Tashkent: Uniting Durability with Transaction Ordering for High-Performance Scalable Database Replication

Sameh Elnikety
School of Computer and Communication Sciences
EPFL
Switzerland

Steven Dropsho
School of Computer and Communication Sciences
EPFL
Switzerland

Fernando Pedone
Faculty of Informatics
Università della Svizzera Italiana
USI
Switzerland

Abstract

In stand-alone databases, the two functions of ordering the transaction commits and making the effects of transactions durable are generally performed in one action, namely in the writing of the commit record to disk. In replicated database systems where all replicas agree on the commit order of update transactions, these two functions are naturally separated; specifically, the replication middleware determines the global commit order, while database replicas make transactions durable. The contribution of this paper is to demonstrate that the traditional separation of commit ordering from durability in replicated designs forces update transactions to be made durable *serially* to disk, a potentially significant scalability bottleneck. Two solutions are possible: (1) keep durability in the database and pass the global commit order from the replication middleware to the database, or (2) move durability from the database to the replication middleware. We show that regardless of the method, uniting ordering and durability greatly improves system scalability.

We implement two example scalable replicated database systems called Tashkent-MW and Tashkent-API to show the benefits of joining global commit order and durability. Tashkent-MW is a pure middleware solution that combines ordering and durability in the middleware and treats an unmodified database as a black box. Tashkent-MW represents a high-performance replication solution suitable for closed-source, off-the-shelf standalone databases. In Tashkent-API, we modify the open source PostgreSQL database API so the middleware can specify the commit order, combining ordering and durability inside the database. We compare both Tashkent systems to a similar replicated system, called *Base*, in which ordering and durability remain separated. Under high update transaction loads at 15 replicas, we show both Tashkent systems greatly improve scalability and outperform *Base* by factors of 5 and 3 times, respectively, in throughput with lower response times.

Categories and Subject Descriptors

H.2.4 Systems – *distributed databases, concurrency.*

General Terms

Measurement, Performance, Design, Reliability.

Keywords

Database replication, Generalized snapshot isolation.

EuroSys'06, April 18-21, 2006, Leuven, Belgium.

1 Introduction

Database replication is a cost-effective technique to improve performance and to enhance availability for some applications. High performance replication designs distribute transactions across many replicas such that all replicas agree on the commit order of update transactions for consistency.

The primary contribution of this paper is to reveal a dependency between *durability* and commit *ordering* of update transactions in such designs. In particular, we show that relying on an off-the-shelf database to make transactions durable (*i.e.*, write their effects to disk) and commit them in a consistent global order requires the synchronous disk writes to be serialized, a significant scalability bottleneck.

We show that the root cause is the separation in replicated systems of commit ordering from the expensive synchronous disk writes that ensure durability. In standalone databases, these two functions are together, permitting group-commits to minimize the number of synchronous writes to disk, which is an important performance optimization.

However, in replicated database systems ordering is determined in the replication middleware – rather than at the database – to ensure a consistent global order. Since it is customary to only move the minimal required functionality to the middleware, it has been natural to leave durability in the database, leading to the separation of durability and ordering and to scalability problems. The solution is to unite durability and ordering in replicated systems.

We propose two approaches to unite these functions. The first approach is to move durability to the middleware layer where ordering is determined. It is attractive to make this approach a pure middleware solution so that it can be used with off-the-shelf databases. This approach is appropriate for open source and closed source databases and for clustering different databases, making it particularly suitable for third-party software providers supplying replication infrastructures. The second approach is to extend the database API so the replication middleware can specify a commit order for update transactions, combining durability and ordering in the database. An open source database, such as PostgreSQL, can be modified appropriately.

We implement instances of both approaches, called *Tashkent-MW* and *Tashkent-API*, respectively, and compare them to an instance of a traditional replication system, called *Base*, where

ordering and durability remain separated. All systems use generalized snapshot isolation (GSI) [3] for concurrency control. They are identical in all other respects. We use PostgreSQL [19], a well-known open source database to compare the performance systems.

In our experimental results, we show both Tashkent systems greatly improve scalability under high update transaction loads, and outperform *Base* by factors of five and three times, respectively, in throughput at 15 replicas and with lower response times.

The rest of the paper is structured as follows. Section 2 gives the necessary background on generalized snapshot isolation. In Section 3 we detail the issue of separating durability from ordering and justify uniting the two. In Section 4 we present the design of the *Base* replication system. In Section 5 we detail the changes necessary to implement Tashkent-MW and Tashkent-API. In Section 6 we discuss fault-tolerance of systems. We discuss the design of the replication middleware in Section 7, and its interface to PostgreSQL in Section 8. In Section 9 we experimentally compare the three systems and analyze their relative performance. Section 10 contrasts related work to this research. Finally, in Section 11 we summarize the main conclusions.

2 Background

Snapshot Isolation (SI) [1] is a concurrency control algorithm for centralized multi-version databases. In snapshot isolation, when a transaction begins it receives a view, called a *snapshot*, of the database for the duration of the transaction. After the snapshot is assigned it is unaffected by concurrently running transactions. A read-only transaction reads from the snapshot and can always commit under SI. An update transaction *T* reads from and writes to its snapshot, and can commit if it has no write-write conflict with any committed update transaction that ran concurrently with it.

Many database vendors use SI, e.g., Oracle, PostgreSQL, Microsoft SQL Server, InterBase [16, 1, 7, 10]. SI is weaker than serializability but in practice most applications run serializably under SI, including the most widely-used database benchmarks TPC-B, TPC-C, and TPC-W. SI has attractive performance properties. Most notably, read-only transactions never block or abort— they do not need read-locks, and they never cause update transactions to block or abort. This property is important for workloads dominated by read-only transactions, such as those resulting from dynamic content Web servers.

Generalized Snapshot Isolation (GSI) [3] extends SI to replicated databases (replicas) such that the performance properties of SI in a centralized setting are maintained in a replicated setting, including the property that read-only transactions do not block or abort, and they do not cause update transactions to block or abort. In addition, workloads that are serializable under SI are also serializable under GSI [3]. Here we summarize the essential details of GSI because it is the concurrency control model used in this paper. We give a more formal description of GSI in Section 7.

Informally, a replica using GSI works as follows. When a transaction starts, the replica assigns its latest snapshot to the transaction. All transaction read and write operations are executed locally on the replica against the assigned snapshot. At commit, the replica extracts the transaction's modifications to the database into a *writeset*. If the writeset is empty (i.e., it is a read-only transaction), the transaction commits immediately. Otherwise, a certification check is performed to detect write-write conflicts among update transactions in the system. If no conflict is found, then the transaction commits, else the transaction aborts.

The *certifier* performs certification and assigns a global total order to the commits of update transactions. Since committing an update transaction creates a new version (snapshot) of the database, the total order defines the sequence of snapshots the database replicas go through. Writesets are propagated to all other replicas to update their state. We refer to writesets of remote transactions during the propagation phase as remote writesets or *updatesets*.

3 How Durability and Ordering Affect Scalability

In the GSI algorithm, all replicas commit update transactions in a global order. However, there is a somewhat surprising dependency between maintaining the global commit order and how the durability function must write state to disk. In this section, we discuss an example of the problem, its implications on performance, and the underlying causes.

3.1 Example

Here we contrast a centralized versus a replicated system to show how separating durability and commit ordering in a replicated system forces serializing the commits which limits scalability in a replicated system.

Centralized SI database. The initial case is a centralized system. If clients concurrently submit two update transactions *T4* and *T9* – whose writesets (modifications), *W4* and *W9*, do not conflict – to a centralized SI database, the database can commit them in any order: *T4* then *T9*, or *T9* then *T4*. In addition, each commit corresponds to a disk write, a disk write for *W4* and another for *W9* to guarantee durability. However, the database IO subsystem can group *W4* and *W9* into a single disk write, which greatly improves performance, i.e., one disk write for both *T4* and *T9*.

One GSI replica. Next, consider a replicated GSI database consisting of one replica and one certifier, both at version zero. If clients submit *T4* and *T9* concurrently to the replica, it executes them and sends their writesets, *W4* and *W9*, to the certifier. The certifier checks the writesets, determines that there are no conflicts, and assigns them an order. Let us assume that the certifier orders *W4* at version 1 and *W9* at version 2, and sends these results back to the replica.

