Strongly consistent replication for a bargain

Konstantinos Krikellas [#‡], Sameh Elnikety [†], Zografoula Vagena [*‡], Orion Hodson [†]

[#]*School of Informatics, University of Edinburgh*

[†]*Microsoft Research*

[*]*Concentra Consulting Ltd*

***Abstract*— Strong consistency is an important correctness property for replicated databases. It ensures that each transaction accesses the latest committed database state as provided in centralized databases. Achieving strong consistency in replicated databases is a major performance challenge and is typically not provided, exposing inconsistent data to client applications. We propose two scalable techniques that exploit lazy update propagation and workload information to guarantee strong consistency by delaying transaction start. We implement a prototype replicated database system and incorporate the proposed techniques for providing strong consistency. Extensive experiments using both a micro-benchmark and the TPC-W benchmark demonstrate that our proposals are viable and achieve considerable scalability while maintaining strong consistency.**

I. Introduction

Database systems are an integral part of enterprise IT infrastructures. In these environments, replication is a commonly used technique in databases to increase availability and performance. In many cases, the database clients are other computer systems. This creates a large and complex distributed system with intricate dependencies and hidden communication channels.

Traditionally, when a database system is replicated to scale its performance, *i.e.*, all replicas execute client transactions, there is an explicit trade-off between performance and consistency. A replicated database may be configured to expose inconsistent data to clients by providing weaker consistency guarantees. Some inconsistencies stem from the fact that there is a (usually considerable) delay between when an update transaction is executed at a replica and when its effects have been propagated to all the other replicas. Weakening consistency allows the replicated database system to achieve higher performance; providing strong consistency penalizes the scalability of the replicated system.

Many real–world applications would benefit from replicated databases being transparent, providing the same semantics as a centralized database. In particular, centralized databases provide strong consistency, which guarantees that each transaction sees the latest committed state of the database. Therefore, replicated database systems should ideally provide the same strong consistency guarantees [7].

We highlight the importance of strong consistency using the following example. Consider two automated clients, $Agent_A$ and $Agent_B$, running under separate administrative domains which implement a simple hidden communication channel. $Agent_A$ executes a transaction T_1 to update a particular database, based on a request received from $Agent_B$, e.g., to trade shares for $Agent_B$. When the transaction commits, $Agent_A$ notifies $Agent_B$ that the operation has completed. Both agents presume that the updates of T_1 are reflected in any direct interaction – or indirect interaction via a third party – with that database. Here communication between $Agent_A$ and $Agent_B$ occurs via a hidden channel, which is increasingly common in applications that span multiple administrative domains. In a centralized database the latest committed database state is immediately available, making strong consistency natively guaranteed. However, replacing a centralized database with a replicated database system exposes these subtle inconsistencies (e.g., $Agent_B$ may not observe the effects of T_1 for a considerable interval), unless strong consistency is provided.

This challenge has been recognized by the database community for years, with early replicated databases using techniques such as eager update propagation to provide strong consistency [20]. According to this eager approach, the commit of an update transaction is executed on all replicas synchronously before responding to the client. In this case, any new transaction *from any client* observes the committed changes. However, committing transactions on all replicas synchronously is expensive, requiring tight replica synchronization. Many replicated systems use weaker consistency guarantees, such as session consistency [12] or relaxed currency [6], [21], to improve performance. Still, correct programs using replicated databases without strong consistency are explicitly written to handle the increased complexity of data inconsistency; handling consistency at the application level is tricky and error prone.

Replicated database systems should ideally provide strong consistency to clients while preserving their scalability potential. To that end, we propose two approaches that guarantee strong consistency in replicated databases while lazily propagating the effects of update transactions. The first approach monitors the progress of propagated updates at each replica and ensures that replicas are updated before starting a new transaction. The second approach extends the first approach and exploits workload specific information, which is easily available in automated environments, to execute transactions as soon as the needed subset of data is current. Both approaches mask the latency required to reflect the effects of an update transaction on all replicas, but might introduce a delay at the start of a transaction to apply the needed updates. This delay

[‡]Work done while author was at Microsoft Research.

allows new transactions to receive the latest committed state of the data they are going to access.

We build a prototype replicated database system and implement our proposals to experimentally evaluate them. The results show that the proposed techniques guarantee strong consistency with scalable performance, outperforming the eager approach for providing strong consistency and matching the performance of session consistency. The latter is a weaker form of consistency guaranteeing that a single client sees its own committed updates.

Strong consistency is concerned with the effects of committed transactions and not with overlapping transactions. It is different from transaction isolation levels [5], [18], such as serializability and snapshot isolation. This work focuses on providing strong consistency: this issue has received much less attention than serializability from the replicated database research community. We delineate the differences between strong consistency and serializability and give examples in the next section.

The main contributions of this paper are the following:

- we propose two scalable techniques that exploit lazy update propagation and workload characteristics to guarantee strong consistency,
- we present a prototype replicated database system and show how to implement the two proposed techniques, and
- we experimentally evaluate the proposed techniques and compare them with eager strong consistency and session consistency, proving the viability and the efficiency of our proposals.

The paper proceeds as follows: in Section II we provide the database model and define strong consistency. We describe the proposed lazy approaches to provide strong consistency in Section III. In Section IV we present the architecture and design of our replicated database prototype and explain how the proposed techniques for strong consistency are implemented. The experimental evaluation of our techniques is presented in Section V. Finally, we discuss related work in Section VI and draw our conclusions in Section VII.

II. Definitions And Assumptions

We present a database and transaction model to formally define strong consistency. We also define session consistency because we later compare the performance of strong consistency to session consistency. We show the difference between strong consistency and serializability, and provide example histories.

A database D is a collection of uniquely identified data items. A transaction T_i starts with the BEGIN statement followed by a sequence of read and write operations on data items, and terminates with either the COMMIT or ABORT statement. We denote T_i's read operation on data item X by $R_i(X)$ and write operation by $W_i(X)$, while begin and commit are denoted as B_i and C_i. The sequence of operations from all committed transactions constitutes a history H.

We adopt the definition of strong consistency in replicated systems from prior work [7], [12], [16], [17]:

Definition 1: **Strong consistency:** For every committed transaction T_i, there is a single-copy history H' such that (a) H' is view equivalent to a history H over a single copy of the database, and (b) for any transaction T_j: if (T_i commits before T_j starts in H) then T_i precedes T_j in H'.

Strong consistency guarantees that after a transaction T_i commits, any new transaction T_j following T_i observes the updates of T_i. This property is immediately provided in a standalone database system.

Next, we describe session consistency. Each client submits a sequence of transactions to the database that defines the client's session (S). H is an interleaved sequence of the transaction operations contained in all sessions. Session consistency is then defined as follows:

Definition 2: **Session consistency:** For every committed transaction T_i, there is a single-copy history H' such that (a) H' is view equivalent to a history H over a single copy of the database, and (b) for any transaction T_j: if (T_i commits before T_j starts in H, and $session(T_i) = session(T_j)$) then T_i precedes T_j in H'.

The definition above shows that session consistency is a relaxed version of strong consistency: it ensures that transactions access a consistent state of the database within the bounds of the session they belong to. Here, a client is guaranteed that the updates of its last transaction are visible to its new one. Session consistency is weaker than strong consistency, in the sense that there is no guarantee for a client's transaction to access the updates of committed transactions originating from other clients.

