Granola: Low-Overhead Distributed Transaction Coordination

James Cowling
MIT CSAIL

Barbara Liskov
MIT CSAIL

Abstract

This paper presents Granola, a transaction coordination infrastructure for building reliable distributed storage applications. Granola provides a strong consistency model, while significantly reducing transaction coordination overhead. We introduce specific support for a new type of *independent* distributed transaction, which we can serialize with no locking overhead and no aborts due to write conflicts. Granola uses a novel timestamp-based coordination mechanism to order distributed transactions, offering lower latency and higher throughput than previous systems that offer strong consistency.

Our experiments show that Granola has low overhead, is scalable and has high throughput. We implemented the TPC-C benchmark on Granola, and achieved 3× the throughput of a platform using a locking approach.

1 Introduction

Online storage systems run at very large scale and typically partition their state among many nodes to provide fast access and sufficient storage space. These systems need to provide persistence, availability, and good performance.

It is also highly desirable to run operations as *atomic transactions*, since this greatly simplifies the reasoning that application developers must do. Transactions allow users to ignore concurrency, since all operations appear to run sequentially in some serial order. Most distributed storage systems do not provide serializable transactions, however, because of concerns about performance and partition tolerance. Instead, they provide weaker semantics, e.g., eventual consistency [14] or causality [26].

This paper presents Granola, an infrastructure for building distributed storage applications where data resides at multiple storage repositories. Granola supports atomic transactions, and provides serializability across all operations. Granola also provides persistence and high availability, along with low per-transaction overhead.

Granola provides transaction ordering, atomicity and reliability on behalf of storage applications that run on the platform. Applications specify their own operations, and Granola does not interpret operation semantics. Granola can thus be used to support a wide variety of storage systems, such as databases and object stores. Granola implements atomic *one-round* transactions. These execute in one round of communication between a user and the storage system, and are used extensively in online transaction processing workloads to avoid the cost of user stalls [7,20,32].

Granola supports three classes of one-round transactions. *Single-repository transactions* execute on a single storage node; we expect that most transactions will be in this class, since data is likely to be well-partitioned. *Coordinated distributed transactions* execute atomically across multiple storage nodes, and commit only if all participants vote to commit; these transactions are what is provided by traditional two-phase commit. We also support a new transaction class, which we term *independent distributed transactions*. These execute atomically across a set of nodes, but do not require agreement, since each participant will independently come to the same commit decision. Examples include an operation to give everyone a raise, an atomic update of a replicated table, or a read-only query that obtains a snapshot of distributed tables.

Granola uses a timestamp-based coordination mechanism to provide serializability for single-repository and independent transactions *without* locking, using clients to propagate timestamp ordering constraints between repositories. This provides a substantial reduction in overhead from locking, log management and aborts, and a consequent improvement in throughput. Granola provides this lock-free coordination protocol while handling single-repository and independent transactions, and adopts a lock-based protocol when handling coordinated transactions. Granola's throughput is similar to existing state-of-the-art approaches when operating under the locking protocol, but significantly higher when it is not.

Granola provides low latency for all transaction types:

we run single-repository transactions with two one-way message delays plus a stable log write, and both types of distributed transactions usually run with only three messages delays plus a stable log write. This is a significant improvement on traditional two-phase commit mechanisms, which require at least two stable log writes, and improves even on modern systems such as Sinfonia [7], which requires at least four one-way message delays (for a remote client) and one stable log write.

Our experiments show that we can provide 3× greater throughput on the TPC-C benchmark compared to using a locking approach.

2 Transaction Model

Granola supports *one-round transactions*, which are expressed in a single round of communication between the client and a set of repositories; we refer to the set of repositories as transaction *participants*. The client application specifies what operations to execute and Granola ensures that these are executed atomically at all participants.

One-round transactions are distinct from general database transactions in two key ways: One-round transactions do not allow for interaction with the client, where the client executes multiple sub-statements (e.g., queries) before issuing a transaction commit; transactions are instead specified as a single operation that executes atomically at each participant. One round transactions also execute to completion at each participant, with no communication with other repositories, apart from at most a single commit/abort vote. Despite these restrictions, one-round transactions are still a powerful primitive. They are used extensively in online transaction processing workloads to avoid the cost of user stalls [7, 20, 32], and map closely to the use of stored procedures in a relational DBMS.

2.1 Why Independent Transactions?

As mentioned, Granola supports three types of one-round transactions: single-repository, coordinated, and independent transactions; this last category is a primary contribution of the Granola protocol.

Coordinated transactions incur significant cost for locking and undo-logging; previous studies have estimated overhead to be 30–40% of CPU load under typical workloads [18]. These also incur overhead when retrying transactions that block or abort due to contention [7]. Our motivation for independent transactions was to explore the most powerful primitive we could provide without incurring this overhead. Independent transactions execute atomically across a set of participants, but do not require locking or undo-logging, and do not contend with other transactions. This provides us a significant performance advantage, as seen in Section 6.2.

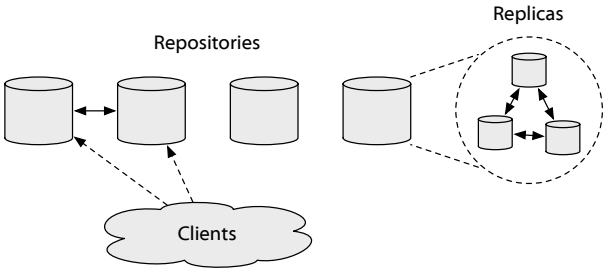

Figure 1: System topology.

Our approach was influenced by the H-Store project [32], which identified a large class of operations in real-world applications, deemed as *one-shot* and *strongly two-phase*, that fit our independent transaction model. This work did not provide a functional protocol for coordinating independent transactions in a distributed setting [19], however. To our knowledge, no previous system provides explicit support for independent transactions.

Independent transactions are appropriate for distributed operations where all participants will make the same local decision whether to commit or abort. This includes distributed read-only transactions, common in read-heavy workloads when data spans multiple partitions, or transactions where the commit decision is a deterministic function of shared data. Application developers commonly replicate tables when partitioning data, to ensure that the majority of transactions are issued to a single partition [20, 32]; independent transactions can be used to atomically update replicated data, or to execute distributed transactions predicated on replicated data.