Upon receiving the responses from the certifier to commit the writesets, the replica must now commit *T4* first to reach version 1, then commit *T9* to reach version 2, the sequence that the certifier determined. Under GSI, the replica cannot change this order; otherwise, a new transaction may receive a snapshot containing the effects of *W9* but not *W4*, a snapshot that never existed globally (i.e., at the certifier). A typical database offers no mechanism to specify a commit order externally, but yet the middleware must not allow the replica to swap the order in which the commits of *T4* and *T9* occur. Without an efficient low-level external mechanism to enforce a particular commit order, the middleware must submit each commit serially, waiting for each commit (which includes a disk write) to complete. Thus, two disk writes, one for *T4* and another for *T9*, are required. This serializes a costly component of executing update transactions at the replica.

Multiple GSI replicas. We continue the example with a replicated GSI system having *N* replicas. One replica receives *T4* and *T9* and then sends *W4* and *W9* to the certifier. The certifier receives the two writesets, and receives writesets from other replicas as well, and creates the following total order: *W1*, *W2*, *W3*, *W4*, *W5*, *W6*, *W7*, *W8*, *W9*. Under GSI, the replica must observe this total order. Not only must the replica commit transaction *W4* before *W9*, but *W1*, *W2*, *W3* must commit before *W4*, and *W5*, *W6*, *W7*, *W8* before *W9*. The replica version follows the sequence: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, which would naively imply 9 disk writes, but this can be improved.

Grouping remote writesets. We can batch the writesets of several remote transactions by combining *W1*, *W2*, *W3* into one transaction *T1_2_3* with writeset (updateset) *W1_2_3* and similarly another transaction *T5_6_7_8* with writeset *W5_6_7_8*. The replica version follows this sequence: 0, 3, 4, 8, 9, which requires 4 disk writes (2**M* disk writes in general for *M* local update transactions at each replica).

It is important to note that the middleware is outside the database and cannot submit *T1_2_3, T4*, *T5_6_7_8, T9* concurrently to a standard database to force committing *T1_2_3* first, *T4* second*, T5_5_7_8* third, then finally *T9*, since databases do not provide such low level mechanisms to the client interface. Allowing the database to commit them in any order would require the middleware to block the start of new transactions (including read-only transactions), lest they observe an inconsistent snapshot (e.g., *T9* commits internally before *T4*). This approach has a major performance drawback as read-only transactions would block waiting for other transactions to finish, voiding the main performance benefit of GSI. Another reason for following the global order is that under certain conditions changing the commit order can result in the final state of the database not being the same. For example, if *T1_2_3* and *T5_6_7_8* both modify a common database item, then the order in which they commit is significant. Therefore, we propose two possible solutions for uniting ordering and durability.

Solution 1: Move durability to the middleware. Another solution is to have the middleware, which decides the ordering, be responsible for making durable any modifications to the database. For example, the middleware can batch all available requests at certification and ensure they are made persistent in the proper order, e.g., batch all nine writesets of the prior example into a single disk write. Furthermore, synchronous writes can be disabled in the replicas, so commits are essentially as fast as in-memory actions and serializing them is not a performance issue.

Solution 2: Pass the ordering information to the database. For example, if the database API interface is extended such that the middleware specifies the commit order, then all writesets could be submitted to the database concurrently.

The key insight is that in both solutions durability is united with ordering. Regardless of whether the two are united in the replication middleware as a pure middleware solution or in the database via an extended API, as we show in the results both solutions permit IO subsystem optimizations that greatly improve throughput.

For this paper, the two example solutions assume that the standalone database (1) supports the SI concurrency control model, (2) has the ability to capture and extract writesets of update transactions (further elaborated on in Section 8), (3) has the ability to enable/disable synchronous writes to disk, and (4) uses write-ahead logging (WAL).

4 Architecture and Design of *Base*

Here we describe a replicated database design, called *Base*, representing a traditional solution in which the middleware performs global ordering but relies on the database replicas for durability. In Section 5, we derive from *Base* two systems, Tashkent-MW and Tashkent-API, that combine ordering and durability. Recovery is an important aspect of any database system; we discuss fault tolerance and recovery for each of the three replicated designs in Section 6.

4.1 Architecture

Base is a replicated database system. It uses GSI and consists of two main logical components both of which are replicated: (1) database replica and (2) certifier. When a replica receives a read-only transaction, the replica executes it entirely locally. When a replica receives an update transaction, it executes it locally - except the commit operation, which requires global certification. Replicas communicate only with the certifier component, not directly with each other. The certifier validates (certifies) update transactions from all replicas and orders them.

The design described is a *pure middleware* solution since no modification to the database source code is required; thus, each database replica can be an off-the-shelf standalone system. Attached to each replica is a *transparent proxy* that intercepts database requests. The proxy appears as the database to clients, and appears as clients to the database. We refer to the proxies and certifiers as the *replication middleware*.

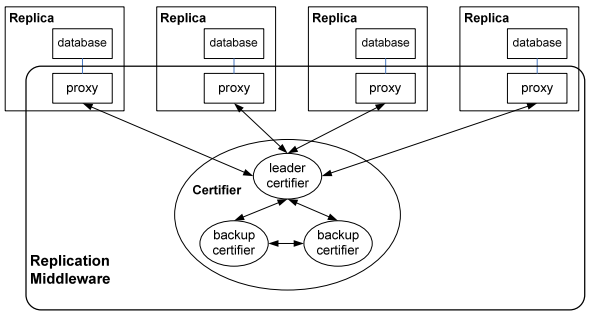

Figure 1 - Architecture of *Base*

Figure 1 shows the architecture of the *Base* system. The two main components, database replica and certifier, are replicated asymmetrically at different replication degrees. Database replicas are replicated mainly for performance, whereas the certifier is replicated mainly for availability. The certifier component could be implemented via an atomic broadcast mechanism incorporated into the proxy at every replica. However, for this study we simply assume a separate certifier component, which itself can be replicated, though our conclusions apply to other configurations.

For update propagation, we use writesets rather than the original SQL text of update transactions. Although for some transactions propagating the original SQL text may be shorter in size than the writeset, it is generally more expensive to re-execute the SQL text at the certifier and then at each replica when propagating the effects of the transaction.

4.2 Processing Update Transactions

Processing read-only transactions is straightforward. Each read-only transaction is assigned a snapshot and it reads from its snapshot. Next, we outline the mains steps of processing an

update transaction using the GSI algorithm (see [3] for the formal specification) as used in *Base*. The pseudo code and its implementation are in Sections 7 and 8. We use the following terminology. We use *version* to count database snapshots. The version at a replica database is called `replica_version`. The database starts at version zero, i.e., at the initial state `replica_version=0`. When an update transaction commits, `replica_version` is incremented. Each transaction has two numbers, its version at start, `tx_start_version`, and the version created at its commit, `tx_commit_version` (`tx_commit_version` is valid for only update transactions). At a replica, *updatesets* are the writesets of committed update transactions from other remote replicas that must be applied locally to bring the replica's state up-to-date before the local transaction can commit.

The GSI algorithm has two parts: the actions at replicas and actions at certifiers. We first present an outline of the actions here, and then present the pseudo code in Section 7**.**

Writeset intersection. At the point of an update transaction commit, the proxy at the replica makes a request for certification to the certifier. The certifier performs writeset intersection, a fast main memory operation that compares table and field identifiers for matches against recent writesets to detect write-write conflicts. For the *committing* transaction T_c with writeset W_c, the certifier compares W_c to the set of writesets committed at versions only more recent (greater) than T_c.`tx_start_version`. Successfully certified writesets (i.e., with no conflicts) are recorded in a persistent log thus creating a global ordering. The log is necessary because we use the crash-recovery model for the middleware (and for the databases too). The state of any replica is always a consistent prefix of the certifier's log. Certification is lightweight: writeset intersection is a quick operation and writing to the persistent log can be done very efficiently.

Durability. In the *Base* design, the durability function is performed in the database. Since the certifier in the middleware determines the order of commits, the middleware commits update transactions serially at each database to ensure the same commit order is followed. In the results we show that this serialization is a scalability bottleneck, one which we address via Tashkent-MW and Tashkent-API.

Responding to replicas. The certifier logs the writeset before it responds to the replica with the (1) result of the requested certification test, (2) along with any updatesets for the intervening update transactions at other remote replicas, and (3) the version at which the transaction commits (if the certification test succeeds). The middleware proxy at the replica applies the updatesets to the database before it commits the certified update transaction.

5 Tashkent-MW and Tashkent-API

Here we describe the modifications to the *Base* system needed to implement Tashkent-MW and then the changes to implement Tashkent-API to combine ordering and durability.

