Ph.D. Thesis Proposal
Database Replication in Wide Area Networks

Yi Lin

Abstract

In recent years it has been shown that database replication is promising in improving performance and fault tolerance of database systems. Data replication means that there exist many copies of the same data. A challenge is replica control, i.e., to keep copies consistent despite updates. Many replica control protocols have been proposed. Most of these protocols have two shortcomings. Firstly, although these protocols perform well for cluster based systems in Local Area Networks (LAN) they are not applicable in Wide Area Networks (WAN) due to the much longer communication delay in WAN. Secondly, database replication must gurantee certain levels of transaction isolation, i.e., to what extent transactions will interfere with each other. Most of the existing protocols guarantee serializability, which has become less popular than Snapshot Isolation (SI), a new transaction isolation level. My thesis aims to propose a replication solution which guarantees SI and works well for WANs. Furthermore, I use a middleware approach which provides replica control outside the database system. This provides flexibility and allows heterogeneous configurations. Existing middleware based approaches have severe restrictions. For instance, some require all operations of a transaction to be known in advance or that transactions must be marked as read-only or update. Furthermore, concurrency control is usually at a coarse level, e.g., table level. My research aims in overcoming these restrictions in order to provide a flexible and transparent solution.

1 Introduction

1.1 What and why database replication in WANs

A replicated database system is composed of many copies of databases distributed across different sites. Each database, being called a replica, can work individually to accept clients requests. The database replicas work cooperatively as a global database system to provide database services to clients at all sites. Clients submit read and write operations to the system. They can bundle several such operations into the unit of a transaction, requiring that they are executed as a single observable action.

Database replication is used for performance and availability. Performance refers to response time and throughput of the system. Nowadays, businesses are becoming more geographically dispersed, yet employees still need access to a single set of coherent data. [25] shows that the execution time for a transaction of the TPC-W benchmark [31] in a centralized database is about 100 milliseconds. This does not include the communication cost between client and database for a transaction. Assuming there is only one message round between clients and databases, we can expect an additional few milliseconds added to the client response time if a client is connected to a local database, while the total response time will be around 200 milliseconds in a WAN setup since message round trip is around 100 milliseconds in WANs. With database replication, data can be replicated to remote sites so that clients in the remote sites can access the data just locally. Thus clients receive fast response since WAN communication does not occur. In order to improve throughput, since each replica can handle client requests, we can add replicas to increase the work capacity of the system. This is especially useful in clusters configuration, but might also be true for WANs.

In regard to availability, if a replica fails, client requests can be routed to other replicas. In case of catastrophic disaster such as fire, in which hardware is destroyed, data will survive as long as one replica, probably in a different geographical site, remains accessible.

In all, database replication in WAN is desirable to be considered.

1.2 Challenge of database replication

The main challenge of database replication is to keep the data copies consistent in the presence of updates. If a client updates a data copy, the update has to be propagated to other copies. If clients connected to different replicas submit updates on the same data items, such updates have to be coordinated to guarantee that the data remains consistent.

An ideal replicated database system is that database replication should be transparent to clients as if there is only one centralized database. This goal results in the challenge of how to guarantee the correct execution of transactions globally as if they are executed on one logical database. In order to speed up read transactions which occur more often than write transactions in most client applications, many replication solutions follow a scheme of Read One Write All Available (ROWAA) in which reads are performed on one replica (the local) and writes on all available replicas (those that have not crashed). With ROWAA, reads are as fast as having a single, local database, while writes trigger a considerable update overhead. This is acceptable if the ratio of read to write is high, and has shown to outperform basically all other approaches (e.g., quorum) [22]. My proposal also follows this scheme.

1.3 Existing work and their shortcoming

How to guarantee that the data in all replicas is consistent (i.e., the same) in the presence of updates is the big challenge of replica control. Some replication protocols provide strong consistency meaning that data must be consistent at any time. This, however, increases response time for updates because replicas must coordinate such updates before the response is sent to the client. Other protocols only provide weak consistency meaning that data may be inconsistent temporarily though it will be consistent finally. This provides fast response for writes but transactions may read stale data when they read local data which does not yet reflect updates performed on remote replicas. Some protocols with weak consistency even require to rollback updates previously applied and committed. This complicates the system and exposes to applications a weird behavior of the systems. Section 3.1 will discuss in more detail existing replica control strategies. Many protocols with strong consistency have been proposed [3, 5, 4, 6, 10, 11, 15, 19, 20, 27]. They guarantee data consistency at any time and provide reasonably good performance. However, these protocols only work well in LANs but not in WANs because they do not consider the long message delay in WANs. My approach will address this issue.

Moreover, many of the existing replica control protocols are not up to date in regard to the current database technology. Although databases allow transactions to execute concurrently and access data simultaneously, transactions may not arbitrarily interfere with each other. Instead, different transaction isolation levels have been defined. They refer to the extend to which concurrent transactions may access the same data items. The strongest transaction isolation level is **serializability** [8]. With serializability, though transactions may execute concurrently, the effect is the same as running the transactions serially one after another. Many databases have concurrency control mechanisms that guarantee serializability. For a replicated system the correctness criteria is **1-copy-serializability**, that is, the entire system behaves as if there were only one logical database providing serializability.

However, recently **Snapshot Isolation (SI)** has emerged as a new isolation level [8]. SI is slightly weaker than serializability and has become quite popular. It requires that transactions read data from a snapshot committed at the time point when they start. Furthermore, if two transactions want to update the same data item at the same time, one will be aborted. SI has been adopted by many database vendors such as Oracle, PostgreSQL, Interbase 4 and Microsoft SQLServer (the upcoming YuKon). Although not being as strong as serializability as defined in the research literature, SI avoids all isolation anomalies as defined by the industrial ANSI standard [7]. Hence, Oracle and PostgreSQL claim that their SI based concurrency control mechanisms actually provide serializability. Although SI has become popular for centralized databases, little has been done on replication with SI. My thesis will look at replica control, providing SI at the global level, what I define as **1-copy-SI**.

Recently, many middleware based approaches for database replication have been proposed (e.g., [5, 4, 21, 11, 20, 29]). The middleware approaches implement replica control algorithms in middleware components which reside between client and databases. Figure 1 shows two typical architectures for middleware approaches. In the centralized architecture (Figure 1.(a)) there is only one middleware component for all databases. In the decentralized one (Figure 1.(b)) there is one middleware component for each database in one site and these middleware components will coordinate with each other. In order to distinguish middleware and database components in one site, hereafter, we call them

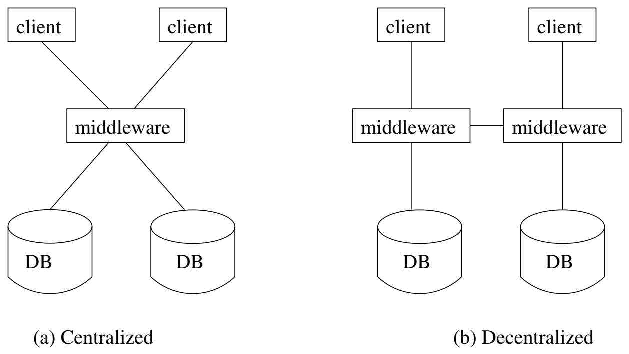

Figure 1. Middleware architectures

middleware replica and *database replica* respectively, unless it is clear in the context. A *site* refers to the combination of its middleware replica (if it exists) and its database replica.

There are many reasons for using middleware approaches. Database systems are huge software systems, access to the source code is limited, and any optimized implementation within the database kernel will lead to a tight integration. Furthermore, middleware based solutions can be developed and maintained independently of database systems, and can potentially be used in heterogeneous environments. It is well known that middleware based replication is simple and flexible. We also follow a middleware approach because of these advantages.

In order to provide global transaction isolation, many of them perform concurrency control at the middleware level. However since a middleware does not have access to the database kernel, it does not know exactly which records are accessed by a transaction, but typically only knows which tables are accessed[1]. Hence, many middleware based approaches [20, 5, 4, 11] restrict the execution of concurrent transactions if they access the same table, although they access different records. My research aims at providing concurrency at the record level.

Apart of this, most existing replication protocols have some restrictions such as read-only transactions must be marked in advance [5, 29], or all operations of a transaction must be known upon submission time [5, 4, 15, 19, 20]. Some protocols require complicated setup or only work in simulation [10, 6].

My thesis aims to propose a replication solution which provides strong consistency under the SI isolation level, and performs reasonably well in WANs. It should overcome the restrictions of current protocols. Furthermore, it should be easy to implement and use.

1.4 Structure of the paper

In the following, Section 2 gives a detail explanations of transactions and transaction isolation levels. Section 3 summarizes the problems of current existing replica control protocols. Section 4 proposes a basic replica control protocol based on SI which works well in LANs. Section 5 extends the protocol to work well in WANs by using some important optimizations. Section 6 shows some experimental results. Section 7 concludes what has been done so far and shows my future schedule.