The common TPC-C benchmark [6], designed to be representative of a typical online transaction processing workload, can be partitioned so that operations consist entirely of single-repository transactions and independent transactions [32]. For example, `new_order` transactions only abort if a request contains an invalid item number, which can be computed locally if the Item table is replicated at each participant. Modifications to the Item table could also be performed using independent transactions.

3 Architecture and Assumptions

This section describes our architecture and application interface. It also discusses our assumptions.

3.1 Architecture

Granola contains two types of nodes: *clients* and *repositories*. Repositories are the server machines that store data and execute transactions, while clients interact with the repositories to issue transaction requests. Repositories communicate with one another to coordinate transactions,

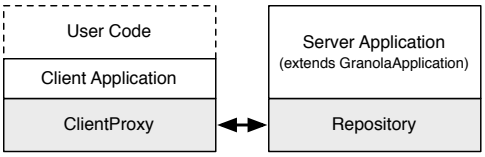

Figure 2: Logical structure.

whereas clients typically interact only with the repositories. This topology is illustrated in Figure 1. Each repository is comprised of a number of replica nodes, to provide reliability; we discuss replication in Section 5.1.

Applications link against the Granola library, which provides functionality for correctly ordering transactions and delivering them reliably to each server application. Applications are layered atop the Granola client and repository code, as shown in Figure 2.

3.2 Client Organization

The *client application* code is provided by the application developer, and supports the desired interface to user code, e.g., support for queries in a database system. The client application is responsible for determining which repositories are required for a given user request, and choosing which transaction class to use.

The application issues transactions by interacting with the Granola *client proxy*, using the interface shown in Figure 3.[1] The application specifies the repository ID (RID) for each participant, the operation to run at each participant, and whether the transaction is read-only. The client proxy interacts with the Granola repository code at the participants, and handles all other client-side functionality, including providing a *TID* (transaction identifier) for each request. The TID uniquely identifies each request, and consists of the client ID and a sequence number that increases for each request from that client. The client proxy also manages timestamps, which are used to ensure serializability as discussed in Section 4.

3.3 Server Organization

The Granola repository code prepares and executes transactions by making upcalls to the server application. The server application must implement the `GranolaApplication` interface shown in Figure 4. Application upcalls supply the transaction operation and retrieve a result, both of which are uninterpreted by Granola. The server application runs in isolation at each repository, and does not need to communicate directly with other repositories, since this functionality is provided by the

[1]While we show a blocking interface for the client proxy, the user can issue multiple requests concurrently; Granola determines the relative ordering of concurrent transactions.

```
// issue trans to given repository
// writes result into provided buffer
void invokeSingle(int rid, ByteBuffer op,
        boolean ro, ByteBuffer result);

// issue trans to set of repositories
void invokeIndep(List<Integer> rids,
        List<ByteBuffer> ops, boolean ro,
        List<ByteBuffer> results);

// issue trans to set of repositories
// returns application commit/abort status
boolean invokeCoord(List<Integer> rids,
        List<ByteBuffer> ops,
        List<ByteBuffer> results);
```

Figure 3: Client API.

Granola repository code. We discuss the use of the server API in Section 4, and the recovery interface in Section 5.4.

3.4 Assumptions

For the purposes of this paper, we assume that the set of repositories is fixed and well-known; system reconfiguration is outlined in our extended description of the protocol [13], along with other protocol details.

While each repository can process many transactions in parallel, each application upcall is executed sequentially, saving considerable overhead over the cost of latching and concurrency control [32]. This approach works well assuming in-memory workloads, as is common in most large-scale transaction processing applications [32]. Multiple repositories may be colocated on a single machine to take advantage of multicore systems.

Granola tolerates crash failures. Our replication protocol depends on replicas having loosely synchronized clock *rates* [25]. We depend on repositories having loosely synchronized clocks for performance, but not correctness.

4 Protocol Design

This section describes the Granola protocol. We first discuss the timestamps used to provide serializability, followed by our protocols for the three transaction classes.

Each repository runs in one of two modes. When there are no coordinated transactions running at a repository it runs in *timestamp mode*. Sections 4.3 and 4.4 describe our protocols for single-repository and distributed transactions as they work in timestamp mode. When the repository receives a request for a coordinated transaction it switches to *locking mode*. Section 4.5 describes how the system runs in this mode and how it transitions between modes.

The repository interacts with the server application differently in the two different transaction modes. In time-

```
Independent Interface
// executes transaction to completion.
// returns false if lock conflict
boolean run(ByteBuffer op, ByteBuffer result);

Coordinated Interface
// runs to commit point and acquires locks.
// returns COMMIT/ABORT vote or CONFLICT
// result is empty unless returning ABORT
AbortType prepare(ByteBuffer op, long tid,
        ByteBuffer result);

// commits trans and releases locks
void commit(long tid, ByteBuffer result);

// aborts trans and releases locks
void abort(long tid);

Recovery Interface
// force-acquires any locks that could be
// required at any point in the serial order
// returns true if no conflict
boolean forcePrepare(ByteBuffer request,
        long tid);
```

Figure 4: Server API.

stamp mode, all transactions are executed using the `run` upcall, which executes the transaction to completion. The `prepare` upcall is used for distributed transactions when in locking mode, to acquire locks on the transaction and determine the commit or abort vote. The response to the `prepare` upcall can indicate COMMIT, ABORT, or CONFLICT. COMMIT indicates that the application has acquired the locks needed by the request while CONFLICT means that some locks cannot be acquired and therefore the client should retry the transaction. ABORT means that the application has decided to abort the transaction based on application logic, e.g., the application refuses to decrement an account balance because the balance is too small. If the application returns ABORT it can also include additional information for the client in the `result` buffer. The ABORT response occurs only for coordinated transaction requests.

In the following description we note where a *stable log write* is required. This step involves the primary replica executing state machine replication, as described in Section 5.1. Unless specified, no other replication occurs and communication is solely between the primary replicas at each repository.

4.1 Timestamps

Granola uses timestamps to order distributed transactions without locking. Each transaction is assigned a timestamp, which defines its position in the global serial order. A transaction is ordered before any transaction with a larger timestamp; if two transactions have the same timestamp, the transaction with the lower TID is ordered first.

