocol such as Paxos, a two-level handshake protocol based on majority quorums [15], and gossip-based protocols that ensure causal dissemination of events [50, 51]. The main advantage of Paxos is that it integrates both the normal operational mode and a possible failure mode. In the case of other protocols, explicit recovery actions must be taken when failures are encountered. For example, in the case of gossip-based protocols, progress cannot be made if one of the datacenter fails. In this case, Paxos may be triggered to modify the global membership information of operational datacenters [32, 52, 53, 54, 55].

2.3 Classifying Existing Multi-datacenter Replication Protocols

Table 1 summarizes the basic components that must be included to design and develop protocols for managing geo-replicated data across multiple datacenters. We leverage this design framework for cross-datacenter geo-replication to classify the three well-known protocols that have been proposed recently, viz., Google's Megastore [5], Google's Spanner [11], and UC Berkeley's MDCC (multi data-center consistency) protocol [49]. Although this classification does not formally establish the correctness of these proposed systems, it however gives us the understanding whether all the necessary design elements are indeed present to ensure correctness. In addition, this step allows us to explore possible variations and modifications.

Megastore was developed to meet the storage requirements of Google's interactive online services. Megastore moved away from a single-entity based data-model (i.e., key-value pairs) to hierarchically organized entity-groups (i.e., shard) where entities within a shard are logically related and shards are replicated across multiple datacenters. Each entity-group functions as a mini-database that provides serializable ACID semantics. The basic Megastore design is intended to support atomic execution of transactions within a single entity group. Transaction and replica synchronization in Megastore is based on Paxos where transactions read the current value of data-items in an entity group and when multiple transactions contend for write operations on the entity-group concurrently, the Paxos protocol ensures that only one transaction succeeds whereas the others fail. The authors claim that Megastore uses "optimistic concurrency control" but our assertion is that Megastore uses mutual-exclusion to enforce serial execution of transactions [56]. Under our classification, Megastore provides: (i) sharded data-model; (ii) single-shard consistency model with serial execution of transactions; and (iii) Paxos based replica consistency. Megastore also proposes using Two-Phase Commit for atomic transactions across entity-groups; however, not enough details are provided.

Spanner [11] is Google's latest proposal to build globally-distributed database over multiple datacenters. In that it uses a sharded data-model with data-item level accesses within a shard. In particular, the protocol uses read and write locks in conjunction with Two-Phase Locking to synchronize concurrent execution of transactions. Furthermore, a transaction execution can span multiple shards and all shards are replicated across multiple datacenters. Since transaction executions span multiple shards, Two-Phase Commit is used to ensure atomic commitment of transactions. Finally, updates to the shards are coordinated by the synchronous replication layer which uses Paxos to ensure consistent ordering of updates to all copies of a shard. In summary, Spanner uses Two-Phase Commit and Two-Phase Locking to provide atomicity and isolation, running on top of Paxos to provide synchronous replication. A similar architecture is also used in Scatter [46].

The MDCC protocol from UC Berkeley [49] also supports atomic execution of multi-sharded transactions with data-item level accesses within each shard. Serializable execution of transactions is enforced using optimistic concurrency control. However, atomic commitment and synchronous replication is achieved by using a variant of Paxos protocol which is referred to as multi-Paxos. In particular, a transaction initiates a single instance of Paxos where the distributed consensus involves all shards and their replicas. This obviates the need for an explicit Two-Phase Commit at the expense of executing on a larger number of cohorts. As an example, if there are three data-objects each with three copies, MDCC will require consensus with nine entities. In contrast, Spanner requires Two-Phase Commit with three shards whereas the replication layer runs three separate instances of Paxos on three copies of each shard.

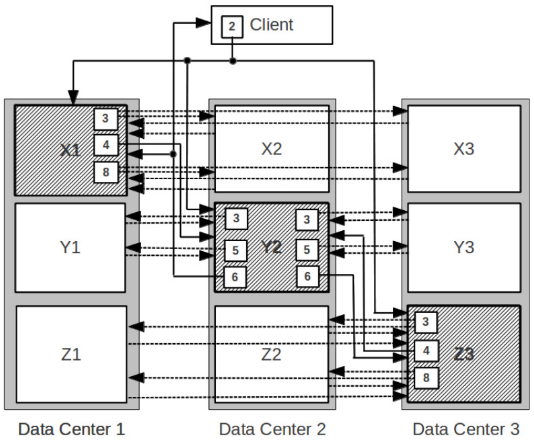


Fig. 1. Typical Two-Phase Commit operations when using Paxos-replicated transactional logs

2.4 Inter-datacenter Communication

Inter-datacenter message exchange's effect on latency dominates over the effect of intra-datacenter communication. For this, it is important to understand the behavior of current designs regarding their wide-area message exchange. We now analyze the number of inter-datacenter messages in Megastore. A single instance of Paxos takes five rounds of messages. In state machine replication, the number of rounds can be reduced to three by designating a *master replica*, a distinguished leader that remains in charge until it fails [57, 12]. Megastore uses this optimization to reduce the number of message rounds to three in cases where there is no contention for the log position [5, 56]. MDCC [49] uses a variant of Paxos to atomically commit transactions while providing replication at the same time. Each record requires a Paxos round to be accepted. A record is accepted if it did not introduce any conflicts. When all records are accepted and learned, then the transaction is considered committed.