2 Background

2.1 Transactions and Concurrency Control

Database clients access the database in terms of transactions. A transaction is the basic execution unit in databases. It contains a collection of read and write operations accessing data records within the database. If a transaction is run successfully, we say that the transaction has committed. All data changes performed by committed transactions are permanent. If a transaction's execution is canceled or its results are not made permanent when it is finished, we say that it has aborted. In this case, none of its changes will remain in the database. Transactions may interleave

[1]SQL statements are declaratively indicating the accessed tables while the particular records to be accessed are determined by predicates.

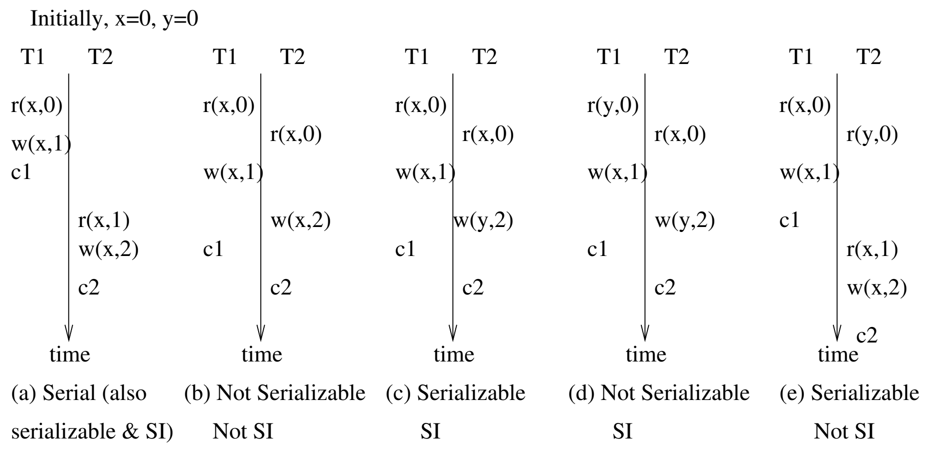

Figure 2. Serializability v.s. Snapshot Isolation

their operations during execution. If two transactions overlap their execution in that neither starts after the other commits/aborts, we say that these two transactions are **concurrent** to each other.

A schedule is a representation of transaction execution over time. Within one database, we assume that execution of operations is serial, that is one after the other. Figure 2 lists 5 different schedules of execution of transactions T_1 and T_2 in a single database. Let's denote reading data item *x* with value a as *r(x,a)*, writing data item *x=b* as *w(x,b)*, committing transaction T_1 as c_1, and aborting T_1 as a_1 hereafter. In Figure 2.(a), T_1 and T_2 execute serially. In the remaining schedules, they execute concurrently.

We say that two operations conflict if they are from two transactions, access the same data item, and at least one operation is a write. If one operation reads and the other writes the same data item, the corresponding two transactions have a **read/write conflict**. If both operations write the same data item, the corresponding two transactions have a **write/write conflict**. There is no conflict if two transactions read the same data item. Two transactions conflict if they have conflicting operations. In all examples of Figure 2, T_1 and T_2 conflict. For instance, in Figure 2.(a), (b) and (e) T_1 and T_2 have read/write and write/write conflicts on data item *x*. In Figure 2.(c), they have only read/write conflict on *x*. In Figure 2.(d), they have read/write conflicts on *x* and *y*. Note that two transactions may conflict but they are not concurrent, as in Figure 2.(a).

Concurrency control is the activity of coordinating the execution of concurrent transactions that potentially interfere with each other. Concurrency control is mainly concerned with *concurrent conflicting* transactions since transactions without conflicts to each other will not interfere, and only concurrent transactions interleave their operations.

2.2 Transaction Isolation Level

2.2.1 Serializability and Snapshot Isolation

In order for transactions not to interfere with each other, we could just execute them one by one serially as in the schedule of Figure 2.(a). We call this a **serial** schedule and denote the execution order as $T_1 \to T_2$. However, concurrent execution allows better resource utilization and increases system throughput. An *isolation level* restricts the order in which in a non-serial schedule the operations of concurrent conflicting transactions may interleave. The strongest isolation level is **serializability** [8]. A schedule which satisfies serializability is called a **serializable** schedule, i.e., it is equivalent to a serial schedule. A serializable schedule orders all conflicting operations in the same way as a corresponding serial schedule that has the same sets of operations. For example, schedules (a),(c) and (e) in Figure 2 are serializable but schedules (b) and (d) are not. The schedule (c) is equivalent to the serial schedule $T_2 \to T_1$ and the schedule (e) to $T_1 \to T_2$. However, schedules (b) and (d) are not equivalent to either $T_1 \to T_2$ or $T_2 \to T_1$.

The typical concurrency control method to provide serializability is **strict two-phase-locking (2PL)**. Locking requires that a transaction obtains a read (or write) lock on each data item before it reads (or writes) that data item. There can be several read locks active on the same data item (allowing concurrent read) but when a write lock is active no other read or write lock may be granted (exclusive write access). In strict 2PL, a transaction releases all locks only

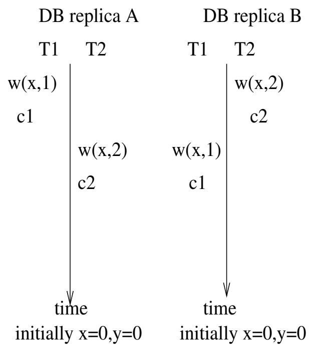

Figure 3. Schedule producing inconsistent data

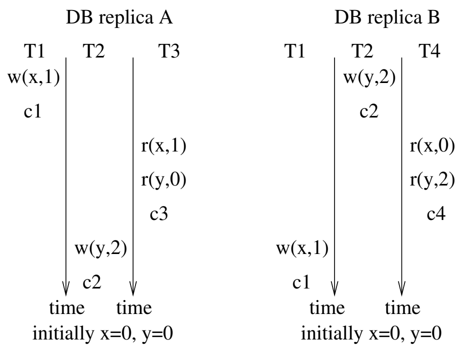

Figure 4. Weak 1-copy-SI schedule

at the time of commit or abort.

In recent years, a slightly weaker isolation level than serializability, **Snapshot Isolation (SI)**, has been proposed [8]. A transaction executing on SI reads data from a snapshot of the committed data as of the time the transaction started. That is, if a transaction T reads data item x it reads the version of x created by a transaction T' which was the last to update x and commit before T started. If two concurrent transactions try to update the same object, one will be aborted. Real systems often detect such write/write conflicts by special forms of locking. With SI, we only need to worry about write/write but not read/write conflicts when determining conflicting transactions, because transactions always read committed data from a committed snapshot. The beauty of SI is that read-only transactions will never request locks, abort or interfere with update transactions. Since in database applications, the number of read operations is usually much higher than that of write operations, the SI approach can save lots of concurrency control overhead compared to standard locking protocol requesting locks for both reads and writes.

For example, schedules (a), (c) and (d) in Figure 2 provide SI while schedules (b) and (e) do not. There are no write/write conflicts for transactions in schedules (c) and (d) so that they are allowed to commit according to SI. However, transactions in schedules (b) and (e) have write/write conflicts and they are concurrent. Thus they should not be able to commit according to SI.

Note that there are schedules that are allowed under SI but they are not serializable (e.g., Figure 2.(d)), and some serializable schedules do not fulfill the SI conditions (e.g., Figure 2.(e)). The SI isolation level is not as strong as serializability since SI does not request read locks during execution. For example, in Figure 2.(d), there are two read/write conflicts which are not detected.

2.2.2 1-copy-serializability and 1-copy-SI

In a replicated database system, it is not enough that each local database system provides serializability or SI in order to guarantee "global correctness".

For example, in Figure 3 there are two schedules over T_1 and T_2 in database replicas A and B respectively. They are serial and hence serializable and SI. However, the data are not consistent in both database replicas after execution, i.e., x==*2* in A and x==*1* in B. The reason is that transactions with write/write conflicts do not commit in the same order in the two database replicas (i.e., $T_1 \rightarrow T_2$ in A and $T_2 \rightarrow T_1$ in B). In order to keep data *consistent* in all database replicas, conflicting write operations must execute in the same order at all database replicas. In order to provide global serializability or SI, even more is needed.

The standard correctness criteria is **1-copy-serializability** [9]. Despite the existence of multiple copies, a data item must appear as one logical copy (*1-copy-equivalent*), and the execution of concurrent transactions is coordinated so that it is equivalent to a serial execution over the logical copy (*serializability*). As long as replica control provides 1-copy-serializability, it is guaranteed that data in all database replicas is consistent. This is because all database replicas execute concurrent transactions in the same serializable order. Many replication solutions aim to provide 1-copy-serializability such as [12, 3, 5, 11, 6, 15, 17, 19, 27, 30].

The popularity of SI in centralized databases motivates me to apply SI on replicated database systems and derive a corresponding global transaction isolation level, i.e., **1-copy-SI** [25]. A global execution schedule provides **1-copy-SI** if the concurrent execution of a set of transactions on the different database replicas is equivalent to executing them on a centralized database providing SI. Intuitively, with 1-copy-SI, all local schedules must provide SI and all transactions with write/write conflicts must be scheduled in the same order. However, although these two conditions are enough to guarantee data consistency, they do not guarantee that the global execution schedule is equivalent to a SI schedule in a centralized database.