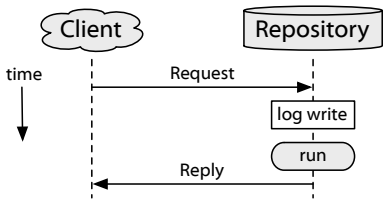

Figure 5: Timeline for single-repository transactions.

Repositories select the timestamp for each transaction based on their clock. Repositories exchange timestamps before committing a given transaction, to ensure that they all assign it the same timestamp. Each transaction result sent to the client contains the timestamp for that transaction, and each request from the client contains the latest timestamp observed by the client, ensuring that timestamp dependencies are maintained. We explain how timestamps are used in the following sections.

4.2 Client Protocol

The client proxy receives transaction invocation requests from the client application via the interface specified in Figure 3. Each client proxy maintains *highTS*, the highest timestamp it has observed in a transaction response, initially 0. The client proxy issues a transaction REQUEST to the repositories specified by the client application, along with the highTS value and the TID.

The client proxy then waits for REPLY messages from the participants. If it receives COMMITs from all participants, it returns the results to the client application. If the proxy receives an ABORT response from some repository, it returns false; if it receives a CONFLICT response, it retries the transaction with a new TID, after waiting a random backoff.

4.3 Single-Repository Transactions

The basic protocol for single-repository transactions has much in common with how existing single-node storage systems work. The protocol timeline is shown in Figure 5.

The protocol for read-write transactions is as follows:

1. When a repository receives a client REQUEST it assigns it a timestamp that is greater than the highTS sent by the client, the timestamp of the most recently executed transaction at the repository, and the current clock value.

2. The repository performs a stable log write to record both the request and the assigned timestamp, so that this information will persist across failures.

3. The transaction is now ready to be executed. Transactions are executed in timestamp order, by making

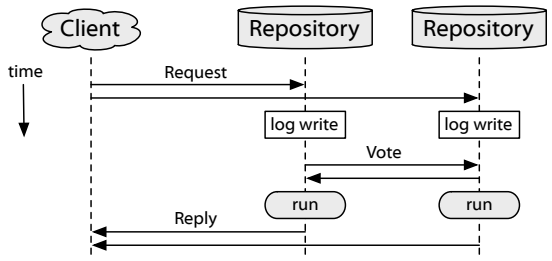

Figure 6: Timeline for independent transactions.

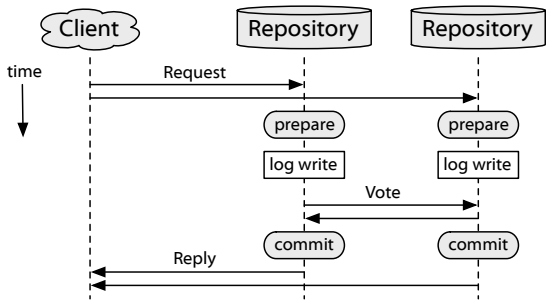

Figure 7: Timeline for coordinated transactions.

a `run` upcall to the application. Once a transaction is executed, a COMMIT reply containing the result of the upcall is sent to the client.

Additional transactions can be processed while awaiting completion of a stable log write; these requests will be executed in timestamp order.

The protocol for read-only transactions is the same as for read-write transactions, except that a stable log write is not required in Step 2 of the protocol. Since read-only transactions do not modify the service state, they can be retried in the case of failure.

4.4 Independent Distributed Transactions

Independent transactions are ordered with respect to all other transactions without locking or conflicts. Granola achieves this by executing each independent transaction at the same timestamp at all transaction participants. We determine the timestamp by using a distributed voting mechanism. Each participant nominates a proposed timestamp for the transaction, the participants exchange these nominations in VOTE messages, and the transaction is assigned the highest timestamp from among these votes.

The protocol for transactions that modify data is as follows; the timeline is shown in Figure 6.

1. The repository selects a *proposed* timestamp for the transaction that is higher than highTS (sent by the client), the timestamp of the most recently executed transaction at the repository, and the current clock value.

2. The transaction request and timestamp proposal are recorded using a stable log write.

3. The repository sends a COMMIT VOTE message containing the proposed timestamp to the other participants. The repository can process other transactions after the vote has been sent.

4. The repository waits for votes from other participants. If it receives a CONFLICT vote, it ceases processing the transaction and sends this response to the client. A conflict vote will be received only from a participant that is operating in locking mode, as described in the next section.

5. Once the repository receives commit responses from all other participants, it assigns the transaction the *highest* timestamp from the votes; this timestamp will be consistent across all participants. The transaction is now ready to be executed.

6. The transaction is executed at the assigned timestamp, in timestamp order, and a reply is sent to the client.

The protocol for read-only transactions is similar, except that the stable log write is not required. Since these transactions do not modify the service state, the client proxy can retry a read-only transaction if a participant fails while executing the protocol.

As mentioned, the repository can process other transactions while waiting for votes. In all cases we, guarantee serializability by executing in *timestamp* order. A transaction won't be executed until after any concurrent transaction with a lower timestamp at the repository. It is thus possible for execution of a transaction to be delayed if a transaction with a lower timestamp has not yet received a full set of votes. Longer-term delays can occur if a transaction participant has failed; recovery from a failed participant is discussed in Section 5.4.

4.5 Coordinated Distributed Transactions

We now describe the protocol used for coordinated transactions, and the impact on single-repository and independent transactions when in locking mode. Coordinated transactions require participants to agree whether to commit or abort. They require locking to support concurrency; otherwise, one transaction might modify state that was used by a concurrent transaction to determine its vote.

The protocol for coordinated transactions is as follows; the timeline is shown in Figure 7.

1. Coordinated transactions first undergo a prepare phase. This is accomplished by issuing a `prepare`

upcall to the application. The application acquires any locks required by the transaction, and returns its vote, holding the locks only if it decides to commit.

2. The repository selects a timestamp for the transaction that is greater than highTS, the timestamp of the last executed transaction and the current clock value.

3. The transaction request, vote, and proposed timestamp are recorded using a stable log write.

4. The repository sends its VOTE along with the proposed timestamp. If the vote is ABORT or CONFLICT, it immediately returns this result to the client, and ceases processing the transaction.

