Message Futures: Fast Commitment of Transactions in Multi-datacenter Environments

Faisal Nawab Divyakant Agrawal Amr El Abbadi

Department of Computer Science, University of California, Santa Barbara, Santa Barbara, CA 93106
{nawab,agrawal,amr}@cs.ucsb.edu

ABSTRACT

Geo-replication of large Internet services is increasingly deployed for better data locality and fault tolerance. Maintaining consistency across datacenters is expensive and requires wide-area communication. This renders current solutions to either settle for weaker forms of consistency or suffer from large delays. In this work we present *Message Futures*, a strongly consistent concurrency control manager with low commit latency to ensure mutual consistency of replicas across datacenters. By judicial message passing of relevant information and at opportune time intervals, Message Futures can also enforce different priority levels of access, where each datacenter experiences a commit latency relative to its priority. In fact, in many common cases, transactions can be committed locally without the need for any communication. An experimental evaluation of Message Futures on a geo-replicated multi-datacenter setting is presented. We show that Message Futures achieves a commit latency around one RTT (Round-Trip Time) for datacenters with identical priority, and a latency comparable to committing locally for high priority datacenters.

Keywords

cloud computing, replication, concurrency

1. INTRODUCTION

Modern Internet services are increasingly deployed over geographically separated datacenters to provide better availability and fault tolerance. Replication across datacenters brings data closer to the end user, reducing last-mile latency and increasing locality. Providing the underlying infrastructure to manage replication and data management is essential to rid developers of concerns of consistency and coordination.

Replication, however, is expensive, requiring management of concurrency, failures, and communication overhead. Protocols to manage replication face significant challenges with large latencies between replicas. Inter-datacenter communication is prone to variation in Round-Trip Times (RTTs) and loss of packets. Most importantly, RTTs are in the order of hundreds of milliseconds, *e.g.*, up to 300ms. Delivering strong guarantees requires awareness of other replicas' states before committing. With such large RTTs it becomes apparent that communication overhead dominates the commit latencies observed by users. Solutions often sacrifice strong consistency guarantees to maintain acceptable response times. Weak consistency, however, make application development significantly more complex. Furthermore, synchronization and concurrency solutions are developed multiple times in every application with considerable vulnerability to errors.

In this paper we propose Message Futures (MF), a completely distributed multi-datacenter transaction management system that provides strong consistency guarantees while maintaining low commit latency. It achieves an average commit latency around one RTT. A transaction is committed when a *commit condition* on mutual information is met. At any point in time, the commit condition is designed to be true for any single object in at most one datacenter. A Replicated Log (RLog) [19] is utilized to share transactions and state information among datacenters continuously. RLogs provide enough information to ensure correct transaction execution. Thus, the continuous exchange of RLogs *allows a datacenter to commit transactions without initiating a new wide-area message exchange with other datacenters.* MF guarantees one-copy serializability [2].

In addition to maintaining strong consistency and low latency, the incorporation of RLog improve MF's resilience to node and communication failures. RLogs guarantee the *happens-before* relation among events. RLogs are propagated continuously and independently. We study the propagation of RLogs to achieve the desired performance properties. In particular, we study how assigning different RLog transmission rates allows us to prioritize datacenters and tailor commit latency. This, in many common cases, enables high priority datacenters to commit transactions immediately while having a minimal effect on other datacenters commit latency.

Our contributions are summarized as the following:

- We propose MF, a concurrency control manager (Section 3). It provides strong consistency guarantees while achieving a commit latency close to a single RTT.
- Correctness of MF is proved (Section 3.3).

6th Biennial Conference on Innovative Data Systems Research (CIDR '13) January 6-9, 2013, Asilomar, California, USA.

- We performed a set of experiments on EC2 nodes across inter-continental datacenters (Section 4). In those experiments, we compare MF's performance with two Paxos-based protocols. We also demonstrate how to tailor the system's commit latency and how to enable transactions to commit immediately.

The rest of this document is divided as follows: First, we overview the system in Section 2 and elucidate the considered datacenter architecture, the main components of an MF instance, and RLogs. Then, we proceed to propose MF in Section 3. We provide an intuitive description followed by a more formal definition of MF operation and a proof of its correctness. An evaluation of our system is detailed in Section 4. Section 5 summarizes related work. Finally, we conclude the paper with a summary in Section 6.

2. SYSTEM OVERVIEW

We consider a large multi-cloud system consisting of multiple datacenters, each with a large number of nodes. In what follows we describe the underlying datacenter architecture, MF design, and RLogs.

2.1 Datacenter architecture

Each datacenter contains a subset of the nodes that belong to the system, referred to as a *cluster*. Each cluster maintains a full replica of the system's data. A key-value store constitutes the underlying storage. The storage can be distributed across different servers in the cluster. Since latency between servers within a single datacenter is small, applying current solutions for intra-datacenter concurrency does not affect the overall performance of a geo-replicated system. Our solution handles inter-datacenter concurrency control. MF assumes that transactions executed within a datacenter are serializable. A complete datacenter outage is a rare occurrence. MF focuses on optimizing the normal case operation, that is when all datacenters are available. Clock synchronization between datacenters is not required by MF.

2.2 Message Futures design

MF is a multi-datacenter concurrency control manager. Components in MF's design include: (1) *The transaction client component* to handle clients requests. (2) *The replication component* to handle RLog replication. (3) *The concurrency component* that performs the concurrency logic to commit or abort transactions. These components operate on common global data structures. Each datacenter, DC_i, maintains the following structures:

- *Local RLog*, L_i, is the locally maintained 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.

2.3 Replicated Logs

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 [19]. 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. RLogs are transitive: events of a datacenter DC_A may reach DC_C through another datacenter DC_B. The algorithm ensures two properties. First, all events are eventually known by all datacenters. Second, if two events have a *happened-before* relation [12], their order is maintained in the RLog.