For example, in Figure 4 initially data items *x* and *y* are *0* in both database replicas. There are 4 transactions, $T_1 : w_1(x, 1)$, $T_2 : w_2(y, 2)$, T_3 and T_4 reading *x* and *y* in *A* and *B* respectively. Note that we are using ROWAA so that T_1 and T_2 will execute their writes at both database replicas but T_3 executes only in *A* and T_4 in *B*. There are two local SI schedules in *A* and *B* respectively. Since T_1 and T_2 do not have write/write conflict, they can execute and commit in any order according to SI. In the figure, T_1 executes and commits before T_2 in *A* and after T_2 in *B*. After execution, *x* is set to 1 and *y* is 2 in both database replicas. Hence, data is consistent.

However, in database replica *A*, T_3 reads *x* and *y* from a snapshot after T_1 commits and before T_2 commits, while in *B*, T_4 reads *x* and *y* from a snapshot before T_1 commits and after T_2 commits. This can never happen in a centralized database SI schedule, in which only one of these two snapshots can hold. The replicated database does not behave as if there is only one database.

Thus, I define two levels of 1-copy-SI, which I will only introduce informally here. [25] presents a formal definition. Both levels of 1-copy-SI satisfy the two conditions mentioned above, i.e. all local schedules must be SI and all transactions with write/write conflicts must be scheduled in the same order at all database replicas. This will guarantee data consistency in all database replicas. **Strong 1-copy-SI** additionally requires to order the transactions in such a way that any two snapshots read by two transactions in two database replicas can both occur in an execution in one centralized database. This can be achieved if all update transactions (no matter if they conflict) are executed and committed in the same order at all database replicas, because this will create the same series of snapshots at all database replicas. The **weak 1-copy-SI** allows a transaction to read a local snapshot not existing in other database replicas. The global schedule shown in Figure 4 provides weak 1-copy-SI. To provide strong 1-copy-SI, T_1 and T_2 should commit in the same order at both database replicas.

Actually, little has been done so far on replica control with SI. [29, 32] use SI in a replicated database but they do not define a global transaction isolation level such as 1-copy-SI. [16] defines global SI. Although the definitions and the reasoning is quite different, their correctness criteria allows exactly the same schedules as strong 1-copy-SI.

3 Related Work

3.1 Categories of Database Replication

There exist many different solutions. Most recent solutions all use a ROWAA approach. The seminal paper of Gray et al. [18] categorizes these replication strategies according to two parameters determining update location and propagation time respectively.

In regard to update location, a **primary** approach only allows data to be updated in one primary site. Thus, in a primary approach, if a client submits updates to a site other than the primary site, the updates will be either refused or redirected to the primary site for execution. Different data items might have different primary sites. In this case, however, transactions that want to update data items with different primary sites are disallowed. In contrast, in an **update everywhere** approach the updates are accepted and executed at the local site to which the updates are submitted. In general, update everywhere approaches are more flexible than primary approaches.

In both approaches, updates must be propagated to other sites. A **lazy** approach allows update transactions to commit before propagating the update to other sites. In contrast, using an **eager** approach update propagation must happen before the transaction commits, and thus within the transaction boundary. An eager approach provides strong consistency because a transaction will not commit until it is certain that it will be able to commit in all other available sites. However, it delays transaction execution. A lazy approach provides only weak consistency because of early commit, but transaction response time in a lazy approach is lower than that in an eager approach.

3.2 Primary approach

In a primary approach update transactions are only allowed to execute at the primary site which performs traditional concurrency control to isolate conflicting transactions. As long as other sites apply and commit updates in the same

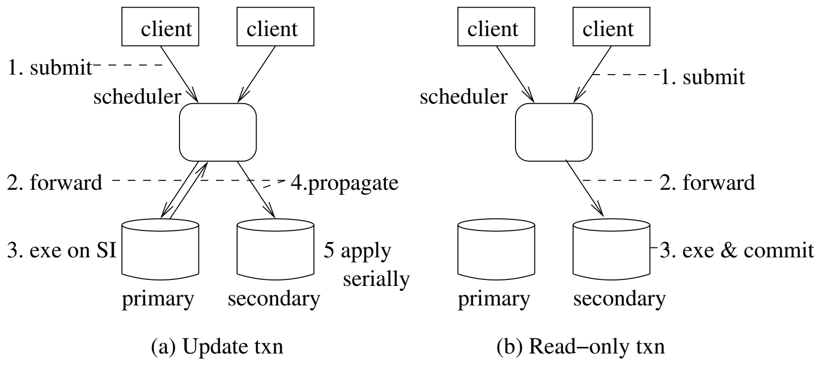

Figure 5. Lazy primary protocol (e.g., Ganymed [29])

order as at the primary site, data will be consistent, no matter if the changes of transactions are propagated lazily (i.e., after commit) or eagerly (i.e., before commit). Since eager propagation delays transaction execution, most primary approaches are lazy [29, 13, 26, 12, 10, 6].

Figure 5 shows the execution model of a typical lazy primary protocol used in the Ganymed [29] system. There is one centralized middleware (i.e., scheduler), one primary database replica, and several secondary database replicas in the system. Transactions can only be submitted to the scheduler. Ganymed assumes that the underlying database systems provide SI (both primary and secondaries). Figure 5.(a) shows how update transactions are handled. The transactions are forwarded to the primary database replica which will execute the transactions on SI, and then the scheduler propagates the changes to all secondary database replicas after commit. Secondary database replicas will apply the changes in the same order as the corresponding transactions are committed at the primary, no matter if they conflict or not. If a transaction is read-only, the scheduler will forward it to a secondary database replica with least load for execution, as shown in Figure 5.(b).

The system provides strong 1-copy-SI. However, there are several shortcomings. First, all requests go through the single middleware resulting in WAN communication between the middleware and clients if the clients are remote, and between the middleware and the database replicas if the database replicas are distributed across the WAN. This communication can happen for each read and write operation. Hence the approach only works well in a LAN. Secondly, secondary database replicas apply all updates serially, not allowing for any parallelism. Third, both middleware and primary database replica are a single point of failure. They are also potential bottlenecks.

Other lazy primary protocols work slightly differently. [13, 26] do not have a middleware but allow clients to submit requests as transactions to local database replicas. If a transaction is read-only, it will execute locally. Otherwise, the local database replicas will forward the transaction to the primary database replica which will execute the transaction (using local concurrency control to provide serializability) and then propagate the changes to secondary database replicas after commit. The secondary database replicas will work as in Ganymed. [13, 26] provide 1-copy-serializability.

All these approaches have the requirement to know if a transaction is read-only or not at the time of submission. It decreases flexibility of applications considerably. Moreover, it may not work well in WANs for transactions with multiple operations if update transactions must be forwarded to the primary site operation by operation. If clients are not local to the primary site, the more operations there are in an update transaction, the more communication cost is included in the response time. It is desirable to send only a constant number of messages for each transaction, as my protocol will do.

[6] use a centralized replication graph for conflict resolution. Conflicting information must be delivered to a centralized site for building the replication graph and the decision is sent back. The communication overhead is large in WANs. [12, 10] allow multiple primaries (i.e., assigning different primaries to different data) and put certain restrictions at the data placement in different sites. This restricts the flexibility of these protocols. Furthermore, if an update transaction is submitted to a secondary site, it is simply aborted or rejected. Hence, the clients must know the locations of the primaries.

Finally, there is a serious problem for lazy primary approaches in failure cases. In case that the primary site crashes before propagating a change but after commit of a transaction, the transaction will not be applied in the secondary sites and data will be inconsistent.

3.3 Update everywhere approach

Update everywhere approaches do not require update transactions to be submitted or forwarded to a primary site for execution. However, it is more difficult to keep data consistent than in primary approaches. This is because in a primary approach conflicts between update transactions are detected in a single site (i.e., the primary) while in an update everywhere approach conflicting update transactions can run concurrently on different sites. Thus, an update everywhere approach requires additional coordination between different sites for concurrency control purposes, which is not trivial. Gray et al. [18] claims that update everywhere approaches may lead to high deadlock and abort rates if many transactions run concurrently on different sites.

Update everywhere approaches can be combined with lazy and eager propagation. Figure 3 is using lazy update everywhere approach. T_1 commits in site A and then propagates the change to site B. T_2 commits in site B and then propagates the change to site A. Thus, data in sites A and B will not be consistent. The inconsistency has to be detected and reconciled, leading basically to a rollback of an already committed transaction. This example shows that lazy update everywhere may lead to a serious problem, reconciliation. Although provided by many commercial systems, it is recommended to use lazy update everywhere only if there are extremely low conflict rates.

The challenge of eager update everywhere approaches is to provide replica control in order to guarantee global transaction isolation. Traditional eager update everywhere protocols use distributed 2PL. [18] has shown analytically and [23] has shown empirically that such an approach does not scale. Recent proposals address the problems of eager update everywhere with two different approaches, a middleware based scheduler or the use of powerful communication mechanisms.