We present an example to estimate the number of inter-datacenter messages in Spanner. After completing all reads, the client submits all updates to the database using Two-Phase Commit. Consider the case when a transaction updates three data objects 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 shards 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. Hashed servers represent the Paxos leaders of their shards.

The required number of wide-area messages is now illustrated. The client picks one leader to be the Two-Phase Commit coordinator. A prepare message is sent from the client to all leaders. Then, the prepare message is logged using Paxos. Logging requires a round-trip message exchange, assuming the leader did not change. Afterwards, Paxos leaders acknowledge the Two-Phase Commit coordinator. This will require one inter-datacenter message. The Two-Phase Commit coordinator now logs the received acknowledgments, which will take a round-trip message exchange. The Two-Phase Commit notifies other leaders of its decision when it committed while sending the commit decision to the client, both taking one wide-area message. At this point, the client knows the transaction is committed, however, more rounds of communication occur by replica leaders to log the commit decision. Thus, the required number of inter-datacenter messages is seven until the client knows the commit decision and nine until the transaction is fully replicated.

3 New Protocols

In this section, we sketch the outline of two new protocols for managing geo-replicated data in multi-datacenters. The first protocol can be viewed as an extension of the original Megastore in that the modified protocol enables *serializable* execution of transactions within an entity-group thus permitting more concurrency in the underlying system. The second protocol uses a radically different approach in that it uses gossip-messages (i.e., message propagation) for synchronizing the execution of distributed transactions over replicated data. Our goal in this section is to underscore the many possible approaches that can be used for geo-replication. In the next section we will describe our approaches to develop a better understanding of the engineering tradeoffs of using different design choices in the development of these protocols.

3.1 Megastore with Optimistic Concurrency Control

The original Megastore system allows multiple transactions to operate concurrently on a replicated shard (or entity-group) at multiple datacenters. However, if multiple transactions attempt to update the shard simultaneously, the Paxos protocol ensures that only one transaction succeeds and the rest are aborted. This is facilitated using Paxos in that all concurrent update transactions compete for the next empty log position in the Paxos log and only one transaction is granted the log position. Although the authors *incorrectly* state that Megastore uses optimistic concurrency control, this corresponds to serial execution of transactions. At best this can be viewed as *optimistic* mutual exclusion in that multiple transactions are allowed to enter the critical section to update the shard but only one transaction succeeds in exiting the critical section by updating the shard and the rest abort.

We now outline the design of extended Megastore [56]. Higher concurrency is achieved by *promoting* the losing non-conflicting transactions to compete for the subsequent log position. During an instance of Paxos, a losing transaction that has not received *majority* of votes for log position k realizes that the value of the winner transaction will write its values in log position k. Therefore, there is no benefit for the loser to continue competing for this log position. Instead, it can try to win log position $k+1$ as long as the loser does not read any value that was written by the winning transaction for log position k. In this case, the loser then initiates the commit protocol for log position $k+1$ with its own value. Otherwise, the client stops executing the commit protocol and returns an *abort* status to the application. If the client does not win log position $k+1$, it can try again for promotion to the next log position as long as its writes do not conflict with the writes of the winners at the log positions k and $k+1$. As the number of tries increases, there is an increased possibility that the transaction will be aborted. With this simple enhancement we are able to support *serializable* execution of transactions in Megastore. Multi-shard execution of transactions can be supported by using two-phase commit over Paxos as was the case in the original Megastore proposal.

We note that when several transactions compete for the log position, there is a possibility that no transaction gets the *majority* during the Paxos commit. In that situation, it may be possible to *combine* non-conflicting transactions into a single log position instead of aborting all competing transactions. However, this approach requires a combinatorial number of comparisons to construct all possible non-conflicting subsets of concurrent transactions that can be combined and choosing a subset with the largest number of transactions. We are developing a protocol that incorporates the above- mentioned enhancements and will conduct experimental evaluations to compare the proposed variants with the original Megastore protocol.