5.1 Tashkent-MW

To unite durability with ordering in the middleware, synchronous writes are disabled at the database replicas and the middleware logs the database modifications (i.e., writesets) for durability. However, the certifier already logs this information via its persistent log – this log is used for a different purpose: to allow certifier recovery. Because enabling/disabling synchronous writes is a standard feature in many databases, it is relatively simple to transform the *Base* design into Tashkent-MW. Furthermore, since *Base* is a pure middleware solution, so then is Tashkent-MW. The commit operations at each database replica are serial, but they are fast in-memory operations. Therefore, system throughput increases and system scalability is greatly improved because the durability function is performed at the certifier which can efficiently group the writesets into a minimal number of synchronous writes. However, there is a cost to moving durability out of the databases and it appears in the recovery procedure. We address this in the next section.

5.2 Tashkent-API

To unite ordering with durability in the database, the API available to the middleware is extended so the commit order of update transactions can be specified. The change is to the SQL "COMMIT" command to permit an optional integer parameter giving the sequence number to the commit (e.g., COMMIT 9 to commit current transaction at version 9). Since we change the interface of the database, Tashkent-API is not a pure replication middleware and the source code of the database must be available.

The flow of actions is the following. When a client commits a transaction, the commit is intercepted by the middleware. As in *Base*, the middleware extracts the writeset of the transaction and sends it for certification. Here, upon receipt of a *commit decision* from the certifier, the Tashkent-API middleware forwards to the database both the commit command and the version number returned from the certifier. Similarly, updatesets are applied in separate transactions with their appropriate commit sequence numbers.

Next we describe the changes to the database to follow the commit order and we use PostgreSQL. Inside the database, commit ordering is enforced as follows. Modifications of a transaction are visible after (1) the commit record is written to disk and (2) the transaction is announced as committed. Thus, to control the sequence of commits we need only control the order in which transactions are announced as committed. We modify the PostgreSQL in Tashkent-API so that all pending commit operations wait upon a semaphore *after* writing their commit records to disk, *i.e.*, we control only the second step, when a transaction is "announced" as committed. The semaphore is initialized to 0 at system start and incremented after each commit. Each commit requests the semaphore with a count matching the specified ordering sequence and blocks until the semaphore count progresses sufficiently. This method requires minimal changes to PostgreSQL (20 lines of source code).

The database may write the commit records to disk in an order different from the order in which transactions are announced as committed. This behavior is identical to the original standalone PostgreSQL system; therefore, database recovery is not affected by this change.

This extended API is a simple interface yet powerful; it should be restricted to the replication middleware only. Normal clients are unaware of system wide commit ordering. If the interface is abused (e.g., issuing COMMIT 9, without ever providing COMMIT 1-8), the database may deadlock, potentially aborting the committing transaction to resolve the deadlock.

5.2.1 Constraints under Tashkent-API

With the new API, the replication middleware can submit multiple commit operations concurrently since ordering is enforced inside the database (after writing the commit record to disk). This concurrent submission of commits is the source of performance improvement for Tashkent-API since the database can group the writing of several commit records. *However, it is not always possible for the middleware to submit transaction commits concurrently*, leading to serializing some commit operations and subsequently reducing the performance gain. This section describes the condition for which a commit must be serialized and how to detect if this condition arises.

The condition may arise when committing two different local update transactions. Should their accompanying two updatesets modify a shared element then the middleware submitting these commits concurrently will generate an "artificial" write-write conflict among local transactions in the database replica, inhibiting the transactions from committing. The root of the problem is that the "artificially conflicting" updatesets, which were generated by remote transactions that *did not* run concurrently originally, are now applied by concurrent local transactions to group their commit records. We illustrate this subtle but important point, which may appear in Tashkent-API, with a concrete example and present a simple detection scheme.

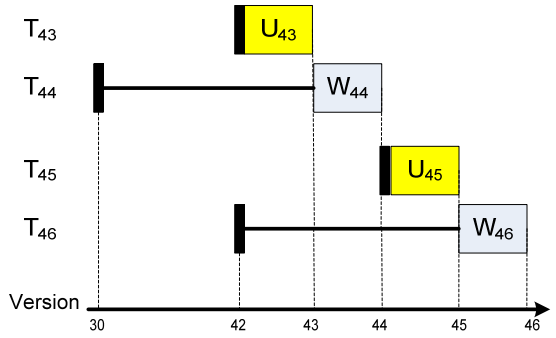

Figure 1: Example of concurrent commits

Figure 1 shows two local *concurrent* transactions, T_{44} and T_{46} at a replica and two updatesets, U_{43} and U_{45}, that are applied as local transactions, T_{43} and T_{45}. The transactions are labeled with the version at which they commit globally. For example, T_{44} starts from a snapshot with version 30 and its writeset W_{44} commits as version 44 globally, but before it can commit, updateset U_{43} is received from the certifier and applied to the replica as local transaction T_{43}.

The proxy can safely submit the commits of transactions T_{44} and T_{46} concurrently at the database along with T_{43} and T_{45} if none of the updatesets (U_{43} and U_{45}) conflict. If there is no conflict, then the database can write the commit records of T_{43}, T_{44}, T_{45}, and T_{46} in one disk write. This is the desirable behavior.

If an artificial conflict exists between U_{43} and U_{45}, e.g., U_{43} sets x=17 and U_{45} sets x=39, then this artificial conflict is detected and U_{45} will be serialized waiting for U_{43}. Therefore, If U_{43} and U_{45} have an artificial conflict, then the proxy commits T_{43} and T_{44} concurrently, waits for the acknowledgement from the database, commit T_{45} and T_{46} concurrently. The best the database can do is to group the commits of T_{43} and T_{44} into one disk write and the commits of T_{45} and T_{46} into a second disk write. Therefore, artificial conflicts decrease opportunities to group commits.

The proxy in Tashkent-API needs to know if it is safe to apply an updateset concurrently with previous updatesets. We adopt a safe (and conservative) solution to detect any potential artificial conflicts. The certifier performs extra work to return a hint, which is extra information on the updateset returned in the response to a certification request. When the certifier receives a certification request, it verifies the received writeset from the current version back to its start version. We extend this verification to updatesets as well; the certifier verifies all updatesets to be returned back to the transaction's start version. The hint, which the certifier returns in its response, is the version back to which the updatesets have been successfully verified.

For example, to certify T_{46}, W_{46} is certified back to version 42 and updateset U_{45} is verified back to version 42 as well. If an updateset has already been verified that far back, then it will not conflict with another updateset: U_{45} will not have an artificial conflict with U_{43}. While this method is conservative, it is simple and requires only information readily available in the transaction itself (its start version). We discuss other conflict types (e.g., between an updateset and a writeset) in Section 8.

If such artificial conflicts between updatesets are frequent then in the worst-case all commits are forced to be serialized and Tashkent-API degrades gracefully to the performance of *Base*. In addition, the overhead of the additional verification checks at the certifier is minimal since we record for each writeset the point to where it has been (further) certified and avoid repeated checks for responses to other replicas.

6 System Fault-Tolerance

We use the crash-recovery model— a node may crash and subsequently recover an unbounded number of times. *Safety* is always guaranteed. If any number of replica or certifier nodes fail, no transaction is lost: all effects of every committed update transaction are durable in the system. *Progress* depends on the number of failures. Read-only transactions can be processed if at least one replica is up. Update transactions can be processed if a majority of certifier nodes are up and at least one replica is up. Next, we discuss how replicas recover, and how the certifiers achieve fault-tolerance and recover.

6.1 Replica Recovery for Tashkent-MW

Moving durability outside the database may interfere with the database recovery subsystem. Informally, turning off durability in a database *should not* affect physical data integrity (correct data pages on disk). In some databases, however, when durability is disabled, physical data integrity may be violated as well, producing corrupt data pages in case of database crash. In this subsection we explain this further and outline a middleware recovery scheme.

Modern databases use write-ahead logging (WAL), which improves their IO performance. For guaranteeing physical data integrity, a database can write a dirty data page only if the corresponding record in WAL is stable on disk. Therefore, if the database crashes while writing a dirty data page and the data page becomes corrupt, then the page can still be recovered using the undo/redo information in WAL.

Thus, WAL is written to disk for two reasons: (1) to commit update transactions and make their effects durable (i.e., for *durability*), and (2) to allow dirty data pages to be safely written to disk without violating integrity (i.e., for *physical data integrity*).