3.3.1 Update everywhere with centralized scheduler

[4] proposes a conflict aware replica control protocol, which is a typical example of an update everywhere protocol using a middleware based scheduler. There is a single scheduler in the system. It is required that all tables to be accessed in a transaction must be indicated at the start of the transaction. Upon start of a transaction, the scheduler assigns a unique version number to the transaction. Then, the scheduler requires locks for the tables to be accessed on each database replica in the order of version number. Thus, all conflicting operations are enforced to execute in an identical order in all database replicas and 1-copy-serializability is obtained. After being successfully scheduled, the client can submit one by one several read and write operations belonging to this transaction. The scheduler forwards each write to all database replicas and returns to the client once one database replica has executed the write. A read is sent to a single database replica. This replica must have executed all previous update operations of this transaction. Therefore, the approach is called conflict aware scheduling.

The distributed versioning protocol [5] is similar to the conflict aware protocol [4] except that [5] uses a distributed version number per table instead of a lock. C-JDBC [11] also works similarly. It does not require the knowledge of all operations of a transaction in advance. The scheduler implements a table-based lock manager and uses strict 2PL. The scheduler waits until it receives responses from all database replicas involved in the operation (one for reads, all for writes) before it returns a response to the client.

Since the centralized middleware represents a similar architecture to the one of Ganymed, all these approaches are not suitable for WANs.

3.3.2 Update everywhere with group communication system

In recent years many replica control protocols have been proposed [20, 23, 30, 3, 2, 24, 27], taking advantage of multicast primitives provided by Group Communication Systems (GCS) [14]. Replicas build a group and multicast messages to all group members. The semantics of the typical multicast primitives provided by GCS can be categorized by two parameters. The *ordering* semantics that are interesting in the context of database replication are *unordered*, *FIFO* (messages of one sender are received in sent order by all members), and *total* (for each two members receiving m and m', both receive them in the same order). The *reliability* semantics are *unreliable* (no guarantee that a message will be received at all members), *reliable* (whenever a member receives a message and does not fail for sufficiently

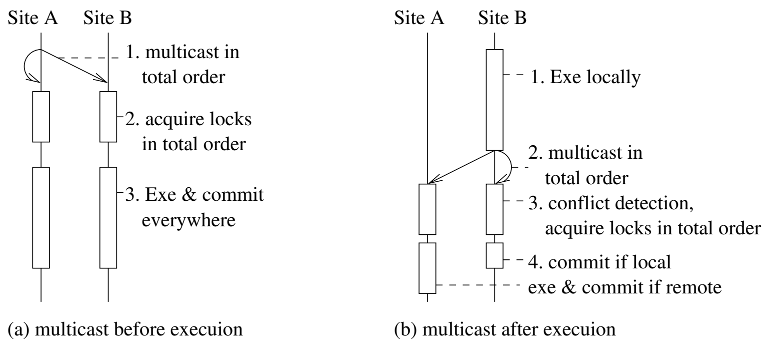
Site A
Site B
1. multicast in total order
2. acquire locks in total order
3. Exe & commit everywhere
(a) multicast before execuion
Site A
Site B
1. Exe locally
2. multicast in total order
3. conflict detection, acquire locks in total order
4. commit if local exe & commit if remote
(b) multicast after execuion

Figure 6. Update everywhere with GCS approach

long time, then all other group members will receive the message unless they fail), and *uniform reliable* (whenever a member *p* receives a message, all other members will receive the message unless they fail, even if *p* fails immediately after the reception).

It has been shown in the previous sections that sending a message for each operation prevents protocols from working well in WANs. If we want to avoid this, we can either multicast the whole transaction before execution (see Figure 6.(a)) or multicast the changes performed by a transaction after execution (see Figure 6.(b)).

[20, 3, 2] work as Figure 6.(a). They require that all operations of a transaction must be known at start time. Read-only transactions are executed locally. A transaction request for an update transaction is multicast in *uniform reliable* and *total* order. Since *total* order guarantees transaction requests are delivered in the same order in all sites, the commit order of transactions can be the same without further coordination among sites. [3, 2] apply the transactions at each site serially. In [20] transactions will request all necessary locks on tables in the same total order in all sites upon receiving the transaction request. [20] defines different primary sites for different transactions. A transaction will be executed only in its primary after all its locks are granted, then its changes will be multicast to other sites. Other sites will apply these changes again in the correct lock order. [20, 3, 2] guarantee 1-copy-serializability.

Postgres-R [32] works as Figure 6.(b). All databases provide SI. A transaction executes first locally. After execution, its writeset is multicast in total order to all sites for validation. If it conflicts with a concurrent transaction whose writeset was delivered earlier, it will be aborted. Otherwise, its writeset will be applied serially at remote sites according to the delivery order which is identical in all sites. This sacrifices the concurrency of the system although it does guarantee strong 1-copy-SI as defined in Section 2.2.2. Postgres-R is integrated into the kernel of PostgreSQL.

GlobData [30] also performs execution before multicast, as Postgres-R. Each data item has a version number which will be increased upon a change being committed. Conflict detection is based on the version number of data accessed by transactions. In order to detect both read/write and write/write conflicts both the version numbers of data items read and the changes are multicast. Conflict detection is done upon receiving such message in total order, possibly leading to aborts. GlobData uses a GCS designed specially for WANs. The Database State Machine [28] works similarly to GlobData. However, unlike GlobData, after local execution, only identifiers of the data items read and written by update transactions and the changed values are multicast to other sites, and read-only transactions just commit locally and their read sets are not multicast. All sites will apply and commit the changes serially if no conflicts are detected. Both GlobData and Database State Machine provide 1-copy-serializability.

All these approaches have potential for WANs because only one message (except two messages in [20]) is sent per transaction. However, the approach shown in Figure 6.(a) requires to distinguish between read-only and update transactions and must know all operations in advance. Hence, it is inflexible. The approach shown in Figure 6.(b) seems to have the great potential for low communication overhead and flexibility. However, GlobData and Database State Machine require to send read sets to provide 1-copy-serializability, what we believe is infeasible and also not needed for 1-copy-SI. Postgres-R is the closest to what we want. However it is implemented within the kernel of the database, increasing the complexity. Hence it will be difficult to implement any further optimizations that might be needed for WANs. The question is whether we can design a middleware based approach with at least the same functionality and optimal for WANs. Moreover, Postgres-R applies changes serially at the remote sites. This sacrifices the concurrency of execution.

3.4 Contribution to be achieved

In summary, the objective of my thesis is to develop a replication solution working well in WANs despite long communication delay. It should provide strong consistency by guaranteeing 1-copy-SI which is based on SI, a popular centralized transaction isolation level. The solution will overcome the restrictions exhibited by current approaches, e.g., read-only transaction do not need to be marked in advance, neither do all operations of a transaction be known in advance. The solution will be middleware based for simple implementation and for adaptability to all kinds of databases. In contrast to most existing middleware based approaches, it will provide concurrency control on a record basis and not a table basis.

4 Basic SI replication protocol for LANs

In this section, I will first present a protocol working in LANs. The protocol guarantees strong 1-copy-SI. However, there is a deadlock problem when it works with real databases. I will show how the problem is solved. This work will be published in [25].

4.1 Basic idea

Figure 1.(a) shows the architecture of the system. There is a centralized middleware instance for all database replicas, each of which provides SI. A client will submit its operations (e.g., begin, read, write, commit/abort) one by one to the centralized middleware.

The execution flow of a transaction is as following. At start of a transaction, the middleware chooses one database replica to execute the transaction (called local database replica). The client submits operation by operation to the middleware which forwards it to the local database replica to execute. Upon receiving the commit of a transaction, the middleware will retrieve the writeset of the transaction from the local database replica. The writeset contains the new values and the identifiers of the data items written by the transaction. If there is no writeset (i.e., the transaction is read-only), the transaction will commit locally. Otherwise, the middleware validates the transaction against concurrent transactions that have already committed. If the validation succeeds (i.e., there is no conflict), the writeset of the transaction will be sent to all database replicas in FIFO order. Upon delivering writesets, remote database replicas will apply the writeset and all database replicas will commit the writesets in FIFO order.

This protocol does provide strong 1-copy-SI because all database replicas agree to commit the transactions in the identical order, which is determined uniquely by the centralized middleware. The challenge of this protocol resides in the validation, i.e., how to detect if two transactions are concurrent and conflicting. Note that the protocol is based on SI so that we are not worried about read/write conflicts. Hereafter, we call two transactions conflicting transactions if and only if they have write/write conflicts.

To detect if two transactions conflict, we can simply look at their writesets. Since a writeset contains identifiers for data items it modified, if two writesets have overlapping identifiers, the transactions have write/write conflicts.

To detect if two transactions are concurrent is more tricky. Recall that it is defined in Section 2.1 that two transactions are concurrent if one starts before the other commits/aborts and vice versa. Since the centralized middleware knows when transactions start and commit, it can detect concurrent transactions. It will be explained in detail in the next section.