Now we will illustrate our adaptation of 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: (1) *Pending transactions* (p) that requested to commit but are neither committed nor aborted yet. (2) *Committed transactions* (c) that were not propagated to all datacenters yet. A transaction, t_i, starts as a pending transaction with an entry record $E_p(t_i)$. Once the transaction commits or aborts, a new event record, $E_c(t_i)$, is added to the RLog. 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. Note also that the clock must be incremented when new events occur and when L_i is propagated.

3. CONCURRENCY CONTROL

In this section we describe the concurrency control manager. We begin by giving an overview and an example scenario of MF. Then, we present the algorithm used to commit transactions. Finally, we establish the correctness of MF.

3.1 Message Futures overview

We start with an intuitive description of MF. Note that 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- 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 a 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- or write-sets. To commit t_i, DC_A waits until the following *commit condition* holds:

DEFENITION 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*

$$T_A[B,A] \geq t_i \rightarrow LPT_A, \quad \forall_B (DC_B \in datacenters)$$

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

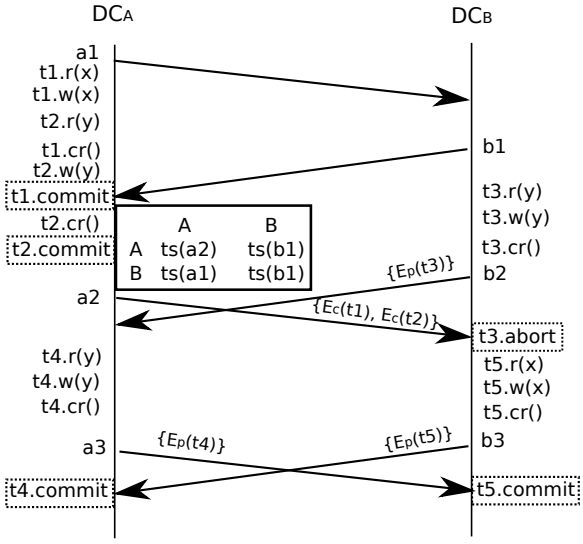


Figure 1: MF example scenario

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.

To illustrate the operational aspects of MF and its correctness, consider a successfully committed transaction, t_i, at datacenter DC_A as an example. The event of transmitting L_A by DC_A is denoted by a_α, where α is a monotonically increasing number; a_α is before a_β if $\alpha < \beta$. The time of an event is represented by the notation $ts(e)$, where e is the event and the returned time is a locally maintained monotonically increasing number. If transaction t_i requests to commit at datacenter DC_A at time $ts(t_i)$, where $ts(a_\alpha) < ts(t_i) < ts(a_{\alpha+1})$, then $t_i{\rightarrow}LPT_A$ is set to $ts(a_\alpha)$. This ensures that the event record of t_i will be included in L_A of every a_β, where β is greater than α. The transaction commits if the commit condition holds. For any other datacenter, DC_B, consider b_α as the first transmission of L_B, where L_B's Timetable satisfies: $T_B[B, A] \geq ts(a_\alpha)$. This is DC_B's part in satisfying t_i's commit condition. Reception of b_α's content by DC_A makes it aware of all transactions, t_{b_pre}, with commit request times less than $ts(b_\alpha)$. No t_{b_pre} conflicts with t_i; otherwise, t_i would have been aborted, since t_{b_pre} is necessarily included in L_B which is transmitted in b_α. Any other transaction, t_{b_post}, that requested to commit after $ts(b_\alpha)$, is necessarily aborted if it conflicts with t_i. This is because $t_{b\_post}{\rightarrow}LPT_B$ is greater than or equal to $ts(b_\alpha)$; t_{b_post} will wait until its commit condition holds. This includes waiting for $T_B[A, B]$ to be greater than or equal to $ts(b_\alpha)$. $ts(b_\alpha)$ is greater than the time when a_α's transmission was received by DC_B. Thus, any L_A satisfying t_{b_post}'s commit condition must have been sent at a time greater than or equal to $ts(a_{\alpha+1})$. L_A sent at a time greater than or equal to $ts(a_{\alpha+1})$ must contain t_i, as we have shown above. Thus, when the record of t_i is received at DC_B, t_{b_post} is aborted if a conflict existed between t_i and t_{b_post}. Since any transaction in DC_B requests to commit either before or after $ts(b_\alpha)$, the commit condition ensures that for any two transactions that are concurrent and conflicting, at most one of them will commit.

We now illustrate a simple operational scenario of MF depicted in Figure 1. 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 $ts(a_2)$.

3.2 Concurrency control protocol

The concurrency component of a datacenter DC_A runs continuously to process L_A and PT_A. Clients read local values and buffer write operations. Conflicts with the local storage are checked by verifying that all read values of t_i have the same version number as the last writes on the local storage. Also, conflicts with other transactions in PT_A, pt, are checked. If a conflict is detected between t_i and pt, t_i is aborted. A conflict exists if a common object, x, exists in pt's write-set and t_i's read- or write-sets. When a transaction, t_i, requests to commit at DC_A, it is added to PT_A (with both read- and write-sets) and L_A (with its write-set only) and $t_i{\rightarrow}LPT_A$ will be set to the current LPT_A. After each RLog transmission, LPT_A is set to the processing time of L_A, which is the entry $T_A[A, A]$. Upon receiving a RLog, L_B, from any DC_B, L_A is updated to reflect the received information, *i.e.*, L_B's events are merged into L_A's events and T_A is updated.