There are two cases. First, most databases offer the option of "disabling all WAL synchronous writes", voiding both durability and physical data integrity guarantees— that is if the database crashes, some committed update transactions may be lost and a data page may be corrupt. When we deploy Tashkent-MW on such databases, we disable all WAL synchronous writes for performance, which voids physical data integrity (but durability is still guaranteed in the middleware). Therefore, to guarantee physical data integrity, the Tashkent-MW middleware periodically asks the database to make a copy, with the middleware recording the database version at the point of its request. Most databases have direct support for taking such a copy even while the database is normally processing transactions, especially in snapshot isolated databases because a database copy is logically equivalent to a database snapshot.

The Tashkent-MW middleware maintains two complete copies of the database. If the database crashes, the middleware restarts the database with the last copy, or the second to last copy (in the case where the database crashed while dumping the last copy). Next, the middleware updates the state of the database by applying all updatesets that have occurred since the version of the copy used for recovery. We use this alternative with PostgreSQL because it uses write-ahead logging and can either enable or disable *all* WAL synchronous writes.

The second case is where databases offer the option of "enable WAL synchronous writes, but disable WAL synchronous writes on commits of update transactions", which guarantees data integrity but does not guarantee durability— that is if the database crashes, it can recover to a previous committed state but some update transactions that committed after the last WAL synchronous write may be lost. If the database offers this feature, Tashkent does not need to take database copies; the database uses its recovery mechanism and then the middleware applies the necessary updatesets to bring the database up-to-date.

6.2 Replica Recovery for *Base* and Tashkent-API

In both *Base* and Tashkent-API, the database uses its standard recovery scheme, redoing/undoing transactions in the database log as necessary. Next, the middleware proxy needs to re-apply update transactions whose commits were forwarded to the database but were not acknowledged. In *Base*, this is at most one transaction because update transactions commit serially. In Tashkent-API, this is at most the set of update transactions whose commit operations were concurrent at the time of the crash.

Reapplying update transactions using their writesets in the global order is always safe. Therefore, if these update transactions are not known (e.g., the proxy crashes with the database), the proxy can obtain them from the certifier's log. At the end of this step, the proxy knows the version of the database. After the database recovers and proxy updates the database, the replica resumes normal operations.

6.3 Certifier Replication and Recovery

The certifier is identical in all three systems. The certifier state is replicated for availability. We replicate the state of the certifier on a small set of nodes using Paxos [13]. Here we briefly describe how the certifier state is replicated and how recovery works. The replication algorithm uses a *leader*. The leader is elected among the certifiers, and is responsible for receiving all certification requests.

Normal Case. When the leader receives certification requests, it performs normal GSI certification and selects which transactions may commit. Then, it sends the new state (i.e., the log records containing writesets of the selected transactions) to all certifiers including itself. All certifiers write the new state to disk and reply to the leader. Upon reception of replies from a majority of certifiers, the leader declares those transactions as committed.

On Failure. When a certifier crashes, a new leader is elected (if necessary) and certification continues making progress whenever a majority of certifiers are up.

On Recovery. When a certifier recovers from a crash, it requests a state transfer from another up certifier to update its state, participates in electing a new leader (if necessary), and logs certification requests to disk.

7 Implementation of Replication Middleware

In this section we discuss the implementation of the replication middleware which uses the GSI algorithm and list its pseudo code. In the next section we discuss the mechanisms used by the proxy to interface to PostgreSQL.

7.1 Certifier

The certifier implements certification of the GSI algorithm. The certifier prevents write-write conflicts and orders transaction commits. When the certifier receives a request to certify transaction *T*, it runs the certification procedure which checks for write-write conflicts between *T*.`writeset` and the writesets of other committed update transactions that ran concurrently with *T*. Certification is a stateful service that maintains (1) a persistent LOG containing tuples of (`writeset`, `tx_commit_version`) for all committed update transactions and (2) `system_version`. Certification has two inputs (`writeset`, `tx_start_version`) and three outputs (updatesets, certification result, `tx_commit_version`). The pseudo code is the following:

On a certification request for (*T*.`tx_start_version`, *T*.`writeset`):

1. The input *T*.`writeset` is tested for intersection against entries in LOG whose `tx_commit_version` > *T*.`tx_start_version`. An intersection occurs if the two writesets overlap (signaling a write-write conflict).
2. **IF** there is no intersection,
   THEN {
   - decision ← "commit",
   - increment `system_version`,
   - *T*.`tx_commit_version` ← `system_version`,
   - append (*T*.`writeset`, *T*.`tx_commit_version`) to persistent LOG for crash-recovery }

   ELSE decision ← "abort".
3. The output of the procedure contains:
   - the writesets which replica did not receive between `tx_start_version` and *T*.`tx_commit_version`,
   - decision (either "commit" or "abort"),
   - *T*.`tx_commit_version`.

The certifier is implemented as a multi-threaded server in C with *worker threads* to receive and process certification requests and send responses. The certifier has a single *writer thread* for writing the certifier log to the disk. The key to high

performance is the single writer thread that batches all outstanding writesets certified by the worker threads and commits the batch to disk via a single *fsync* call.

7.2 Transparent Proxy

We use a proxy in front of each database to intercept incoming database requests. The proxy tracks the database version `replica_version`, maintains a small amount of state for each active transaction, invokes certification, and applies the updatesets. We begin by listing the pseudo code of the GSI algorithm at the replica. Then, show how some of the steps are implemented in the proxy. The steps a replica takes to process a transaction *T* are the following:

A- On *T* begin:
1. *T* receives a snapshot of the database,
2. *T*.`tx_start_version` ← `replica_version`.

B- On *T* read and update operations:
1. Read and update operations are executed on *T*'s snapshot.

C- On *T* commit:
1. *T*.`writeset` is extracted and examined.
2. **IF** *T*.`writeset` is empty (i.e., *T* is read-only),
   THEN *T* commits,
   ELSE invoke *certification* (*T*.`tx_start_version`, *T*.`writeset`)
3. Certification returns three outputs:
   (a) updatesets,
   (b) "commit" or "abort",
   (c) *T*.`tx_commit_version`.
4. Replica applies the updatesets (a), which are the effects of update transactions (their writesets) from other replicas in the system.
5. **IF** the second output (b) is "commit",
   THEN { *T* commits, and `replica_version` ← *T*.`tx_commit_version (c)`}.
   ELSE *T* aborts.

The labels refer to the corresponding parts of the pseudo code.

[A1] Intercepting BEGIN. When the proxy intercepts "BEGIN", which signals a new transaction, it creates a new record (`tx_start_version`, `tx_commit_version`), and assigns `tx_start_version = replica_version`. The proxy forwards "BEGIN" to the database.

[C2] Intercepting COMMIT. When the proxy intercepts "COMMIT" for transaction *T*, it extracts the writeset of the transaction. If the writeset is empty (i.e., a read-only transaction), "COMMIT" is forwarded to the database; else the proxy invokes certification by sending `T.writeset`, `T.tx_start_version` to the certifier, and waits for the output from the certifier.

[C3] Processing certifier output. The proxy receives updatesets, the certification decision and `T.tx_commit_version`.

[C4] Updatesets. Updatesets are first stored in a memory-mapped file, called `proxy_log`, a new transaction is started containing the updatesets, the transaction is sent to the database, and then the proxy waits for the reply. The database executes the transaction, applies the updates, and sends the commit to the proxy.

[C5] Finalizing the COMMIT. If the certifier decision is "abort" the proxy forwards "ABORT" to the database, aborting transaction *T*. On the other hand, if the certifier decision is "commit", the proxy forwards "COMMIT" to the database, waits for its reply. The database commits the transaction. The proxy updates `replica_version = T.tx_commit_version`. Additionally, for Tashkent-API, `T.tx_commit_version` is added as a parameter to the "COMMIT".

Note that steps [C4] and [C5] are serialized both in *Base* and in Tashkent-MW. The proxy applies the updatesets, waits for the reply from the database, sends the transaction commit, and again waits for the reply from the database. For Tashkent-MW, this serialization is quick since the database acts essentially as an in-memory database. For Tashkent-API, steps [C4] and [C5] can be concurrent since the commit command includes the commit order.

Local certification. Local certification is an optimization that can reduce work at the certifier by reducing the number of versions against which a writeset is certified. Before the proxy sends a certification request to the certifier with parameters (`T.writeset`, `T.tx_start_version`), the proxy certifies `T.writeset` against the updatesets in the `proxy_log`, since the proxy may have received new updatesets while the transaction was executing. If the local certification succeeds, `T.tx_start_version` is advanced appropriately and certification is invoked. Otherwise, *T* is aborted, obviating unnecessary certification.