Although the definitions of strong and session consistency are associated with serializability in prior work [7], [12], it is important to distinguish between strong consistency and the isolation level. Both are correctness properties and they are different: both, one, or none can be provided. Strong consistency captures the externally visible order of transaction commits that clients observe. In contrast, serializability and other transaction isolation levels are concerned with how the database interleaves the operations of concurrent transactions [8], [9], [30], [38].

To further clarify the differences between strong consistency and the isolation levels, we use the following histories involving two transactions, T_1 and T_2, executing on two distinct replicas, Rep_1 and Rep_2, respectively.

- $H_1 = \{B_1, W_1(X=1), C_1, B_2, R_2(X=0), C_2\}$
- $H_2 = \{B_1, W_1(X=1), C_1, B_2, R_2(X=1), C_2\}$
- $H_3 = \{B_1, R_1(X=0), R_1(Y=0), B_2, R_2(X=0), R_2(Y=0), W_1(X=1), W_2(Y=1), C_1, C_2\}$.

In history H_1, T_2 starts before the update of T_1 on X is propagated to replica Rep_2. T_2 accesses the old value of X. This history is not strongly consistent but it is serializable and the equivalent serial history is $\{T_2, T_1\}$, though clients submit T_1 first and then T_2 to the replicated system. If strong consistency is enforced, execution follows history H_2, which corresponds to the serial history $\{T_1, T_2\}$. In this case, Rep_2

must be updated with the effects of T_1 before T_2 starts, allowing T_2 to read the latest value of X.

Finally, history H_3 is strongly consistent and snapshot isolated, but is not serializable [5]. Both T_1 and T_2 read the latest values of X and Y, but there is no equivalent serial history.

III. Providing Strong Consistency

This section describes three approaches for providing strong consistency. We start with the traditional eager approach and then propose two novel approaches employing lazy update propagation. We abstract the details of the replicated database system to show the generality of our proposals, and provide concrete implementations in Section IV.

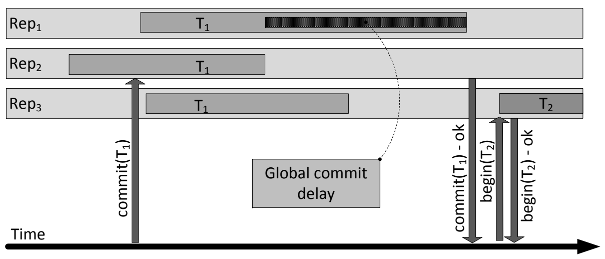
A. Eager approach

Traditionally, strong consistency is provided by committing each update transaction at all replicas before notifying the originating client. Strong consistency is guaranteed here because once a client learns that its transaction has committed, each new transaction observes the client's committed changes. According to this approach, the execution time for an update transaction includes a delay, termed *global commit delay*, representing the latency to commit the transaction at the additional replicas (i.e. all replicas except for the originating replica) and to collect the corresponding acknowledgments. This delay is dictated by the slowest replica to apply and commit the transaction during each commit round. This approach has therefore poor scalability, unless the workload is dominated by read-only transactions.

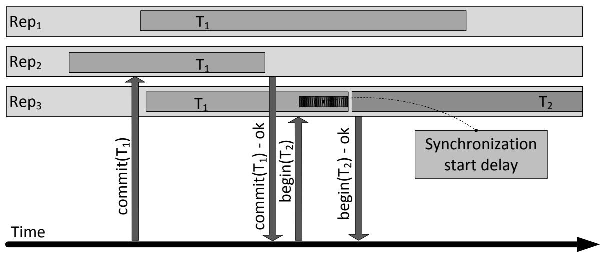
B. Lazy coarse-grained approach

The previous discussion suggests that waiting for all replicas to commit is not efficient. An alternative is to return as soon as the update transaction has committed at the hosting replica, while its effects are lazily propagated to the other replicas. In this case, when a new transaction is dispatched to a replica, there may be pending updates that have not been locally applied yet. To enforce strong consistency, we propose that the replica delays starting the transaction until it updates its state. We term this delay *synchronization start delay*. Effectively, we shift the waiting from the *global commit delay* to waiting for the *single* receiving replica to apply the needed updates. This approach is termed *coarse-grained* because the transaction start is delayed until all previous updates are applied.

We explain how this approach works assuming that the replicated system has a mechanism to monitor the progress of update transactions and to tag each new transaction with the required database state that should be accessed. When an update transaction commits at its host replica, the state of the local database is updated. The replicated system keeps track of the updates that the transaction performs and propagates them to other replicas in the background. The replica hosting the transaction sends the commit success acknowledgment to the client with no delay, avoiding the wait encountered in the eager approach. Afterwards, when a replica receives



(a) Eager approach



(b) Lazy approach

Fig. 1. Comparison of approaches providing strong consistency

a new transaction request, it first applies the updates from transactions that have committed (if any) and then executes the new transaction. Here, the system ensures that the new transaction receives the latest committed state of the database, so any update committed by any client on any replica will be "visible" to the new transaction.

***Theorem** 1:* The lazy coarse-grained approach guarantees strong consistency.

To contrast the differences between the eager and the lazy approaches, we use the example in Figure 1. The system has three replicas, Rep_1, Rep_2, and Rep_3. Transaction T_1 commits on Rep_2 and then transaction T_2 starts on Rep_3. The eager approach (Figure 1(a)) requires transaction T_1 to commit at all replicas before responding to the client; transaction delay is dictated by the slowest replica (Rep_1). Then T_2 starts immediately, as Rep_3 already contains the updates of T_1. Conversely, the lazy approach allows Rep_2 to notify the client as soon as transaction T_1 is locally committed, though the other replicas have not been updated yet. When Rep_3 receives the request for a new transaction, it may need to delay the transaction until the updates of T_1 have been applied. Note that Rep_1 has not been updated when T_2 is started. The lazy approach shrinks synchronization delays.

Compared to the eager approach, the lazy coarse-grained approach reduces the degree of replica synchronization, leading to a more efficient system. Under this approach, transaction execution includes a synchronization start delay at the receiving replica, rather than a global commit delay that is set by the slowest replica. We show that this is a favourable trade-off in Section V.

Compared to session consistency, the coarse-grained approach provides a stronger consistency guarantee at the cost of higher transaction delays. Session consistency delays transaction start until the previous updates of the same client have been applied to the replica, while the coarse-grained approach delays transaction start until updates from *all* clients have been applied.

C. Lazy fine-grained approach

The lazy coarse-grained approach for strong consistency requires the replicas to apply all committed updates before starting a new transaction. We shall now examine how this restriction can be relaxed, *i.e.*, whether we can apply only a subset of the committed updates to synchronize a replica at transaction start while maintaining strong consistency. A transaction typically accesses a small portion of the total data-set managed in the database; so the system can achieve the same result by updating only the needed data-set before the transaction starts. This approach reduces the number of updates required at transaction start and allows the transaction to start early.

An issue that needs to be addressed, however, is how the system determines the exact transaction data-set *a priori*. In practice, this only holds in the trivial case where each transaction accesses exactly the same records in each table on every execution. But in automated environments, such as in e-commerce applications, a predefined set of transactions is used. Each transaction consists of a sequence of *prepared* statements, *i.e.*, SQL statements that access a specific set of tables but different records depending on the statement parameters. In such an environment, we take advantage of the fact that a database is logically partitioned to tables: we statically extract the *table-set* that the transaction accesses. Since the table-set is a superset of the transaction's data-set, strong consistency is preserved when a replica installs the pending updates for the tables included in each transaction's table-set before executing the transaction.