A concurrency server, at DC_A, runs continuously and at each iteration performs three tasks in the following order:

Process transactions, lt, in L_A: This step is demonstrated in Algorithm 1. Two types of transactions in L_A are considered: (1) Committed transactions that were not

Algorithm 1: Processing transactions in L_A.

```
1  function ProcessLocalLog()
2      for each transaction lt in L_A do
3          if (lt was already processed AND is not pending) OR (lt
           is local) then
4              continue to next lt
5          for each transaction pt in PT_A do
6              if conflictExist (pt, lt) then
7                  abort pt
8          if lt is committed then
9              apply lt's write − set to storage
```

Algorithm 2: Processing transactions in PT_A.

```
10 function ProcessPendingTransactions ()
11     for each transaction pt in PT_A do
12         if for all datacenters, DC_B, T_A[B, A] ≥ pt→LPT_A
           AND read-set values are not changed in local storage
           then
13             apply pt's write − set to storage
14             add E_c(pt) to L_A
15             remove pt from PT_A
```

processed before, and (2) pending transactions that did not commit/abort yet. Local transactions in L_A are not considered (line 3). Other transactions are processed according to their order in L_A and checked for conflicts with transactions, pt, in PT_A (lines 5-7). If a conflict was detected between lt and pt, then pt aborts. A conflict exist if a common object, x, exists in lt's write-set and pt's read- or write-sets. Finally, if lt is a committed transaction, incorporate its write-set to the local storage (line 9).

Process transactions, pt, in PT_A: Determine if any pt can commit (Algorithm 2). pt can commit if the commit condition holds (line 12). Committing a transaction includes incorporating its write-set to local storage, adding a committed event record, $E_c(pt)$, to L_A, and removing pt from PT_A (lines 13-15).

Garbage collection and termination: If a transaction, t_i is committed/aborted then discard $E_p(t_i)$. If t_i is committed/aborted and is also known to all datacenters, then discard $E_c(t_i)$.

3.3 Correctness

We now show that MF preserves one-copy serializability. We use Serialization Graphs (SGs) [2]. Nodes in a SG represent committed transactions and edges represent conflicts. An edge (t_i, t_j) represents one of the following three types: (1) *Write-read (wr) edges* where t_j directly *read-depends* on t_i. (2) *Write-write (ww) edges* where t_j directly *write-depends* on t_i. (3) *Read-write (rw) edges* where t_j directly *anti-depends* on t_i. A Multi-Version, Multi-Copy (MVMC) history is one-copy serializable if no cycles exist in its SG. Acyclicity of SG will be proven by showing that any edge (t_i, t_j) translates to an ordering of the commit record of t_i, $E_c(t_i)$, preceding ($<_s$) $E_c(t_j)$ in RLog. Since the order of events in RLogs maintains the *happens-before* relation, and from the acyclicity of the *happens-before* relation, proving this mapping establishes the acyclicity of SG. This is an intuitive mapping since the ordered list of events in RLog actually represents a totally ordered history of committed transactions. We begin by listing some properties of MF to aid us in proving this mapping.

PROPERTY 1. *For a transaction t_j at DC_A, define the set of previously committed transactions by TX_p, where*
$\forall_{t \in TX_p}$ (write-set(t) $\cap$ read-set(t_j) $\neq \phi$) *OR* (write-set(t) $\cap$ write-set(t_j) $\neq \phi$). *All transactions in TX_p have event records in L_A at the time of t_j's commit.*

The property asserts that prior to transaction t_j's commit, DC_A accumulates the history of all previously committed transactions with at least one object x in their write-set that is also in t_j's read- or write-sets.

PROPERTY 2. *The set of committed transactions, TX_c, that were concurrent at any point in time satisfies the following:*
$\forall_{t_j \neq t_i \in TX_c}$ (write-set(t_j) $\cap$ read-set(t_i) = write-set(t_j) $\cap$ write-set(t_i) = ϕ).

This property states that concurrent transactions can commit only if they have disjoint object sets. Now, we will consider each edge type and prove our mapping:

Write-read edges. All transactions read from their local storage. Consider the wr-edge (i, j). A transaction t_j at datacenter DC_A contains version x_v in its read-set. Property 2 indicates that t_i and t_j are not concurrent, thus t_i committed before t_j's commit request. Given Property 1 we also know that all transactions that have key x in their write-sets are in L_A. This includes transaction t_i that wrote the most recent version, *i.e.*, x_v. Thus, the order $E_c(t_i) <_s E_c(t_j)$ is guaranteed in L_A and there is no transaction t_m that writes to x having an order $E_c(t_i) <_s E_c(t_m) <_s E_c(t_j)$. Since the RLog maintains the order of the *happens-before* relation, $E_c(t_i) <_s E_c(t_j)$ will hold for all RLogs.

Write-write edges. Consider two transactions, t_i and t_j, with a common object, x, in their write-sets. Consider an edge (i, j) in SG. We infer from Property 2 that t_i and t_j are not concurrent; t_i committed before t_j requests a commit. Furthermore, Property 1 indicates that L_A accumulated the set of all previously committed transactions, TX_p, that have at least one common object in their write-sets with t_j. Thus, t_i which wrote the most recent version of x before t_j is necessarily in TX_p. Thus, the order $E_c(t_i) <_s E_c(t_j)$ is ensured in L_A and there is no transaction t_m that writes to x with the order $E_c(t_i) <_s E_c(t_m) <_s E_c(t_j)$. The order holds for all RLogs due to the preservation of the *happens-before* relation.