5. The repository waits for votes from other participants. If it receives an ABORT or CONFLICT vote it makes an `abort` upcall to the application which releases locks and reverts any changes, sends the ABORT or CONFLICT REPLY to the client, then ceases processing the transaction.

6. Once the repository has received COMMIT votes from all other participants, it assigns the transaction the highest timestamp from the votes, and immediately executes the transaction by issuing a `commit` upcall to the application. The application releases any locks, and returns the transaction result. The repository can then send a COMMIT REPLY to the client.

Locking Mode. The protocol for single-repository and independent transactions is different in locking mode. Independent transactions are processed using the coordinated transaction protocol with `prepare` and `commit` upcalls. The client proxy will retry an independent transaction if it receives a CONFLICT response; it will never receive an ABORT response for an independent transaction.

We avoid holding locks for single-repository transactions by attempting to execute them as soon as they have been assigned a timestamp, but before the transaction is logged, by using a `run` upcall. The application checks locks for a `run` upcall if issued concurrently with other distributed transactions. If the `run` upcall is successful, the repository issues a stable log write to record the timestamp for the transaction before responding to the client. If there is a lock conflict, the application returns `false`, and the repository responds to the client with a CONFLICT response then discards the transaction. No log write is required for the aborted transaction.

The protocol for read-only single-repository transactions is the same, except that the stable log write is not required. The response to the client must be buffered until the most recent stable log write is complete, to avoid externalizing the effects of any previous transaction that has not yet been logged. We discuss the recovery implications of executing a transaction before logging in Section 5.2.

Switching Modes. There may be single-repository and independent transactions in progress when a coordinated transaction arrives while the repository is in timestamp mode. We handle this situation by issuing `prepare` upcalls for the independent transactions, and following the locking protocol. If all prepares succeed, we go on to process the coordinated transaction; otherwise, we block the coordinated transaction and remain in timestamp mode until all earlier prepares succeed.

It is desirable to switch back to timestamp mode as soon as possible, to avoid the cost of locking. Once all coordinated transactions have completed, the repository issues an `abort` upcall for any current independent transactions. This releases their locks, and allows the repository to transition into timestamp mode. The independent transactions will be executed using `run` upcalls in timestamp order, once their final timestamps are known.

4.6 Consistency

Granola provides serializability for transactions. In timestamp mode our consistency model is straightforward: Timestamps define a *total ordering* of transactions, guaranteeing serializability; all participants observe a transaction to execute at a single common timestamp. Clients propagate timestamps between repositories, to ensure that a transaction is not assigned a timestamp lower than any transaction that may have preceded it. Locking is not required when in timestamp mode, since each transaction executes serially with a single application upcall at each participant.

Granola allows each repository to execute its part of an independent transaction without knowing what is happening at the other participants: it only knows that they will ultimately select the same timestamp. This means that it is possible for a client to observe the effect of a distributed transaction T at one repository before another participant has executed it. Since any subsequent request from the client will carry a highTS value at least as high as T, however, we can guarantee that this request will be delayed if necessary and execute after T at any participant.

Locks are required when handling coordinated transactions, to ensure that a transaction does not observe or invalidate the state used to determine the commit or abort vote for a concurrent transaction. Repositories may execute transactions out of timestamp order when in locking mode, since locking is sufficient to guarantee serializability [16]. Timestamps thus may not represent the *commit* order of transactions, but still represent a valid *serial* order, since any transactions that execute out of timestamp order are guaranteed not to conflict.

Our transaction protocol does not provide *external* consistency [23], meaning that consistency is not guaranteed when communication occurs outside the system. Granola relies on clients including their highTS value on each request, which will not be included on out-of-band communication. External consistency can be provided for client-to-client communication if these messages include the highTS value of the sender. While violations of external consistency are possible for communication that occurs completely outside the system, such violations are unlikely since they are only possible within a small window of time, proportional to the clock skew between repositories [13].

5 Failures and Recovery

This section discusses the mechanisms used to handle individual node failures. We also discuss what happens when problems such as a network partition cause repositories to become unresponsive for an extended period of time.

5.1 Replication

Granola requires that stable log writes remain durable, and that repositories recover quickly from failure. We accomplish this using state machine replication [30]; while disk writes could have been used for durability, they do not provide fast recovery from failure. Our replication protocol uses an improved version of Viewstamped Replication [24, 28]; Paxos [22] provides an equivalent consensus protocol and could also have been used.

Repositories are replicated using a set of $2f+1$ replicas to survive f crash failures. One replica is designated the *primary*, and carries out the Granola protocol. *Backup* replicas carry out a *view change* to select a new primary if the old one appears to have failed.

Stable log writes involve the primary sending the log message to the backups and waiting for f replies, at which point the log is stable. The primary does not stall while waiting for replies; it continues processing incoming requests in the meantime. The replication protocol uses batching of groups of log messages [10] to reduce the number of messages sent to backups.

Each log message records the request, proposed timestamp and vote for each transaction. The primary piggybacks the final timestamp assigned to each transaction on subsequent log messages, to facilitate pruning the log at the backups. The backups execute transactions in timestamp order once the final timestamp is known.

Our protocol does not require disk writes as part of the stable log write, since we assume that replicas are failure-independent. Information is written to disk in the background, e.g., as required by main memory limitations. Failure-independence can be achieved by locating replicas in disjoint locations, or by equipping them with battery-backed RAM.

5.2 Repository Recovery

This section discusses issues that arise due to failovers.

- When a failover occurs, it's possible that the old primary does not know it has been superseded. This can be a problem for read-only single-repository transactions since they may observe stale data if run at the old primary. We avoid this problem by using the leases mechanism introduced in Harp [25], which guarantees there is only ever a single primary.

- The new primary might not know about recent read-only independent transactions for which the old primary sent votes. The new primary may thus receive such a request and select a different timestamp, and as a result the client proxy may receive replies with different timestamps. In this case the client proxy discards the replies and reissues the transaction with a higher TID.

- The new primary will know all votes sent by the previous primary for distributed read/write transactions, but may not know the final timestamp assigned to some recently completed transactions. The new primary recovers this information by resending its votes to solicit votes it is missing.