Enforcing consistency on the transaction's table-set suggests an acceptable trade-off: most likely the table-set is a small subset of the database but it likely contains more records than the ones that will be accessed.

***Theorem** 2:* The lazy fine-grained approach guarantees strong consistency.

In terms of performance, the fine-grained approach can outperform or match the coarse-grained approach, depending on the number of tables accessed and the frequency of update transactions. For instance, if a set of tables is updated by consecutive transactions, the differences between the two lazy approaches will be negligible. Conversely, if a transaction accesses read-only tables, the fine-grained approach allows earlier start, without waiting for any pending updates. In contrast, the coarse-grained approach requires installing all committed updates, even though their absence does not violate strong consistency for the specific transaction.

Even though session consistency provides a weaker consistency guarantee than the lazy fine-grained approach, the latter may yield better performance. In most cases the updates made by the client's last transaction will be applied faster than the updates for the new transaction's table-set, as client interaction includes delays due to application processing, network data transfer and client-side processing time. Still, if we return to the example of the transaction that accesses read-only tables, the fine-grained approach allows replicas to start this transaction immediately. In contrast, session consistency requires the replica to first apply the updates of the client's previous transaction(s), though these are not to be accessed by the new transaction.

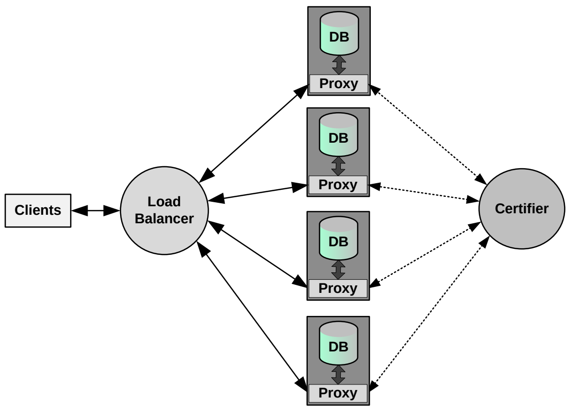


Fig. 2. Multi-master database replication middleware

IV. Prototype Architecture and Implementation

To evaluate the viability and the efficiency of the proposed approaches for strong consistency against session and eager strong consistency, we implement a replicated database prototype and use it as a test platform. Our system is aligned with the state-of-the-art in modern middleware-based replication [13], [14], [27], [31], [35] and follows the *multi-master* architecture. The main challenge of a replicated database system is to efficiently keep the different copies consistent in the presence of update transactions, so this configuration allows every replica to independently process client read-only and update transactions. The replication middleware is responsible for replica control to resolve system-wide conflicts and for propagating the effects of update transactions.

Our prototype provides transaction isolation by adopting generalized snapshot isolation (GSI) [16]. This concurrency control algorithm extends snapshot isolation (SI) [10], [18] in a manner suitable for replicated databases. It allows transactions to use local snapshots of the database available on the replica, while still providing the main correctness and performance properties of snapshot isolation. Both GSI and SI are multi-version concurrency control algorithms, imposing a total order on the commits of update transactions. Note that GSI is weaker than serializability but conditions exist to check if a workload runs serializably – and to make it serializable if needed – under GSI. As an example, the workloads of the TPC-C and TPC-W transaction benchmarks run serializably under SI and GSI [16], [19].

Since we need to keep track of the transaction updates for consistency purposes, we count database versions: the database starts at version 0, and the version is incremented when an update transaction successfully commits. Each replica in the system proceeds through this version sequence though possibly at different speed.

The architecture of our prototype is depicted in Figure 2. The system consists of the certifier, the load balancer and a number of database replicas, each hosting a standalone DBMS and a proxy. We describe each module next.

Load balancer. The load balancer is the intermediary between clients and replicas; it hides the distributed nature of the cluster. The load balancer receives transactions from client applications, dispatches them to replicas and relays replica responses back to clients. It routes transactions to replicas according to its load balancing policy, which is selecting the replica with the least number of active transactions. The load balancer does not use workload information for routing, *e.g.*, to maximize transaction conflict independence. Its design is minimalistic for scalability.

Certifier. The certifier [14] performs the following tasks: (*a*) it decides whether an update transaction commits, (*b*) it maintains the total order of committed update transactions, (*c*) it ensures the durability of its decisions, and (*d*) it forwards the updates of every committed transaction to the other replicas, in the form of a refresh transaction containing the transaction writeset (*i.e.*, the set of records the transaction has inserted, updated, or deleted). A transaction T can commit if its writeset does not write-conflict with the writesets of transactions that committed since T started (see also [14], [27]).

The certifier handles the database version counter V_{commit}. Each time it certifies a transaction to commit, it increases this counter and tags the transaction writeset with the current database version. Then, it notifies the originating replica of the version at which the transaction should commit and forwards the refresh writeset to the other replicas. That way, all replicas commit transactions according to the global order determined by the certifier.

Replicas. Each replica consists of a proxy and a standalone DBMS configured to provide snapshot isolation. The proxy intercepts all requests to the DBMS, including both client transactions and refresh transactions. It maintains a version number V_{local} reflecting the version of its local database. V_{local} is incremented whenever the proxy updates the state of the database by committing a client update transaction or a refresh transaction.

For client transactions the proxy forwards the SQL statements to the database and relays responses to clients via the load balancer. Upon the reception of a transaction commit request from a client, the proxy first examines the transaction writeset. If the writeset is empty, the transaction is read-only, so the proxy commits it locally and immediately notifies the client. If it is an update transaction, (*a*) it needs to be checked whether it conflicts with any transaction (possibly originating from another replica) that has committed after the start of the transaction and (*b*) its writeset needs to be forwarded to the other replicas, if it is certified to commit. These two tasks are handled by the certifier: the proxy forwards the writeset to the certifier and awaits the latter's decision on committing or aborting the transaction. Upon receiving the decision from the certifier, the proxy commits or aborts the transaction to the local database instance and sends the outcome to the client. The proxy also has a refresh transaction handler that receives refresh writesets from the certifier. Refresh writesets are queued and sequentially applied to the local DBMS. Their commits are interleaved with committing local update transactions, so as to maintain the global transaction order as determined by the certifier.

Replicated databases employing transaction certification need to address the *hidden deadlock* problem [27]. The certifier imposes a global ordering to transaction commits; however, transactions acquire locks in a different order inside each replica. This may lead to a deadlock involving remote and local transactions (*i.e.*, spanning the proxy and the standalone DBMS) inside replicas that cannot be handled by the transaction controller of the standalone DBMS. To prevent the hidden deadlock problem, the proxy performs early certification [14]. If a client statement updates the database content, the proxy extracts the partial writeset of the update statement and checks whether it conflicts with the pending (*i.e.*, received but not applied yet) refresh writesets. In the case of a conflict, the client's update transaction is aborted. In addition, when the proxy receives a new writeset, it checks it against the partial writesets of the currently active local client transactions and immediately aborts any conflicting transaction(s).

Fault-tolerance. We briefly outline the fault-tolerance aspects of this design. We assume the crash-recovery failure model [2], [29] in which a host may fail independently by crashing, and subsequently recover. The certifier is lightweight and deterministic, and therefore can be easily replicated for availability (rather than for performance) [14] using the state machine approach [39]. The load balancer is lightweight because it maintains only a small amount of soft state, including the number of connections currently opened per replica and session and version accounting (as we shall shortly introduce). The database replicas, in contrast, maintain hard (i.e. persistent) state that is orders of magnitude larger than the load balancer's state. Therefore, a standby load balancer can be used for availability.