Read-write edges. Consider the case of three transactions, t_i, t_j, and t_k accessing an object x. Assume that t_j and t_i request to commit at DC_A and DC_B respectively. Assume there is a ww-edge from t_k to t_j and a wr-edge from t_k to t_i, then there should be a rw-edge from t_i to t_j. We need to prove the mapping of those transactions to an order $E_c(t_k) <_s E_c(t_i) <_s E_c(t_j)$. We already proved that $E_c(t_k) <_s E_c(t_i)$ and $E_c(t_k) <_s E_c(t_j)$. We now need to only prove that $E_c(t_i) <_s E_c(t_j)$. Suppose to the contrary that $E_c(t_j) <_s E_c(t_i)$. This is only possible if $E_c(t_j)$ was already in L_B when t_i commits. There are three cases of

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

Table 1: RTT latencies between different datacenters in milliseconds.

possible times for t_j's reception at DC_B: (1) $E_c(t_j)$ was in L_B before t_i requested to commit. This case is not possible, since MF compares the read versions against the ones in storage, and having $E_c(t_j)$ in L_B necessarily means that t_j's write-set was incorporated in DC_B. (2) $E_c(t_j)$ was added to L_B while t_i was a pending transaction in PT_B. This is also not possible because all pending transactions are checked for conflicts with any externally received transaction. So, when $E_c(t_j)$ was first received at DC_B, transaction t_i would have been aborted. (3) $E_c(t_j)$ arrived at DC_B after t_i commits. However, when t_i commits, the entry $E_c(t_i)$ is added to L_B. Thus, $E_c(t_j)$ is necessarily added after $E_c(t_i)$, hence $E_c(t_i) <_s E_c(t_j)$. A contradiction is observed for all cases. Having no transaction t_m that writes to x with an order $E_c(t_i) <_s E_c(t_m) <_s E_c(t_j)$ follows from our proof of ww-edges.

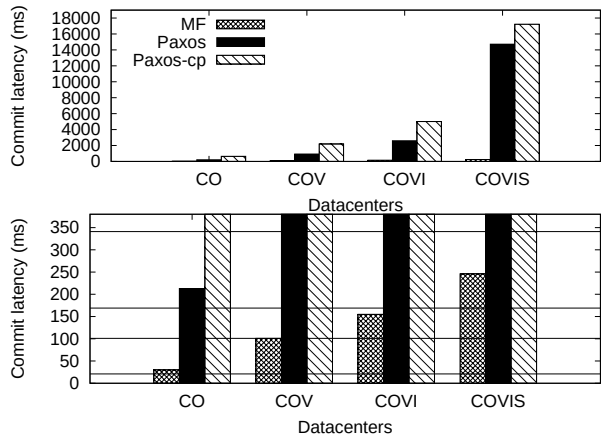
The following theorem shows that our mapping is sufficient to prove one-copy serializability.

THEOREM 1. *MF ensures one-copy serializability.*

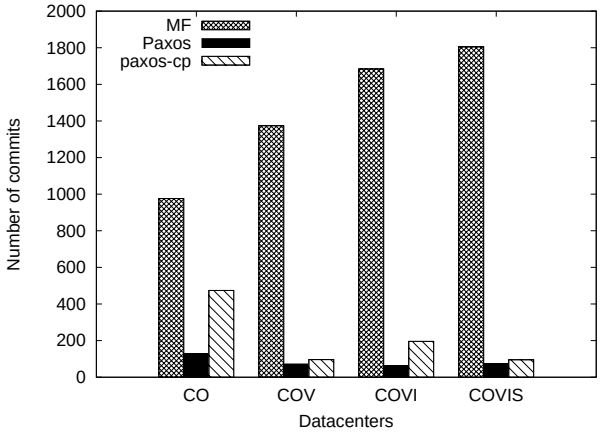
Proof: MF guarantees the mapping of SG edges, (i, j), to an order $E_c(t_i) <_s E_c(t_j)$ in RLogs as we showed above. The order of events in RLogs guarantees the *happens-before* relation between events. The *Happens-before* relation is acyclic. Since in SG, there is an edge from t_i to t_j only if $E_c(t_i) <_s E_c(t_j)$, a cycle in SG will amount to a violation of the *happens-before* relation exhibited in RLogs. Thus, SG is acyclic, hence MF is one-copy serializable. ■

4. EXPERIMENTAL EVALUATION

We performed an evaluation of MF on Amazon EC2. The results are compared with a Paxos-based log replication protocol and an optimized version of it, called Paxos-CP [16]. Our implementation of the Paxos-based log replication protocol is based on megastore [1] and we refer to it as the Paxos commit protocol (or simply Paxos) in the rest of the paper. Paxos-CP is a variant of the Paxos commit protocol that allows more concurrency via *promotion* and *combination*. We use machines in five EC2 datacenters: California (C), Oregon (O), Virginia (V), Ireland (I), and Singapore (S). RTT latencies across datacenters are shown in Table 1. Each datacenter runs an instance of MF. We run experiments on combinations of those datacenters. A combination is represented by a grouping of the initials of the used datacenters; experiment CI for example is a scenario of two replicas: one in California and the other in Ireland. HBase [10] is used as the underlying key-value store. Yahoo! cloud serving benchmark (YCSB) [3] is leveraged to evaluate the system. Since YCSB does not support transactions, an extended version of YCSB is used to support transactions and transactional workloads [6].



(a) Average commit latency.



(b) Number of successful transaction commits.

Figure 2: Transactions average commit latency and number of commits for scenarios with different numbers of datacenters.

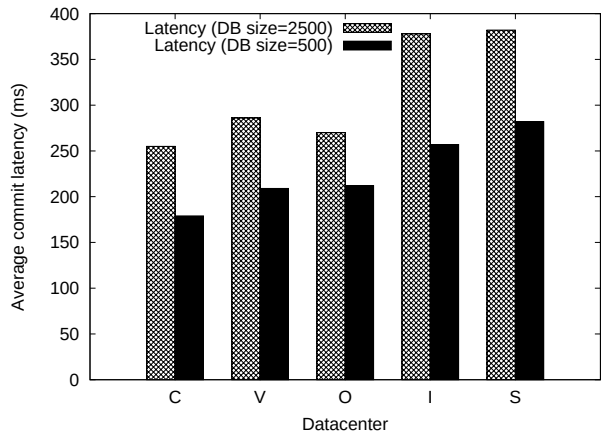
Our experiments focus on measuring the commit latency and the amount of concurrency in the system. Our results show that MF commits transactions, of a datacenter i, quickly with a latency close to RTT^i_{max}, where RTT^i_{max} is the maximum RTT experienced by datacenter i and any other datacenter in the scenario. We use the notion RTT_{max} to denote the maximum RTT^i_{max} for all datacenters i. Other experiments detail the behavior of MF while varying contention, write skewness, and propagation intervals. Also, we report the results of varying propagation intervals in a selected datacenter and observe immediate transaction commits for many cases. Thus, a low average commit latency is demonstrated for the datacenter with the longer propagation interval. Results are detailed next.