3.2 Message Futures: Fast Transaction Commitment in Multi-datacenters

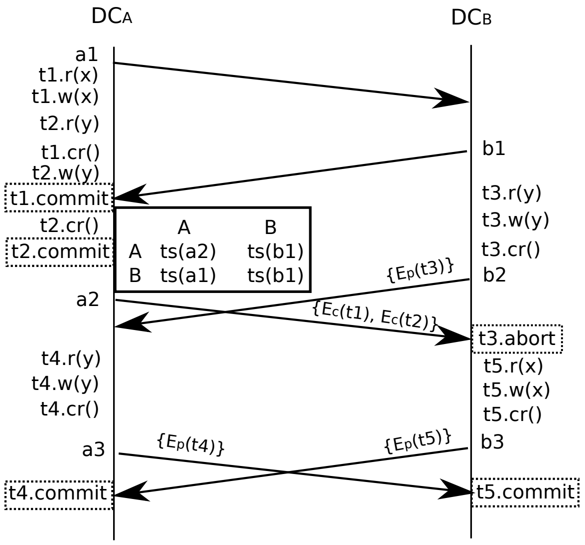
We now propose Message Futures (MF), a cross-datacenter geo-replication protocol that supports serializable execution of transactions in a multi-shard model where shards are replicated across multiple datacenters. The protocol can use either two-phase locking or optimistic concurrency control for intra-datacenter synchronization of transactions within a single data-center. Replica consistency and inter-datacenter synchronization is achieved by deploying a gossip-based protocol for distributed commitment of transactions at all datacenters. One of the advantages of gossip-based message communication is that an event log (e.g., transaction execution) can be propagated among datacenters transitively while ensuring causal relationship of events in a distributed system [51].

A naive and straightforward adaptation of gossip-messages would be to execute transactions locally and initiate a distributed commit of a transaction by appending a transaction commit request to the log (referred to as the replicated log or RLog for brevity) and wait for other datacenters to respond to this request. After receiving the requisite responses either directly or transitively from other datacenters, the fate of the transaction can be determined and the second round of atomic commitment can be facilitated again via log propagation. Although this design ensures serializable execution, it incurs very high latency to commit transactions. We instead propose a novel approach called *Message Futures* that *potentially* eliminates the need for blocking to achieve consensus from other datacenters when transaction are ready to commit. We now present an overview of our proposed protocol. Each datacenter, DC_i, maintains the following structures:

- *Local RLog*, L_i, is the local view of the global RLog.
- *Pending Transactions list*, PT_i, contains local *pending transactions*. These are transactions that requested to commit but are still neither committed nor aborted.
- *Last Propagated Time*, LPT_i, is the timestamp of the processing time of the last sent L_i at DC_i.

RLogs maintain a global view of the system that can be used by datacenters to perform their concurrency logic. RLogs consist of an ordered sequence of *events*. All events have timestamps. Each transaction is represented by an event. RLogs are continuously propagated to other datacenters. An algorithm used to efficiently propagate RLogs is presented in [51]. An $N \times N$ Timetable, T_i, is maintained by L_i, where N is the number of datacenters. Each entry in the Timetable is a timestamp representing a bound on how much a datacenter knows about another datacenter's events. For example, entry $T_i(j,k) = \tau$ means that datacenter DC_i knows that datacenter DC_j is aware of all events at datacenter DC_k up to timestamp τ. An event in L_i is discarded if DC_i knows that all datacenters know about it. The transitive log propagation algorithm ensures two properties about events in the system. First, all events are eventually known at all datacenters. Second, if two events have a *happened-before* relation [58], their order is maintained in the RLog.

Each datacenter is represented by one row and one column in the Timetable. Each transaction, t_i, is represented as an event record, $E_{type}(t_i)$, in the RLog, where $type$ is either *Pending* ($E_p(t_i)$) or *Committed* ($E_c(t_i)$). A pending event is maintained until the transaction commits or aborts. A committed event is maintained in the RLog until it is known to all datacenters.



	A	B
A	ts(a2)	ts(b1)
B	ts(a1)	ts(b1)

Fig. 2. MF example scenario

Each datacenter, DC_A, transmits L_A, its local RLog, continuously regardless of the existence of new events. Consider a pending transaction t_i at DC_A. When t_i requests to commit, the current *Last Propagated Time*, LPT_A, is attached to t_i and is referred to as $t_i \rightarrow LPT_A$. Then, t_i with its read-set and write-sets are appended to the local Pending Transactions list, PT_A, while only the write-set is appended to L_A. Whenever DC_A receives RLog, L_B, it checks for conflicts between transactions, t_i, in PT_A and t' in L_B.

If a conflict exists, t_i is aborted. A conflict exist if a common object, x, exists in t''s write-set and t_i's read-set or write-sets. To commit t_i, DC_A waits until the following *commit condition* holds:

Definition 1. *A pending transaction t_i in PT_A commits if all read versions of objects in t_i's read-set are identical to ones in local storage, and $T_A[B, A] \geq t_i \rightarrow LPT_A$, $\forall_B$ $(DC_B \in datacenters)$*

That is, all objects in t_i's read-set have the same versions as those in the local storage and datacenter DC_A knows that all datacenters, DC_B, are aware of DC_A's events up to time $t_i \rightarrow LPT_A$. Conflicts that include t_i's write-set are detected earlier when remote transactions are received and their conflicts are detected.