We next present how strong consistency is implemented in our prototype using the lazy coarse- and fine-grained approaches. For reference, we also describe the implementations for the eager approach for strong consistency and for session consistency.

A. Lazy coarse-grained strong consistency (CoarseSC)

CoarseSC implements the lazy coarse-grained approach of Section III-B for strong consistency. The load balancer is the intermediary between replicas and clients, so it is extended to enforce strong consistency. It maintains the database version V_{system} reflecting updates of the database state that clients may observe. Each time a replica commits a transaction, the proxy tags its response to the load balancer with the current value of V_{local}. The load balancer maintains the database version of the latest transaction committed and acknowledged to the clients. When a client sends a request for a new transaction, the load balancer tags it with the the current value of V_{system}. In turn, when the replica receives the request for a

TABLE I
DATABASE AND TABLE VERSIONS

Transaction	**Updated tables**	**Database version** V_{system}	**Table A version** V_A	**Table B version** V_B	**Table C version** V_C
		0	0	0	0
T_1	A	1	1	0	0
T_2	B,C	2	1	2	2
T_3	B	3	1	3	2
T_4	C	4	1	3	4
T_5	B,C	5	1	5	5
T_6	A	6	6	5	5

new transaction, it checks the version tag and compares it with its current version V_{local}. If $V_{local} < V_{system}$ there are pending writesets so the transaction start is delayed until applying them. This technique ensures that, when the transaction begins, the writesets of all transactions committed so far have already been applied to the local database, so strong consistency is preserved.

B. Lazy fine-grained strong consistency (FineSC)

FineSC implements the lazy fine-grained approach of Section III-C for strong consistency. This approach reduces the synchronization start delay by requiring only the updates affecting the transaction's table-set to be applied before transaction start. For this purpose, the load balancer and the proxy maintain, for each table t in the database, a version number V_t that reflects the latest version of table t. The proxy updates V_t when locally committing a transaction (local or refresh): it sets V_t to V_{commit} for all tables contained in the transaction's writeset. After committing a client transaction the proxy tags its response to the load balancer with the updated table versions. Note that the transaction table-set contains the tables that the transaction accesses, but access to them may be read-only, so not all versions V_t for the transaction table-set need to be updated.

The load balancer needs the transaction table-set at transaction start. We implement this by storing the transaction table-set information in the database; the load balancer queries the database once to retrieve this information and stores it in a dictionary mapping transaction identifiers to corresponding table-sets[1]. Clients tag their requests for a new transaction with the transaction identifier. The load balancer uses it to retrieve the transaction table-set and then, for each table t accessed in the transaction, checks the corresponding version V_t. The *highest* such V_t defines the *minimum* version V_{start} that contains the latest updates for all the tables accessed by the transaction, as required by the lazy fine-grained approach. This version is passed to the replica and used by its proxy to delay transaction start, if necessary, so as to ensure strong consistency.

To illustrate how versions are maintained, consider a database with three tables, (A, B, C), over which transactions are executed. Table I shows how versions V_{system} and V_{table} are managed after a number of update transactions, T_1, T_2, T_3, T_4, T_5 and T_6, are certified to commit in the system with the given order. This order is conveyed to the load balancer through replica responses. The database and table versions start from 0. When the first update transaction T_1 commits, V_{system} is incremented to 1, and V_A is then updated to $V_A = V_{system} = 1$, while the versions of V_B and V_C remain 0 since B and C are not updated by T_1. The second update transaction T_2 updates tables B and C and then commits at $V_{system} = 2$. Consequently, V_B and V_C are updated to $V_B = V_C = 2$. Similarly, T_3 commits at $V_{system} = 3$ and changes $V_B = 3$, T_4 commits at $V_{system} = 4$ and changes $V_C = 4$ and T_5 commits at $V_{system} = 5$ and changes $V_B = V_C = 5$.

Next consider a new transaction T_6, which reads from and writes to table A only. T_6 starts after T_5 commits and is dispatched to replica i. CoarseSC requires replica i to reach $V_{local} = 5$, the version of the latest committed transaction. However, it is sufficient to have $V_{local} = 1$ to execute T_6 under strong consistency because T_6 does not access other tables. In effect, any version $V_{local} \geq 1$ is current enough for T_6. This flexibility reduces the synchronization start delay since the delay is until $V_{local} = 1$ rather than $V_{local} = 5$.

C. Session consistency (SessionC)

SessionC provides session consistency to clients. Our system exploits database version to provide session consistency, as proposed in prior work [12]. We assume that each request for a new transaction is tagged with the client's session identifier (SID). The load balancer keeps account of sessions by maintaining a dictionary that maps SIDs to database versions. The proxy tags transaction commit confirmations with the current value of V_{local}. The load balancer uses this version to update the corresponding entry in the session dictionary: map(SID)= V_{local}. Next time the same client sends a request for a new transaction, the load balancer intercepts the request and tags it with the latest database version $V_{session}$ =map(SID) for this session. This version is forwarded to the replica and used for synchronization at transaction start. This technique ensures that, when a transaction begins, the updates of the client's last transaction have already been applied to the local database, so the client sees monotonically increasing versions of the database.

[1]Another alternative is to tag requests for new transactions with the table-set to be accessed [3], [28]

D. *Eager strong consistency (EagerSC)*

EagerSC implements the eager approach of Section III-A for strong consistency. We exploit the certifier to monitor transaction commits, instead of passing control to replicas, because it is less loaded than the replicas. As described before, the certifier decides which transactions commit and forwards the refresh writesets to all replicas. In this configuration, it also maintains a counter per committed transaction, denoting the number of replicas that have committed the transaction. Each time a transaction is certified to commit, the certifier creates a counter for the transaction and sets it to 0. After the replica commits a transaction (local or refresh), the proxy notifies the certifier and the counter for this transaction is incremented. When the counter matches the number of replicas in the system, the certifier notifies the originating replica that the transaction is globally committed. Then the originating replica notifies the client about the outcome of the transaction.

V. Experimental Results

Our main objective is to show that the proposed CoarseSC and FineSC approaches provide strong consistency efficiently, i.e. they have better performance than the EagerSC approach. The performance of the SessionC approach represents an upper bound since it provides weaker consistency.

A. *Experimental Setup*

Testbed. We use a cluster of machines to deploy the replicated database system. Each machine runs Windows Server 2008 and has one Intel Core 2 Duo 2.2 GHz CPU, 4 GB DDR2 SDRAM of main memory and a Seagate Barracuda 250 GB 7200rpm disk drive. All machines are interconnected using a Gigabit Ethernet switch. We monitor the system load using the Windows Sysinternals utilities. We implement the proxy, certifier, and load balancer as multi-threaded C# programs. We use a separate machine to host the certifier and vary the number of database replicas between one and eight replicas. Each replica runs an instance of the Microsoft SQL Server 2008 Enterprise database server configured to execute transactions at snapshot isolation level. Transaction durability is enforced by the certifier; we turn off log-forcing in replicas, in line with the proposals of [14].

Workload. We experiment with two workloads: (*a*) a customized micro-benchmark and (*b*) the TPC-W benchmark [41]. The micro-benchmark allows us to explore the behavior of various system configurations using simplified and controllable schemata and transactions. We quantify the performance of the replicated system and assess the cost of providing session or strong consistency on various transaction mixes. The TPC-W benchmark is a more realistic workload simulating an online book store. It therefore provides evidence of the performance advantages of our proposed techniques for strong consistency against the eager approach in demanding e-commerce environments. Both workloads run serializably under generalized snapshot isolation which is provided by the replicated system [16].

