Middleware based Data Replication providing Snapshot Isolation

Yi Lin[*]
McGill Univ.
Montreal
ylin30@cs.mcgill.ca

Bettina Kemme[*]
McGill Univ.
Montreal
kemme@cs.mcgill.ca

Marta Patiño-Martínez[†]
Univ. Politecnica de Madrid
mpatino@fi.upm.es

Ricardo Jiménez-Peris[†]
Univ. Politecnica de Madrid
rjimenez@fi.upm.es

ABSTRACT

Many cluster based replication solutions have been proposed providing scalability and fault-tolerance. Many of these solutions perform replica control in a middleware on top of the database replicas. In such a setting concurrency control is a challenge and is often performed on a table basis. Additionally, some systems put severe requirements on transaction programs (e.g., to declare all objects to be accessed in advance). This paper addresses these issues and presents a middleware-based replication scheme which provides the popular snapshot isolation level at the same tuple-level granularity as database systems like PostgreSQL and Oracle, without any need to declare transaction properties in advance. Both read-only and update transactions can be executed at any replica while providing data consistency at all times. Our approach provides what we call "1-copy-snapshot-isolation" as long as the underlying database replicas provide snapshot isolation. We have implemented our approach as a replicated middleware on top of PostgreSQL replicas. By providing a standard JDBC interface, the middleware is completely transparent to the client program. Fault-tolerance is provided by automatically reconnecting clients in case of crashes. Our middleware shows good performance in terms of response times and scalability.

1. INTRODUCTION

Cluster based data replication is used in many data intensive applications for fault-tolerance and scalability. New replicas are added to the cluster to handle increasing load, and if replicas fail, available replicas are able to take over.

[*]This work has been partially funded by MDER-Programme PSIIRI: Projet PSIIRI-016/ADAPT, by NSERC Rgpin 23910601, and by FQRNT 2003-NC-80398.

[†]This work has been partially funded by the European Commission (Adapt project) IST-2001-37126, by Microsoft Research Cambridge MS-2003-193, and by the Spanish Research Council (MEC) TIN2004-07474-C02-01.

The challenge of replication is replica control, i.e., keeping copies consistent. Basically all recent solutions perform reads on one replica and writes on all (available) replicas since this performs best for read-intensive applications. The well-known paper of Gray et. al [16] categorizes replication strategies by two parameters. The first parameter determines where updates take place. In a primary copy approach, each object has a primary replica which performs all updates and propagates them to the secondary replicas. Using update everywhere, each replica accepts updates. The second parameter determines when replicas coordinate. In an eager approach updates are executed at all replicas before transactions commit while lazy approaches delay coordination until after transactions commit. Each of the approaches has its own problems. Eager approaches delay transaction execution, and lazy approaches have consistency problems if sites fail before propagating changes. Primary copy approaches require updates to be forwarded to the primary, and update everywhere requires complex concurrency control or conflict resolution. Most commercial database systems provide a suite of in-house replication solutions where eager approaches are mainly developed for high availability and lazy approaches for fast local access and scalability.

The detailed analysis of [16] comes to the conclusion that eager solutions generally do not scale and that both eager and lazy update everywhere have unacceptable conflict rates. Their analysis has motivated many researchers to revisit the issue of replication, e.g., [3, 4, 7, 8, 9, 10, 12, 17, 18, 20, 21, 23, 24, 25, 27, 29]. The principle goal of most of these new algorithms is to eliminate the problems associated with the category to which the solution belongs to, in order to provide both 1-copy-serializability (or other well defined isolation levels), and achieve good performance. One common approach for cluster based replication is to provide a hybrid propagation that is both eager and lazy. The commit is returned to the client once the transaction is committed at one replica but before all replicas have executed the transaction. However, transactions do coordinate before commit. This avoids loss of transactions in the failure case, avoids the complexity of conflict resolution needed in lazy update everywhere schemes, and at the same time provides short response times. The overhead for such hybrid approaches is acceptable when communication is fast [3, 17, 20, 34].

Although many new solutions have been proposed, only some of them have been validated by real implementations. Many implementations, both from research and industry, are middleware based where replication is controlled in a layer between clients and database replicas (e.g., [3, 7, 9, 15, 20, 27, 29]. This simplifies the development since the internals of the database are mostly inaccessible. And if they are, they are complex and difficult to change. Furthermore, middleware solutions can be maintained independently of the database system, and can potentially be used in heterogeneous settings [27]. A main challenge, however, is to coordinate replica control with concurrency control. Most update everywhere systems use locking, and since the SQL statements visible at the middleware do not necessarily indicate the exact records to be accessed, locking is table or predicate based. Additionally, many systems have special requirements. For instance, in [3] a transaction has to indicate all tables it will access at the start. In [20], the programs containing the transactions must run in the same context as the middleware. Many primary copy solutions rely on the concurrency control of the underlying database system (e.g., [8, 10, 12, 24, 27, 29]). However, in order for the middleware to forward update transactions to the primary and distribute read-only load over the secondary copies transactions should indicate at start time whether they are read-only (e.g., by using `Connection.setReadonly()` in JDBC).

In this paper, we propose SI-Rep, a replication middleware that has all the advantages of middleware approaches but avoids the restrictions of existing solutions. We follow a hybrid, update everywhere approach. Applications do not need to be adjusted but simply use the standard database interface. Furthermore, SI-Rep implements concurrency control on a record level. Our approach is useful for applications that require fault-tolerance and data consistency at all times even if update rates are considerably high while still requiring fast response times and high throughput.

Our approach uses snapshot isolation (SI) as isolation level [5]. SI is becoming more widely available in commercial database systems. With SI, a transaction T reads from a snapshot of the database which contains all updates that were committed at the time T started. Only conflicts between write operations are detected, and if two concurrent transactions want to update the same object, one will be aborted. The popularity of SI is due to the fact that reads never conflict with writes since they read from a snapshot. Compared to locking, SI is especially attractive for middleware based concurrency control. Firstly, read operations do not need to be tracked. Secondly, conflicts on write operations can be detected in an optimistic way after transaction execution. These are important properties since the middleware only sees the SQL statements but does not know the records which are going to be accessed before execution.

Our approach performs concurrency control at two levels. We assume the underlying database replicas provide SI, and SI-Rep detects conflicts among transactions running at different replicas. Transaction execution is as follows. A transaction is first executed at one replica. At the end of execution we extract the updated records in form of a writeset. Writeset extraction is a standard mechanism in many commercial replication solutions (e.g., [22]) implemented via triggers or log-sniffing. Although commercial systems usually export writesets only after commit, the functionality per se exists, and we provide a pre-commit extraction similar to the ones developed in other research prototypes [20, 27]. After retrieving the writeset, SI-Rep performs a validation to check for write/write conflicts with transactions that executed at other replicas and that have already validated. If validation succeeds, the transaction commits at the local replica and the writeset is applied at the remote replicas in a lazy fashion. Otherwise it is aborted at the local replica.

In this paper, we follow a stepwise approach to explore our solution. We first develop a formalism that allows us to reason about SI in a replicated system (Section 2). Then, we present a simple algorithm providing SI at the global level using a centralized middleware (Section 3). Then we analyze how database systems actually implement SI (Section 4). This allows for optimization of our basic solution but also requires some adjustments in order not to run into deadlocks. We have implemented SI-Rep on top of PostgreSQL (Section 5). Our implementation is decentralized, that is, a middleware replica is running in front of each database replica. A client can connect via standard JDBC to any replica and is automatically failed over to another replica in case of crash. Middleware replicas communicate through a group communication system providing a total order multicast. Writesets are multicast to all middleware replicas which perform validation in the total order writesets are received in order to guarantee that all replicas make the same validation decisions. Experiments show that our approach has very good performance (Section 6). Related work and conclusions are presented in Sections 7 and 8.

2. SNAPSHOT ISOLATION IN A REPLICATED SYSTEM

In this section, we introduce some formalism that allows us to reason about SI in replicated systems. We follow concepts introduced in [1, 5, 14, 31, 30]. A transaction T_i starts with a begin b_i, followed by a sequence of read $r_i(x)$ and write $w_i(x)$ operations on objects x, and terminates with a commit c_i or an abort a_i. For simplicity of description, we assume a transaction does not read an object x after it has written it and it reads and writes each object at most once[1]. We denote as readset RS_i the set of all objects read, and as writeset WS_i the set of all objects written by T_i. We say a transaction T_i *executes before* T_j (and T_j *executes after* T_i), if T_i commits before T_j starts. If T_i neither executes before or after T_j, then T_i and T_j are *concurrent*.