Conservative assigning of versions is safe under GSI. The proxy always assigns its most recent value of `replica_version` to new transactions (step [A1] above). However, during the execution of steps [C4] and [C5], the database commits the transaction before informing the proxy. Therefore, it is possible that the database has already performed the commit and gives the new transaction a newer snapshot of the database. This is safe because under GSI because certification is correct as long as the transaction is labeled with a version that is the same or earlier than its actual version, i.e., the certifier detects all write-write conflicts.

Bounding Staleness. Updatesets are sent in response to update transaction commits. In the case where an update transaction has not been received for a period of time (e.g., a few seconds), a proxy can proactively request updatesets from the certifier to bring the database up-to-date.

Proxy Implementation. The proxy is implemented as a multi-threaded Java program providing a JDBC interface, which means that all access to the replicated tables must go through this driver.

8 Middleware Interface to PostgreSQL

8.1 Proxy-Database Interface

The interaction between the proxy and database is database-specific. We describe here the mechanisms used to implement the proxy interface for PostgreSQL. Using another database would require changes, though the underlying mechanisms are common in most databases.

[C1] Writeset Extraction. Extracting the writeset of a transaction is not defined in the SQL standard, but many databases provide mechanisms to extract the writesets, such as triggers in PostgreSQL, direct support in Oracle, and log sniffing in Microsoft SQL Server.

We use PostgreSQL's database triggers to extract the writesets. We define triggers on the following events "INSERT", "UPDATE", and "DELETE" for replicated tables. When a trigger is fired it: (1) captures the new row in case of

"INSERT", (2) captures the primary key(s) and the modified columns for "UPDATE", or (3) captures the primary key for "DELETE". The trigger also records the table name and operation type. These changes are stored in a memory mapped file in order to give access of partial writesets to the proxy (see Eager Pre-certification below).

Recovery. For recovery in Tashkent-MW, the proxy sends a "DUMP DATA" command to the database periodically. The proxy stores `replica_version`, timestamp, dump data, end-of-file marker with a checksum in a file. If the database crashes, the proxy restarts the database using the appropriate dump file.

Soft Recovery. The proxy may send "COMMIT" to the database, and the database decides to abort the transaction due to exceptional circumstances, such as running out of disk space, performing garbage collection to delete old snapshots, or a crash of one database process. In such cases, the proxy aborts all active transactions, and re-applies the updatesets and the previously aborted transaction sequentially in order. If soft recovery fails, the proxy performs normal system recovery.

8.2 Deadlock Avoidance

A deadlock may develop between the writeset of a local transaction and the updateset from a remote transaction because PostgreSQL uses write locks. This deadlock scenario exists in all three designs *Base*, Tashkent-MW, and Tashkent-API. We explain the locking mechanism in PostgreSQL, how deadlocks can arise, and a middleware solution for this problem.

Write Locks in PostgreSQL. Since PostgreSQL (and other centralized SI databases) immediately sees all partial writesets of active update transactions, it uses write locks to *eagerly* test for write-write conflicts during transaction execution rather than at commit time. The first transaction that acquires a write lock on a database item may proceed, blocking the competitors who want to update the same database item. If the lock holder commits, all competitors abort. If the lock holder aborts, one of the competitors may proceed.

Traditional Deadlock Scenario. In a centralized database, deadlock may develop between two update transactions *T1* and *T2*. Suppose *T1* updates **x**, and holds the write-lock on item **x** and *T2* updates **y** and holds the write-lock on **y**. Then, *T2* attempts to update **x** and *T1* attempts to update **y**. Neither *T1* nor *T2* can proceed as each is waiting for the other.

Deadlock between a Writeset and a Updateset. In a replicated system, a similar deadlock situation can arise if *T2* is on a remote replica, has been certified, and its writeset is being applied as an updateset to the replica where *T1* is currently executing and holding a lock.

Some databases allow tagging transactions with priorities. If such a mechanism is available then avoiding a deadlock is straight forward: we mark updatesets with high priority, aborting any conflicting local transaction. However, PostgreSQL does not have priorities; therefore, we can either (a) do nothing letting PostgreSQL handle deadlocks between updatesets and local update transactions (we run soft recovery if PostgreSQL aborts an updateset), or (b) detect and prevent deadlocks eagerly in the middleware as an optimization.

Eager Pre-certification. In PostgreSQL, which does not support tagging transactions with priorities, the proxy can avoid deadlocks by eagerly detecting write-write conflicts between updatesets and writesets of local transactions. Conceptually, we leverage the local certification functionality at the proxy (Section 7.2) to eagerly certify every write in a transaction as it occurs against the pending updatesets. Similarly, updatesets received at the replica are verified against the current group of partial writesets. If a write-write conflict is found in any case, the proxy aborts the conflicting local update transaction, which allows the updateset to be executed.

We point out that local certification and eager pre-certification of the writesets does not increase the total number of low-level certification comparisons system wide, since the writesets advances their `tx_start_version` when locally certified against recent updatesets, and they hence do not have to be recertified globally against the same updatesets at the certifier. Local certification and eager pre-certification change *when* (earlier than the commit time) and *where* (on the replica rather than on the certifier) some certification comparisons are performed, and therefore, they have the benefit of distributing the certification load to the replicas (though the time for certifying a writeset is typically an order of magnitude less than that of executing the transaction itself).

9 Performance Evaluation

9.1 Methodology

Under generalized snapshot isolation, read-only transactions are executed completely locally on the receiving replica. Thus, the scalability of the replicated system is limited by the rate of update transactions and the overhead of maintain consistency. We assess the performance of uniting durability with ordering using three benchmarks that vary widely in their update rates to compare Tashkent-MW and Tashkent-API to *Base*.

AllUpdates Benchmark. We developed a benchmark where clients rapidly generate back-to-back short update transactions that do not conflict. The benchmark is relatively small and highly focused on fast, simple updates; it reflects high-throughput memory-centric behavior. This benchmark, called AllUpdates, creates a very heavy update-only workload to generate maximum system consistency maintenance overhead and represents a worst case workload for a replicated system. The average writeset size is 54 bytes for each update transaction in this benchmark.

TPC-B Benchmark. TPC-B is a benchmark from the Transaction Processing Council [23] that uses transactions containing small writes and one read. In contrast to the AllUpdates benchmark, TPC-B transactions has both reads and writes, plus its workload contains write-write conflicts. AllUpdates has no conflicts and TPC-W has very few conflicts. The writeset size is 158 bytes for each update transaction in the TPC-B benchmark.

TPC-W Benchmark. TPC-W is also a standard benchmark from the Transaction Processing Council designed to evaluate e-commerce systems. It implements an on-line bookstore. TPC-W has three workload mixes. The shopping mix (with 20% updates) is the main mix for reporting results. In contrast to the other two benchmarks, the relatively heavy weight transactions of TPC-W make CPU processing the bottleneck. The average size of an update transaction writeset in TPC-W is 275 bytes.

Thus, the three benchmarks AllUpdates, TPC-B, and TPC-W represent a spectrum of workloads differing in their transactions complexities and conflict profiles.

System specification. We use a cluster of machines, each running the 2.6.11 Linux kernel on a single Intel Xeon 2.4GHz CPU with 1GB ECC SDRAM, and a 120GB 7200pm disk drive. The machines are connected through a switched 1Gbps

Ethernet LAN. We use a leader certifier and two backups for fault tolerance (see Section 6.3). We use the PostgreSQL 8.0.3 database configured to run transactions in the snapshot isolation level (which is the strictest isolation level in PostgreSQL where it is called the "*serializable transaction isolation level"*). The database working set fits entirely in main-memory.

IO channel for durability. To guarantee durability we use synchronous disk writes with the Linux system calls *write()* and *fsync()*, such that the *fsync* call returns only after the data has been flushed to disk (i.e., to the disk media rather than to the disk cache). On our test systems *fsync* takes about 8ms but the actual time varies depending on where data resides on disk (6ms-10ms). Each machine in the cluster contains only one disk which is used for three IO activities: (1) reading the database pages, (2) writing dirty database pages, and (3) writing log records (for which *fsync()* maybe used). In some experiments, we dedicate the disk for logging by putting the databases of AllUpdates and TPC-B in a ramdisk to remove the effect of activities (1) and (2). Unfortunately, due to the size of TPC-W environment, we cannot use ramdisk with this benchmark, but we fully discuss the implications this has on our results.

9.2 AllUpdates

Here we compare our three replicated systems, *Base*, Tashkent-MW, and Tashkent-API, using the AllUpdates benchmark. Figure 2 shows the throughput results for the three systems. The x-axis is the number of replicas.