4.2 Basic protocol

The basic protocol is based on the assumption that write/write conflicts of transactions will be detected at the commit time and thus, writesets of transactions can be retrieved after their execution. The detailed protocol running in the centralized middleware is shown in Figure 7. The middleware maintains a queue (ws_list) for all the transactions which have been validated but not committed yet. The writesets of these transactions will be sent to all database replicas for being executed and committed in identical order. However, different database replicas may apply and commit the writesets at different speed. The middleware has to keep track of the latest status of each database replica by

```
1. Initialization:
   (a) $next\_tid := 1$
   (b) $ws\_list := \{\}$
   (c) $\forall$ DB replica $R^k$ : $tocommit\_queue_k := \{\}$
   (d) $\forall$ DB replica $R^k$ : $lastcommitted\_tid\_k := 0$
   (e) $wsmutex, \forall databasereplica R^k$ : $dbmutex\_k$
2. Upon receiving an operation $Op_i$ of $T_i$
   (a) if $Op_i$ is begin, then
       i. choose $R^k$ at which $T_i$ will be local
       ii. obtain $dbmutex\_k$
       iii. $T_i.cert := lastcommitted\_tid\_k$
       iv. begin $T_i^k$ at $R^k$
       v. release $dbmutex\_k$
       vi. return to client
   (b) else if $Op_i$ is read or write, then
       i. execute in local $R^k$ and return to client
   (c) else (commit)
       i. $T_i.WS := getwriteset(T_i^k)$ from local $R^k$
       ii. if $T_i.WS = \emptyset$, then
           • commit and return
       iii. end if
       iv. obtain $wsmutex$
       v. 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 $ws\_list$
           • $\forall R^k$: append $T_i$ to $tocommit\_queue\_k$
           • release $wsmutex$
       vi. else
           • release $wsmutex$
           • abort $T_i^k$ at $R^k$
       vii. end if
   (d) end if
3. Upon $T_i$ is first in $tocommit\_queue\_k$
   (a) if $T_i$ is remote, then
       • begin $T_i^k$ at $R^k$
       • apply $writeset_i$ to $R^k$
   (b) end if
   (c) obtain $dbmutex\_k$
   (d) commit at $R^k$
   (e) $lastcommitted\_tid\_k++$
   (f) release $dbmutex\_k$
   (g) if local, return to client
   (h) remove $T_i$ from $tocommit\_queue\_k$
```

Figure 7. Simple SI replication protocol for LANs

two data structures, a queue (*tocommit_queue_k*) which contains writesets to be executed and committed at database replica R^k and an identifier ($lastcommitted\_tid\_k$) which is the last committed transaction *tid* at R^k.

When a transaction T_i begins, it will be assigned a local database replica and a $T_i.cert$ which is the last committed transaction id of this replica (step 2a). We denote T_i^k as the copy of T_i executing in R^k. The read and write operations of T_i will be executed in the local database replica (step 2b). When a commit operation arrives (step 2c), the writeset of the transaction will be retrieved by the middleware from the local database replica (step 2(c)i). If the writeset is empty, this is a read-only transaction so that it can commit in the local database replica and the client receives a response (step 2(c)ii). If T_i is an update transaction, a validation is performed against all transactions which have passed their validations (step 2(c)v).

The validation will check if there is a concurrent conflicting transaction that passed validation already. To determine if an *uncommitted* transaction T_i is concurrent to a transaction T_j that already validated, we just need to compare $T_i.cert$ and $T_j.tid$. If $T_i.cert$ is larger than or equal to $T_j.tid$, they are not concurrent. Otherwise, we know T_i has started before T_j validated and thus T_i and T_j must be concurrent. The validations of transactions must be serial. Otherwise, a transaction might not validate against a concurrent conflicting transaction validating at the same time.

If there is a concurrent conflicting transaction, T_i will be aborted (step 2(c)vi). Otherwise, T_i succeeds the validation and is allowed to commit. A unique identifier *tid* is assigned to T_i, and T_i is put into ws_list and *tocommit_queue_k* for all database replicas. The writeset of T_i will be applied in remote database replicas serially (step 3a). In the local database replica, it only needs to commit. All database replicas will increase their $lastcommitted\_tid\_k$ at the time of committing T_i (step 3e). Note that committing a transaction at a database replica (step 3d and 3e) and starting a transaction (step 2(a)iv and 2(a)iii) are mutually exclusive to each other.

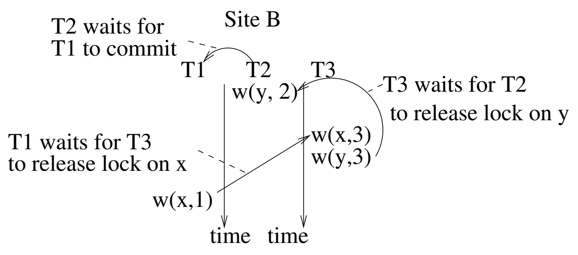
Site B
T2 waits for T1 to commit
T1
T2
T3
w(y, 2)
T3 waits for T2 to release lock on y
T1 waits for T3 to release lock on x
w(x,3)
w(y,3)
w(x,1)
time
time

Figure 8. Deadlock on strong 1-copy-SI

The protocol detects concurrent conflicting transactions in the centralized middleware, and only allows the transaction that validated first to commit. All writesets are applied serially and all transactions are committed serially in the same order at all database replicas. Hence, data consistency and strong 1-copy-SI is guaranteed. Please see [25] for a proof.

4.3 Deadlock problem when working with real databases

The previous section shows in detail how the protocol works to provide data consistency and strong 1-copy-SI. It assumes that write/write conflicts will be detected at the time of commit so that each transaction can perform write operations without being blocked. However, most existing databases such as Oracle and PostgreSQL will detect write/write conflicts during execution by means of strict 2PL. In these real databases, when a transaction T_2 tries to write a data item, it will be blocked from execution if there is already another transaction T_1 writing the same data item (i.e., holding the lock on this data item). If T_1 commits, T_2 will be aborted. If T_1 aborts, the lock will be released and T_2 will be able to continue. If there is a local deadlock, one of the transactions will be aborted. Using such a database as replica, the protocol of the previous section might run into a deadlock spanning across the database and the middleware. Let's look at an example.

In Figure 8, there are three transactions, $T_1 : w(x, 1); T_2 : w(y, 2); T_3 : w(x, 3), w(y, 3)$. T_1 executes locally in database replica *A*, and T_2 and T_3 in *B*. T_1 and T_2 execute completely. T_3 performs *w(x,3)* and waits on T_2 to release its lock on *y*. Now in the middleware T_1 validates first, then T_2, and they have to commit in this order at database replicas *A* and *B*. T_1's writeset is now applied at database replica *B*. T_1 is blocked by T_3 because T_3 is holding a lock on *x*. As a result, there is a deadlock spanning across the database and the middleware in that T_2 waits for T_1's commit in the middleware, T_1 is waiting for T_3 on *x*, and T_3 is waiting for T_2 on *y* in the database.

4.3.1 Solution to deadlock

The deadlock described in the previous example is due to the fact that T_2 must commit after T_1 even though they do not have write/write conflict. In order to avoid the deadlock, we have to allow concurrent commit of non-conflicting transactions, i.e., T_1 and T_2 can commit concurrently in the previous example. Since T_2 and T_1 do not have write/write conflicts, no matter in which order they commit, the data will be consistent at both database replicas. However, if T_1 and T_2 happen to commit in different order at two database replicas, the global schedule will provide weak 1-copy-SI, since there may be two transactions reading two contradicting snapshots at two database replicas respectively, as has been discussed in Section 2.2.2.

If we still want to keep strong 1-copy-SI guarantee, we have to delay the start of local transactions. The basic idea is as following. We allow concurrent commit of non-conflicting transactions at each database replica in order to break deadlocks. Thus, transactions may commit in different order at different database replicas and the snapshots provided by different database replicas may be different. However, we only allow local transactions to read data from certain snapshots that can be produced by a SI schedule in one centralized database. Note the problem is that, if a transaction T_i commits before a transaction T_j who validated before T_i, it may create a snapshot not existing at other database replicas. We call this a **hole** in the commit order. We disallow any local transactions to read this snapshot by disallowing start of transactions if there are holes. The start of these transactions will be delayed until there are no holes, i.e., there are no uncommitted transactions who were validated before a committed transaction.

For example, in Figure 8, T_1 validates before T_2. We allow concurrent commit of non-conflicting transactions so that T_1 may commit before T_2 in database replica *A* while T_1 commits after T_2 in *B*. *B* commits T_1 and T_2 in an order different from the validation order. If a local transaction T_4 wants to start after T_2 commits but before T_1 commits, its start will be delayed. Otherwise, T_4 could read data from a snapshot not existing in A. T_4 will be allowed to start after T_1 commits. Thus, it will read data from a snapshot after both T_1 and T_2 commit. Hence, all transactions in all database replicas will see snapshots that conform to the validation order. Thus, strong 1-copy-SI is guaranteed.

4.3.2 Concurrent execution of writesets and liveness problem

Although the concurrent commit of non-conflicting transactions solves the deadlock problem, the concurrent execution of remote writesets introduces an additional problem, namely a liveness problem. We might delay the start of a local transaction indefinitely if there are always holes (i.e., livelock). A solution to this liveness problem is to disallow new holes when there are already local transactions waiting to start. Only transactions that remove holes or do not create new holes are allowed to commit. Of course, this has to be coordinated in such a way as to guarantee that no new deadlocks can arise. [25] discusses this issue in more detail.