2.1 Snapshot Isolation Schedules

In order to provide SI, the database system maintains several versions of each object. Whenever a transaction writes an object x it creates a new version. When a transaction T_i reads x, it reads the last committed version of x as of start of T_i. That is, T_i reads the version created by transaction T_j such that T_j executes before T_i, and there is no other transaction T_k that also wrote x, executes before T_i and commits after T_j. Furthermore, if transaction T_i wants to write object x, and there is a concurrent transaction T_j that wrote x and already committed, then T_i aborts. In the following, by indicating that two transactions conflict we mean that they have write operations on the same object.

Our goal is to define as an SI-schedule an execution that is allowed by a snapshot isolation scheduler. Since SI does not provide serializable schedules in the classical sense (see [5, 1, 14]), and reads might access older versions, defining a schedule becomes more difficult. For example, assume an initializing transaction T_0 creates a version of each object. Now assume the set of transactions $\mathcal{T} = \{T_1 = (b_1, r_1(x), w_1(x), c_1), T_2 = (b_2, r_2(y), r_2(x), w_2(y), c),\ T_3 =$

[1]Our implementation does not impose these restrictions.

$(b_3, w_3(x), c_3)\}$. The following shows a possible execution of these transactions under a SI-scheduler reflecting the physical time at which operations take place. We label each data version read with the transaction that created it.

$b_1, r_1(x_0), b_2, r_2(y_0), w_1(x_1), c_1, b_3, w_3(x_3), c_3, r_2(x_0), w_2(y_2), c_2$

Although $r_2(x_0)$ takes place physically very late, it takes place logically just after the begin of T_2 since transactions read from a snapshot. Also, since write operations are not visible before commit, they take place logically just before commit. Hence, we can rewrite above schedule to

$b_1, r_1(x), b_2, r_2(y), r_2(x), w_1(x), c_1, b_3, w_3(x), c_3, w_2(y), c_2$

where a read $r(x)$ reads what the last committed $w(x)$ wrote before the read. In fact, given the set of transactions with corresponding read and writesets, the above schedule can be shortened to a sequence of begin and commit operations: $b_1, b_2, c_1, b_3, c_3, c_2$.
b_i of transaction T_i implicitly indicates when the read operations of T_i take place, and c_i implicitly indicates the time the write operations take place. We call this example schedule SE and use it later. This motivates the following definition.

Definition 1 (SI-Schedule). *Let $\mathcal{T}$ be a set of committed transactions, where each transaction T_i is defined by its readset RS_i and writeset WS_i. An* SI-schedule *S over $\mathcal{T}$ is a sequence of operations $o \in \{b, c\}$. Let $(o_i < o_j) \in S$ denote that o_i occurs before o_j in S. S has the following properties. (i) For each $T_i \in \mathcal{T} : (b_i < c_i) \in S$. (ii) If $(b_i < c_j < c_i) \in S$, then $WS_i \cap WS_j = \emptyset$.*

A similar definition has been made in [14] where it is called a scheduler oriented history. For simplicity of definition we only consider committed transactions[2]. When reasoning about the algorithms, we of course, consider aborts. The above SE schedule can only be an SI-schedule if T_2's writeset neither overlaps with T_1 nor T_3's writeset.

In classical serializability theory, the equivalence of two schedules is an important concept, and both conflict- and view-equivalence are well known equivalence criteria. We can reason in a similar way about SI-schedules.

Definition 2 (SI-Equivalence). *Let S^1 and S^2 be two SI-schedules over the same set of transactions $\mathcal{T}$. S^1 and S^2 are SI-equivalent if for any $T_i, T_j \in \mathcal{T}$ the following holds.*
(i) If $WS_i \cap WS_j \neq \emptyset : (c_i < c_j) \in S^1 \Leftrightarrow (c_i < c_j) \in S^2$.
(ii) If $WS_i \cap RS_j \neq \emptyset : (c_i < b_j) \in S^1 \Leftrightarrow (c_i < b_j) \in S^2$.

Condition (i) indicates that the order of two commit statements matters if writesets overlap since this determines the final writes. Since we also want each prefix of committed transactions to be an SI-schedule (similar to definitions in serializability theory), *all* write/write-conflicting transactions must be committed in the same order. Condition (ii) leads to both schedules having the same reads-from relation.

For instance, our example $SE = b_1, b_2, c_1, b_3, c_3, c_2$, is SI-equivalent to $b_2, b_1, c_1, b_3, c_2, c_3$. The order of two begin statements (b_1/b_2) never matters. The order of c_2/c_3 does not matter since WS_2 and WS_3 do not overlap. However, we cannot change the order of b_2/c_1 since T_2 reads an object written by T_1. $b_1, b_2, b_3, c_1, c_2, c_3$ seems to be SI-equivalent to SE. Since T_3 does not read anything written by T_1 the order of c_1/b_3 does not matter. However, it is not an SI-schedule since $b_3 < c_1 < c_3$ and WS_1 and WS_3 overlap. Equivalence, however, is only defined over SI-schedules.

[2]Aborts would require to build committed projections.

2.2 1-copy-SI

We assume each database replica produces its own SI-schedule over the transactions executed at this replica. Since we follow a read-one-write-all approach, each update transaction has one local replica that performs all its operations. The transaction is called local at this replica, and remote at the other replicas. Only the write operations are applied at these remote replicas. Hence, all replicas execute the same set of update transactions, but an update transaction T_i has a readset RS_i only at one replica while it has the same writeset WS_i at all replicas. Read-only transactions, in contrast, only exist at the local replica. This fact is formalized through a ROWA mapper function *rmap* that takes a set of transactions $\mathcal{T}$ and a set of replicas $\mathcal{R}$ as input and transforms $\mathcal{T}$ into set of transactions $\mathcal{T}'$. *rmap* transforms each update transaction $T_i \in \mathcal{T}$ into a set of transactions $\{T_i^k | R^k \in \mathcal{R}\}$. In this set there is exactly one local transaction $T_i^l = T_i$ (T_i is local at R^l). The rest are remote transactions T_i^r, where $WS_i^r = WS_i$ and $RS_i^r = \emptyset$ (T_i is remote at R^r). A read-only transaction is transformed into a single local transaction $T_i^l = T_i$. We call $\mathcal{T}^k = \{T_i^k | T_i^k \in \mathcal{T}'\}$ the set of transactions executed at replica R^k.

In serializability theory, a replicated system is 1-copy-serializable if the execution in the replicated system is equivalent to a serial execution on a one-copy database. We follow a similar reasoning and require the execution in the replicated system to be equivalent to a global SI-schedule that can be produced by a centralized SI-scheduler (1-copy SI).

Definition 3 (1-Copy-SI). *Let $\mathcal{R}$ be a set of replicas following the ROWA approach. Let $\mathcal{T}$ be the set of submitted transactions for which $T_i \in \mathcal{T}$ committed at its local replica. Let S^k be the SI-schedule over the set of committed transactions $\mathcal{T}^k$ at replica $R^k \in \mathcal{R}$. We say that $\mathcal{R}$ provides* 1-copy-SI *if the following properties hold. (i) There is a ROWA mapper function,* rmap, *such that $\cup_k \mathcal{T}^k = rmap(\mathcal{T}, \mathcal{R})$.*
(ii) There is an SI-schedule S over $\mathcal{T}$ such that for each S^k and $T_i^k, T_j^k \in \mathcal{T}^k$ being transformations of $T_i, T_j \in \mathcal{T}$:
(a) if $WS_i^k \cap WS_j^k \neq \emptyset : (c_i^k < c_j^k) \in S^k \Leftrightarrow (c_i < c_j) \in S$,
(b) if $WS_i^k \cap RS_j^k \neq \emptyset : (c_i^k < b_j^k) \in S^k \Leftrightarrow (c_i < b_j) \in S$.