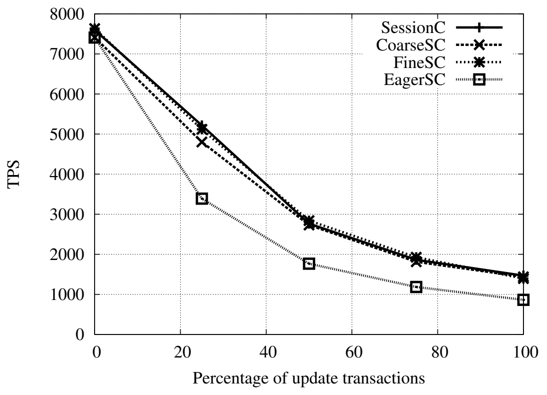


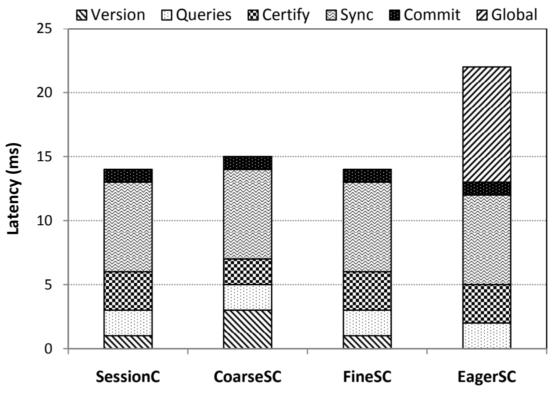
Fig. 3. Throughput results for the micro-benchmark (8 replicas)

Metrics. We adopt the following metrics: system throughput, which is the number of completed transactions per second (TPS), and response time, denoting the elapsed time between starting a transaction and receiving the acknowledgement of successful transaction commit (in ms). We also break transaction delay down into stages (in ms). Read-only transactions have three stages: (*a*) a *synchronization start delay* until the latest version is reached (denoted as "version", not present in EagerSC), (*b*) a stage for executing the transaction SQL statements (denoted as "queries") and (*c*) a stage for committing the transaction to the local DBMS (denoted as "commit"). Update transactions have six stages: (*a*) "version", (*b*) "queries", (*c*) a stage for querying the certifier (denoted as "certify"), (*d*) a stage for committing previous local or refresh transactions according to the global commit order (denoted as "sync"), (*e*) "commit", and (*f*) a final *global commit delay* for update transactions that is only present in EagerSC (denoted as "global"). The "queries" and "sync" stages indicate the performance of the local DBMS; the "certify" delay shows the certification cost; the "versions" and "global" delays indicate the synchronization cost for each configuration.

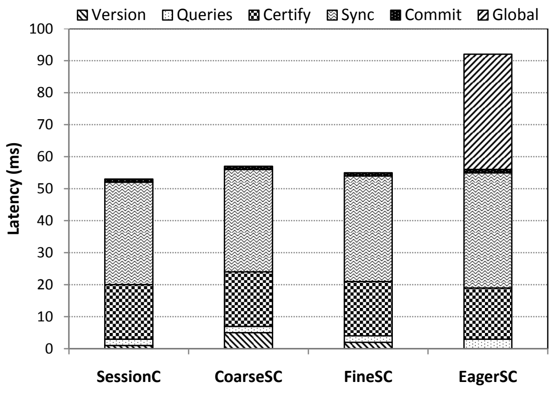
Each experiment consists of 10 separate runs. Each run included a warm-up interval of 10 minutes followed by a measurement interval of 10 minutes. We report average measured values, with the deviating being less than 5% in all cases.

B. *Micro-benchmark*

The mix of read-only and update transactions is a crucial factor for the performance of the replicated system. Read-only transactions run in one replica locally while update transactions need to be certified and propagated to all replicas. Update transactions stress the consistency maintenance subsystem, highlighting the synchronization delays. In order to examine these effects, we use a database consisting of 40 tables with $10,000$ records each. The common table schema comprises the primary key (integer), an integer field and a text field of 100 characters. The workload consists of 40 transactions. Each transaction either retrieves or updates a random record from one table. The ratio of read-only/update transactions varies between $0/40$ and $40/0$. We use 8 replicas and 80 clients and each client issues randomly selected transactions. Transactions are sent back-to-back in a closed loop.



(a) 25% update mix



(b) 100% update mix

Fig. 4. Latency breakdown for the micro-benchmark

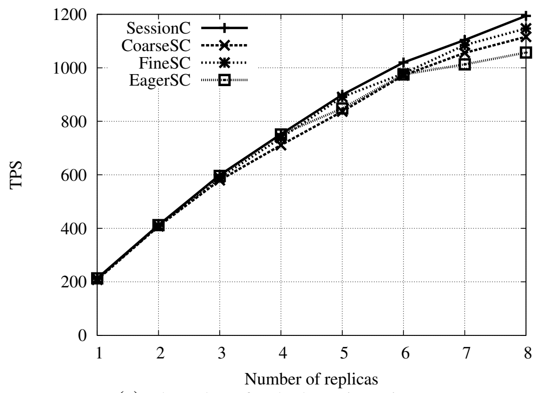
We present the throughput results for the micro-benchmark in Figure 3. The X-axis shows the ratio of read-only to update transaction from 0/40 till 40/0. The Y-axis shows the throughput in TPS. The figure has four curves, one per each consistency configuration. At the 0/40 point, the workload is read-only and all approaches have the same performance. As the ratio of update transactions increases along the X-axis, throughput decreases because processing update transactions is more expensive than processing read-only transactions. Both proposed configurations for strong consistency, CoarseSC and FineSC, achieve comparable performance with SessionC and substantially outperform EagerSC, the difference being 40% when the ratio of updates is higher than 25%. FineSC's throughput is higher than CoarseSC's, the difference being close to 5%, and it matches the throughput of the SessionC in all cases. The results above show that our proposed techniques combine strong consistency with the performance of session consistency.

To evaluate the cost of each transaction stage and, hence, understand the underlying performance of different aspects of the replicated DBMS, we look at the transaction latency breakdown for the 25% and 100% update mixes in Figures 4(a) and 4(b) respectively. The X-axis shows the configuration measured and the Y-axis shows the delay for each transaction stage in ms. For read-intensive mixes, as shown in Figure 4(a), the average query execution time excluding synchronization delays is similar for all configurations; the differences lie in the *synchronization start* and *global commit* delays. SessionC and CoarseSC have a common start-up delay for both read-only and update transactions, with 1 ms delay for SessionC and 3 ms for CoarseSC. FineSC has zero delay for transactions accessing read-only tables, but its start-up delay is 4 ms for transactions on updated tables, as the replica needs to wait for refresh transactions to complete before reaching the consistent version. The total delay is slightly shorter than CoarseSC's and the same as SessionC's. In EagerSC all transactions start immediately, but the *global commit* delay for update transactions is long (32 ms), since it is determined by the slowest replica on each transaction. The latency for EagerSC is therefore 36% more than the latency for the other configurations, increasing the total transaction delay and thus justifying the difference in throughput.