We would like to note at this point that concurrent execution of writesets increases concurrency and hence decreases response time. All other similar protocols (e.g., Postgres-R [32], GlobData [30] and Database State Machine [28]) just apply writesets serially.

5 SI replication protocol for WANs

In the previous section, I have discussed a basic replication protocol working well in LANs. In this section, I will present some optimizations which make the protocol work well in WANs.

5.1 Optimization 1: decentralized middleware with GCS

The basic protocol in Figure 7 does not work well in WANs because of its centralized middleware for exactly the same reasons as other middleware based approaches (e.g., Ganymed [29] or conflict aware scheduler [4]). Therefore, I propose an approach with decentralized middleware replicas based on the basic protocol in Figure 7.

The architecture is as in Figure 1.(b). Each site has a middleware replica connecting to a local database replica. Clients will submit their requests to their corresponding local middleware replicas. As in the basic protocol (Figure 7), the requests will be executed optimistically in the local database replicas. Upon commit, the writeset of a transaction will be retrieved and the middleware replicas perform validation. This validation has to be coordinated in order to guarantee that the transaction commits or aborts at all database replicas. In Section 3.3.2, I have introduced total order multicast which guarantees that all messages will be delivered to all group members in the same order. We can take advantage of this characteristics of GCS to guarantee that all middleware replicas take the same validation decisions.

In each middleware, after the writeset of a local transaction is retrieved, it will be multicast in total order to all middleware replicas including the sender. Because of total order delivery semantics, writesets will be delivered to all middleware replicas in the same order. Let's perform validation according to the delivery order one by one. Thus, in each middleware replica, each transaction will be validated against those transactions delivered before it. If two transactions are concurrent and conflicting, and the first to be delivered passed validation, then the second has to fail validation. For that, we have to have a mechanism to determine at each middleware replica whether two transactions, even if executed at two different sites, are concurrent or not. If we are able to do this and since all transactions are validated in the same order in all middleware replicas, it is guaranteed that all middleware replicas will take the same decision to commit and abort transactions depending on if their validations succeed or fail.

5.1.1 Example

Let's look at an example here. In Figure 9, T_1 and T_2 both modify a data item x so they have write/write conflict. They are concurrent to each other because they start simultaneously in sites A and B respectively. The writesets of T_1 and T_2 will be multicast in total order after execution. Assume the writeset of T_1 is delivered before that of T_2. At each site's middleware replica, T_1 performs its validation against no transaction. So T_1 will pass the validation. In A's database replica, T_1 just needs to commit. In B's database replica, the writeset of T_1 will be executed and committed. Now assume the writeset of T_2 is delivered. T_2 will fail its validation at both middleware replicas since the concurrent

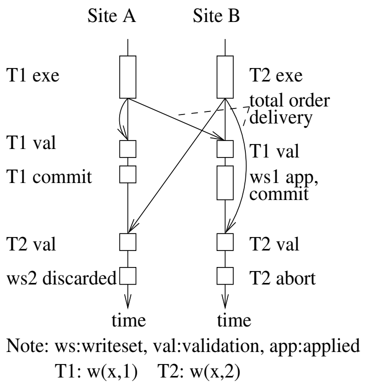

Figure 9. Decentralized approach with GCS

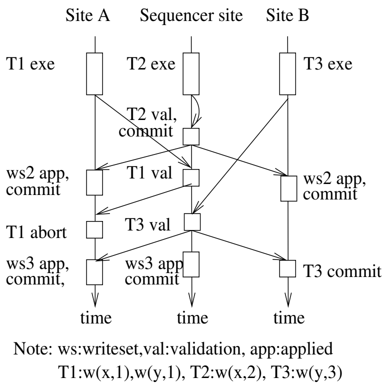

Figure 10. Decentralized approach without GCS

conflicting transaction T_1 has been validated before. So in B's database replica, where T_2 has executed, T_2 will abort. In A's middleware replica, the writeset of T_2 will be discarded. The example shows that both sites reach the same decision to commit T_1 but abort T_2.

The mechanism to determine that T_1 and T_2 are concurrent is similar to the centralized case. Transactions receive a transaction identifier if they pass validations. At start time the transaction receives a *cert* timestamp that is equal to the validation timestamp of the last transaction that committed at the local database replica. This *cert* timestamp is piggybacked with the writeset and helps identifying concurrent transactions.

5.1.2 Fail-over and recovery

[25] discusses a mechanism that was developed at Universidad Politecnica de Madrid and I am planning to adopt this approach. A client is connected to a middleware replica. We need to consider the failures of the middleware replica and the underlying database replica. For simplicity, we assume that the client can be notified of both failures. When the middleware replica or underlying database replica crashes, the client will be redirected to another middleware replica. If the crash happened in the middle of execution of a transaction, the client will be informed about the abort of the transaction. If the client had already submitted the commit request the transaction might have committed before the crash or aborted. The client will transparently receive the correct outcome. This is implemented in a special JDBC driver which handles the connection between the client and the middleware replica. Part of this driver code runs at the client, other parts run on the middleware replica. For details please refer to [25] and the special JDBC driver developed for the ADAPT project [1].

Currently the recovery procedure and joining of new sites are not considered yet. They will be done in the future.

5.2 Optimization 2: disregard GCS, use sequencer for validation

5.2.1 Basic idea

Section 5.1 presents a decentralized approach by using total order delivery provided by GCS. However, GCS is a complicated software. The total order delivery provided by GCS is an expensive process due to coordination overhead between different members. Some of this coordination overhead is unnecessary for replica control purposes. This motivates me to not use GCS for WAN but instead integrate the features of the GCS that the replica control uses into the replica control system itself.

[14] gives a comprehensive survey of different total order algorithms. I also did extensive experiments on testing the performance of different total order algorithms in WANs. Due to space limitations, I will not show my experiments here. My experiments discover that a sequencer based total order algorithm is generally more stable, faster and more scalable than the rest of algorithms in WANs. In the typical sequencer based algorithm, a message is sent to a sequencer, which attaches a sequence number and then multicasts it to all members. Thus, all sites can deliver messages in the same order according to the sequences attached to the messages. The simplicity and good performance

of sequencer based total order algorithm motivates me to use a middleware replica as a sequencer in order to achieve total order validation. The basic idea is as following.

There is a sequencer site in the system. All middleware replicas will send their writesets to the sequencer middleware replica for validation. If the validation fails, the sequencer will send back an abort decision to the transaction's local middleware replica. Otherwise, it will forward the writeset and the commit decision to all middleware replicas in FIFO order (e.g, through TCP/IP socket). Since there is only one sequencer, the decision will be unique in all sites.

With this, we have actually a combination between a centralized and a distributed middleware. Clients are connected to a local middleware replica in order to keep communication between client and middleware fast. Additionally, a dedicated middleware replica is responsible for concurrency control. Communication overhead to this dedicated middleware replica is constant per transaction and does not depend on the number of operations in the transaction.

5.2.2 Example

Figure 10 shows an example using a sequencer instead of GCS. T_1, T_2 and T_3 start simultaneously in 3 sites respectively. They are concurrent to each other because they do not see each other's commits. T_1 and T_2 have write/write conflict on a data item *x*. T_1 and T_3 have write/write conflict on a data item *y*. The writesets of these three transactions will be sent to the sequencer's middleware replica after their executions. Assume the arriving order of writesets at the sequencer's middleware replica is T_2, T_1 and T_3. T_2 will perform validation first and succeed because there is no concurrent conflicting transaction validated so far. T_2 will be able to commit and send its writeset to other sites' middleware replicas in FIFO order. Their corresponding database replicas will execute and commit the writeset without validation. T_1 will perform its validation after T_2 in the sequencer's middleware replica. The validation will fail since concurrent conflicting transaction T_2 is allowed to commit. So the writeset of T_1 will be discarded in the sequencer's middleware replica and an abort decision will be sent back to *A*, T_1's local site. Upon receiving the abort message, *A*'s database replica will abort T_1. Then comes the validation of T_3 in the sequencer's middleware replica. Although there is a concurrent transaction T_2 which has passed the validation, T_2 does not conflict with T_3. Hence, T_3 passes its validation, its writeset will be executed, and T_3 commits in the sequencer. At the same time, its writeset will be sent to *A* and *B*. *A*'s database replica will execute and commit the writeset while *B*'s database replica only needs to commit T_3. The example shows that all three sites reach the same decision to commit $T_2 \rightarrow T_3$ and abort T_1.

5.2.3 Further optimization, eliminating abort decision message

We can further optimize the validation by eliminating the need for the sequencer to send back an abort decision for transaction T_i to its local site. This is because in case a T_i fails validation at the sequencer there must have been a concurrent conflicting transaction T_j that passed validation and was sent to all sites. Upon receiving the writeset of T_j the local site of T_i can determine the conflict and abort T_i without the explicit abort decision from the sequencer. For example, in Figure 10 the sequencer does not need to send back the abort decision message of T_1. Upon *A* receiving the writeset of T_2, it is able to detect that T_1 should be aborted because of conflict with T_2.