Message Futures performance. Our first set of experiments consists of four scenarios, CO, COV, $COVI$, and $COVIS$. We will report the average commit latencies and the total number of commits. In this study we compare MF's results to those obtained from Paxos and Paxos-CP. A relatively conservative workload is used for the comparison. This is due to Paxos and Paxos-CP's inability to scale for even such a conservative workload and to enable us to observe the characteristics of MF in the normal case oper-

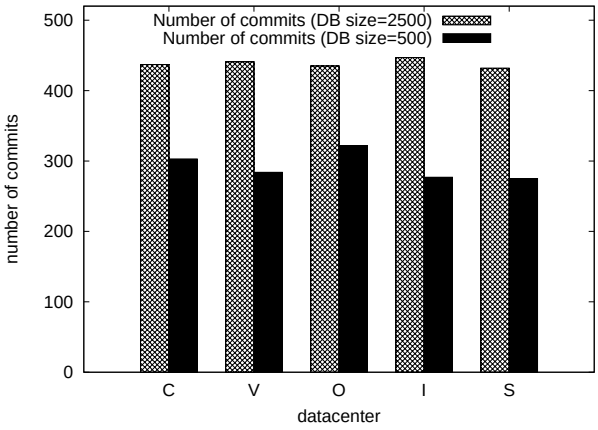
ation. The baseline workload employs five client threads in each datacenter. The rate of requests is 10 transactions per second for each datacenter; for five datacenters the overall rate is 50 transactions per second. We run the experiments for a total of 500 transactions in each datacenter. Each transaction accesses 5 objects uniformly from a pool of 1000 keys. Operations are 50% reads and 50% writes. The interval between two RLog transmissions is called the *propagation interval*. Unless stated otherwise, the propagation interval of any datacenter is set to 100ms, hence 10 RLogs are sent per second.

The results of this experiment are shown in Figure 2. Average commit latencies are shown in Figure 2(a). The figure is divided into two parts. The top part shows the whole range of obtained results. To enable observing MF results more closely, the lower part is a magnification of the range which shows commit latencies lower than 360ms. To facilitate comparing with RTT_{max}, RTT_{max} values are plotted as horizontal lines for all scenarios. Commit latencies for MF are close to RTT_{max} and even below it for larger number of datacenters. This is because a datacenter, DC_i, achieves an average commit latency relative to the its RTT^i_{max} value, which is smaller than or equal to RTT_{max}. Also, more aborted transactions yield lower average commit latency, since most aborts in MF are immediate. MF performs considerably better than Paxos and Paxos-CP, where the average commit latency is more than 14 seconds for $COVIS$ compared to 246ms for MF. The number of commits experienced by the system is affected by the increase in the number of datacenters and transactions per second (Figure 2(b)). MF, however, manages to achieve an average number of 361 commits in the worst case ($COVIS$), whereas Paxos and Paxos-CP were only able to achieve an average number of 64 and 237 commits respectively in their best case (CO). Note that MF's commit ensures that the commit record is persistent in at least one datacenter, whereas Paxos and Paxos-CP ensures that the commit record is persistent in at least a majority.

The commit latency depends on the issuing datacenter. As we have shown, the commit latency is a function of RTT^i_{max}. However, RTT^i_{max} values are different for datacenters in the same scenario. From Table 1 for example note that RTT^C_{max} equals 173ms, where RTT^S_{max} equals 341ms, almost double the value of datacenter C. Thus, we expect the commit latency of a datacenter to be different than other datacenters according to its RTT^i_{max} value. To illustrate this, results clustered according to the issuing datacenter are shown in Figure 3. Two sets of results are shown for different database sizes, namely 500 and 2500 data objects. It is clear that datacenters C, V, and O perform better than their counterparts with higher RTT^i_{max} values. In Figure 3(a), the results of individual latencies show that C, V, and O achieve about two thirds the latency of I and S for the case of 2500 data objects. The effect is less apparent for the higher contention case, *i.e.*, where the number of data objects is 500. This is due to the higher abort rate. Since most aborts are either immediate or require much lower latency, they are not as much affected by the wide-area communication latency. A point to consider here is that I and S achieve commit latencies that are close to their respective RTT^i_{max}, whereas C, V, and O are not. This is due the effect of our choice of



(a) Latency for committed transactions.

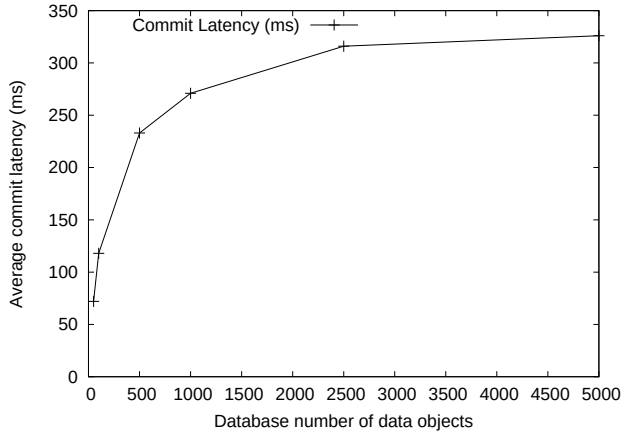


(b) Number of successful transaction commits, out of 500 transactions.