We now illustrate a simple operational scenario of MF depicted in Figure 2. The scenario consists of two datacenters, DC_A and DC_B. The passage of time is represented by going downward. Arrows are RLog transmissions. Events in the RLog are shown over the arrow. If no events exist, nothing will be shown. The corresponding Timetable is also displayed in one case for demonstration purposes. The notation on the sides are operations performed or issued at the datacenter. $t_i.operation(key)$ represents performing an operation on the object key for transaction t_i. Client operations are *read* (r), *write* (w), and *commit request* (cr). Commits and aborts are shown inside dotted boxes. As introduced above, RLog transmissions are represented by the notation δ_i, where δ is the lower case character of the datacenter's name and i is a monotonically increasing number.

Consider transaction t_1 of DC_A. It reads and writes object x and then requests a commit. $t_1 \rightarrow LPT_A$ is set to $ts(a_1)$. DC_A waits until the commit condition (Definition 1) holds. When L_B, sent at b_1, is received at DC_A, the commit condition is satisfied and t_1 commits. Transaction t_2, which also started after a_1, requests a commit. $t_2 \rightarrow LPT_A$ is also set to $ts(a_1)$. Since it has requested to commit after the reception of the RLog transmission at $ts(b_1)$, the commit condition holds at the time it requested to commit, hence t_2 commits immediately. Transaction t_3 requests to commit at DC_B. $t_3 \rightarrow LPT_B$ is set to $ts(b_1)$ when a commit is requested. However, when L_A of a_2 arrives at DC_B, a conflict with transaction t_2 is detected. In this case, t_3 is aborted. Finally, we show the case of transactions t_4 and t_5. When a commit is requested for both of them, $t_4 \rightarrow LPT_A$ is set to $ts(a_2)$ and $t_5 \rightarrow LPT_B$ is set to $ts(b_2)$. When each datacenter receives the other datacenter's RLog, it contains the information of the pending transaction of the other datacenter. However, no conflict is detected. At that point, the commit condition holds for both of them and both t_4 and t_5 commit. We also included a demonstration of T_A at time $ts(a_2)$.

The performance of Message Futures based protocol depends on the frequency of log propagation. In our initial evaluations, by tuning the propagation interval we are able to achieve commit latency close to the maximum round-trip times among inter-datacenter communications. Furthermore, by making the propagation asymmetric where one datacenter propagates its log much more infrequently compared to the remaining datacenters, we can simulate a master-slave configuration of replicas. In fact the master datacenter with large propagation delays experiences lower commit latencies and in many cases it can commit its transactions immediately. Note that by adjusting the propagation intervals appropriately, this protocol can be extended in such a way that master-ownership on per shard (or a collection of shards) can be dispersed over multi-datacenters. We are exploring these extensions and developing analytical formulations to optimize propagation intervals in terms of number of message exchanges and commit latency. We are also conducting extensive evaluations to quantify the performance and overhead of this protocol in comparison to others.

4 Prospective Research: Protocol Correctness, Evaluation, and Enhancements

In this section, we provide an overview of our research methodology to conduct a systematic research investigation of cross-datacenter geo-replication protocols. First, we present an abstract framework that will be used to establish the correctness of replication protocols. In particular, we will establish the properties of a protocol based on its specifications and will verify that these properties collectively can be used to prove if the protocol is correct. Next, we present the details of an implementation platform that will be used to evaluate the performance of the proposed and existing protocols. Finally, we identify some of the pragmatic enhancements that need to be incorporated with any geo-replication protocols before they can be used in practice.

4.1 A Framework to Establish Correctness

The protocols considered so far in this paper all include the necessary design components to ensure *plausibly* correct executions of distributed transactions over replicated data. However, as we have shown these design components can be integrated in a variety of ways. For example, Spanner [11] uses a layered approach where at the lowest layer it implements synchronous replication using Paxos and at the upper layer it uses two-phase commit in conjunction with two-phase locking for correct execution of transactions. Megastore [5] uses Paxos for both replica synchronization and concurrency prevention. Given this vast variation in the overall design of such protocols, it is indeed necessary to formally establish the correctness of these protocols. This is clearly warranted since multi-datacenter architectures are likely to become an integral part of our national infrastructures. We therefore present an abstract framework that can be used to reason the correctness of multi-datacenter protocols.