- In locking mode, the new primary may execute transactions in a different order than the old primary since the execution order in locking mode depends on the order in which votes are received. Both the old and new primaries will only execute transactions in an order consistent with the timestamp order, however, and hence will not deviate in the serial ordering.

- In locking mode, single-repository transactions are executed before being logged and externalized, and thus may not persist into the new view. In this case the previous execution of the transaction is forgotten. When the old primary recovers, it must undo the execution of any such transactions; this can be accomplished by reverting to a checkpoint and replaying transactions from the log.

5.3 Client Recovery

Clients can ensure uniqueness of sequence numbers (which comprise TIDs) by writing periodic sequence number ranges to disk, and only using sequence numbers within each range. Alternatively, a client can recover its timestamp after failure by synchronizing its clock and

ensuring that a period of time equal to the maximum expected clock skew has elapsed. After this it can adopt its clock value as its new latest-observed timestamp, which will be at least as high as the timestamp it knew before it failed. Note that here we are depending on synchronized clocks for correctness, where otherwise we have not needed this assumption.

If there are no explicit consistency constraints between client sessions, the client proxy can instead set its latest-observed timestamp to 0 when recovering from failure.

5.4 Long-term Failures

Since Granola provides strong consistency, we may have to stall because of failure. Replication masks the failure of individual nodes, but cannot provide availability if an entire replica group becomes unavailable. Permanent loss of a replicated repository may result in data loss and will likely require human intervention. This section describes how the rest of the system can make progress after failure of a replicated repository, in particular the situation where a non-failed repository is unable to obtain votes from a failed or unresponsive participant.

When a repository notices that a participant is unresponsive, it first attempts to resolve the transaction by resending its votes to any other participants; these participants will respond with the transaction status if they completed the transaction. If this is unsuccessful after a timeout, however, we proceed as follows.

Locking Mode. The repository needs to hold locks for *incomplete* distributed transactions that are awaiting votes from the failed participant until the participant recovers; subsequent transactions will be sent a CONFLICT REPLY if they conflict with these locks. The repository periodically resends each vote, and will commit or abort an incomplete transaction if notified by another participant that knows the transaction outcome.

Timestamp Mode. Recovery is more complicated if the repository is in timestamp mode, since it will not have acquired locks for the incomplete transactions, and must thus execute transactions in timestamp order. This results in a set of *blocked* distributed transactions, for which the repository has a full set of votes, but cannot yet execute because they are queued behind an *incomplete* transaction that currently has a lower timestamp based on votes so far. Single-repository transactions and read-only independent transactions are not included in these sets, since they can be sent a CONFLICT REPLY without causing repository state to diverge.

The repository first attempts to transition into locking mode by issuing a `prepare` upcall for each blocked transaction, in the current timestamp order, stopping if there is a lock conflict. If the `prepare` is successful for an incomplete transaction, the repository will continue holding the locks. If the `prepare` is successful for a blocked transaction, the repository executes it immediately by issuing a `commit` upcall, responds to the client, and removes it from the queue; this is safe because any request later in the queue will end up with a later timestamp and run after it, and this request doesn't conflict with any requests earlier in the queue.

If the repository is able to prepare all transactions without any lock conflicts, then it transitions into locking mode and follows the locking mode recovery protocol above. If it finds a conflict, however, it must cease preparing the remaining transactions. In the absence of conflicts, the repository can acquire locks for transactions in any order, since none of the transactions interfere, but if there is a lock conflict, the locks acquired for a transaction may depend on the order in which it is run. For example, consider a transaction that reads a stock level from one database table, then updates one of two different tuples depending on whether or not the stock is above a certain level; in this case the lockset depends on the relative ordering of transactions that modify the level.

We acquire locks in an *order-independent* way by requiring applications to implement a `forcePrepare` upcall. This function acquires locks the transaction might acquire in *any* execution order. In our stock level example, the application would acquire locks on both tuples, regardless of the stock level. For many applications, transactions are already order-independent, and this superset is the same as the set of locks acquired by a regular `prepare` upcall, particularly for transactions that update fields that are known ahead of time [7]. In other applications it might involve escalating lock granularity, e.g., from tuple-level locks to sets of locks or even to table locks.

The `forcePrepare` upcall acquires all locks even if there is a lock conflict, to allow the repository to accumulate locks for all blocked and incomplete transactions.

The repository iterates through the incomplete and blocked transactions (excluding blocked transactions that were completed while performing the `prepare` upcalls) and issues `forcePrepare` upcalls for them in the current timestamp order. The upcall returns true if there is no lock conflict; in the case of a blocked transaction this means it does not conflict with any transaction possibly ahead of it in the final serial order, and the repository can execute it, thus releasing its locks, and respond to the client.

Once locks have been acquired for all blocked and incomplete transactions, the repository can transition into locking mode and resume accepting new transactions.

6 Evaluation

We implemented Granola in Java, and deployed it on 10 2005-vintage 3.2 GHz Xeon servers with 2 GB RAM, with a cluster of 10 more powerful 2.5 GHz Core2 Quad servers with 4 GB RAM to provide client load. These machines were connected by a gigabit LAN with a network latency of ~ 0.2 ms. Multiple clients are colocated on a given machine to fully load the repositories, and wide-area network delay is emulated by delaying outgoing packets in our network library. Replication is emulated by running the protocol locally with appropriate network delay. We use this platform to examine Granola's scalability, latency, and resistance to lock conflicts and overhead.

We compare Granola's performance against the transaction coordination protocol from Sinfonia [7], which uses a more traditional lock-based version of two-phase commit. We use this coordination protocol by having clients issue single-repository transactions to the repositories (memory nodes) and issue distributed transactions to a single coordinator (application node). We used a single high-powered 16-core 1.6 GHz Xeon server with 8 GB RAM to serve as the coordinator, to avoid aborts due to contention from multiple masters.

Our implementation of Sinfonia differs in that we don't require clients to explicitly provide the read/write locksets for a transaction, and instead allow general operations through the use of `prepare` and `commit` upcalls. While we refer to this implementation as Sinfonia in our benchmarks, it could be used to represent any similarly-efficient implementation of two-phase commit. No locking is performed for single-repository transactions when running in isolation; the implementation must check whether a single-repository transaction conflicts with any active locks when running concurrently with distributed transactions.