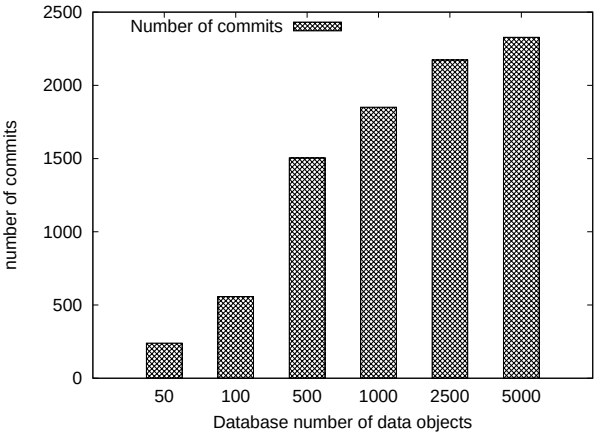
Figure 3: Detailed performance of each datacenter in a CVOIS scenario.

propagation latency which has a more visible effect for lower RTT^i_{max} values. It will be shown in a later experiment (Figure 6) how enforcing a lower propagation interval will make the commit latency of datacenter C close to RTT^C_{max}. The number of commits, shown in Figure 3(b), do not show the same variation and dependence on RTT^i_{max}. The number of commits are only slightly better for C and O for scenarios with higher contention.

Contention effect. In the next set of experiments we would like to study the effect of contention on the performance of MF. This is tested by varying the size of the database from 50 to 5000 data objects on a $CVOIS$ scenario. Figure 4 shows obtained results. Observe how the effect of high contention is apparent for values smaller than 500 data objects. The results then gradually converge to performance results close to scenarios with no contention. For example, the number of commits, as shown in Figure 4(b), increase by 7% when the number of data objects is increased from 2500 to 5000. This is at the cost of a 3% increase in commit latency (Figure 4(a)). Two main observations are to be taken from these results. First, the effect of increasing the number of data objects diminishes as the size of the database increases. Second, there is a trade-off between commit la-

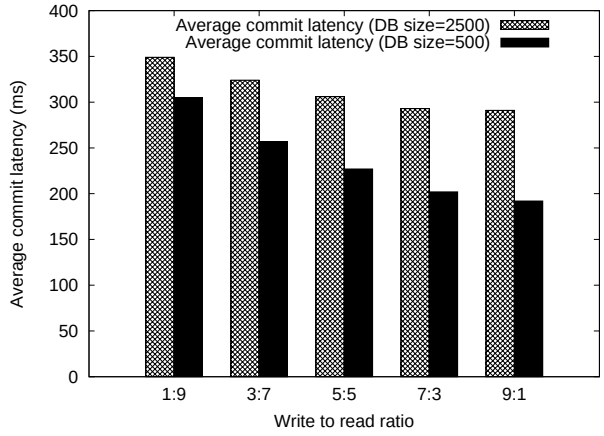


(a) Latency for committed transactions.

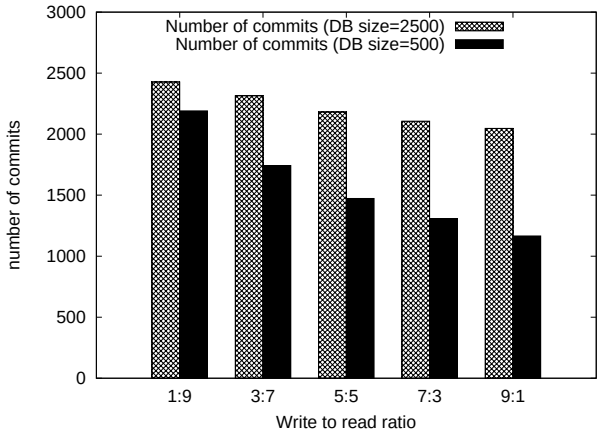


(b) Number of successful transaction commits, out of 2500 transactions.

Figure 4: The effect of contention on Message Futures.



(a) Latency for committed transactions.



(b) Number of successful transaction commits, out of 2500 transactions.

Figure 5: The effect of Write-to-read ratio on Message Futures.

tency and number of commits with respect to the size of the database. This is an important factor for deciding a suitable granularity for a given database.

Write to read ratio. In all previous experiments an operation can be either a read or a write with equal probability. While this can be considered as a workload with a high number of writes, it is necessary to investigate the behavior of MF for different write to read ratios. The results for varying the write to read ratio are shown in Figure 5 for a $CVOIS$ scenario. Two sets of experiments were performed with different database sizes, *i.e.*, 500 and 2500 data objects. Commit latency results, shown in Figure 5(a), demonstrate the degrading effect of increasing write to read ratio. This effect is larger for scenarios with higher contention. Consider going from a write to read ratio of 1:9 to 9:1. A degradation of 16.6% is observed for a database with 2500 data objects compared to a degradation of 37% for a database with 500 data objects. The number of commits experiences similar behavior where a degradation of 15.5% is observed for a size of 2500 compared to 46.8% for a size of 500 data objects.

Propagation interval effect. Now we study the effect of the propagation interval on MF. To illustrate this effect, we plot the Cumulative Distribution Function (CDF) of transactions' commit latencies for Datacenter C in a $CVOIS$ scenario for different propagation interval values. In Figure 6, the propagation interval is increased from 10ms to 200ms. Note that the propagation interval changed in this set of experiments is for all datacenters, hence it is a global propagation interval. Increasing the propagation interval causes an increase in commit latencies. Furthermore, increasing the global propagation interval causes the commit latency values to be more variant. For example, with a global propagation interval of 200ms, the bulk of commit latency values range from 300 to 540ms, whereas a global propagation interval of 10ms causes the range of the bulk of commit latency values to be from 170 to 250ms. The range of values of the latter is almost one third of the former.