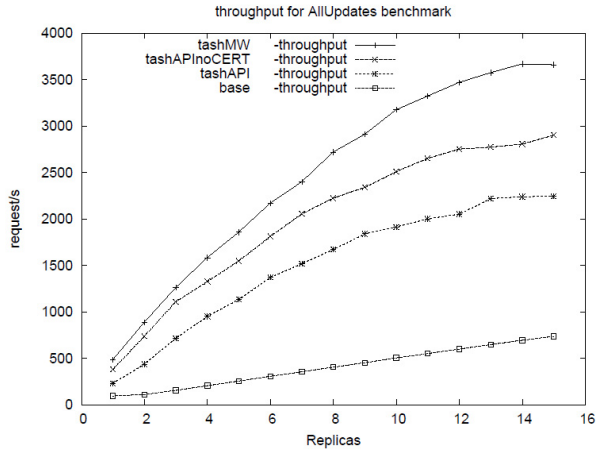

Figure 2: Throughput for AllUpdates (shared IO).

In Figure 2, the curve for *Base* is at the bottom and grows linearly at the rate of just over 49 req/sec and up to 735 req/sec at 15 replicas. This matches the limits of each replica's IO channel to process synchronous writes. At 8 ms, the limit is 125 IO/sec since all commits are serialized. If there are updatesets returned in the certification process (which is the case even at two replicas) then two writes are required to commit each local update transaction, one for the grouped updatesets and one for the local transaction, for an expected limit of 50-60 local transaction commit per second depending on the time to complete the fsync (more on this below).

In contrast, Tashkent-MW (top curve: tashMW), where ordering and durability are united in the middleware, the certifier can group multiple commits in a single synchronous write. At 15 replicas, the certifier is grouping an average of 29 requests per fsync for a total throughput of 3657 req/sec, or 5.0x that of *Base*.

Tashkent-API (tashAPI curve), which combines ordering with durability in the database, also performs much better than *Base*, achieving 2240 req/sec, or 3.0x that of *Base*, but does not achieve the same throughput of Tashkent-MW. One possible reason for this difference is that in Tashkent-API there is an inherently longer round-trip latency for each client request because there are actually two disk writes in the path of each update transaction: one at the certifier for middleware recovery and one in the database for durability. This extra fsync delay extends the response time and depresses the throughput somewhat (*Base* is not likewise limited as serialization is by far its bottleneck). While durability in the certifier is necessary for middleware recovery, we run with it off to determine the extent of its effect, as shown by the fourth curve *tashAPInoCERT* for which the certifier performs normal certification but does not write to its disk. Without the delay at the certifier, the throughput increases to 2901 req/sec, but still does not rise to that of Tashkent-MW.

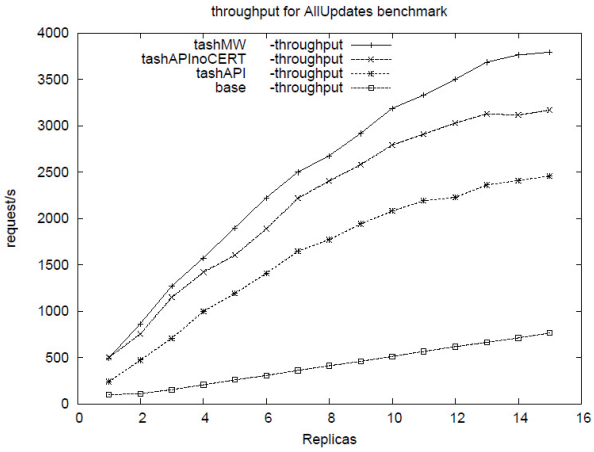

Figure 3: Throughput for AllUpdates (dedicated IO).

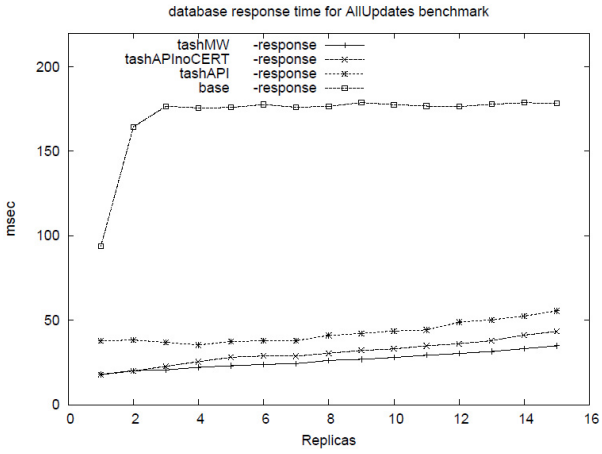

Figure 4: Response for AllUpdates (dedicated IO).

Another possible explanation for the difference could be the IO channel. With a single disk, the IO channel is shared between three different IO streams: reading the database pages, writing back database dirty pages, and logging. Since logging IO is in the critical path when durability is in the database, we simulate the effect of having a dedicated IO channel for logging. To do this we create a dedicated logging channel by putting the database in ramdisk.

Figure 3 has the throughput when logging does not contend with the other two streams. The relative behavior is similar as before. However, all the curves adjust up somewhat because less contention improves throughput, though the effect is minor as AllUpdates runs essentially from memory, resulting in very little activity for reading and writing database pages in the steady state.

The remaining performance difference between Tashkent-MW and Tashkent-API is due to that fact they write somewhat different log information and in different ways. PostgreSQL uses a multiprocess architecture, and the certifier uses lighter weight multithreaded IO design. We expect that the performance differences would be eliminated with proper changes to PostgreSQL.

Figure 4 shows the response times for AllUpdates with a dedicated IO channel. For *Base*, we see the jump of the response time between one replica and two replicas. This reflects two fsyncs per transaction in the replicated system: one for the grouped updateset and one for the commit of the local transaction. With 10 clients at each replica (having near 10 concurrent local requests at each replica), this adds an additional delay of 10 fsyncs to each client request (over 80 msec) at one replica and 20 fsyncs (over 160 msec) from two replicas and onwards. For the Tashkent systems, the delay increases slowly as more updatesets are processed at each replica.

In summary, under this worst-case update workload from AllUpdates, both systems Tashkent-MW and Tashkent-API show impressive scalability improvements when ordering and durability are united. Tashkent-MW and Tashkent-API have improvements of 5.0x/3.0x (shared IO) and 5.0x/3.2x (dedicated IO) over *Base*.

Certifier Scalability. At 15 replicas for the Tashkent-API system (as shown in figure 3), the certifier disk is less than 50% utilized and the certifier CPU utilization is below 20%, suggesting that the certifier is lightweight and can handle a higher workload.

9.3 TPC-B

TPC-B is also an update intensive benchmark, but the transactions include reads from the database as well. The throughput is shown in Figure 5 when the IO channel is shared. Again we see the same relative performance of the systems: Tashkent-MW highest performing (2.6x *Base*), *Base* lowest, and Tashkent-API (1.3x *Base*) in between the two.

To explore the difference between Tashkent-MW and Tashkent-API, we again remove the fsync in the certifier and show its improvement in the curve *tashAPInoCERT*, resulting in higher throughput.

We also plot the results in Figure 6 when the IO channel is dedicated (i.e., the database is in ramdisk). Here all curves are higher, indicating a higher activity for reading and writing database pages than in AllUpdates. However, there is still a significant difference between Tashkent-MW and Tashkent-API and the difference is not due to only the additional latency at the certifier as the curve *tashAPInoCERT* shows little additional throughput with the dedicated logging channel.

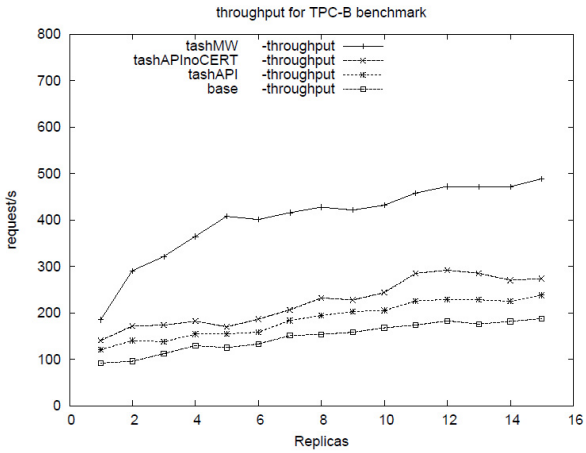

Figure 5: Throughput for TPC-B (shared IO).