We first examine the performance characteristics of Granola on a set of microbenchmarks, followed by an evaluation on a more complex transaction processing benchmark. Our figures show 95% confidence intervals for all datapoints.

6.1 Micro-benchmarks

Our micro-benchmarks examine a counter service. Each update modifies either a counter on a single repository, or counters distributed across multiple repositories. We vary the conflict rate for coordinated transactions by adjusting the ratio of conflicting updates issued by a client. Since the protocol for read-only operations is similar to many other distributed storage systems, these benchmarks focus exclusively on read/write transactions.

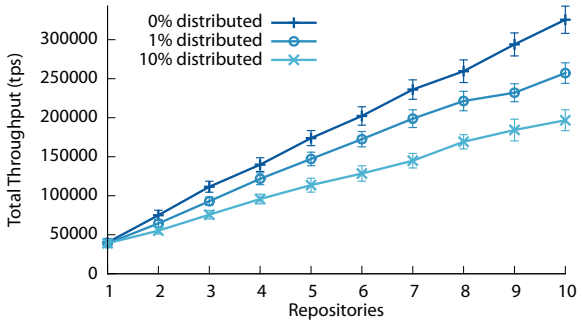

Figure 8: Throughput scalability.

6.1.1 Base Performance

We measured a maximum throughput of approximately 50,000 tps (transactions/sec) with a per-transaction latency of under 1 ms, increasing to 65,000 tps with under 10 ms latency, for single-repository transactions on a single compute core. Throughput was CPU-bound, and we observed more than twice this throughput on a more powerful machine. We also examined a wide-area configuration with approximately 10 ms emulated one-way delay between replicas, which resulted in similar throughput and a baseline per-transaction latency of approximately 22 ms. Wide-area replication did not impose a throughput penalty despite additional per-request latency, since our replication protocol can handle transactions in parallel.

6.1.2 Scalability

We illustrate Granola's scalability in Figure 8. Clients issue each request to a random repository, with between 0 and 10% of requests issued as distributed 2-repository independent transactions. This figure shows a local-area replication configuration. Configurations with 10 ms one-way delay between repositories and between replicas gave similar throughput but required significantly more clients to load the system, due to higher request latency.

There is a slight drop in per-repository throughput when moving from one repository to multiple repositories, due to the overhead of buffering transactions with higher `highTS` values, but this stabilizes at higher numbers of repositories. This occurs in our microbenchmark since clients and repositories are colocated on the same LAN, and hence the clock skew can be greater than the latency between requests; this effect is less likely when there is a network delay between clients and repositories.

Per-client throughput is lower for distributed transactions, owing to the additional communication delay; distributed-transaction latency was found to be consistently twice the latency of single-repository transactions.

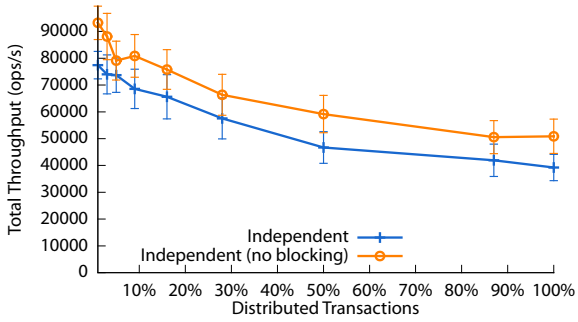

Figure 9: Throughput with distributed transactions.

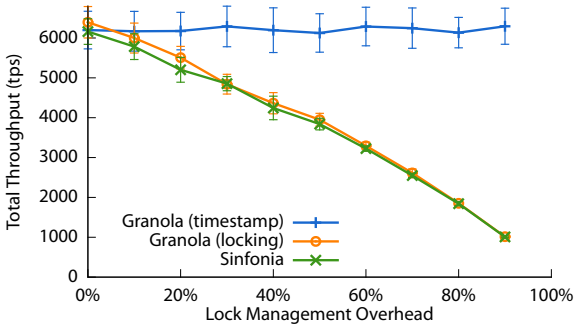

Figure 10: Throughput with locking/logging overhead.

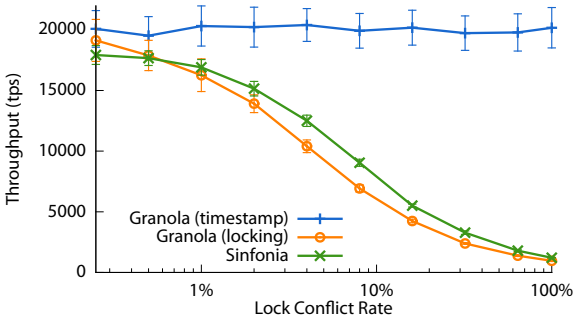

Figure 11: Throughput with varying lock conflict rate.

6.1.3 Distributed Transactions

Figure 9 shows the impact of workloads with between 0 and 100% distributed transactions. For each data-point, we examine workloads composed of independent transactions concurrently with single-repository transactions in timestamp mode; we examine locking mode in subsequent sections. We compare the overhead of the timestamp protocol against a version of Granola that never blocks a transaction to wait for earlier timestamps to complete, and thus does not provide consistency.

This figure shows throughput in terms of *operations* per second, since each distributed transaction involves running an operation on each repository. An optimal coordination scheme would exhibit constant throughput, independent of the fraction of distributed transactions. Granola scales well with an increase in distributed transactions, but we observe throughput less than this optimal value, due to communication and processing overhead. We observe a 10–20% reduction in throughput from waiting for earlier transactions to complete, due to timestamp constraints. We examine distributed transaction throughput on a more realistic workload in Section 6.2.

6.1.4 Locking

Locking introduces overhead in two key ways: the execution cost of acquiring locks and recording undo logs; and the wasted work from having to retry transactions that abort due to lock conflicts. We examine both these components separately in the following two sections, and examine locking on a realistic workload in Section 6.2.