Property (i) guarantees that the set of committed transactions is a ROWA mapping of a subset of submitted transactions. A submitted update transaction either commits at all replicas or at none, and a submitted read-only transaction commits at the local replica only. Property (ii) requires an SI-schedule S produced by a centralized scheduler over the set $\mathcal{T}$ of committed transactions. Each local schedule S^k must be somehow equivalent to this global schedule. Since $\mathcal{T}^k$ and $\mathcal{T}$ are not exactly the same set of transactions, we cannot take exactly the definition of SI-equivalence, however, the idea is the same. Since $\mathcal{T}^k$ and $\mathcal{T}$ have exactly the same writesets, each local schedule should have the same order as S in regard to all conflicting writesets. Since only the transactions local at R^k have readsets, only those local transactions must have the same reads-from relationship as in S. The position of the begin statements of remote transactions in S^k is irrelevant (as long as S^k is an SI-schedule) since remote transactions do not read anything. Note that also the position of commit statements of read-only transactions is irrelevant both in S^k and in S (except $b < c$, of course), since no writeset is associated with them.

From now on, we write T_i instead of T_i^k if it is clear from the context that we refer to T_i's execution at replica R^k.

3. A BASIC ALGORITHM

We develop our solution in three steps over the next three sections respectively. We first present a Simple Replica Control Algorithm (SRCA). The basic idea is that a centralized middleware forwards each transaction to one of the database (DB) replicas for execution, retrieves the writeset, checks for conflicts with concurrent transactions, and either aborts the transaction, or commits it at the local DB replica and applies the writeset at the remote replicas. We assume each DB replica provides an interface that allows the extraction and application of writesets. Writesets contain the changed objects and their identifiers. Furthermore, we assume that each DB replica provides SI where write/write conflicts are only detected at the end of transactions. We relax this assumption in the next section. We also do not consider failures but look at them in Section 5.

Fig. 1 shows the SRCA algorithm executed by the middleware. All begin, read, write, and commit operations are submitted to the middleware (we ignore abort requests for simplicity). The middleware maintains a list of already validated transactions (*ws_list*). A transaction T receives an identifier $T.tid$ upon validation. Although all successfully validated transactions will be committed at the different DB replicas in validation order the replicas might run at different speed. Hence, the middleware keeps for each replica R^k a queue *tocommit_queue_k* which contains the writesets to be executed and committed at R^k, and the identifier *lastcommitted_tid_k* of the last committed transaction at R^k.

Upon the begin of a new transaction T_i (step I.1) one DB replica is chosen to be the local replica. Before the start of T_i at the DB replica, we get a mutex (I.1.b) that avoids that the begin is concurrent with any commit operations on the replica (II.2). With this, we can exactly determine when T_i starts at the database (I.1.c), and hence are able to determine concurrent transactions. This is reflected by setting $T_i.cert$ to the transaction that was the last to commit at the local replica before T_i started. Read and write operations are then simply forwarded to the DB replica which reads from a snapshot and writes new object versions (I.2). When the commit request is submitted, the middleware retrieves the writeset from the local replica (I.3.a). If it is empty, T_i is simply committed locally (I.3.b). Otherwise, the middleware starts a validation phase (I.3.c-e). Only one transaction can be in validation phase. T_i's writeset is compared against all writesets of concurrent transactions that validated before (maintained in *ws_list*). If the middleware finds a conflict, the transaction is aborted. Otherwise, the transaction receives its *tid* value, and the writeset is added to all queues (*tocommit_queue_k*). It will be applied at different speeds at individual replicas (II). At the local replica, of course, the writeset does not need to be applied. Still, the commit order in regard to other transactions must be maintained. Hence, the local transaction only commits when all the writesets stored in the queue at the time of validation have been applied. In summary, validation phases start when the writesets are retrieved, validation is an atomic process but runs concurrently to committing and applying the writesets. However, applying writesets by itself occurs again in a serial fashion. Note that in order for clients to read their own writes, a transaction should only be assigned to a replica if all previous transactions of the same client are already committed at this replica.

Fig. 2 shows an example that leads to an abort. The set of transactions is again $\mathcal{T} = \{T_1 = (b_1, r_1(x), w_1(x), c_1), T_2 = (b_2, r_2(y), r_2(x), w_2(y), c), T_3 = (b_3, w_3(x), c_3)\}$. T_1 is local at replica $R1$, and T_2 and T_3 are local at replica $R2$. In the figure, grey boxes reflect writes, and white boxes represent reads, begin or commit. The middleware keeps *tocommit_queue_k* for each replica (Q1 and Q2). The figure shows the temporal evolution of the queues and transaction execution from left to right. T_1 starts at $R1$ and reads and updates x. At $R2$, T_2 starts and reads y. Upon T_1's commit request, the middleware retrieves the writeset, validation succeeds, and T_1 receives $T_1.tid = 1$. T_1 is appended to $Q1$ and $Q2$. Since T_1 is the first in $Q1$, T_1 commits at $R1$ and is removed from $Q1$ (*lastcommitted_tid_1 = 1*). T_3 now starts at $R2$. Although T_3 begins after T_1 commits in $R1$, it is concurrent to T_1 in $R2$ since T_1's updates are not yet applied in $R2$. Hence, $T3.cert = 0$. When T_3 now submits commit at timepoint A, T_3's validation at the middleware fails since $T_3.cert = 0 < T_1.tid = 1$ and the writesets over-

```
Initialization:
$next\_tid := 1$
$ws\_list := \{\}$
$\forall R^k : tocommit\_queue\_k := \{\}$
$\forall R^k : lastcommitted\_tid\_k := 0$
$wsmutex, \forall R^k : dbmutex\_k$
I. Upon operation request $OP$ from $T_i$
  1. if $OP == b_i$ then
    a. choose $R^k$ at which $T_i$ will be local
    b. obtain $dbmutex\_k$
    c. $T_i.cert := lastcommitted\_tid\_k$
    d. begin $T_i^k$ at $R^k$
    e. release $dbmutex\_k$
    f. return to client
  2. else if $OP$ is a read or write request
    a. execute at local replica $R^k$ and return to client
  3. else (commit)
    a. $T_i.WS :=$ getwriteset($T_i^k$) from local $R^k$
    b. if $T_i.WS = \emptyset$, then commit and return
    c. obtain $wsmutex$
    d. if $\nexists T_j \in ws\_list$ such that
       $T_i.cert < T_j.tid \wedge T_i.WS \cap T_j.WS \neq \emptyset$
         •$T_i.tid := next\_tid++$
         •append $T_i$ to $ws\_list$
         •$\forall R^m$: append $T_i$ to $tocommit\_queue\_m$
         •release $wsmutex$
    e. else
         •release $wsmutex$
         •abort $T_i^k$ at $R^k$
II. Upon $T_i$ first in $tocommit\_queue\_k$
  1. if $T_i$ is remote at $R^k$
    a. begin $T_i^k$ at $R^k$
    b. apply $T_i.writeset$ to $R^k$
  2. obtain $dbmutex\_k$
  3. commit at $R^k$
  4. $lastcommitted\_tid\_k++$
  5. release $dbmutex\_k$
  6. if local, return to client.
  7. remove $T_i$ from $tocommit\_queue\_k$
```

Figure 1: SRCA: Simple Replica Control Algorithm

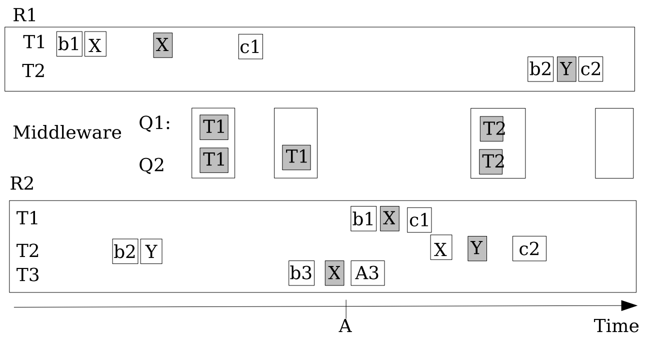

Figure 2: SRCA Sample Execution

lap. Hence, T_3 is aborted at R_2. At the same time, $R1$ applies T_1's writeset and commits T_1. T_1 is removed from $Q2$ and *lastcommitted_tid_2* set to 1. Although T_2's read is after T_1's write it does not read the value written by T_1 since the transactions are concurrent in $R2$. After T_2's execution its validation succeeds ($T_2.tid = 2$) since it has not write/write conflict with T_1. T_2 is appended to $Q1$ and $Q2$ and later committed at both replicas.