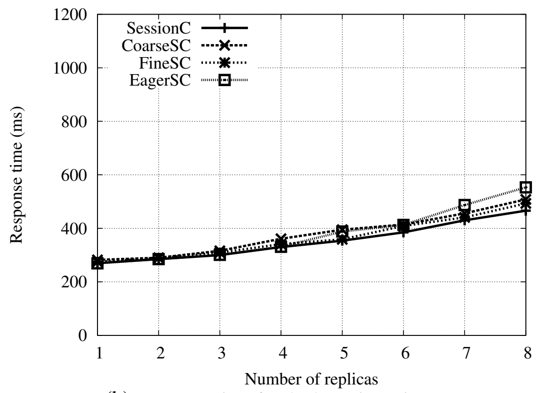
Moving to the update only mix (Figure 4(b)), all transactions are propagated to all replicas, so the cost for certifying update transactions and applying refresh writesets is substantially increased for all configurations when compared to the 25% update mix; the sum of the "Certify", "Sync" and "Commit" stages increases from 12 ms to 51 ms. Notice that Figure 4(b) has a different scale for the Y-axis than Figure 4(a). The start-up delay for FineSC is 2 ms on average, while for CoarseSC it is 5 ms. This is expected as each transaction in this benchmark accesses one table, so FineSC requires only the updates of the specific table to be applied before the transaction starts. On the other hand, CoarseSC requires all tables to be updated before starting the new transaction and therefore requires a higher delay. EagerSC is the slowest configuration; the *global commit* delay reaches 35 ms, an order of magnitude higher than the synchronization latency of the other configurations. It is obvious that the eager approach penalizes performance; the techniques we propose match the performance of SessionC while providing strong consistency.

C. TPC-W Benchmark

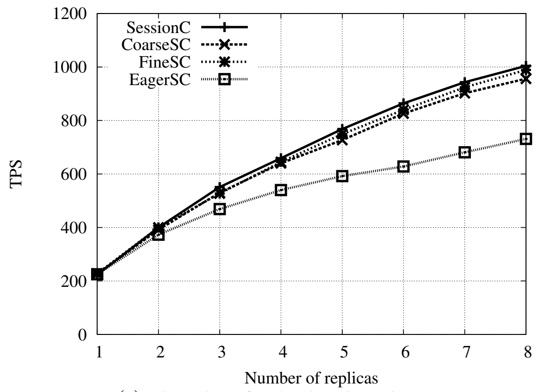
We now switch to the TPC-W benchmark from the Transaction Processing Council [41]. TPC-W is designed to evaluate e-commerce systems and it implements an on-line bookstore. It consists of three workload mixes that differ in the relative frequency of the transaction types. The browsing mix has 5% update transactions, the shopping mix has 20% update transactions, and the ordering mix has 50% update transactions. We present results from all mixes. The shopping mix is the most representative mix, while the ordering mix is update-intensive and therefore is the most challenging mix for replicated databases. TPC-W is widely used to evaluate replicated database systems [3], [14], [15], [27], [35]. To drive the replicated database system, we use one machine running the application server and another hosting a remote terminal emulator (RTE). The application server (IIS 7.0) executes the requested `ASP.NET` pages which access the database. The RTE is a multi-threaded `C#` program in which each thread represents



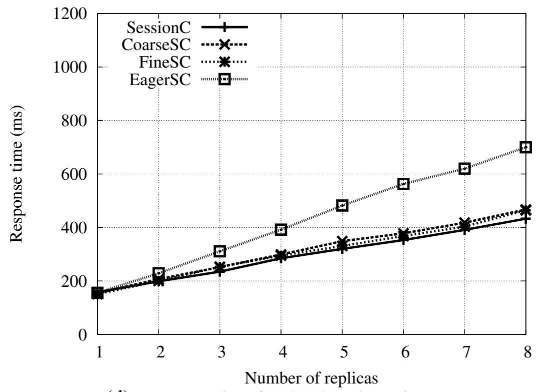
(a) Throughput for the browsing mix



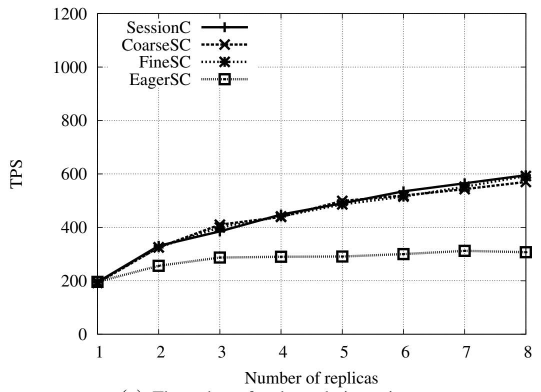
(b) Response time for the browsing mix



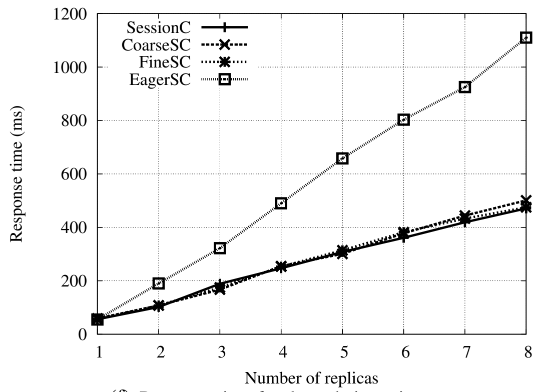
(c) Throughput for the shopping mix



(d) Response time for the shopping mix



(e) Throughput for the ordering mix



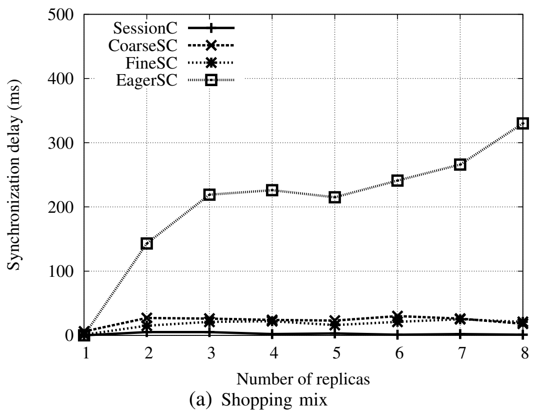
(f) Response time for the ordering mix

Fig. 5. TPC-W throughput and response time results (scaled load)

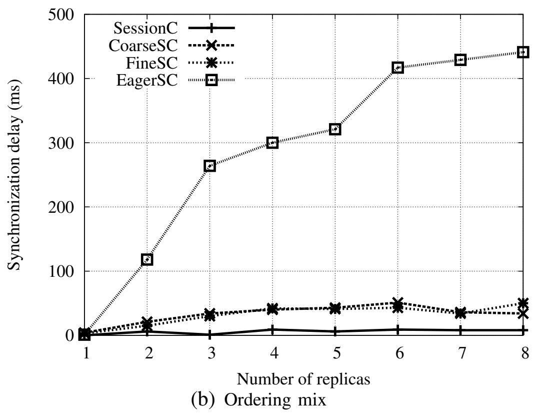
one client issuing requests in a closed loop. The client machine is lightly loaded with CPU utilization below 40%.

The TPC-W database standard scaling parameters are 200 EBS (emulated browsers) and $10,000$ items and the database size is 850 MB. Client think time between consecutive requests follows a negative exponential distribution with an average of 200 ms. We conduct two sets of experiments. In the first set we replicate the database to get higher throughput by scaling the load (represented by the number of clients) with the number of replicas as follows: (a) 100 clients/replica for the browsing mix, (b) 80 clients/replica for the shopping mix and (c) 50 clients/replica for the ordering mix. In the second set we use replication to reduce response time by employing a fixed load, therefore the number of clients in the system is constant (*i.e.*, 100, 80 and 50 for the three transaction mixes) regardless of the number of replicas.