Lock Management Overhead. Figure 10 shows the throughput of Granola and our version of Sinfonia on a two-repository topology as a function of lock management cost. We examine a workload composed entirely of distributed transactions, and run Granola in both timestamp mode and locking mode. This experiment measures lock overhead with no lock contention; we set transaction execution cost to be equivalent to the cost of a `new_order` transaction in TPC-C, and vary the lock overhead as a fraction of this execution cost.

As expected, Granola in timestamp mode is unaffected by locking overhead, while both Sinfonia and Granola in locking mode drop in throughput as lock management overhead increases. Typical values for lock management cost in OLTP workloads are in the vicinity of 30–40% of total CPU load [18, 19]. At this level Granola gives 25–50% higher throughput than lock-based protocols, even in the absence of lock contention.

Lock Contention. We investigate the impact of lock *contention* in Figure 11, on a two-repository topology with 100% distributed transactions. We control the lock conflict rate by having transactions modify either a private or shared counter. This experiment measures lock contention in isolation, and we tailor our application to have negligible lock management overhead.

Throughput for Sinfonia and Granola in locking mode deteriorates fairly rapidly as lock conflict rate increases, due to the cost of retrying aborted transactions; note the logarithmic x-axis. Throughput for these transactions drops to approximately 1000 tps at 100% contention, which correlates with the average 1 ms latency we observed for each non-aborted transaction.

Unlike our other experiments, Sinfonia's maximum throughput at low contention is limited here by our use

of a single coordinator, which serves as a bottleneck. Sinfonia achieves slightly higher throughput than Granola in locking mode when there is *high* contention, since the coordinator presents a consistent transaction ordering to repositories, unlike in Granola where transactions may be received by repositories in conflicting orders. We evaluated an extension to Granola to support the use of lightweight coordinator nodes, but observed minimal benefit in typical workloads [13].

6.2 Transaction Processing Benchmark

We evaluate performance on an application based on the TPC-C transaction processing benchmark [6]. This benchmark models a large order-processing workload, with complex queries distributed across multiple repositories. Our implementation stores the TPC-C dataset in-memory and executes transactions as single-round stored procedures.

We used the C++ implementation of TPC-C from the H-Store project [32] for our client and server application code.[2] The codebase that we used was designed for a single node deployment and had no explicit support for distributed transactions. By interposing the Granola platform between the TPC-C client and server, we were able to build a scalable distributed database with minimal code changes; code modifications were constrained to calling the Java `ClientProxy` from the C++ client, responding to transaction requests from the GranolaApplication server, and translating warehouse numbers to repository IDs.

We adopt the data partitioning strategy proposed in H-Store. This partitioning ensures that all transactions can be expressed as either single-repository or independent transactions. We were able to disable locking and undo logging when evaluating Granola, since TPC-C involves no coordinated transactions. We also compare Granola and Sinfonia against a version of Granola that is set to always operate in locking mode, to measure the impact of lock-based concurrency control.

Scalability. We examine scalability in Figure 12. This experiment uses a single TPC-C warehouse per repository, and increases the number of clients to maximize throughput. 10.7% of transactions in this benchmark are issued to multiple participants.

All systems exhibit the same throughput in a single-repository configuration, since they both have similar overhead in the absence of locking or distributed transactions. Throughput drops for the lock-based protocols on multiple nodes, however. The TPC-C implementation is highly optimized and executes transactions efficiently, hence the lock overhead imposes a significant relative penalty; the overhead of locking and allocating undo records was approximately equal to the cost of executing each operation, in line with similar measurements on the same workload [20]. Throughput reduction is also heavily impacted by the cost of retrying transactions that abort due to lock conflicts. Sinfonia offers slightly lower performance than Granola in locking mode, due to the additional overhead and latency of communicating with the coordinator.

[2]This implementation does not *strictly* adhere to the TPC-C spec., e.g., does not implement client "wait times" between requests [19].

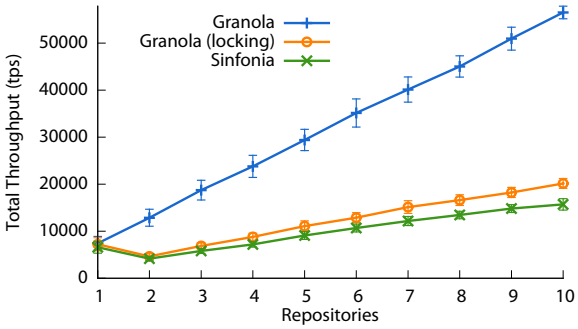

Figure 12: Scalability of TPC-C implementation.

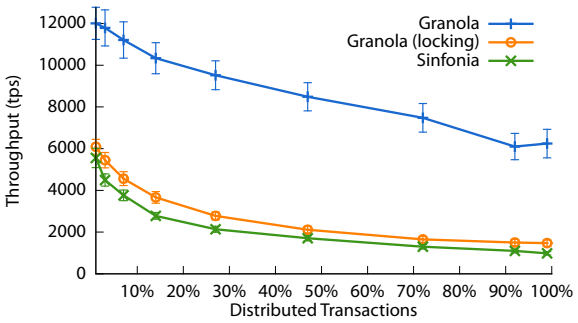

Figure 13: TPC-C throughput `new_order` transactions.

We observe a relatively constant latency regardless of system size, with average distributed transaction latencies of 2.9, 3.2, and 4.9 ms for Granola, Granola (locking) and Sinfonia respectively. Sinfonia encounters higher latency due to the additional communication delay.

Distributed Transactions. We further examine coordination overhead by modifying TPC-C to vary the proportion of distributed transactions. This workload is composed entirely of `new_order` transactions and we adjust the likelihood that an item in the order will come from a remote warehouse [20]. The results for this benchmark are shown in Figure 13, for a two warehouse configuration on two repositories.

These results echo the previous benchmark, with the performance difference dominated by lock conflicts and lock management overhead. Granola achieves better resilience to distributed transactions in this benchmark than in our microbenchmarks, since overhead in TPC-C is dominated by transaction execution costs rather than proto-

col effects. Throughput for Granola in timestamp mode decreases by approximately 50% when moving from 0 to 100% distributed transactions. This represents a very low performance penalty for distributed transactions, since each given distributed transaction involves execution cost on two repositories instead of one.