THEOREM 1. *Assuming that the underlying DB replicas provide snapshot isolation, SRCA provides 1-copy-SI.*

Proof Sketch We first show that, apart of read-only transactions, all replicas commit the same set of transactions. If validation of a transaction T_i succeeds at the middleware it is appended to *tocommit_queue_k* of each replica R^k. Transactions in *tocommit_queue_k* are handled one after the other. Let T_i be the first in the queue. If T_i is a remote transaction, no other transaction commits between T_i's start and T_i's commit at R^k. Hence, commit is successful. If T_i is local at R^k, then T_i has already started at R^k. If T_i conflicted with any transaction (local or remote) that committed at R^k since T_i's start, R^k would abort T_i when the commit request is submitted since R^k provides SI. But at validation, the middleware had already checked whether there was such transaction, and if yes, would have aborted T_i. Hence, once a transaction is added to *tocommit_queue_k*, it will commit at R^k. This guarantees property (i) of 1-copy-SI.

We now build an SI-schedule S fulfilling property (ii) of 1-copy-SI in the following way. Assume the system after a set of transactions was submitted to the system and all transactions finished execution at all replicas. Let $\mathcal{T}$ be the subset that committed. Let S^k be the schedule and $\mathcal{T}^k$ be the set of committed transactions at replica R^k. Now we choose any replica R^m. For any two update transactions $T_i^m, T_j^m \in \mathcal{T}^m$, if $(c_i^m < c_j^m) \in S^m$ we set $(c_i < c_j)$ in S. Furthermore, we perform for each replica R^k the following actions. For each local transaction $T_i^k \in \mathcal{T}^k$ and update transaction $T_j^k \in \mathcal{T}^k$, if $(c_j^k < b_i^k) \in S^k$ we set $(c_j < b_i)$ in S, and if $(b_i^k < c_j^k) \in S^k$, we set $(b_i < c_j)$ in S. Furthermore, if T_i^k is read-only, we set c_i directly after b_i in S. Since each transaction is local at exactly one replica, and update transactions commit at all replicas, S contains exactly one begin and one commit operation for each committed transaction. S fulfills (ii.a) since all replicas commit all update transactions in exactly the same order, namely the order in which they are appended to the queues. (ii.b) is trivially fulfilled by construction of S. We have to show now that S is an SI-schedule over $\mathcal{T}$. Assume it is not, then there exists a $(b_i < c_j < c_i) \in S$ and $WS_j \cap WS_i \neq \emptyset$. For that to happen, however, there must be $(b_i^k < c_j^k < c_i^k) \in S^k$ where T_i is local in R^k according to construction of S. This is not possible since all local schedules are SI-schedules.

4. WORKING WITH REAL DATABASES

So far, we have assumed that the database checks write/write conflicts at commit time. However, the commercial systems we are aware of (Oracle and PostgreSQL) check for write conflicts during transaction execution by the means of strict 2-phase-locking. A simplified version of the execution is as follows. Whenever a transaction T_i wants to write a tuple x it acquires an exclusive lock, and performs a version check. If the last committed version of x was created by a concurrent transaction, T_i aborts immediately. Otherwise it performs the operation. If a transaction T_j holds a lock on x when T_i requests its lock, T_i is blocked. When T_j commits the lock is granted to T_i. However, T_i fails the version check and aborts because of T_j's update being the last committed version. If T_j aborts, T_i's version check might succeed or still fail (due to another committed but concurrent transaction that updated x). These early conflict checks have positive and negative aspects for the middleware. In the following, we keep reasoning informal for space reasons.

4.1 The good thing

At the middleware, we can take advantage of the fact that the DB replica already performs some validation during the execution of a local transaction. When T_i finishes execution at local replica R^k, SRCA validates T_i against *all* concurrent transactions that validated before T_i. For some of those the DB has already performed the validation. Let T_j be a transaction concurrent to T_i. T_j can be either local or remote at R^k. If T_j is local, both the execution of T_i and T_j finished at R^k. Hence, they do not conflict otherwise the database would have blocked one of the two transactions. Hence, the middleware does not need to validate T_i against other transactions local at R^k. If T_j is remote and has already committed at R^k the same holds. To show this, assume T_i and T_j conflict on tuple x and T_j is concurrent to T_i. Assume T_j acquired the lock on x within the DB replica before T_i requested it, then T_i fails the database internal version check upon T_j's commit, and abort before the middleware can perform validation. If T_i was the first to acquire the lock, then T_j is still waiting for T_i to release the lock at the time the middleware validates T_i. But then T_j is not yet committed and we assumed it is. This means, if T_j is remote and committed, it cannot conflict with T_i, and hence, the middleware does not need to validate against T_j. That is, the only case we have to validate is if T_j is a remote transaction and has not yet committed. In this case, however, T_j is still stored in *tocommit_queue_k*. Therefore, the following simple adjustments can be made to SRCA.

Adjustment 1: The middleware only validates a transaction T_i which is local at R^k against all remote transactions in *tocommit_queue_k*. Data structures *ws_list* and *lastcommitted_tid_k* are not needed. No *dbmutex* is set since there is no need to keep track of transaction start.

4.2 The bad thing

The locking within the database can lead to blocking. Firstly, remote transactions might be blocked by local transactions. Assume a transaction T_i executing locally at R^k and holding a lock on x. Now a remote transaction T_j is applied at R^k and also updates x. T_j will be blocked. Ideally, T_i should be aborted since it conflicts with T_j and has not yet been validated. The middleware will detect this conflict once T_i finishes execution and validates. At this time, T_j is still in *tocommit_queue_k*, and hence, T_i fails validation and aborts. T_j receives the lock and performs the version check within the database which succeeds since T_i aborted.

Secondly, it is possible to have deadlocks between local transactions that have not yet finished execution, and remote transactions that apply their writesets. The database detects such deadlock and abort any of the transactions. If the remote transaction is aborted, the middleware has to reapply the writeset until the remote transaction succeeds.

Thirdly, and this is the most serious problem, SRCA might have a deadlock involving a cycle across the middleware and the database. Assume within the database replica R^k a local transaction T_i holds a lock on x. At the same time, a concurrent local transaction T_j holds a lock on y. A transaction T_r remote to R^k is successfully validated at the middleware. Assume $WS_r = \{y\}$. The writeset is submitted to R^k. Within R^k, T_r attempts to lock y but is blocked since T_j holds this lock. Now T_i finishes execution and validation succeeds at the middleware (it does not conflict with T_r). However, T_i's commit is not submitted to R^k since T_r is still executing at R^k (T_i is not first in *tocommit_queue_k*). Now assume transaction T_j requests a lock on x. Since T_i holds the lock on x, T_j is blocked. Within the database there is no deadlock (T_j waits for T_i, T_r waits for T_j). However, at the middleware layer T_i waits for T_r, leading to a "hidden" deadlock spanning both the middleware and the database.

4.3 Solutions to the hidden deadlock problem

4.3.1 Early writeset retrieval

If T_j, after updating tuple y, informs the middleware about this update, then the middleware, upon validating T_r, can detect the conflict with T_j and ask T_j to abort. However, this is unattractive for two main reasons. Firstly, it is not that simple to abort a transaction at any time. Usually the client can only abort between the submission of different statements. If T_j has already submitted its operation on x and blocks, the middleware might not be able to abort T_j (this is actually the behavior of PostgreSQL). Furthermore, the access of x and y might actually occur within one SQL statement in which case T_j blocks on x before it can provide the middleware with the information that it updated y.

4.3.2 Concurrent commit

The problem of "hidden" deadlock is due to the sequential execution of writesets and commits. The question is whether this is really necessary. If there are two transactions T_i and T_j queued in *tocommit_queue_k* and they do not conflict there seems to be no reason to not execute them concurrently. In particular, if a transaction T_i local to R^k is successfully validated, we can be sure that it does not conflict with any transaction in *tocommit_queue_k*. Hence, we can commit T_i immediately at R^k and still provide property (ii.a) of 1-copy-SI. In the example above, since T_r and T_i do not conflict, T_i can commit locally at R^k and release the lock on x, T_j gets the lock on x, fails the version check, and aborts. At this time, T_r gets the lock on y and can commit. Early commit does not only seem to avoid the deadlock problem but also decreases the response time since a local transaction T_i at R^k can commit immediately after validation without waiting for remote transactions queued in *tocommit_queue_k* to finish. We can even extend this to remote transactions. Whenever a remote transaction T_i queued in *tocommit_queue_k* does not conflict with any transaction queued before T_i, T_i can be applied at R^k and it is guaranteed to commit[3].