In a multi-datacenter architecture, each datacenter has its own multi-version datastore comprising sharded data. All shards are replicated on multiple datacenters, and hence, there are both multiple copies and multiple versions of each data-item within a shard. Yet, when a client (an application instance) executes a transaction, it should appear that (1) there is only one copy and one version of each data item, and (2) within the scope of its transaction, the client is the only one accessing those data items. These two properties are captured by the notion of *one-copy serializability* [59]. In a multi-version, multi-copy (MVMC) datastore, when a client performs a read operation, it reads a single version of a single copy of a data item. When a write operation is applied to the cloud datastore, a new version of the item is created at one or more datacenters. An *MVMC transaction* is a partially ordered set of read and write operations, with their corresponding version and copy attributes, ending with a single *commit* or a single *abort* operation. We say a transaction t *reads-x-from* transaction s if t reads the version of x (at one copy) that was written by s (at one or more copies). An *MVMC history* is a set of MVMC transactions with a partial order. The partial order obeys the order of operations within each transaction and maintains the *reads-from* relation, i.e., if transaction t reads version i of x from transaction s at copy A, then the write of version i at copy A precedes the read of version i at copy A, and no other write occurs between these operations at copy A.

Definition 2. *A multi-version, multi-copy history H is **one-copy serializable** if there exists a single copy, single version serial history S such that H and S have the same operations, and t_i reads-x-from t_j in H* iff *t_i reads-x-from t_j in S.*

Our goal is to prove that the system and protocols for Multi-datacenter replication guarantee one-copy serializability. In general, all systems implement a concurrency control protocol with a write-ahead log. In addition to its set of data items, each shard has its own write-ahead log that is replicated at all datacenters. The write ahead log is divided into *log positions* which are uniquely numbered in increasing order. When a transaction that contains write operations commits, its operations are written into a single log position, the *commit position*. Read-only transactions are not recorded in the log. For each write in the committed transaction, the commit log position serves as the timestamp for the corresponding write operation. While the log is updated at commit time, these write operations may be performed later by a background process or as needed to serve a read request.

To guarantee correct execution of transactions, we must be sure that transactions are only written to the log if they are correct with respect to the one-copy serializability property. Formally, we require that a concurrency control protocol ensure the following properties.

(L1) The log only contains operations from committed transactions.

(L2) For every committed transaction that contain a write operation, all of its operations are contained in a single log position.

(L3) An entry will only be created in a log position if the union of this log entry and the complete prefix of the log prior to this log entry is a one-copy serializable history.

We require that transactions are consistently replicated across multiple datacenters. To achieve consistent replication, when a transition commits, we replicate the new log entry at every datacenter. The replication algorithm must satisfy the following property.

(R1) No two logs have different values for the same log position.

To guarantee correctness, we need an additional assumption that relate to the handling of read requests.

(A1) Within a transaction, all read operations read from the same log position; i.e., the transaction reads the latest writes performed up through the specified read position in the log.

We state the following theorem that can be formally established to verify that the properties defined above are sufficient to guarantee one-copy serializability.

Theorem 1. *For the transactional data store with replication at multiple datacenters, if the underlying protocol guarantees properties (L1) - (L3), (R1), and (A1), then the datastore guarantees one-copy serializability.*

During the course of our research we will use the above correctness framework to establish correctness of the proposed protocols.

4.2 Implementation Testbed and Performance Evaluation

Developing an infrastructure of geo-replication solutions is essential to our evaluation plan. An infrastructure that is available to the public, including other research groups, will allow validation of results and the ability to extend experiments. Amazon AWS is a prominent cloud computing platform that Amazon makes available to researchers and educators worldwide through their research and education grants. A central part of AWS is Amazon EC2 which allows users to rent virtual machines in Amazon's cloud. Users can either use local storage or network-attached storage called Elastic Block Storage (Amazon EBS). EC2 instances offer many variations of number of cores and main memory. An important feature of EC2 for the purposes of this section is the number of datacenters available and the geo-separation among them. Currently, EC2 allows creating instances in eight data centers physically located in California, Oregon, Virginia, Ireland, Singapore, Japan, Australia, and Brazil. Availability of Amazon's geographically distributed platforms will enable us to test the effectiveness of multi-datacenter geo-replicated datastores in a real-life setting.