Performance prioritization In Section 3 we showed how a transaction can immediately commit if the commit condition is already satisfied. Commit latency and regions of immediate commits highly depend on the dynamics of transmitting and receiving RLogs. For example, immediate commits will only occur when the commit condition with respect to all other datacenters is satisfied. Smart scheduling of RLog transmission will increase the duration of immediate commits. In the following experiment we use a fixed rate

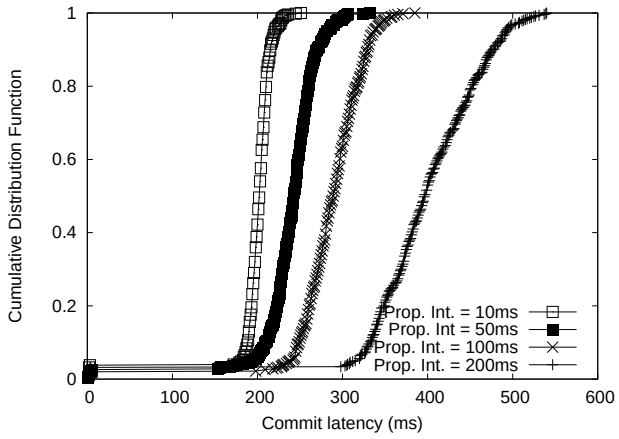


Figure 6: Cumulative density function of commit latency of 500 transactions at datacenter C for four global values of propagation intervals.

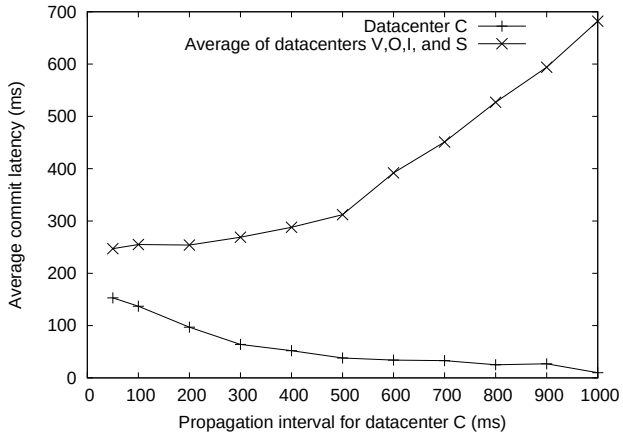


Figure 7: Transactions average commit latency while increasing the propagation interval of datacenter C in a CVOIS scenario.

scheduler with different rates to observe the effect of increasing the propagation interval of one datacenter and how that enables us to achieve higher performance results for it. In this experiment, we report the results obtained for C and compare it against the average of all other datacenters in a $COVIS$ scenario. Datacenters other than C have their propagation interval fixed at 10ms in all runs to observe the effect of C's propagation interval in isolation. The results are shown in Figure 7. As the propagation interval of C increases, an improvement of C's average latency is experienced. This improvement is at the expense of an increase of the average commit latency of all other datacenters. An important point to observe in C's propagation interval values is 500ms. C's commit latency at this point is 75% less than the run with a propagation interval of 50ms for C. Before this point, the average of other datacenter increases slowly, *i.e.*, an increase of 65ms. However, after the 500ms point, although C's commit latency continue to decrease, this is at a higher cost for other datacenters' commit latencies. Observe that compared to the point at 500ms, when C's propagation interval increases to 1000ms its average commit latency improves by dropping another 74% of its latency to be 10ms

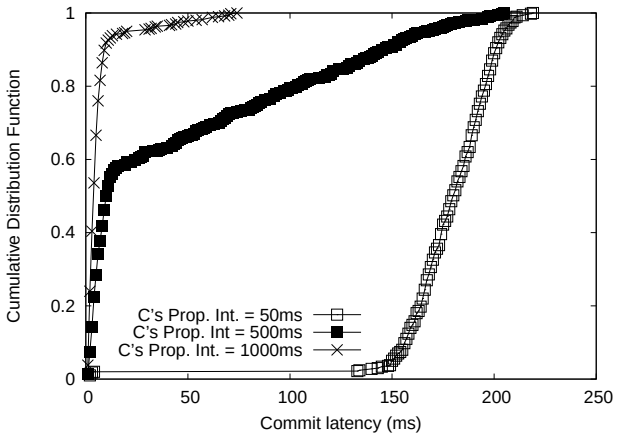


Figure 8: Cumulative Distribution Function of commit latencies of 500 transactions in datacenter C with different propagation interval values.

whereas the average of other datacenters increase by 119%. Changing the propagation interval does not show a clear effect on the number of commits for C; a number of commits between 376 and 426 is maintained for all runs. The effect on other datacenters is a slowly decreasing average number of commits from 386 to 345 successful commits.

To study the effect of increasing the propagation interval of one datacenter we performed another set of experiments to observe the commit latency values of all transactions. This will allow us to extract the number of transactions that committed immediately and show how other transactions behave. The experiments were performed on a $CVOIS$ scenario where C is the only datacenter issuing transactions. Datacenters other than C transmit RLogs with a fixed interval of 10ms. The results are shown in Figure 8. A CDF of obtained commit latency values of scenarios with different propagation intervals are plotted. Note how increasing the propagation interval increases the amount of immediate commits. About 60% and 90% immediate commits are recorded for a propagation interval of 500ms and 1000ms, respectively. Also, note that transactions collectively performs better as the propagation interval increases.

5. RELATED WORK

State of the art solutions vary significantly. In this section, we summarize the literature of concurrency management for multi-datacenter environments. Then, we attempt to describe MF's approach in contrast to other work.

Two-Phase Commit (2PC), and other traditional protocols, were not intended for geographically separated datacenters. 2PC requires a coordinator, introducing a single point of failure. Lock and release exchanges require at least two RTTs for each transaction in addition to blocking other stores.