7 Related Work

There has been a long history of research in transactional distributed storage, including a rich literature on distributed databases [9, 15, 27]. These systems provide support for interactive transactions, whereas Granola targets a simpler single-round transaction model that can nonetheless be used to support a wide number of applications [7, 32].

Relaxed Consistency. Many systems [17, 21, 29, 34] relax the consistency guarantees provided by traditional databases, in order to provide increased scalability and resilience to network or hardware partitions. The growth of cloud computing has led to a resurgence in popularity of large-scale storage systems with weaker consistency, typified by Amazon's eventual-consistency Dynamo [14] and a wealth of others [2–5, 12]. These systems target high availability and aim to be "always writable", but sacrifice consistency and typically offer constrained transaction interfaces, such as Dynamo's read/write distributed hash table interface.

Per-Row Consistency. Systems such as SimpleDB [1] and Bigtable [11] provide consistency within a single row or data partition, but do not provide ACID guarantees between these entities. A significant downside to relaxed-consistency storage systems is the complicated application semantics presented to clients and developers when operating with multiple data items. More recent protocols such as COPS [26] and Walter [31] attempt to simplify application development by providing stronger consistency models: *causal+* and *parallel snapshot isolation* respectively. These models do not prevent consistency anomalies however, and require the developer to reason carefully about the correctness of their application.

Strong Consistency. Megastore [8] represents a departure from the traditional wisdom that it's infeasible for large-scale storage systems to provide strong consistency. Megastore is designed to scale very widely, uses state-machine replication for storage nodes, and offers transactional ACID guarantees. As in SimpleDB [1], Megastore ordinarily provides ACID guarantees within a single entity group, but also supports the use of standard two-phase commit to provide strong consistency between groups. CRAQ [33] primarily targets consistency for single-object updates, but mentions that a two-phase commit protocol could be used to provide multi-object updates. Granola is more heavily optimized for transactions that span multiple partitions, and provides a more general operation model.

Granola is most similar in design to Sinfonia [7]. Sinfonia also supports reliable distributed storage with strong consistency over large numbers of nodes. Sinfonia's *mini-transactions* express transactions in terms of read, write and predicate sets, whereas Granola supports arbitrary operations and does not require a priori knowledge of lock-sets. Granola also provides fewer message delays in the transaction coordination protocol. The most significant difference is Granola's support for independent distributed transactions; this is of considerable benefit in suitable workloads, avoiding conflicts and lock overhead.

The H-Store project argues for the relevance of transactions that fit the independent model, and observes that these transactions can be handled without locking [32]. The protocol sketched in the position paper was not a full distributed implementation however, and does not work with failures, delays and clock skew; later complete implementations used different techniques and did not optimize for independent transactions [20, 36]. Granola introduces a novel transaction coordination protocol based on timestamp exchange to provide the first complete protocol that supports independent transactions in a distributed setting.

The Calvin transaction coordination protocol [35] was developed in parallel with Granola, and provides similar functionality. Rather than using a distributed timestamp voting scheme to determine execution order, Calvin delays read/write transactions and runs a global agreement protocol to produce a deterministic locking order.

8 Conclusion

Granola is a distributed transaction infrastructure that provides serializability for one-round transactions. This simplifies the reasoning required by application developers and users of the system. Granola also supports a general operation model, which allows development of arbitrary storage applications.

Granola implements new protocols that provide lower overhead for transaction coordination than in previous work. Distributed transactions complete with one stable log write and only three message delays.

Most significantly, this paper introduces explicit support for independent distributed transactions. Independent transactions appear in many common workloads and allow the development of high-performance distributed applications. Granola uses a timestamp-based implementation of independent transactions that provides a substantial performance benefit, due to an absence of lock conflicts and management overhead.

Acknowledgments

We thank Dan Ports, the anonymous reviewers, and our shepherd, Jon Howell, for their helpful feedback. This research was supported under NSF grant CNS-0834239.

References

[1] Amazon SimpleDB. `http://aws.amazon.com/simpledb/`.

[2] Apache Cassandra. `http://cassandra.apache.org`.

[3] Apache CouchDB. `http://couchdb.apache.org`.

[4] Apache HBase. `http://hbase.apache.org`.

[5] MongoDB. `http://www.mongodb.com`.

[6] TPC benchmark C. Technical report, Transaction Processing Performance Council, February 2010. Revision 5.11.

[7] M. K. Aguilera, A. Merchant, M. A. Shah, A. C. Veitch, and C. T. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM TOCS*, 2009.

[8] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J. M. Lon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[9] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *IEEE TKDE*, March 1990.

[10] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM TOCS*, 2002.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI*, 2006.

[12] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *VLDB*, 2008.

[13] J. Cowling. *Low-Overhead Distributed Transaction Coordination*. PhD thesis, MIT, June 2012.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[15] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. l Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE TKDR*, March 1990.

[16] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *CACM*, Nov. 1976.

[17] R. Guy, J. Heidemann, W. Mak, T. Page Jr., G. Popek, and D. Rothneier. Implementation of the Ficus replicated file system. In *USENIX*, 1990.

[18] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, 2008.

[19] E. P. C. Jones. *Fault-Tolerant Distributed Transactions for Partitioned OLTP Databases*. PhD thesis, MIT, 2012.

[20] E. P. C. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, June 2010.

[21] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM TOCS*, Nov. 1992.

[22] L. Lamport. The Part-Time Parliament. Technical Report Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, Sept. 1989.

[23] K.-J. Lin. Consistency issues in real-time database systems. In *System Sciences*, 1989.

[24] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical report, MIT CSAIL, Cambridge, MA, 2012.

[25] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *SOSP*, 1991.

[26] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[27] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM TODS*, December 1986.

[28] B. Oki and B. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *PODC*, 1988.

[29] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, David, and C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE TC*, 1990.

[30] F. B. Schneider. The state machine approach: A Tutorial. Technical Report TR 86-600, Cornell University, Dept. of Computer Science, Ithaca, N. Y., Dec. 1986.

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

[32] M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, 2007.

[33] J. Terrace and M. J. Freedman. Object storage on CRAQ: high-throughput chain replication for read-mostly workloads. In *USENIX ATC*, 2009.

[34] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, and M. J. Spreitzer. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP*, 1995.

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

[36] VoltDB Inc. VoltDB. `http://voltdb.com`.