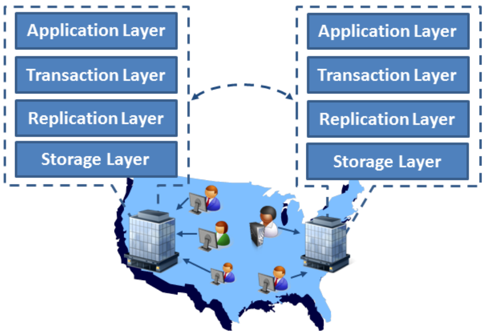


Fig. 3. The geo-replication stack of two datacenters

Operating over the infrastructure is the concurrency solution. In this section, a multi-layer stack abstraction is assumed as shown in Figure 3. Each layer provides an interface to the neighboring layers. By defining those interfaces, development of one layer can be carried independently from other layers. Furthermore, deployments can exploit this layering to plug components in each layer in such a way that is suitable for their application. The bottom most layer, closest to the hardware infrastructure, is the *storage layer*. This layer consists of the database or key-value store that maintain the data. This layer has access to the local storage in addition to the network-attached storage. On top of the storage layer are the *transaction layer* and the *replication layers*. These layers handle concurrency control and communication among different instances. There are several different possible configurations of replication and transaction layers as shown in Figure 4. A transaction layer can be independent from the replication layer. The transaction layer can be on top of the replication layer as shown in Figure 4(a), meaning that the transaction layer relays transactions to the replication layer that finally interface with the storage layer. This is the configuration used in Spanner [11]. Alternatively, as shown in Figure 4(b) a replication layer can be on top of the transactional layer. The transaction and replication layers can also be configured so they are adjacent to each other as shown in Figure 4(c), where both units access the storage layer while being independent from each other. In this case there is an interface between the two units. This configuration is used in Message Futures. Finally, the transaction and replication layers can be intermingled into a single unit as shown in Figure 4(d), hence there is no clear distinction between the replication and transactional logic. This is analogous to the design of MDCC [49] and Megastore [5]. The layer on top of the transactional layer is the application layer which provides the interface for clients to access the system. The application layer is designated for single users which can be used by clients to issue operations and request the commitment of transactions. Furthermore, in the course of developing our solutions presented earlier we have built many components that can be plugged into the infrastructure to complete the stack. Some of these implementations are designs for 2PC and Paxos that we plan to release to the community as a bundle with our infrastructure. In addition, we leverage current open source solutions to act as components in our implementation platform. For example, HBase [42] is a suitable candidate of a key-value store setting in the storage layer.

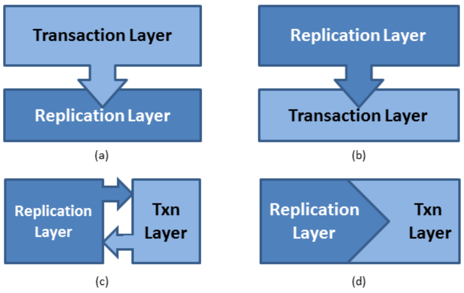


Fig. 4. Different configurations of the transaction and replication layers

In addition to the implementation platform, it is critical to develop an evaluation framework to compare configurations and implementations. Existing benchmarks are not suitable for evaluating multi-datacenter replication for the following reasons. First, most evaluation frameworks are not distributed and are designed for single node execution. To measure impact of writes and reads on disjoint data centers, benchmark *workers* should be local, or near, to replication components. Using a single site for generating load will skew results. Second, many existing benchmarks use a blocking multi-threaded model. The high latencies encountered in geo-replication results in lower throughput due to blocking. Increasing the number of active threads can result in extremely bursty behavior, which can saturate and throttle services. An ideal evaluation framework would utilize asynchronous workers. Third, outside of the aforementioned issues, existing benchmarks are good for evaluating throughput and latency. A geo-replication benchmark should also evaluate the percentage of stale reads, update propagation time, and aborted transactions due to consistency conflicts. Our plan is to incorporate existing benchmarks, such as YCSB and TPC-C, into a custom framework that addresses these issues. The evaluation framework will be open-sourced, and allow for custom workloads to be incorporated.

4.3 Pragmatic Enhancements

So far in our development we have focused on multi-datacenter protocols that ensure strong consistency, i.e., atomic execution of transactions over multiple shards and synchronous updates to all replicas of the shards. Most system implementations consider pragmatic enhancements to the basic protocols that will result in better performance. Maintaining multiple versions for each shard allow for numerous opportunities to process read operations efficiently. For example, Spanner [11] utilizes the availability of synchronized clocks (i.e., TrueTime in spanner), timestamps, and version numbering judicially to support *pre-declared read-only* transactions and *snapshot read* operations. In particular, both read-only transactions and snapshot reads can be executed without any locking overhead. In the same vein, Yahoo's PNUTS system [2], uses timestamps and versions to implement the *timeline consistency* model for data. Using this model, fast read operations can be supported using past versions of data. In our research, we will explore similar enhancements in the context of the geo-replication protocols proposed in this paper. In particular, given the dominant read-only nature of many of the applications, fast read-only transactions and fast read operations will in general be a valuable enhancement.

The other research direction is based on the observation that although all commercial DBMSs guarantee serializable execution of transactions, most real DBMS deployments and installations use what is widely known as *snapshot isolation* [60, 61]. Snapshot isolation ensures that concurrent transactions observe the most up-to-date consistent view of the database for reads and must not have any write-write conflicts. Ensuring snapshot isolation of transactions in replicated environments is in general considered a hard problem. Recently, a weaker notion of snapshot isolation, referred to as Parallel Snapshot Isolation (PSI) [47] has been introduced specifically for geo-replicated systems. PSI mandates that transactions observe the most up-to-date consistent local view of the database (which may be older than the global up-to-date consistent view), must not have write-write conflicts globally, and that commits are causally ordered (a transaction is propagated to other replicas after all transactions that committed before it began). We plan to explore enhancements to the proposed protocols for both snapshot isolation [62] and PSI based executions of transactions.