Paxos [13] is a consensus algorithm that could be leveraged to implement concurrency control for multi-datacenters [1, 9, 11, 16]. Paxos takes at least two rounds of communication to reach consensus on a value, a prepare phase and an acceptance phase. It is a majority algorithm, meaning that it can reach consensus by coordinating with at least a ma-

jority of nodes. Gray and Lamport [9] propose using Paxos to commit protocols in order to avoid the blocking problem incurred in 2PC. Another way to use Paxos for concurrency control is via log replication [1, 16]. In those protocols, participants compete on writing to log positions. By reaching consensus on every position in the log, correctness is maintained, since all participants agree on the transaction order. MDCC [11] 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. Paxos, though master-less, requires at least two RTTs for transaction commits. Also, it cannot handle highly concurrent scenarios as is, since transactions contend even if they were not in conflict. Optimizations were proposed to minimize latency by using a variant called Fast Paxos [14]. In it, the master is bypassed, thus allowing for collisions. Collisions require master intervention and retraction to classic Paxos, leading to poorer performance for conflicting transactions.

A Transactional layer that sits over a replication layer was also studied and implemented in large-scale systems [5, 7, 8]. Google Spanner [5] is a recently introduced global-scale concurrency control manager used by Google. Data are partitioned and each partition is replicated across datacenters. Leaders are assigned to each partition. Transactions read from the leader and read objects are locked. Transactions commit by running 2PC on partition leaders. In this paradigm, 2PC and Two-Phase Locking are used to achieve concurrency control and Paxos is used to achieve replica consistency. Scatter [8] is also another solution that uses 2PC over Paxos in a fashion similar to Spanner. It is a key-value store that uses Distributed Hash Tables.

Geo-replication of data stores was tackled by recent work [4, 15, 17]. These systems sacrifice strong consistency for better performance. Data stores are partitioned and each partition is assigned a master. Thus, requests close to the master are served without being affected by the large latency between data centers. However, this approach is only effective if data can be localized efficiently and requests access a restricted partition. PNUTS [4] uses a reliable Pub/Sub message passing system to transmit updates asynchronously across data centers. PNUTS, however, is a key-value data store and does not support transactions. Walter [17] extends Snapshot Isolation to a variant called Parallel Snapshot Isolation (PSI). PSI relaxes the snapshot read condition to be the last committed version in the local data center rather than the last committed version globally. Also, write-write conflicts should not be concurrent in any single site. The final property of PSI is the preservation of commit causality across sites. COPS [15] delivers causal consistency with convergent conflict handling, called *causal+*. Causal consistency is observed by maintaining the causal dependency of operations and ensuring that the propagation of operations from one data center to the other follows the causal order in the local data store. Calvin [18] is a recently proposed protocol for distributed transactions that trades throughput for latency. Global transaction ordering is achieved by ordering transactions in batches. Unfortunately, this makes transactions experience large commit latencies.

In our work we abstract a multi-datacenter environment as a hierarchy of two levels. Those are the inter- and intra-datacenter levels. There are two main characteristics that differentiate those two levels: (1) coordination between nodes are much more expensive across datacenters. (2) machine failures are more frequent than datacenter outages. Observing those differences indicates that an inter-datacenter concurrency manager should target minimizing the cost of wide-area replication, whereas an intra-datacenter concurrency manager should target increasing fault-tolerance. The paper focuses on the former target by proposing MF to be an inter-datacenter concurrency manager. By abstracting each datacenter as a single entity we minimize the required communication and by designing with wide-area latency awareness we minimize the cost of wide-area coordination. Traditional Paxos solutions do not provide such an abstraction, thus increasing message complexity and inter-datacenter coordination cost. Solutions with a transactional layer on top of a replication layer increases the required wide-area messaging. Systems that follow this paradigm require planned machine placement to overcome the cost of wide-area messaging, *i.e.*, placing a majority of machines in close proximity to each other. However, placing a majority in close proximity will make the system vulnerable to natural disasters. Furthermore, since the majority is at close proximity, any client, no matter where it is, will be required to coordinate with at least one of them. Our approach divides the problem into two levels, where each level is transactional, and replication is achieved by executing transactions at each datacenter.

6. CONCLUSION

We proposed Message Futures, a geo-replicated concurrency control manager for multi-datacenter environments. It exhibits low commit latency of transactions, with latency close to the RTT of the system. Proactive message passing of state and transaction information enables datacenters to infer enough information to commit transactions. RLogs are incorporated in the MF design to increase fault tolerance and leverage its conservation of the *happens-before* relation. We demonstrated how our scheme can be used to prioritize the performance of different datacenters. A datacenter with a larger propagation interval experiences lower commit latencies and the immediate commitment of transactions in many cases. An evaluation on an inter-continental setting was performed to demonstrate MF's performance and validate our claims.

7. ACKNOWLEDGMENTS

The authors would like to thank Aaron Elmore and Alex Pucher for the initial discussions on the design, the anonymous reviewers, and Ken Salem for their insightful comments. This work is partially supported by NSF Grant 1053594. Faisal Nawab is funded by a fellowship from King Fahd University of Petroleum and Minerals. We Also would like to thank Amazon for access to Amazon EC2.

8. REFERENCES

[1] Jason Baker, Chris Bond, James Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly

available storage for interactive services. In *CIDR*, pages 223–234, 2011.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

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

[4] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

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

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

[7] D.K. Gifford. Information storage in a decentralized computer system. *Dissertation Abstracts International Part B: Science and Engineering[DISS. ABST. INT. PT. B- SCI. & ENG.], 143*, page 1981, 1981.

[8] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 15–28. ACM, 2011.

[9] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

[10] HBase. `http://hbase.apache.org`, 2011. [Online; acc. 18-Jul-2011].

[11] Tim Kraska, Gene Pang, Michael J. Franklin, and Samuel Madden. Mdcc: Multi-data center consistency. *CoRR*, abs/1203.6049, 2012.

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

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

[14] Leslie Lamport. Fast paxos. *Distributed Computing*, 19:79–103, 2006. 10.1007/s00446-006-0005-x.

[15] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[16] Stacy Patterson, Aaron J. Elmore, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *PVLDB*, 5(11):1459–1470, 2012.

[17] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[18] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[19] Gene T.J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, PODC '84, pages 233–242, New York, NY, USA, 1984. ACM.