1) Replication for higher throughput: We first examine performance when the load scales with the number of replicas. We report separately throughput and response time results for the browsing, shopping and ordering mix in Figure 5. We also report the average *synchronization delay* for the shopping and ordering mix in Figure 6. Synchronization delay is defined as the *synchronization start* delay for SessionC, CoarseSC and FineSC and the *global commit* delay for EagerSC. In all figures, the X-axis gives the number of replicas that varies between one and eight, while the curves represent the perfor-

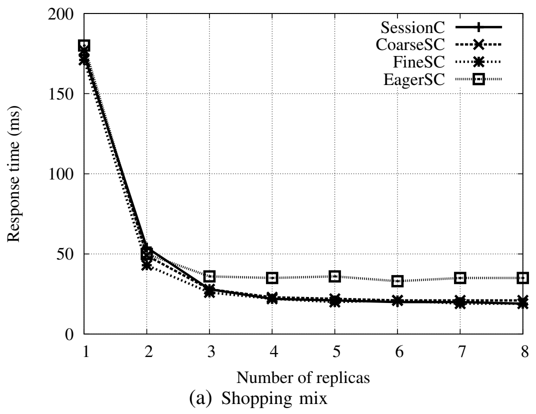


(a) Shopping mix

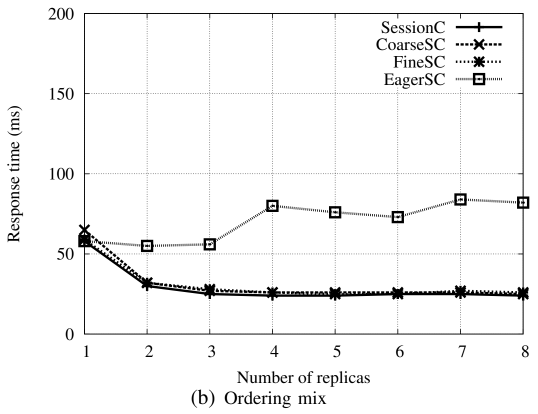


(b) Ordering mix

Fig. 6. TPC-W synchronization latency results (scaled load)



(a) Shopping mix



(b) Ordering mix

Fig. 7. TPC-W response time results (fixed load)

mance of each configuration. The Y-axis shows throughput in TPS or response time in ms or synchronization delay in ms.

The TPC-W browsing mix (5% update transactions) is dominated by read-only transactions. The results of Figures 5(a) and 5(b) show that the differences between the various configurations are negligible, with EagerSC achieving marginally lower throughput for 7 and 8 replicas. The low frequency of update transactions allows the replicas to synchronize quickly in all configurations, so performance increases almost linearly and reaches a 7x increase in throughput for eight replicas.

Next we turn to the TPC-W shopping mix (20% update transactions), as depicted in Figures 5(c) and 5(d). All configurations display considerable scalability, the lazy ones increasing throughput at eight replicas by five times, compared to the throughput of one replica. However, EagerSC's throughput is lower than the throughput of the other configurations, as the *global commit* delay ($143 - 330\ ms$ as shown in Figure 6(a)) restricts the system's performance. At the same time, CoarseSC and FineSC are close to SessionC, as the delay for the latest version ranges between 18 and 30 ms for CoarseSC and between 15 and 25 ms for FineSC. When eight replicas are used, CoarseSC and FineSC nearly match the throughput of SessionC, while EagerSC is almost 30% slower, so our techniques prove their efficiency against EagerSC on a workload with a 20% update transaction mix. Response time increases linearly for all configurations, but in EagerSC it deteriorates faster, being 50% higher than the latency for the other configurations.

As the transaction mix becomes more update-dominated, EagerSC's performance is penalized by the delay for applying the refresh writesets to all replicas before completing each update transaction. This is verified by the results of the TPC-W ordering mix (50% update transactions), as shown in Figures 5(e) and 5(f). SessionC, CoarseSC and FineSC display similar throughput and achieve almost linear scalability as updates are lazily propagated. Using eight replicas increases throughput by three times when compared to a single replica because it is an update-intensive workload. The difference between SessionC and our proposed techniques CoarseSC and FineSC is marginal, while CoarseSC and FineSC provide strong consistency. The synchronization delay for SessionC (10 ms, see also Figure 6(b)) is smaller than for CoarseSC and FineSC (50 ms), but this is a small percentage of the total response time. In contrast EagerSC can barely scale its performance, as the *global commit* delay dominates execution time and reaches 450 ms for 8 replicas. This shows that the eager approach for guaranteeing strong consistency restricts performance when the workload is update-intensive.

2) Replication for lower response time.: We now turn to the system's behaviour when the load is fixed, as depicted in Figure 7. Due to space constraints we show the response time graphs for the shopping and ordering mixes and skip the browsing mix. The response time results for the shopping mix of Figure 7(a) show that, for all configurations, response time gradually decreases and stabilizes when using five or more replicas. However, EagerSC requires 35 ms on average for each transaction while the other configurations respond in 20 ms. The difference is due to the higher *global commit* delay for the eager approach, when compared to the negligible *synchronization start* delay of the other configurations. This effect becomes more apparent in the ordering mix, as shown in Figure 7(b). In this case, response time for CoarseSC, FineSC and SessionC gradually decreases and reaches 26 ms. On the contrary, adding more replicas in EagerSC *increases* response time up to 84 ms. Since EagerSC requires all replicas to commit each transaction before reporting back to the client, response time for update transactions is determined by the slowest replica, so more replicas mean higher delays.

D. Discussion

The results presented above for the micro-benchmark and the TPC-W benchmark verify the efficiency of the proposed techniques for strong consistency. The eager approach for strong consistency can practically be used only when the workload contains almost exclusively read-only transactions. In contrast, the CoarseSC and FineSC configurations reach the performance of the SessionC configuration, achieving considerable scalability in a wide range of workloads while providing strong consistency to clients. In many cases, FineSC matches the performance of SessionC, so synchronization on the table-set combines the best performance with strong consistency. The lazy fine-grained approach is therefore the best system configuration, provided that the application can provide the necessary workload information (*i.e.*, transaction table-set) to the replicated system. Still, even if this condition is not met, the coarse-grained approach guarantees strong consistency with a small penalty in performance.

VI. Related work

In this section we compare our work to related work on recent middleware-based replicated databases as well as to consistency criteria within distributed systems.

A recent paper [11] provides an overview of middleware-based database replication and qualitatively compares the state-of-the-art in academia and the commercial world. The paper explains that existing systems either employ eager approaches (e.g. [24]), or lazy approaches that provide weaker forms of consistency. Examples of the second category include the following: Tashkent [14], which provides generalized snapshot isolation, replicated database systems that provide 1-copy SI [27], [42] and systems that provide session consistency (e.g. [13], [35], [37]). None of these systems provides strong consistency.

Session consistency in a replicated system [12] guarantees for each client to receive snapshots with monotonically increasing versions. Every client sees its own updates. Moreover, from the client's point of view, its successive transactions receive snapshots that never go back in time. However, there is no provision for correctness when clients participate in a workflow or have dependencies among each other. Strong consistency addresses these issues. The performance of the two proposed techniques matches the performance of session consistency while providing a stronger correctness property.