References

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

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

[3] Muthukkaruppan, K.: The underlying technology of messages (2011) (acc. October 5, 2011)

[4] McKusick, K., Quinlan, S.: Gfs: evolution on fast-forward. Commun. ACM 53(3), 42–49 (2010)

[5] Baker, J., Bond, C., Corbett, J., Furman, J., Khorlin, A., Larson, J., Leon, J.M., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing scalable, highly available storage for interactive services. In: Conf. Innovative Data Systems Research, pp. 223–234 (2011)

[6] Das, S., Agrawal, D., El Abbadi, A.: G-Store: A scalable data store for transactional multi key access in the cloud. In: Proc. 1st ACM Symp. Cloud Computing, pp. 163–174 (2010)

[7] Das, S., Agrawal, D., El Abbadi, A.: Elastras: An elastic transactional data store in the cloud. In: USENIX Workshop on Hot Topics in Cloud Computing (2009); An expanded version of this paper will appear in the ACM Transactions on Database Systems

[8] Amazon.com: Summary of the Amazon EC2 and Amazon RDS service disruption in the US East Region (2011) (acc. October 5, 2011)

[9] Butcher, M.: Amazon EC2 goes down, taking with it Reddit, Foursquare and Quora (April 2011) (acc. October 5, 2011)

[10] Greene, A.: Lightning strike causes Amazon, Microsoft cloud outage in Europe. TechFlash (August 2011)

[11] Corbett, J., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al.: Spanner: Google's globally-distributed database. To Appear in Proceedings of OSDI, 1 (2012)

[12] Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live: an engineering perspective. In: Proc. 26th ACM Symp. Principles of Distributed Computing, pp. 398–407 (2007)

[13] Lamport, L.: Paxos made simple. ACM SIGACT News 32(4), 18–25 (2001)

[14] van Renesse, R.: Paxos made moderately complex. Technical Report (2011)

[15] Gifford, D.: Weighted voting for replicated data. In: Proceedings of the Seventh ACM Symposium on Operating Systems Principles, pp. 150–162. ACM (1979)

[16] Stonebraker, M.: Concurrency Control and Consistency in Multiple Copies of Data in Distributed INGRES. IEEE Transactions on Software Engineering 3(3), 188–194 (1979)

[17] Thomas, R.H.: A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. ACM Transaction on Database Systems 4(2), 180–209 (1979)

[18] Bernstein, P.A., Goodman, N.: An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases. ACM Transactions on Database Systems 9(4), 596–615 (1984)

[19] Herlihy, M.: Replication Methods for Abstract Data Types. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology (May 1984)

[20] Birman, K.P.: Replication and Fault-tolerance in the ISIS System. In: Proceedings of the Tenth Symposium on Operating Systems Principles, pp. 79–86 (December 1985)

[21] El Abbadi, A., Skeen, D., Cristian, F.: An Efficient Fault-Tolerant Protocol for Replicated Data Management. In: Proceedings of the Fourth ACM Symposium on Principles of Database Systems, pp. 215–228 (March 1985)

[22] El Abbadi, A., Toueg, S.: Availability in partitioned replicated databases. In: Proceedings of the Fifth ACM Symposium on Principles of Database Systems, pp. 240–251 (March 1986)

[23] Garcia-Molina, H., Barbara, D.: How to assign votes in a distributed system. Journal of the Association of the Computing Machinery 32(4), 841–860 (1985)

[24] Herlihy, M.: A Quorum-Consensus Replication Method for Abstract Data Types. ACM Transactions on Computer Systems 4(1), 32–53 (1986)

[25] Liskov, B., Ladin, R.: Highly Available Services in Distributed Systems. In: Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing, pp. 29–39 (August 1986)

[26] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic Algorithms for Replicated Database Maintenance. In: Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing, pp. 1–12 (August 1987)

[27] Jajodia, S., Mutchler, D.: Dynamic Voting. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 227–238 (June 1987)

[28] Carey, M.J., Livny, M.: Distributed concurrency control performance: A study of algorithms, distribution, and replication. In: Proceedings of the Fourteenth Conference on Very Large Data Bases, pp. 13–25 (August 1988)

[29] Agrawal, D., El Abbadi, A.: Reducing storage for quorum consensus algorithms. In: Proceedings of the Thirteenth International Conference on Very Large Data Bases, pp. 419–430 (August 1988)

[30] El Abbadi, A., Toueg, S.: Maintaining Availability in Partitioned Replicated Databases. ACM Transaction on Database Systems 14(2), 264–290 (1989)

[31] Agrawal, D., El Abbadi, A.: The Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data. In: Proceedings of Sixteenth International Conference on Very Large Data Bases, pp. 243–254 (August 1990)