5.2.4 Fail-over and recovery

Again, as in the fail-over procedure in Section 5.1.2, we use a special JDBC driver that handles the connection between client and middleware replicas. If the special JDBC driver running on the client side detects a failure of the middleware replica during the execution of a transaction *T*, it will simply inform the client about an abort of *T* since the commit request has not been sent. If the failure is detected after the commit request has been submitted and no response is returned yet, there might be two cases:

1. **If the client is connected to a middleware replica that is not sequencer**, the driver will be redirected to the sequencer middleware replica and ask for the decision on *T*.

   (a) *If the sequencer middleware replica also fails*, refer to Case 2.

   (b) *Else if the sequencer middleware replica has not yet received* T*'s writeset*, the writeset was lost during transmission. No commit decision is made and none of the other middleware replicas has seen *T*. Hence, an abort message is returned to the client.

   (c) *Else (if the sequencer middleware replica had received the writeset)*, it will finally make a commit/abort decision on *T* and broadcast the decision to all other middleware replicas. So the driver can just respond to the client with the decision.

2. **If the client is connected to the sequencer middleware replica**, the rest of non-sequencer middleware replicas have to vote and select one as new sequencer middleware replica. The situation is similar to the coordinator crash in a 2-phase-commit protocol. From the point of view of all non-failed middleware replicas, they might or might not receive an explicit decision on *T* made by the original sequencer middleware replica. Hence, the new sequencer will have to collect all writesets (including those sent by the original sequencer and those being sent but no decision was yet returned) at all middleware replicas in order to make a consistent decision.
   (a) *If all middleware replicas have received the writeset and the commit decision from the original sequencer*, the client will receive the corresponding decision.
   (b) *Else if only some middleware replicas receive the writeset and the commit decision*, the client also receives the corresponding decision. The new sequencer middleware replica will forward the writeset and the decision to all middleware replicas which missed this information.
   (c) *Else (if none of the middleware replicas receive the writeset and the decision)*, it is possible that a). the original sequencer middleware replica has passed the validation and committed *T*, but failed before sending the writeset and the commit decision, or b). the original sequencer database replica aborted *T*. There are no means to determine what happened. We have to resort to the support of hardware. Nowadays Uninterruptible Power Supplies (UPS) is used extensively for continually supplying power in case of a unexpected power-off. Once UPS is activated by a poweroff, we let another site to take over the sequencer site gracefully, without outstanding decisions.

Additionally, it might happen that the sequencer middleware replica crashes after receiving writesets sent by a non-sequencer middleware replica which does not crash. The non-sequencer middleware replica will detect the failure of the sequencer and ask the new sequencer for decision. This case is handled in the same way as Case 2.

5.3 Optimization 3: double validation

In the basic protocol shown in Figure 7, a transaction is validated only once, i.e., after its writeset has been delivered. For example, in Figure 11.(a) after the delivery of its writeset, transaction T_a performs its validation against two concurrent transactions T_1 and T_2. Let's look closely at Figure 11.(a). T_1 is delivered before T_a multicasts its writeset while T_2 is delivered after the multicast. It could be better if T_a performs its validation against T_1 right after execution and before multicast. Thus, if T_1 conflicts with T_a, T_a will abort itself earlier and there is no need to multicast its writeset. Note that the second validation after the delivery of the writeset is still necessary because there may be some concurrent transactions whose writeset were not yet delivered at the time of the first validation but arrived before T_a's writeset (e.g., T_2 in Figure 11). The corresponding example of double validation is shown in Figure 11.(b) and (c). In (b), T_a conflicts with T_1 and is aborted early. In (c), T_a conflicts only with T_2 and hence, its writeset is multicast.

The double validation optimization is applicable both when using GCS and the sequencer approach. Before sending the writeset of a transaction to the sequencer middleware replica, each middleware replica can perform locally the first validation of the transaction against transactions which have been validated and then propagated by the sequencer middleware replica. Only if the first validation does not fail, the writeset will be sent to the sequencer middleware replica for further validation.

Double validation can provide the opportunity to abort transactions earlier and eliminate the unnecessary message exchange. Since the message latency is high in WANs, this optimization might be good for performance.

5.4 Optimization 4: hybrid protocol for WAN applications with several LANs

5.4.1 Basic idea

The protocols discussed so far make the assumption that the communication latency between any two sites is approximately the same. For example, each site is connected with each other via a WAN. However, in the real world, it is more

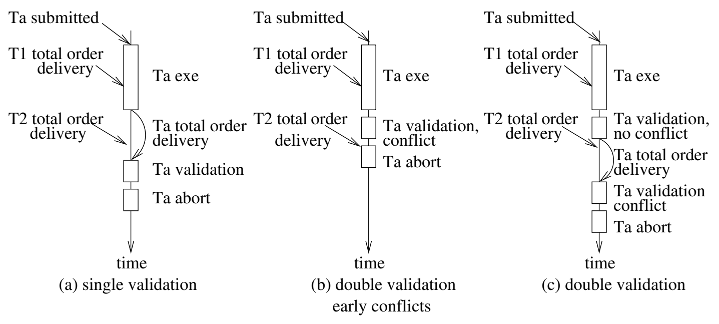

Figure 11. Double validation optimization

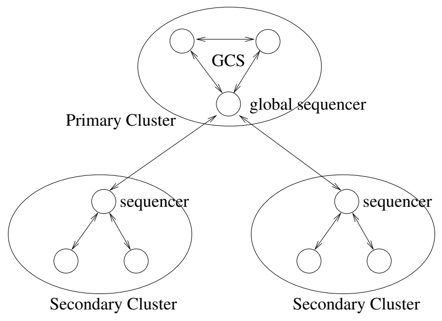

Figure 12. Hybrid protocol

reasonable to consider that there might be more than one site in one LAN. We call all sites within a LAN a cluster. Some clusters may have more sites than others. For example, a Chinese news website would like to have many sites in the company's headquarter located in Beijing and only a couple of sites dispersed around the world because most of its readers are in China. Since the communication latencies between different sites variate a lot, especially between sites in the same LAN and across the WAN, we can take advantage of this characteristics in designing a hybrid protocol for such advanced applications.

Suppose there are clusters (LANs) with more than one site and many clusters in the whole system. The basic idea of the hybrid protocol is shown in Figure 12. Let's assign a cluster as the primary cluster and others as secondary clusters. A transaction will be validated within its local cluster first and then sent to the primary cluster for further validation (i.e., double validation). Within the primary cluster, we use the replica control protocol with GCS (as discussed in Section 5.1). Within the secondary clusters, we use the protocol with a sequencer (as discussed in 5.2). For the coordination between the primary and secondary clusters, we follow the same idea of using a sequencer as in Section 5.2. But we have to modify the protocol slightly to fit the problem.

When a transaction is submitted to a site in a secondary cluster, it follows the same procedure as discussed in Section 5.2 until it passes the second validation in the sequencer of the local cluster. After that, it can not commit yet because there may be some concurrent conflicting transactions in other clusters. Hence, its writeset has to be sent to a site (named as global sequencer) in the primary cluster. Once the global sequencer in the primary cluster receives a writeset from secondary clusters, it will multicast the writeset to all sites in the primary cluster. All sites in the primary replica will process the writeset as before. The global sequencer will do additional work. It will send all writesets of the transactions which have passed the second validation in the primary cluster to all sequencers of the secondary clusters. These sequencers will forward the writeset transactions to all sites in their individual clusters.

When a transaction is submitted to a site in the primary cluster, it just follows the protocol as discussed in Section 5.1. The global sequencer will also send the writesets of transactions to other sequencers of secondary clusters if they pass the second validation.

5.4.2 Discussion

The hybrid protocol takes into account the different network latencies between different sites. The hybrid protocol groups sites according to their physical locations, e.g., within one cluster. A secondary cluster only needs to communicate with the primary cluster. They communicate with each other through a representative (i.e., sequencer). A writeset message to be broadcast to all sites in a secondary cluster *C* is only sent to *C*'s sequencer, which will then broadcast the message within the LAN. Thus, the WAN network bandwidth will be saved. The sites in the primary cluster will response to their clients very fast because the multicast does not involve the sites in the remote clusters and the final decision to commit or abort can be made locally in the primary cluster. The sites in secondary clusters can abort transactions very fast if there are conflicts detected locally. But to commit a transaction, they still have to communicate with the primary cluster.

The approach could be generalized to tree structures with a depth larger than two for certain configurations. With that we might be able to release the global sequencer. Further study is needed into this issue.

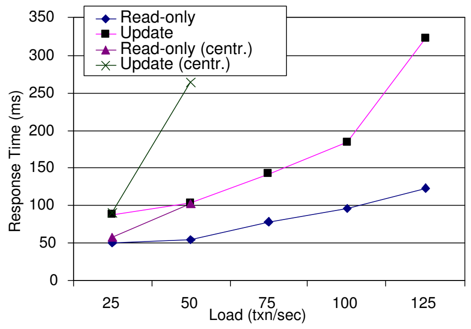

Figure 13. Response times for TPC-W

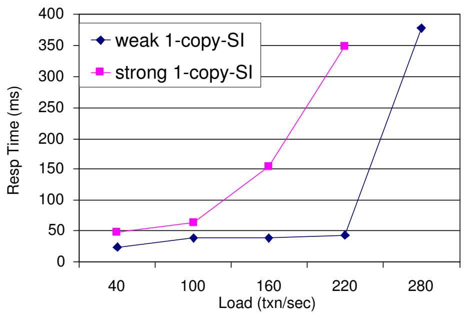

Figure 14. Response times of updates for small DB

6 Performance Evaluation

Currently I have implemented the basic protocol with the optimizations of GCS and double validation but without fail-over support. I tested the protocol on different workloads in LANs in order to understand its performance behavior. Further experiments in WANs will be conducted in the future. In this section, I first present results with running the TPC-W benchmark to see how the protocol behaves under a real application. Then, in order to compare protocols providing weak and strong 1-copy-SI, I run experiments on my own simplified benchmark. All experiments are 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 each 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 workload.