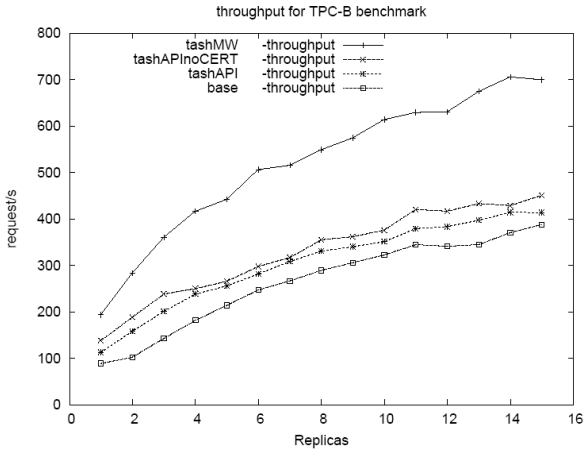

Figure 6: Throughput for TPC-B (dedicated IO).

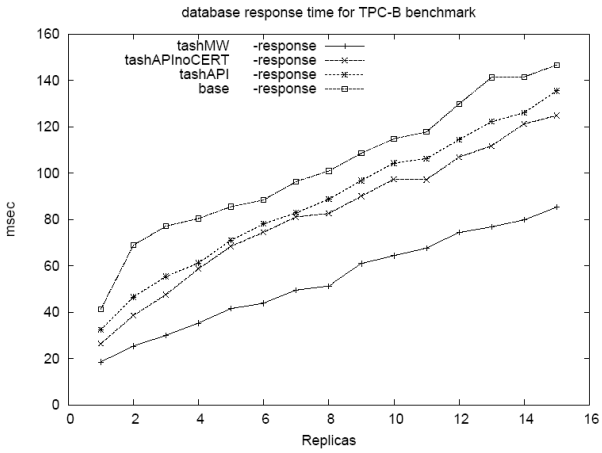

Figure 7: Response time for TPC-B (dedicated IO).

Unlike AllUpdates, all the performance difference is not wholly attributed to the way PostgreSQL performs IO. TPC-B generates "artificial" conflicts among updatesets, while AllUpdates does not. As detailed in Section 3, such artificial

conflicts among updatesets prevent Tashkent-API from submitting them concurrently to the database along with the commits of local update transactions, reducing opportunities to unite ordering and durability. When such an artificial conflict is detected under Tashkent-API, the updatesets must be submitted serially, waiting for at least one fsync, which gracefully degrades its performance towards that of *Base* as the conflict rate increases.

Finally, the response times in Figure 7 show a steady rise as replicas are added. This reflects the overhead of applying writesets at the replicas and is not due to load at the certifier. The certifier is only lightly loaded even at 15 replicas.

9.4 TPC-W

In Figure 8 we show the throughput of TPC-W with the shopping mix (20% updates) for a shared IO channel. There is no difference in throughput between Tashkent-API and *Base* (we discuss Tashkent-MW below). The reason is that at the maximum throughput of 240 tps, there are only 48 updates per second system wide. At 15 replicas, this means each replica generates only about 3 updates per second on average (requiring around 2*3 fsync() calls per second), much lower than what is needed to saturate the local logging channel. Thus, Tashkent-API has no opportunity to group updatesets in the synchronous writes. At very low update rates, separating ordering and durability does not create a bottleneck. Their response times are identical as well (we omit the response time graph for space limitation).

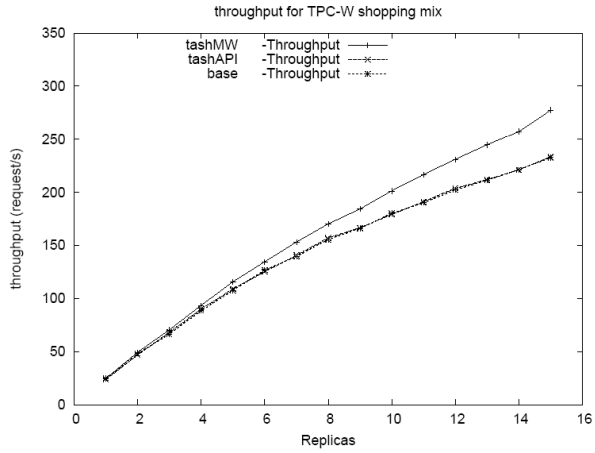

Figure 8: Throughput for TPC-W, shopping mix (shared IO).

The performance of Tashkent-MW is better than both Tashkent-API and *Base*. With the shared IO channel and the larger database, Tashkent-API and *Base* experience significantly higher critical path fsync delays due to non-logging IO congestion. Using a dedicated logging channel would alleviate this congestion for Tashkent-API and *Base*. Unfortunately, we were unable to run TPC-W in ramdisk due to its large size. If we could, we expect both Tashkent-API and *Base* would match the performance of Tashkent-MW.

9.5 Recovery for Tashkent-MW

Since Tashkent-MW requires database checkpoints for its recovery, we report its cost but only for TPC-W (PostgreSQL, database size is ~700 MB) since the other benchmarks are small and are able to recover much faster. In a 15 replica system, a replica requires about 230 seconds to dump a complete copy of the database *while it continues processing transactions*. That replica's throughput degrades by 13% during the 230 seconds. After a crash, restarting a replica requires 140 seconds to restore from the dump file. The missing writesets (updatesets) that occurred since the dump was created (this includes writesets during the down time of the replica) can be applied to the database at a rate of 900 writesets per second. At 15 replicas, the rate of updates is 56 writesets per second (20% of 280 tps for the shopping mix). For H hours of down time, the total recovery time is approximately $(2.3 + 3.7H)$ minutes. For the certifiers, the growth rate of the history log for 15 replicas is 201,600 writesets per hour at an average of 275 bytes per writeset, or 56 MB per hour. The log must be transferred from one of the backups and is essentially a file transfer and takes about 1 second in our LAN.

9.6 The affects of aborts

Here we show that the throughput of the two Tashkent systems degrades gracefully in the presence of aborts. Aborts can occur due to (real) conflicts between concurrent transactions whose writesets overlap (i.e., write-write conflicts). The certifier detects these conflicts and responds with an abort to the later occurring request. We show the effects of aborts using the AllUpdates benchmark because TPC-B and TPC-W have very few (real) conflicts and subsequently low abort rates.

In AllUpdates, we can force a system wide abort rate by having the certifier randomly abort requests at a given rate. If a certification request is selected to be aborted, the abort occurs after the full certification check so that all computational overhead at the certifier is incurred.

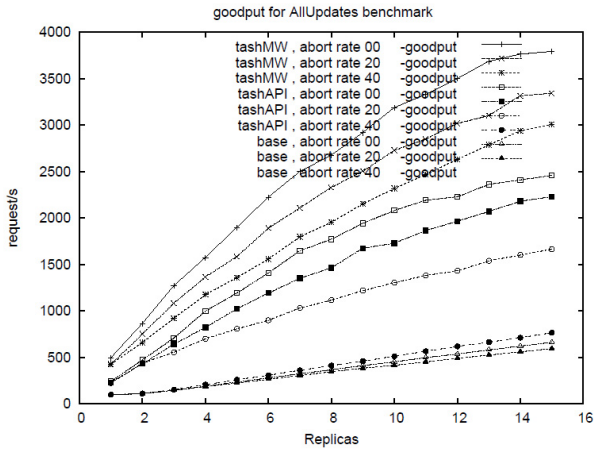

Figure 9. Certifier goodput under different abort rates (dedicated IO).

In Figure 9 we use *exaggerated* abort rates to demonstrate how the "goodput" (the throughput of non-aborted requests only) is affected. On the three systems, we force three different abort rates: 0%, 20%, and 40%. The top three lines are Tashkent-MW, the middle three are Tashkent-API, and the bottom three are for *Base*. In all systems, the throughput degrades gracefully and proportionally with the increasing abort rates.

9.7 Summary

We show that under high update workloads separating durability from ordering can create a significant scalability bottleneck. Under these conditions, combining these two functions, either in the middleware or in the database, alleviates this bottleneck and can greatly improve performance. We also demonstrate that the Tashkent-MW solution may be more robust than Tashkent-API if the workload has significant artificially conflicting writes that would force Tashkent-API to serialize its updates and behave more like *Base*. Furthermore, when the update rate is low, as in TPC-W, combining ordering and durability does not improve performance because at low rates the IO for durability is not a bottleneck.

10 Related Work

In this section, we first discuss related work on serializability in SI and GSI. Then, we contrast Tashkent to the two main types of database replication: eager replication and lazy replication. Finally, we compare Tashkent to replicated systems that use snapshot isolation.