Committing non conflicting transactions as soon as validation and execution have finished avoids deadlocks. A transaction T_i only has to wait in *tocommitted_queue_k* to start execution if T_i is remote at R^k and there is another transaction T_j before T_i in *tocommitted_queue_k* that conflicts with T_i. However, T_i cannot be involved in a "hidden" deadlock since it is remote at R^k, has not yet started at R^k and hence, does not yet hold any locks at R^k. With this, the following adjustment can be made to SRCA.

Adjustment 2: Step II of SRCA does not start when T_i is the first in *tocommit_queue_k*, but when there is no conflicting transaction ordered before T_i in the queue.

The problem with this approach is that it does not provide 1-copy-SI. If we allow transactions to commit in different order at different replicas, then we could have transactions T_i, T_j and schedules S^k and S^m at replicas R^k and R^m such that $(c_i^k < c_j^k) \in S^k$ and $(c_j^m < c_i^m) \in S^m$ if $WS_i \cap WS_j = \emptyset$. Assume now, there is a transaction T_a local at R^k and $(c_i^k < b_a^k < c_j^k) \in S^k$, and a transaction T_b local at R^m and $(c_j^m < b_b^m < c_i^m) \in S^m$. Assume that $WS_i{=}\{x\}$, $WS_j{=}\{y\}$ and $RS_a{=}RS_b{=}\{x, y\}$ and all transactions commit. T_a and T_b might also perform updates but this is irrelevant. A global SI-schedule over T_i, T_j, T_a and T_b cannot fulfill (ii.b) of 1-copy-SI since this would mean $c_i < b_a < c_j < b_b < c_i$ which is impossible. The problem is that by starting T_a and T_b between the commits of T_i and T_j, they induce an indirect conflict between T_i and T_j, and the conflict order is different at the different replicas.

4.3.3 Synchronizing start and commit order

We can solve the problem by sometimes delaying the start of transactions. Let T_i be validated before T_j, and T_i and T_j do not conflict. We allow T_j to commit before T_i at replica R^k, and say that the commit order at R^k has a hole. The hole is closed when T_i commits. We allow a local transaction T_a at R^k to start only when there are no holes in the commit order. Hence, if T_j commits before T_i at R^k then T_a has to wait to start until T_i also commits at R^k. In this case, we have the following guarantee: if $(c_i^k < c_j^k) \in S^k$ of any replica R^k, then either T_i validated before T_j or $\nexists T_a$ local at R^k and $(c_i^k < b_a^k < c_j^k) \in S^k$. Hence, by only allowing transactions to start when there are no holes in the commit order, we guarantee that indirectly induced conflicts always

[3]Note that although it looks like that when enqueuing a transaction T_i into a queue, there are never conflicting transactions queued before T_i, this is not true. Assume a replica R^k executes T_i updating x, commits T_i and then starts local transaction T_j also updating x. T_j passes validation and is appended to all queues. Now assume a replica R^m for which both T_i and T_j are remote. When T_j is appended to *tocommit_queue_m*, T_i might still reside in the queue. At R^m we may not apply T_j's writeset before T_i commits because otherwise both transactions might run concurrently in the database (and then one is aborted) or T_j might execute before T_i which results in the wrong final write.

lead to a dependency according to the validation order.

If we do this, we allow concurrent execution of writesets and concurrent commits. However, we might delay the start of a transaction, possibly indefinitely if there are always holes (i.e. a livelock). A solution to this liveness problem is to disallow new holes at certain timepoints. Eventually, all existing holes will close, and all waiting transactions can be started. However, we must be careful to choose timepoints at which disallowing new holes does not lead to a hidden deadlock. These timepoints can be determined as follows. Let A be the set of local transactions waiting to start at R^k because of holes in R^k's commit order. Let B be the set of local transactions at R^k that are already running. No new transactions can be added to B as long as there are holes. We allow new holes to be created until B is empty. B will eventually be empty (due to the absence of hidden deadlocks by terminating local transactions immediately after validation even if this creates holes). Once B is empty, we delay the commit of further transactions until all holes have disappeared (i.e. only transactions that remove holes or do not create new ones are allowed to commit). This does not lead to hidden deadlocks since there are only remote transactions delayed in *tocommit_queue_k* which have not yet started and acquired locks at R^k. Once there are no holes, we start all transactions in A moving them to B before allowing any further commit. Then, we again allow holes in the commit order. The following *adjustemnt 3* describes the required coordination.

- Only start a local transaction T_i at R^k if there are no holes in the commit order of R^k.
- Commit a transaction T_i at R^k only if no local transaction is waiting to start at R^k or T_i is local at R^k or no new hole is created by commit of T_i.

5. IMPLEMENTATION

5.1 Architecture

So far, we have assumed a centralized middleware (Fig. 3.a) which is attractive due to its simplicity but it is a single point of failure. Alternatively, we could have a primary and a backup middleware, and all clients are switched over to the backup if the primary fails (Fig. 3.b). However, failover might be quite complicated because in case of failures the connections between the primary and the DB replicas are broken. Typically, upon connection loss, database systems abort the active transaction on the connection. At the time the primary crashes, a given transaction T_i might be committed at some DB replicas, active at others, and not even started at some. The backup has to make sure that such transactions are eventually committed at all replicas.

A third solution, and this is the one implemented in SI-Rep, is a completely decentralized middleware (Fig. 3.c). For each DB replica R^k there is a middleware replica M^k which is connected only to R^k and communicates with all other middleware replicas. A client is connected to one middleware replica and, in case of crash of this middleware replica, is reconnected to any of the available replicas. We discuss failures in more detail in Section 5.4. The middleware provides a standard JDBC driver to the client.

5.2 Communication

Communication between the middleware replicas must be efficient and able to handle failures. For that purpose, we use

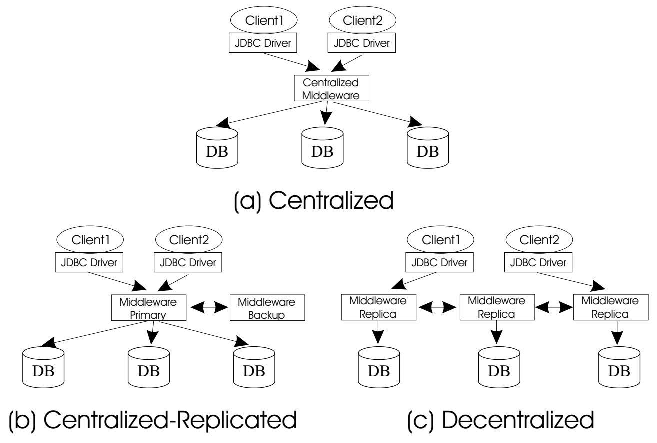

Figure 3: Middleware Architectures

group communication technology in a similar way as previous data replication proposals (e.g., [25, 17, 20, 34]). Group communication systems (GCS) provide powerful primitives and have shown to be a useful abstraction for replicated and fault-tolerant systems [6]. In here, we only introduce the concepts of GCS that are needed in our context. A GCS provides multicast primitives which multicast a message to all members of a group. One of these primitives delivers all messages in the same total order to all members. Using total order multicast, any two messages $m1$ and $m2$ multicast to a group (even if from different senders) are delivered to all members in the same order, that is, either $m1$ before $m2$ or vice versa. Usually senders are also group members and then receive their own messages in correct order. A GCS also provides group maintenance. For instance, it detects member crashes and recoveries and informs the surviving members about such membership changes. In regard to crashes, a GCS also provides different levels of delivery guarantees. All levels deliver all messages to all members if there are no failures. Additionally, choosing uniform reliable delivery, even if a member receives a message and then crashes, then all members that do not crash will not only receive the message but receive it before they are informed about the crash. In our implementation all middleware replicas belong to the same group and communicate through uniform reliable, total order multicast. GCS also provide mechanisms to handle network partitions and determine a single primary view. However, we currently do not consider network partitions. Other replication proposals have looked into how to use GCS to handle partitions and we believe that similar extensions can be applied to our system [2].