Postgres-R [24] provides one-copy serializability but with no guarantees for strong consistency. Several proposals [1], [23], [25], [32], [33] advocate using group communication systems, such as Totem or Horus [36], for replicated databases. Most of this work is simulation based and none provides strong consistency. In this paper, we build a system and implement strong consistency using an eager approach. We go further by proposing two lazy update propagation mechanisms for strong consistency and evaluate them experimentally.

The relaxed-currency model [21] has been extended to systems containing replicated databases and middle-tier caching in the form of relaxed-currency serializability [6]. The transaction model is extended so that clients pass a "freshness" constraint to the replicated system that bounds how up-to-date a replica must be to provide a consistent database state to a new transaction. This approach requires the clients to tag their transaction requests with the appropriate constraints. In contrast, our proposals allow the replicated system to handle consistency internally. The coarse-grained approach requires no extension of the transaction model, while the fine-grained approach uses (potentially static) workload information and requires clients to tag their transaction requests with the transaction identifier.

Commitment ordering [37] is a condition for achieving global serializability among transactions spanning autonomous systems that may have different concurrency control models. It relies on atomic commitment whose implementation is at least as expensive as the eager approach presented earlier in which a commit is applied at all replicas. A similar synchronization overhead applies when read-only and update transactions are atomically broadcasted to all replicas [43] to enforce strong consistency.

Immediate operations [26] can be used to enforce synchronization at transaction start. However, this technique incurs a high communication cost, in order to specify the latest version of each item to be read across all replicas. Our proposed techniques examine transaction data-sets on a coarser (database and table) granularity to reduce synchronization overhead and enhance scalability.

Several consistency models in distributed systems have been proposed, for example Linearizability [22] and Sequential Consistency [4], targeting different environments such as shared registers, shared distributed memory, and multi-processors. These models specify properties of individual operations and their correct scheduling, rather than properties of transactions, and therefore these models are not directly applicable to replicated databases.

Bayou [34], [40] is a distributed system that uses causal ordering constraints on read and write operations to manage data freshness to client sessions. It provides eventual consistency, rather than strong consistency, in the sense that servers converge to the same state in the absence of updates. Our proposals provide strong consistency regardless of the workload characteristics.

VII. Conclusions

This paper advocates using strong consistency in replicated database systems as provided in centralized databases. Strong consistency has been traditionally provided using eager approaches that do not scale well. We propose two approaches that employ lazy update propagation while providing strong consistency. The coarse-grained approach is generally applicable and enforces strong consistency by possibly delaying transaction start. The fine-grained approach further exploits workload information to reduce transaction start delay. We build a replicated database prototype to evaluate the presented approaches and compare them to the eager approach as well as to a method that provides weaker consistency, called session consistency. Experimental results using a customized micro-benchmark and the TPC-W benchmark show that the proposed approaches for strong consistency scale better than the eager approach while matching the performance of session consistency.

Acknowledgments

We thank Antony Rowstron (Microsoft Research), Alan Fekete (University of Sydney), and Fernando Pedone (University of Lugano) for their insightful suggestions and valuable feedback. We also thank John Douceur, Tim Harris, Alan Geller, and Sreenivas Addagatla (Microsoft Research) for their constructive comments.

References

[1] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases (extended abstract). In *Euro-Par*, 1997.

[2] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distrib. Comput.*, 13(2):99–125, 2000.

[3] C. Amza, A. Cox, and W. Zwaenepoel. Distributed Versioning: Consistent Replication for Scaling Back-end Databases of Dynamic Content Sites. In *Middleware*, 2003.

[4] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, 1994.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[6] P. A. Bernstein, A. Fekete, H. Guo, R. Ramakrishnan, and P. Tamma. Relaxed-currency serializability for middle-tier caching and replication. In *SIGMOD*, 2006.

[7] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *The VLDB Journal*, 1(2):181–240, 1992.

[8] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update Propagation Protocols For Replicated Databases. In *SIGMOD*, 1999.

[9] Y. Breitbart and H. F. Korth. Replication and Consistency: Being Lazy Helps Sometimes. In *PODS*, 1997.

[10] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *SIGMOD*, 2008.

[11] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: the gaps between theory and practice. In *SIGMOD*, 2008.

[12] K. Daudjee and K. Salem. Lazy Database Replication with Ordering Guarantees. In *ICDE*, 2004.

[13] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *VLDB*, 2006.

[14] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *EuroSys*, 2006.

[15] S. Elnikety, S. Dropsho, and W. Zwaenepoel. Tashkent+: memory-aware load balancing and update filtering in replicated databases. In *EuroSys*, 2007.

[16] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database Replication Using Generalized Snapshot Isolation. In *SRDS*, 2005.

[17] A. Fekete. Formal models of communication services: A case study. *IEEE Computer*, 26(8):37–47, 1993.

[18] A. Fekete. Allocating isolation levels to transactions. In *PODS*, 2005.

[19] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[20] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, 1996.

[21] H. Guo, P.-A. Larson, R. Ramakrishnan, and J. Goldstein. Relaxed currency and consistency: how to say "good enough" in SQL. In *SIGMOD*, 2004.

[22] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[23] J. Holliday, D. Agrawal, and A. E. Abbadi. The Performance of Database Replication with Group Multicast. *Fault-Tolerant Computing, International Symposium on*, 0:158, 1999.

[24] B. Kemme and G. Alonso. Don't Be lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *VLDB*, 2000.

[25] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. Database Syst.*, 25(3):333–379, 2000.

[26] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, 1992.

[27] Y. Lin, B. Kemme, M. Pati no-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD*, 2005.

[28] K. Manassiev and C. Amza. Scaling and Continuous Availability in Database Server Clusters through Multiversion Replication. In *DSN*, 2007.

[29] R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the Crash-Recover Model. Technical report, EPFL, 1997.

[30] E. Pacitti, P. Minet, and E. Simon. Replica Consistency in Lazy Master Replicated Databases. *Distrib. Parallel Databases*, 9(3):237–267, 2001.

[31] M. Pati no-Martinez, R. Jiménez-Peris, B. Kemme, and G. Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4):375–423, 2005.

[32] F. Pedone, R. Guerraoui, and A. Schiper. Transaction Reordering in Replicated Databases. In *SRDS*, 1997.

[33] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distrib. Parallel Databases*, 14(1):71–98, 2003.

[34] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, 1997.

[35] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Middleware*, 2004.

[36] D. Powell. Group communication. *Commun. ACM*, 39(4):50–53, 1996.

[37] Y. Raz. The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Mangers Using Atomic Commitment. In *VLDB*, 1992.

[38] O. T. Satyanarayanan and D. Agrawal. Efficient Execution of Read-Only Transactions in Replicated Multiversion Databases. *IEEE Trans. on Knowl. and Data Eng.*, 5(5):859–871, 1993.

[39] F. B. Schneider. Replication management using the state-machine approach. *Distributed systems (2nd Ed.)*, pages 169–197, 1993.

[40] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

[41] Transaction Processing Performance Council. The TPC-W benchmark, 2009. `http://www.tpc.org/tpcw/`.

[42] S. Wu and B. Kemme. Postgres-R(SI): Combining Replica Control with Concurrency Control Based on Snapshot Isolation. In *ICDE*, 2005.

[43] V. Zuikeviciute and F. Pedone. Correctness criteria for database replication: Theoretical and practical aspects. In *OTM Conferences (1)*, pages 639–656, 2008.