[32] Jajodia, S., Mutchler, D.: Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database. ACM Transactions on Database Systems 15(2), 230–280 (1990)

[33] Agrawal, D., El Abbadi, A.: The Generalized Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data. ACM Transaction on Database Systems 17(4), 689–717 (1992)

[34] Agrawal, D., El Abbadi, A.: Resilient Logical Structures for Efficient Management of Replicated Data. In: Proceedings of Eighteenth International Conference on Very Large Data Bases, pp. 151–162 (August 1992)

[35] Gray, J., Helland, P., O'Neil, P., Shasha, D.: The Dangers of Replication. In: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, pp. 173–182 (June 1996)

[36] Agrawal, D., El Abbadi, A., Steinke, R.: Epidemic Algorithms in Replicated Databases. In: Proceedings of the ACM Symposium on Principles of Database Systems, pp. 161–172 (May 1997)

[37] Stanoi, I., Agrawal, D., El Abbadi, A.: Using broadcast primitives in replicated databases. In: Proceedings of the 1998 IEEE International Conference on Distributed Computing Systems, pp. 148–155 (May 1998)

[38] Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. Operating Systems Review 44(2), 35–40 (2010)

[39] Burrows, M.: The chubby lock service for loosely-coupled distributed systems. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI 2006, pp. 335–350. USENIX Association, Berkeley (2006)

[40] Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: Proc. 2010 USENIX Conference, USENIXATC 2010, p. 11. USENIX Association, Berkeley (2010)

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

[42] HBase (2011), `http://hbase.apache.org` (acc. July 18, 2011)

[43] Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., et al.: Windows azure storage: a highly available cloud storage service with strong consistency. In: Proc. Twenty-Third ACM Symp. Operating Systems Principles, pp. 143–157. ACM (2011)

[44] Curino, C., Jones, E.P.C., Popa, R.A., Malviya, N., Wu, E., Madden, S., Balakrishnan, H., Zeldovich, N.: Relational cloud: a database service for the cloud. In: CIDR, pp. 235–240 (2011)

[45] Bernstein, P.A., Cseri, I., Dani, N., Ellis, N., Kalhan, A., Kakivaya, G., Lomet, D.B., Manne, R., Novik, L., Talius, T.: Adapting microsoft sql server for cloud computing. In: ICDE, pp. 1255–1263 (2011)

[46] Glendenning, L., Beschastnikh, I., Krishnamurthy, A., Anderson, T.: Scalable consistency in scatter. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP 2011, pp. 15–28. ACM, New York (2011)

[47] Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP 2011, pp. 385–400. ACM, New York (2011)

[48] Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP 2011, pp. 401–416. ACM, New York (2011)

[49] Kraska, T., Pang, G., Franklin, M.J., Madden, S.: Mdcc: Multi-data center consistency. CoRR abs/1203.6049 (2012)

[50] Fischer, M., Michael, A.: Sacrificing serializability to attain high availability of data in an unreliable network. In: Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, pp. 70–75. ACM (1982)

[51] Wuu, G.T., Bernstein, A.J.: Efficient solutions to the replicated log and dictionary problems. In: Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, PODC 1984, pp. 233–242. ACM, New York (1984)

[52] Kaashoek, M.F., Tanenbaum, A.S.: Group Communication in the Amoeba Distributed Operating Systems. In: Proceedings of the 11th International Conference on Distributed Computing Systems, 222–230 (May 1991)

[53] Amir, Y., Dolev, D., Kramer, S., Malki, D.: Membership Algorithms for Multicast Communication Groups. In: Segall, A., Zaks, S. (eds.) WDAG 1992. LNCS, vol. 647, pp. 292–312. Springer, Heidelberg (1992)

[54] Amir, Y., Moser, L.E., Melliar-Smith, P.M., Agarwal, D.A., Ciarfella, P.: The Totem Single-Ring Ordering and Membership Protocol. ACM Transactions on Computer Systems 13(4), 311–342 (1995)

[55] Neiger, G.: A New Look at Membership Services. In: Proceedings of the ACM Symposium on Principles of Distributed Computing (1996)

[56] Patterson, S., Elmore, A.J., Nawab, F., Agrawal, D., Abbadi, A.E.: Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. PVLDB 5(11), 1459–1470 (2012)

[57] Lamport, L.: The part-time parliament. ACM Trans. Computer Systems 16(2), 133–169 (1998)

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

[59] Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)

[60] Adya, A., Liskov, B., O'Neil, P.E.: Generalized isolation level definitions. In: ICDE, pp. 67–78 (2000)

[61] Lin, Y., Kemme, B., Jiménez-Peris, R., Patiño Martínez, M., Armendáriz-Iñigo, J.E.: Snapshot isolation and integrity constraints in replicated databases. ACM Trans. Database Syst. 34(2), 11:1–11:49 (2009)

[62] Wu, S., Kemme, B.: Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In: ICDE, pp. 422–433 (2005)