GCS themselves are complex software systems, and have their own overhead. In particular, to provide total order and uniform reliable delivery, some form of agreement protocol has to be run. However, in contrast to atomic commit protocols for databases, no logging is involved, and these systems are very efficient, at least for LAN environments. We are currently using Spread [32]. In here, the delay for a uniform reliable multicast does not exceed 3 ms in a LAN even for message rates of several hundreds of messages per second. Reconfiguration of the system after a crash can take up to a couple of seconds depending on the timeout interval chosen to detect crashes.

```
Initialization:
$lastvalidated\_tid := 0$
$ws\_list := \{\}$
$tocommit\_queue\_k := \{\}$
$wsmutex$
I. Upon operation request for $T_i$ from local client
  1. If select, update, insert, delete
    a. If first operation of $T_i$ wait until no holes in commit
       order and then begin $T_i^k$ at $R^k$
    b. execute operation at $R^k$ and return to client
  2. else (commit)
    a. $T_i.WS :=$ getwriteset($T_i^k$) from local $R^k$
    b. obtain $wsmutex$
    c. if $T_i.WS = \emptyset$, then commit and return
    d. if $\exists T_j \in tocommit\_queue\_k \wedge T_i.WS \cap T_j.WS \neq \emptyset$
         •release $wsmutex$
         •abort $T_i^k$ at $R^k$ and return to client
    e. $T_i.cert := lastvalidated\_tid$
    f. release $wsmutex$
    g. multicast $T_i$ using total order multicast
II. Upon receiving $T_i$ in total order
  1. obtain $wsmutex$
  2. if $\exists T_j \in ws\_list$ such that
     $T_i.cert < T_j.tid \wedge T_i.WS \cap T_j.WS \neq \emptyset$
       •release $wsmutex$
       •if $T_i^k$ local then abort $T_i^k$ at $R^k$ else discard
  3. else
       •$T_i.tid := ++lastvalidated\_tid$
       •append $T_i$ to $ws\_list$ and $tocommit\_queue\_k$
       •release $wsmutex$
III. Upon $\forall T_j, T_j$ before $T_i$ in $tocommit\_queue\_k$ :
     $T_i.WS \cap T_j.WS = \emptyset$, and either $\nexists T_a$ waiting to start
     at $R^k$ or $T_i$ is local or $T_i$ does not create a new hole.
  1. if $T_i$ is remote at $R^k$
    a. begin $T_i^k$ at $R^k$
    b. apply $T_i.writeset$ to $R^k$
  2. commit at $R^k$
  3. if local, return to client.
  4. remove $T_i$ from $tocommit\_queue\_k$
```

Figure 4: SI-Rep: SRCA-Rep at M^k

5.3 Replica Control

We have implemented a decentralized version of SRCA, called SRCA-Rep (including the adjustments 1-3 of Section 4). When a transaction is submitted to a middleware replica M^k it is first executed at the local DB replica R^k. At the end of execution M^k retrieves the writeset and multicasts it to the other middleware replicas. All middleware replicas now perform validation and must take the same decision. In order to do so, we send writesets using total order multicast. That is, if two middleware replicas concurrently multicast two writesets WS_i and WS_j, then either all middleware replicas (including the senders) receive WS_i before WS_j or vice versa. We require all middleware replicas to validate transactions in writeset delivery order.

Fig. 4 describes SRCA-Rep running on middleware replica M^k. A client of M^k submits the operations of its transactions only to M^k which executes them locally. JDBC does not have a begin statement. Instead, the first operation after a commit/abort automatically starts a new transaction[4]. In SRCA-Rep, we need an explicit start in order to synchronize with commits (step I.1.a). Upon the commit request, M^k retrieves the writeset from the DB replica and multicasts it to all other middleware replicas in total order (steps I.2.a and I.2.g). Each middleware replica M^k has to perform validation for all writesets in delivery order, and then apply remote writesets and commit transactions (steps II and III). It only maintains a single $tocommit\_queue\_k$ for its local R^k.

In order to reduce the validation overhead, SRCA-Rep validates in two steps, first a *local validation* at M^k at which T_i is local, and later a *global validation* at all replicas. Local validation occurs at the same timepoint as validation occurs in the centralized solution, namely directly after the writeset is retrieved (I.2.d). In only validates against transactions in the local $tocommit\_queue\_k$ as discussed in adjustment 1. Only if this local validation succeeds, M^k multicasts T_i's writeset (I.2.g). Since middleware replicas can send writesets concurrently, several writesets of other transactions might be delivered between sending and receiving T_i's writeset. When validating locally, M^k had not considered these writesets. Instead this is the task of global validation (II.2). By setting $T_i.cert$ accordingly just before the multicast, at the time T_i's message is received, T_i is really only validated against transactions who were multicast concurrently but received before T_i. Once T_i is successfully validated it can be (applied if remote) and committed once there is no transaction with overlapping writeset before it in $tocommit\_queue\_k$. The synchronization between start and commit as described in adjustment 3 must be performed.

5.4 Client Interaction and Failures

A client is connected to one middleware replica via a standard JDBC interface. The SI-Rep JDBC driver is designed for LANs where we assume no network partitions but we provide automatic failover in case of site or process crashes. We are not aware of any other other replication solution based on GCS that performs such transparent failover. The middleware as a whole has a fixed IP multicast address where the middleware replicas are the final recipients. With this, IP addresses of individual replicas do not need to be known in advance. Upon a connection request, the SI-Rep JDBC driver multicasts a discovery message to the multicast address. Replicas that are able to handle additional workload respond with their IP address/port. The driver connects to one of them, keeping the others for failure cases.

We can consider different types of crashes. The machine on which a middleware/DB replica pair resides crashes, only the middleware process crashes or only the DB crashes. In all three cases, we consider the entire pair unavailable. If the middleware replica crashes all its client connections are lost. The drivers on the clients will detect this and automatically connect to another replica. At the time of crash the connection might have been in one of the following states.

1. There was currently no transaction active on the connection. In this case, failover is completely transparent.
2. A transaction T was active and the client has not yet submitted the commit request. In this case, T was still local on the middleware/DB replica that crashed, and

[4]This is the behavior if `autocommit=off`. Otherwise each statement should be executed in its own transaction. For simplicity, we only show the case for `autocommit=off`.

the other replicas do not know about the existence of T. Hence, it is lost. The JDBC driver returns an appropriate exception to the client program. But the connection is not declared lost, and the client can restart T.

3. A transaction T was active and the client has already submitted the commit request which was forwarded to the middleware replica. In this case, the state at the remaining available replicas might be as follows:
   a.) They have not received T's writeset, and hence, do not know about the existence of T, and T must be considered aborted.
   b.) They have received T's writeset. If validation succeeds, they commit T.

   Note that uniform reliable delivery guarantees that if the local replica received the writeset and committed T before the crash, then all (available) remote replicas receive the writeset and hence, also commit T.

Let's have a closer look at case 3. If clients are directly connected to the database and the database crashes after a commit request but before returning the confirmation, clients do not know whether the transaction aborted or committed. In our case, however, we are able to provide such feature. When a new transaction starts at a middleware replica, the replica assigns a unique transaction identifier and returns it to the driver. Furthermore, the identifier is forwarded to the remote middleware replicas together with the writeset. Each replica keeps these identifiers together with the outcome of the transaction (commit/abort determined at validation). If now a crash occurs during a commit request, the JDBC driver connects to a new replica and inquires about the in-doubt transaction by sending the transaction identifier. If the new replica had not received the writeset, it does not know about the identifier, and hence, informs the driver that the transaction did not commit. The driver returns the same exception to the client as if the commit was not yet submitted at the time of crash. If the new replica has the identifier, it checks for the outcome and returns the outcome to the driver which forwards it to the client program. In this case, failover was completely transparent.

Note that due to the asynchrony of message exchange it might be possible that the middleware receives the inquiry about a transaction from a driver and only after that it receives the writeset for the transaction. In order to handle this correctly, the replica does not immediately return to the JDBC driver if it does not find the transaction identifier. Instead, it waits until the GCS informs it about the crash of the old replica. According to the properties of the GCS, the new replica can be sure that it either receives the writeset before being informed about the crash or not at all. Hence, it can inform the driver accordingly.

So far, recovery and the joining of new nodes is offline. Transaction processing has to come to a halt, and then a complete database copy is transferred from one of the DB replicas to the new or recovering replica. Only then, transaction processing can proceed. In order to perform online recovery, the middleware probably has to log writesets. This would allow it to send the writesets a crashed replica has missed upon recovery. It would also allow for recovery from total failure where the different replicas might have committed different subsets of transactions just before the crash.

5.5 Database Interaction

We are currently using PostgreSQL as our underlying database system providing SI. We have extended PostgreSQL for writeset management [20]. Our implementation intercepts the query execution after an update happens on a tuple and forwards the modified tuple to a writeset management module for marshalling. This module exports two methods that can be called by users, one to retrieve the writeset and one to apply a writeset. [27, 33] are other PostgreSQL based replication tools using triggers for writeset retrieval. Currently, some developers associated with the PostgreSQL development team have started implementing a writeset feature with which users can retrieve writesets in readable format before or after commit. They will not use triggers but will set hooks in the code that allow access to modified tuples [19]. We will use this feature once available.