6.1 TPC-W

TPC-W [31] simulates an online bookstore with three kinds of workloads that vary in the ratio of update vs. read-only transactions. I choose the ordering workload that consists of 50% update and 50% read-only transactions. The database has eight tables, and the size of each table is determined by the items and clients in the system. My configuration has 1000 items and 144 000 client records. This results in a relatively small database of around 200MBytes.

Figure 13 shows the average response time over all update and over all read-only transactions with the increasing load in the system. Strong 1-copy-SI is guaranteed. There are five connected sites. Each site has five clients. 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 without any concurrency control and writeset retrieval etc.). The benchmark has many short queries giving 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 transactions per second (tps), the centralized and the replicated systems have more or less the same response times since the systems are only lightly loaded. At this load, the overhead of the middleware (communication/validation) is compensated by the fact that queries are distributed over 5 sites compared to the centralized system. At 50 tps, however, the centralized system is already saturated while the replicated system can handle 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%). The results shown here are similar to those shown by other middleware based replication solutions (e.g., [4, 20, 29]). What makes my approach different to the previous approaches is that we can achieve this without requiring to predeclare any transaction properties. Furthermore, if in WANs, my approach will work much better.

6.2 Comparing weak and strong 1-copy-SI

In this experiment, I run two versions of the protocol (i.e., one providing weak, the other providing strong 1-copy-SI). Recall that the protocol with weak 1-copy-SI allows transactions to start immediately upon arriving at middleware replicas and non-conflicting transactions which have passed validations to commit concurrently, while strong 1-copy-SI has additional synchronization overhead in delaying the start until no holes exist and synchronizing start and commit. By comparing these two versions of the protocol, we can get an idea of how costly the synchronization overhead is. The benchmark used has only a small database, 10MBytes. There are 10 tables and two transaction types, one update transaction performing 10 simple updates and one query scanning a table.

Figure 14 shows the response times for update transactions with increasing loads for 5 sites. The results for queries were similar in relative term, and hence, are not presented here. In this setting, we can see that the system with strong 1-copy-SI has worse response time than that with weak 1-copy-SI. The system with strong 1-copy-SI is saturated at 160 tps while the system withe weak 1-copy-SI at 220 tps. My experiment also showed that a centralized system is saturated at 100 tps. For the system with strong 1-copy-SI, I also analyzed in detail how much time different actions of a transaction spent within its response time, which is not shown here due to space limitations. I observed that quite some time is spent in the synchronization of start and commit statements. Especially this synchronization overhead is high when the load is high. On average, there are holes at around 10-15% of the times a transaction wants to start. These holes are mainly generated by local transactions that immediately commit after validation, overtaking remote transactions whose writesets were received earlier but who still have to apply them before commit. Whenever there are holes, starting transactions have to wait. Whenever transactions start, commit operations have to wait.

Currently, I use mutexes for start and commit of transactions. With this, all start/commit statements are serially executed. But we could allow concurrent start if there were no holes, and also concurrent commits (only some commits might be delayed if they produce holes when transactions want to start). I am currently reimplementing the algorithm to really allow for the true possible concurrency.

7 Conclusion and future work

7.1 Conclusion

Currently, middleware based protocols are widely developed for database replication due to simplicity and flexibility. However, they are not appropriate for WAN applications due to the large communication latency in WAN. Besides that, the existing protocols have many limitations which restrict them from practical use, mainly as 1) all operations of transactions must be known in advance, 2) transactions must be marked as read-only or update in advance, 3) concurrency control is at table level instead of record level.

My thesis proposes a replica control protocol working well in WANs. The paper also discusses the protocol for the WAN applications in which there are several clusters and each may have more than 1 replica. The fail-over procedures are also discussed carefully.

The basic idea of the protocols is to execute the whole transaction locally and then to send a writeset message with additional information such as primary keys to other sites. Executing the whole transaction locally eliminates the needs for knowing all operations and distinguishing read-only transactions in advance. Only one writeset message to be sent for one transaction reduces the communication overhead in the transaction response time. Thus, it has the potential to work well in WANs. The primary keys contained in the writesets enables the protocol to do concurrency control at record level instead of table level

Regarding global transaction isolation level, most of previous work attempts to achieve 1-copy-serializability. I am using a global transaction isolation level, called 1-copy-SI based on Snapshot Isolation. My protocol can provide two levels of 1-copy-SI. Both levels guarantees data consistency. But strong 1-copy-SI can further guarantee that the replicated database system works as if there is one database.

7.2 Future work

Currently, I have designed the system without considering recovery. To be a complete system, I have to consider recovery procedures in the future. Right now, I have built a first prototype and the implementation of update everywhere with GCS has been done, without fail-over support. My next schedule is:

1. By Sep of 2005: 1). Finish implementation and evaluation of all optimizations except the hybrid protocol, 2). Provide formal descriptions and correctness proofs, 3). Provide JDBC driver with fail-over support. The last is currently cooperated with a master project student.

2. By Dec of 2005: Finish implementation and evaluation of the hybrid protocol for WANs with fail-over support.
3. By May of 2006: Finishing recovery procedures.

References

[1] ADAPT. (Middleware Technologies for Adaptive and Composable Distributed Components) homepage: http://adapt.ls.fi.upm.es/adapt.htm.

[2] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. On the Performance of Consistent Wide-Area Database Replication. Technical Report CNDS-2003-3, CNDS, John Hopkins University, 2003.

[3] Y. Amir and C. Tutu. From Total Order to Database Replication. In *Proc. of ICDCS*, 2002.

[4] C. Amza, A. L. Cox, and W. Zwaenepoel. Conflict-Aware Scheduling for Dynamic Content Applications. In *USENIX Symp. on Internet Tech. and Sys.*, 2003.

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

[6] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, Consistency, and Practicality: Are These Mutually Exclusive? In *ACM SIGMOD Conf.*, 1998.

[7] ANSI. ANSI X3.135-1992, American National Standard for Information Systems Database Language SQL, Nov. 1992.

[8] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. ONeil, and P. ONeil. A Critique of ANSI SQL Isolation Levels. In *Proc. of SIGMOD*, pages 1–10, San Jose, USA, May 1995. ACM Press.

[9] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

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

[11] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *USENIX Conference*, 2004.

[12] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 469–476, New Orleans, Louisiana, February 1996.

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

[14] X. Defago, A. Schiper, and P. Urban. Comparative Performance Analysis of Ordering Strategies in Atomic Broadcast Algorithms. *IEICE Trans. Inf. and Syst.*, E86-D(12), Dec. 2003.

[15] E. Pacitti and T. Ozsu and C. Coulon. Preventive Multi-master Replication in a Cluster of Autonomous Databases. In *Euro-Par*, 2003.

[16] S. Elnikety, F. Pedone, and W. Zwaenepoel. Generalized Snapshot Isolation and a Prefix-Consistent Implementation. In *Technical Report*, EPFL, 2004.

[17] U. Fritzke and P. Ingels. Transactions on Partially Replicated Data based on Reliable and Atomic Multicasts. In *Proc. of ICDCS*, 2001.

[18] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proc. of SIGMOD*, 1996.

[19] J. Holliday, D. Agrawal, and A. E. Abbadi. The Performance of Database Replication with Group Communication. In *FTCS*, 1999.

[20] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving Scalability of Fault Tolerant Database Clusters. In *ICDCS'02*.

[21] K. Böhm and T. Grabs and U. Röhm and H.J. Schek. Evaluating the Coordination Overhead of Replica Maintenance in a Cluster of Databases. In *Proc. of Euro-Par*, 2000.

[22] B. Kemme. *Database Replication for Clusters of Workstations*. PhD thesis, Dept. of Computer Science, Swiss Federal Institute of Technology Zurich, 2000.

[23] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication. In *VLDB'00*.

[24] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *ICDCS'99*.

[25] Y. Lin, B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. Middleware based data replication providing snapshot isolation. In *SIGMOD*, June 2005.

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

[27] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting Atomic Broadcast in Replicated Databases. In *Proc. EuroPar*, 1998.

[28] F. Pedone, R. Guerraoui, and A. Schiper. The Database State Machine Approach. *Distributed and Parallel Databases*, 14:71–98, 2003.

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

[30] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong Replication in the GlobData Middleware. In *Workshop on Dependable Middleware-Based Systems*, 2002.

[31] Transaction Processing Performance Council. TPCW Benchmark.

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