10.1 SI, GSI, and Serializability

Serializability [17, 2] is the main database correctness criteria. Snapshot isolation (SI) provides a weaker form of consistency than serializability. Several researchers [5, 7, 4] have recently demonstrated that, under certain conditions on the workload, transactions executing on a database with SI produce serializable histories. Those conditions hold for many applications (including TPC-W, TPC-C, and TPC-B, TPC-A benchmarks). They argue that even if these conditions do not hold for an application, transaction templates in the application can be easily (automatically) modified to be serializable under SI if the transaction templates are known, as in Web applications. In practice, database developers understand SI and are capable of using SI serializably.

The notion of generalized snapshot isolation (GSI) is introduced in [3]. Generalized snapshot isolation preserves SI isolation guarantees. In particular, the authors prove that if an application runs serializably under SI, it runs serializably under GSI. However, no implementation is reported. In our experimental results section, the workload assigned to the databases is serializable under GSI as previously indicated [3, 4].

10.2 Database Replication

Gray et al. [9] have classified database replication into two schemes: *eager* and *lazy*. Eager replication provides consistency, usually at the cost of limited scalability. Lazy replication increases performance by allowing replicas to diverge, possibly exposing clients to inconsistent states of the database.

The two Tashkent systems use GSI and avoid write-write and write-read conflicts globally while appearing as a single snapshot isolated database to clients. In addition, update reconciliation is not needed and the effects of committed update transactions are not lost in the system due to a replica failure.

10.3 Replication in Snapshot Isolated Databases

Kemme et al. [12] discussed how to implement different isolation levels (including serializability and snapshot isolation) in replicated databases using group communication primitives. In addition, they implemented Postgres-R [11], and integrated the replica management with the database concurrency control [24, 14] in Postgres-R(SI).

Postgres-R(SI) uses a replicated form of snapshot isolation. In contrast to Postgres-R(SI) [24, 14], the two Tashkent systems have the key feature of uniting ordering and durability, they *never* block read-only transactions and, in addition, Tashkent-MW is a *pure* middleware solution. Postgres-R(SI) commits update transactions sequentially, which limits scalability if durability is guaranteed by the database but ordering is in the middleware, requiring one fsync for each commit. The Tashkent systems remove this fsync bottleneck and still provide durability. In Postgres-R(SI) the replication protocol is tightly coupled with the concurrency control. For example, their replication middleware accesses PostgreSQL lock tables and replicas map internal transaction IDs to global transaction IDs. The validation function in Postgres-R(SI) (similar to our certification) is replicated with each database; whereas in our work the certifier and the replica component can be replicated asymmetrically to enhance certifier availability and replica performance. In large scale systems under heavy loads, co-locating the certifier with each database replica marginally improves the certifier availability (over asymmetric replication) but makes the certifier compete on resources with the database at each replica, possibly reducing replica performance.

Plattner et al. [18] presented Ganymed, a master-slave scheme for replicating snapshot isolated databases in clusters of machines. In Ganymed all update transactions are handled by a master and read-only transactions are distributed among slaves. Ganymed serializes commits on the master using one fsync for each commit as well as the commits at the slaves. Both the master and the slaves could benefit from the extended API described in this paper. However, a single master system such as Ganymed is limited to the throughput of a single machine to fully process all update transactions. The Tashkent systems process most of each update transaction locally at a replica and only certify the core writesets at the certifier. Certifying the core writesets is an order of magnitude less work than executing the transaction. Thus, the Tashkent designs distribute much of the update transaction workload.

11 Conclusions

This paper identifies a limitation to scalability concerning durability and commit ordering in replicated database designs in which all replicas agree on which update transactions commit and on the order of their commits. We analyze the dependency between maintaining the global commit order and the durability. By uniting durability with ordering, the two actions be done in one phase, which greatly improves scalability.

We use two example solutions that unite durability and ordering. We present *Base* a traditional replicated database system where durability and commit ordering are separate. Then, we present the design and implementation of Tashkent-MW and Tashkent-API, where durability and ordering are united. In Tashkent-MW, durability is united with ordering in the replication middleware and provides a *pure* middleware solution for high-performance replicated databases.

In Tashkent-API, we detail how to extend the standard database API to permit passing the commit order information to the database from the middleware. In this solution, additional care must be taken to ensure the middleware proxy does not generate artificial conflicts between updatesets to be submitted

concurrently. This constraint can prevent some opportunities to combine durability and ordering. In the presence of these artificial conflicts (an artifact of the conflict profile of the workload) the performance of Tashkent-API degrades gracefully to that of *Base*.

We implement the Tashkent systems on top of PostgreSQL and assess their performance relative to *Base*. At low update rates when durability is not a bottleneck, the Tashkent systems perform similar to *Base*, as to be expected. However, under high update transaction loads uniting durability and ordering becomes significant; we show that both versions of Tashkent greatly improve scalability and outperform *Base* by factors of 5 and 3 times , respectively, in throughput and with lower response times.

12 Acknowledgments

We thank Prof. Willy Zwaenepoel from EPFL for his insightful suggestions and valuable feedback. We also thank the anonymous reviewers for their constructive feedback. This research was partially supported by the Swiss National Science Foundation grant number 200021-107824: System Support for Distributed Dynamic Content Web Services.

References

[1] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In proceedings of the SIGMOD International Conference on Management of Data, May 1995.

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

[3] Elnikety, S., F. Pedone, and W. Zwaenepoel, Database Replication Using Generalized Snapshot Isolation. IEEE Symposium on Reliable Distributed Systems (SRDS 2005), Orlando, Florida, Oct. 2005.

[4] Alan Fekete. Allocating Isolation Levels to Transactions. ACM Sigmod, Baltimore, Maryland, June 2005.

[5] Alan Fekete. Serialisability and snapshot isolation. In proceedings of the Australian Database Conference, pages 201–210, Auckland, New Zealand, January 1999.

[6] L. Frank. Evaluation of the basic remote backup and replication methods for high availability databases. Software Practice and Experience, 29:1339–1353, 1999.

[7] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. In proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, pages 173–182, June 1996.

[8] Lei Gao, Mike Dahlin, Amol Nayate, Jiandan Zheng, and Arun. Iyengar. Application specific data replication for edge services. In Proceedings of the twelfth international conference on World Wide. Web, pages 449–460. ACM Press, 2003.

[9] J. N. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal (Canada), June 1996.

[10] K. Jacobs. Concurrency control, transaction isolation and serializability in SQL92 and Oracle7. Technical report number A33745, Oracle Corporation, Redwood City, CA, July 1995.

[11] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In proceedings of 26th International Conference on Very Large Data Bases (VLDB 2000), Cairo, Egypt, September 2000.

[12] Bettina Kemme and Gustavo Alonso. A suite of database replication protocols based on group communication primitives. In proceedings 18th International Conference on Distributed Computing Systems (ICDCS), Amsterdam, The Netherlands, May 1998.

[13] L. Lamport. The Part-time Parliament. ACM Transactions on Computer Systems, 16(2):133-169, May 1998.

[14] Y. Lin, B. Kemme, M. Patiño-Martínez, R. Jiménez-Peris. Middleware based Data Replication providing Snapshot Isolation. ACM Int. Conf. on Management of Data (SIGMOD), Baltimore, Maryland, June 2005.

[15] Oracle parallel server for windows NT clusters. Online White Paper.

[16] Data Concurrency and Consistency, Oracle8 Concepts, Release 8.0: Chapter 23. Technical report, Oracle Corporation, 1997.

[17] Christos Papadimitriou, The theory of database concurrency control. Computer science press. 1986.

[18] Christian Plattner, Gustavo Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In proceedings of the 5th ACM/IFIP/USENIX International Middleware Conference, Toronto, Canada, October 2004.

[19] PostgreSQL, SQL compliant, open source object-relational database management system. http://www.postgresql.org/.

[20] C. Pu and A. Leff. Replica control in distributed systems: an asynchronous approach. SIGMOD Record (ACM Special Interest Group on Management of Data), 20(2):377–386, June 1991.

[21] Robbert van Renesse, and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. Sixth Symposium on Operating Systems Design and Implementation (OSDI '04). USENIX Association, (San Francisco, California, December 2004), 91—104.

[22] SCHNEIDER, F. B.. Implementing fault-tolerant services using the state machine approach: a tutorial. In ACM Computing Surveys. 22 (4):299–319, December 1990.

[23] Transaction Processing Performance Council – http://www.tpc.org/.

[24] Shuqing Wu and Bettina Kemme: Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation. In proceedings of International Conference on Data Engineering (ICDE), April 2005.

[25] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000), Taipei, Taiwan, April 2000.