The JDBC driver provided by SI-Rep forwards requests to the middleware. The middleware then uses PostgreSQL's original JDBC driver to submit requests to the database.

One problem of using JDBC to connect to the database is that it does not provide any explicit `begin` statement but assumes a transaction starts at the first operation after a commit/abort. SRCA-Rep has to synchronize the start of a transaction with the commit of other transactions to guarantee no holes. However, starting the transaction with the first operation might already lead to a hidden deadlock. Therefore, we currently submit a dummy query statement on an empty table as a fast and non-blocking start.

6. PERFORMANCE EVALUATION

We tested SI-Rep on three different workloads in order to understand its performance behavior. We first provide results using the TPC-W benchmark to see how SI-Rep behaves under a real application. In order to better understand the behavior of the system in certain circumstances we used our own, simplified benchmarks. All results were run on a cluster of standard PCs (Pentium 4, 2.66 GHz with 0.5 GByte RAM) running Linux. In each test run a certain number of clients are connected to one middleware replica. Within a transaction, a client submits the next SQL statement immediately after receiving the previous one, but it sleeps between submitting two different transactions in order to achieve the desired system wide load. All tests were run until a 95/5 confidence interval was achieved.

6.1 TPC-W

TPC-W [11] simulates a bookstore with three different kinds of workloads that vary in the ratio of update vs. read-only transactions. We have chosen the ordering workload that consists to 50% of update transactions and 50% of read-only transactions. The database has eight tables, and the size of each table is determined by the items and emulated browsers in the system. Our configuration has 1000 items and 40 emulated browsers. This results in a relatively small database of around 200 MBytes.

Figure 5 shows the average response times for update and read-only transactions with increasing system load and five replicas. Additionally, the response times for a centralized system are also presented (it still uses our middleware but the middleware simply forwards requests to the single database and does not perform any concurrency control, writeset retrieval, etc.). The benchmark has many short queries, which gives queries on average a smaller response time. As expected, the response time increases with the load in the system until the system is saturated. At 25 trans-

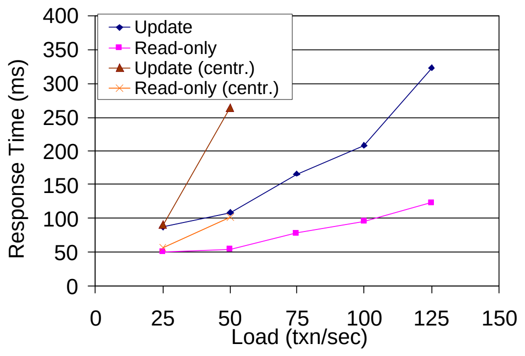

Figure 5: Response Times for TPC-W

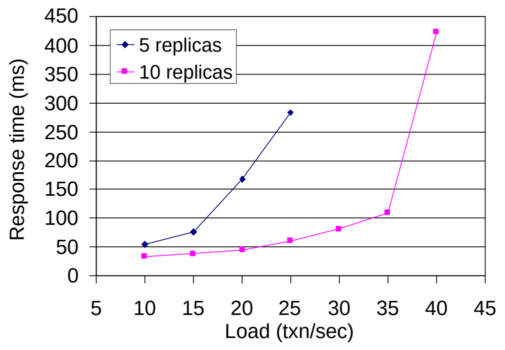

Figure 6: Large Database

action per second (tps), the centralized and the replicated system have more or less the same response times since the system is only lightly loaded. At this load, the overhead of the middleware (communiation/validation) is compensated by the fact that queries are distributed over 5 replicas compared to the centralized system. At 50 tps, however, the centralized system is already saturated while the replicated system can handle loads up to 100 tps with acceptable performance. Although the database is relatively small, conflict rates were small, and very few aborts took place (far below 1%). This shows that the approach has potential to provide scalability and increased performance over a centralized system while providing fault-tolerance and full data consistency.

6.2 Large Databases

In this section, we look at a scenario with a large database of 1.1 GBytes. Each database has 10 tables. There are two transaction types. One is an update transaction with 10 update operations, the other is a query with medium execution requirements, and the update/query ratio is 20/80. The application is read intensive and highly I/O bound. Hence, we can expect that by adding more replicas we can either increase the maximum achievable throughput or decrease the response time of individual transactions since the read load can be distributed over all replicas.

Fig. 6 shows the response time of update transactions with increasing load for 5 and 10 replicas. Results are similar for read-only transactions but response times were generally longer due to the choice of query. The results for a centralized system are not shown since the maximum achievable throughput is around 4 tps with a response time of over 300 ms for update transactions. We did not use any indexes or performed other tuning, hence the performance of PostgreSQL on this configuration is rather limited. In contrast to a single server configuration, a 5-replica system can handle a load up to 20 tps without exceeding a response time of 200 ms. A 10-replica system can achieve such response times up to 35 tps. Hence, we can in fact achieve the scalability that is required for read-intensive applications.

6.3 Analyzing the Overhead of Replication

In this experiment, we want to analyze the overhead of replica control. For that purpose we set up a configuration that stress-tests the system. The database is very small with only 14 MBytes, again having 10 tables. This time, we only run update transactions performing 10 simple updates.

For this experiment, we run two versions of SRCA-Rep. The original one, and one that we call SRCA-Opt. SRCA-Opt does not perform the synchronization between starting transactions and committing transactions but allows transactions to start even if there are holes in the commit order. That is, it only implements adjustments 1 and 2. While each transaction runs under SI within its local database, 1-copy-SI might be lost. By comparing SRCA-Opt and SRCA-Rep we want to get an idea of how costly the synchronization overhead is. We also compare the result with an existing middleware based replica control algorithm based on GCS [20] which we have reimplemented into our system. It requires a transaction to run in the same context as the middleware, and all tables a transaction accesses must be known in advance to the replica control system. Clients submit parametrized requests to execute a transaction. Upon receiving a client request for an update transaction, the middleware multicasts it to all middleware replicas with total order. All replicas acquire all necessary table level locks for the transaction in delivery order. Only one replica (determined by some policy) executes the transaction, retrieves the writeset and multicasts it to the remote replicas (in FIFO order) where the writeset is applied once the locks are granted at these replicas. The local middleware returns to the client once the transaction has executed and committed locally. This algorithm requires two messages per transaction. We have evaluated a configuration where a transaction accesses three different tables (which is a bit less than the number of tables accessed by a typical transaction in TPC-W).

Fig. 7 shows the response times for update transactions with increasing loads for 5 replicas and a centralized system. In this setting, SRCA-Rep and SRCA-Opt have similar response times at small loads but SRCA-Rep is worse than SRCA-Opt at higher loads. SRCA-Rep spends time in the synchronization of begin and commit statements. This synchronization overhead is high when there are many transactions. On average, there are holes at around 4-8% of the times a transaction wants to start. These holes are mainly generated by local transactions that immediately commit after validation, overtaking remote transactions in the *tocommit_queue_k*. Whenever there are holes, starting transactions have to wait. Whenever transactions start, commit operations have to wait. SRCA-Opt does not have this synchronization problem. However, we do not expect that there are many applications running on such a small

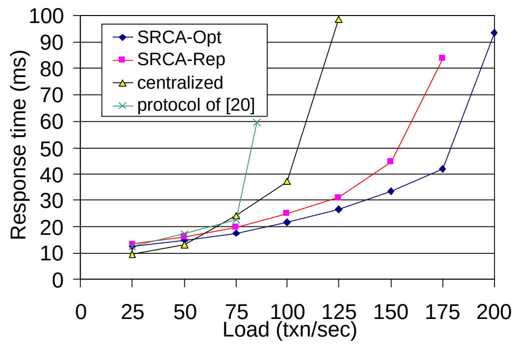

Figure 7: Update intensive workload

database with such a high load of short update transactions (13 Mio transactions per day). In fact, we also tested SRCA-Opt on the large database of the previous section, and the performance differences between SRCA-Rep and SRCA-Opt were very small. However, in update intensive workloads, SRCA-Opt might be a better alternative even it does not provide full 1-copy-SI. This might be comparable with approaches in centralized systems where at high workloads lower levels of isolation are chosen (e.g., READ COMMITTED) to speed up performance.

Compared to the centralized system, SRCA performs worse at low throughput since SRCA requires communication and additional validation. But it can achieve a higher maximum throughput, what is surprising with 100% updates. This is due to the fact that remote replicas only apply writesets in comparison to executing the complete SQL update statements. Applying writesets takes only around 20% of the time it takes to execute the entire transaction. Hence, having more than one replica is still able to alleviate the load on each replica.

SRCA (-Rep and -Opt) has similar response time compared to the replication approach of [20] but achieves higher maximum throughput. There are several differences in the approaches. The protocol of [20] acquires table level locks and the resulting lock contention is the reason why it saturates earlier. At the saturation point too many requests are queued and the system deteriorates. At low loads and conflict rates, however, response times are similar. Although it needs more messages than SRCA, this overhead is compensated by having less client/middleware interaction (SRCA has communication for each statement of a transaction while in the protocol of [20] a client only sends one request per transaction). [20] does not provide transparent replication and a standard JDBC interface. Instead, application programs must be embedded within the middleware system and analyzed to determine the tables to be accessed. These results show that the transparency provided by SRCA does not come at a price of less performance.

We did not compare against any of the open-source replication solutions for PostgreSQL because they are all lazy where updates are only propagated after a considerable delay. Hence, the semantics of these solutions is very different to ours. We tested the system against Postgres-R [34] which provides kernel-based eager replication. The results were very similar to SRCA-Rep since their main difference lies in the validation process while the principal transaction execution is similar (see Section 7). In fact, SRCA-Rep was slightly faster than Postgres-R but we believe the main reason are some inefficiencies in the prototype of Postgres-R which would not occur in a more professional implementation.

7. RELATED WORK

Basically all commercial systems provide one or more replication solutions. Eager solutions are typically for fault-tolerance and often provide only a primary/backup system where backups only have limited reading capabilities (e.g., DB2 HADR and Oracle Data Guard). There exist a wide range of lazy solutions for various kinds of applications. They are often primary copy and it is the responsibility of the client to connect to the primary if it wants to update data (secondary copies simply reject updates). For lazy, update everywhere approaches many different conflict resolution strategies are provided but the users are recommended to only use such approaches if conflict rates are very low or conflict resolution is relatively easy for the application in question. Our approach, in contrast, is aimed at applications where replication should be used for both fault-tolerance and scalability, and conflict rates are high enough to require to be addressed before transaction commit.

In regard to research, there exists a huge body of replication literature and we are able to cover only some of the most recent work. In regard to lazy, update everywhere replication, a main issue is efficient update propagation guaranteeing eventually consistent data [28, 26]. Using a hybrid approach, SI-Rep has always consistent data.

In regard to lazy, primary copy replication, much research has focused on data placement strategies for systems with multiple primaries guaranteeing serializability [16, 10, 4, 8, 23]. Since we use an update everywhere scheme, we do not have these concerns. [12, 27] analyze how to guarantee session consistency where clients see their own changes when reading from secondary copies. [24] studies the effects of different update propagation techniques on the freshness of data. The focus of [29] is to provide efficient query execution and freshness properties for queries. SI-Rep provides data freshness by its hybrid update propagation. All primary copy approaches can only accept updates at the primary. This restriction does not exist in our system. Furthermore, many of the approaches do not consider fault-tolerance which is provided by our system in a transparent manner.

In regard to eager and hybrid approaches, [4] uses a global serialization graph for conflict detection. However, they do not indicate how to integrate their approach with existing database technology. [3, 9, 20] provide replication at the middleware layer and perform pessimistic concurrency control at a coarse level. Additionally, some of them require transactions to pre-declare some properties. These limitations are exactly what we try to eliminate in our system. Our performance measurements show that we can achieve this without performance penalty. [34] provides SI based replication integrated into the kernel of PostgreSQL. Although the general execution is similar using total order multicast to support conflict detection, the validation itself is very different since [34] is kernel based, and our solution is middleware based. Also, [34] does not provide transparent failover. [21, 13] provide replication based on SI. However, their algorithms are on a very abstract level and ignore the challenges one has to face when working with a real system.

[13] also provides a correctness criteria with similar properties to 1-copy-SI. However, their definitions and reasoning are quite different to ours. [27] extends the notion of SI for primary copy replication to allow read-only transactions to access older snapshots at secondary copies. [31, 30] discuss how to provide global SI and serializability in a federated database if the individual database systems provide SI. However, the solution in [31] is quite conservative and might potentially disallow many schedules that provide SI. The solution in [30] requires to understand reads-from relationships by analyzing SQL statements.

8. CONCLUSION

In this paper, we present a middleware-based replica control mechanism that provides snapshot isolation for the entire replicated system. The middleware has its own concurrency control system complementing the one of the underlying systems, and detecting conflicts on a tuple level. The system is compatible with existing database systems that provide snapshot isolation through a lock-based conflict detection mechanism. Our prototype SI-Rep is implemented on top of PostgreSQL providing a standard JDBC driver interface to the clients. The system is fault-tolerant, provides full data consistency, and has very good performance outperforming a centralized system for all tested workloads. We are currently extending our system to provide recovery without interrupting transaction processing. We are also examining load-balancing issues.

9. REFERENCES

[1] A. Adya, B. Liskov, and P. E. O'Neil. Generalized isolation level definitions. In *ICDE*, 2000.

[2] Y. Amir and C. Tutu. From Total Order to Database Replication. In *Proc. of Int. Conf. on Distr. Comp. Systems (ICDCS)*, July 2002.

[3] C. Amza, A. L. Cox, and W. Zwaenepoel. Consistent replication for scaling back-end databases of dynamic content web sites. In *Middleware*, 2003.

[4] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, consistency, and practicality: Are these mutually exclusive? In *ACM SIGMOD*, 1998.

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

[6] K.P. Birman. *Building Secure and Reliable Network Applications*. Prentice Hall, 1996.

[7] K. Böhm, T. Grabs, U. Röhm, and H.-J. Schek. Evaluating the coordination overhead of synchronous replica maintenance. In *Euro-Par*, 2000.

[8] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *ACM SIGMOD*, 1999.

[9] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *USENIX Conference*, 2004.

[10] P. Chundi, D. J. Rosenkratz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *ICDE*, 1996.

[11] Transaction Processing Performance Council. TPC Benchmark W.

[12] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *ICDE*, 2004.

[13] S. Elnikety, F. Pedone, and W. Zwaenepoel. Generalized snapshot isolation and a prefix-consistent implementation. Technical report, EPFL, 2004.

[14] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable.

[15] Foedero Technologies Inc. Foederoreplica 1.0, 2004. http://www.foedero.com/FoederoReplica.html.

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

[17] J. Holliday, D. Agrawal, and A. El Abbadi. The performance of database replication with group communication. In *IEEE FTCS*, 1999.

[18] J. Holliday, R. Steinke, D. Agrawal, and A. Abbadi. Epidemic algorithms for replicated databases. *TKDE*, 15(5), 2003.

[19] J. Wieck. Personal Communication on Slony II.

[20] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *ICDCS*, 2002.

[21] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM TODS*, 25(3), 2000.

[22] C. Liu, B. G. Lindsay, S. Bourbonnais, E. Hamel, T. C. Truong, and J. Stankiewitz. Capturing global transactions from multiple recovery log files in a partitioned database system. In *VLDB*, 2003.

[23] E. Pacitti, P. Minet, and E. Simon. Replica consistency in lazy master replicated databases. *Distributed and Parallel Databases*, 9(3), 2001.

[24] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *VLDB Journal*, 8(3), 2000.

[25] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting Atomic Broadcast in Replicated DBs. In *EuroPar'98*.

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

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

[28] M. Rabinovich, N. H. Gehani, and A. Kononov. Scalable update propagation in epidemic replicated databases. In *EDBT*, 1996.

[29] U. Röhm, K. Böhm, H-J. Schek, and H. Schuldt. FAS - a freshness-sensitive coordination middleware for a cluster of olap components. In *VLDB*, 2002.

[30] R. Schenkel and G. Weikum. Integrating snapshot isolation into transactional federations. In *CooPIS'00*.

[31] R. Schenkel, G. Weikum, N. Weissenberg, and X. Wu. Federated transaction management with snapshot isolation. In *FMLDO*, 1999.

[32] Spread. homepage: http://www.spread.org/.

[33] J. Wieck. Slony-I, A replication system for PostgreSQL. White Paper. http://gborg.postgresql.org/project/slony1.

[34] S. Wu and B. Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*, 2005.