

# 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 functions of ordering the transaction commits and making the effects of transactions durable are performed in one single action, namely the writing of the commit record to disk. For efficiency many of these writes are grouped into a single disk operation. In replicated databases in which all replicas agree on the commit order of update transactions, these two functions are typically separated. Specifically, the replication middleware determines the global commit order, while the database replicas make the transactions durable.

The contribution of this paper is to demonstrate that this separation causes a significant scalability bottleneck. It forces some of the commit records to be written to disk *serially*, where in a standalone system they could have been grouped together in a single disk write. Two solutions are possible: (1) move durability from the database to the replication middleware, or (2) keep durability in the database and pass the global commit order from the replication middleware to the database.

We implement these two solutions. Tashkent-MW is a pure middleware solution that combines durability and ordering in the middleware, and treats an unmodified database as a black box. In Tashkent-API, we modify the database API so that the middleware can specify the commit order to the database, thus, combining ordering and durability inside the database. We compare both Tashkent systems to an otherwise identical replicated system, called *Base*, in which ordering and durability remain separated. Under high update transaction loads both Tashkent systems greatly outperform *Base* in throughput and response time.

## Categories and Subject Descriptors

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

## General Terms

Measurement, Performance, Design, Reliability.

## Keywords

Database replication, Generalized snapshot isolation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

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

Copyright 2006 ACM 1-59593-322-0/06/0004...\$5.00.

## 1 Introduction

Database replication is a cost-effective technique to improve performance and to enhance availability. We are concerned here with fully replicated designs in which any transaction, read-only or update, may execute on any (single) replica and all replicas agree on the commit order of the updates.

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 committing transactions in a global order and relying on an off-the-shelf database to make transactions durable (*i.e.*, write their effects to disk) requires commit records to be written to disk *serially*, a significant scalability bottleneck.

The root cause is the separation in replicated systems of commit ordering from the disk writes that ensure durability. In standalone databases, these two functions are performed together, permitting group-commits to minimize the number of synchronous writes to disk. In replicated databases, however, ordering is determined in the replication middleware – rather than at the database – to ensure a consistent global order. Durability has typically been left in the database.

We propose two approaches to unite these functions. The first approach is to move durability to the middleware layer where ordering is determined. This solution can be pure middleware, so that it can be used with off-the-shelf databases. The second approach is to extend the database API so that the replication middleware can specify a commit order for update transactions, thus combining durability and ordering in the database.

We implement instances of both approaches, called *Tashkent-MW* and *Tashkent-API*, respectively, and compare them to an otherwise identical replication system, called *Base*, in which ordering and durability remain separated. All systems use *generalized snapshot isolation* (GSI) [3] for concurrency control, and are derived from PostgreSQL [19].

Both Tashkent systems greatly improve scalability under high update transaction loads. For instance, at 15 replicas under an update-intensive workload the Tashkent systems outperform *Base* by factors of five and three in throughput, while also providing lower response times.

This paper makes the following contributions:

1. We identify the dependency between durability and ordering and demonstrate the performance impact if they are not united.
2. We design and implement Tashkent-MW, a pure middleware solution that unites durability and ordering in middleware.

3. We extend the database API and implement Tashkent-API which unites durability and ordering in the database.
4. We assess both systems for their benefits and costs.

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 present Tashkent-MW and Tashkent-API. We discuss the implementation of the replication middleware in Section 6. In Section 7 we discuss fault-tolerance of the systems. We discuss the replication middleware interface to PostgreSQL in Section 8. In Section 9 we experimentally compare the three systems and analyze their relative performance. Section 10 contrasts this research to related work. Finally, in Section 11 we summarize the main conclusions.

## 2 Background

*Snapshot Isolation* (SI) [1] is a concurrency control model for centralized multi-version databases. In SI, when a transaction begins it receives a *snapshot* of the database. After the snapshot is assigned, it is unaffected by concurrently running transactions. A read-only transaction reads from the snapshot and can always commit. An update transaction 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 [1, 7, 10, 16]. SI is weaker than serializability, but in practice most applications run serializably under SI, including the most widely-used database benchmarks TPC-A, TPC-B, TPC-C, and TPC-W. SI has attractive performance properties. Most notably, read-only transactions never block or abort, and they never cause update transactions to block or abort.

*Generalized Snapshot Isolation* (GSI) [3] extends SI to replicated databases such that the performance properties of SI in a centralized setting are maintained in a replicated setting. In particular, read-only transactions do not block or abort, and they do not cause update transactions to block or abort. Both of these properties are important for scalability. In addition, workloads that are serializable under SI are also serializable under GSI [3].

Informally, a replica using GSI works as illustrated in Figure 1. When a transaction starts at a database replica, 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*. A writeset captures the minimal set of actions necessary to recreate a transaction's modifications. If the writeset is empty (i.e., the transaction is read-only), the transaction commits immediately. Otherwise, a certification check is performed to detect write-write conflicts among update transactions in the replicated system. If no write-write 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 replicas to update their state. We refer to writesets of remote transactions during the propagation phase as *remote*

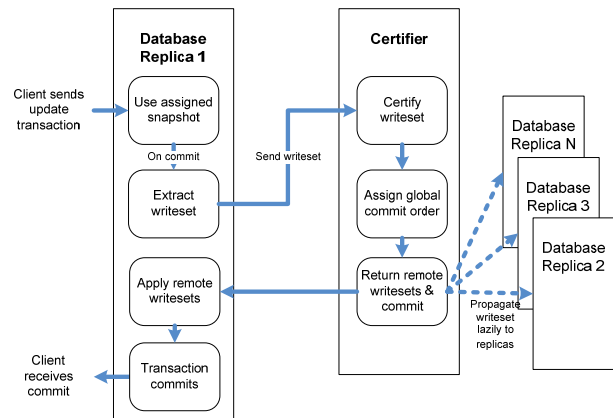


Figure 1: Processing an Update Transaction in GSI

*writesets*. Before committing a local update transaction, if the replica is not up-to-date, it must apply remote writesets to bring that replica's state up-to-date.

## 3 How Durability and Ordering Affect Scalability

In the GSI algorithm, there is a dependency between maintaining the global commit order and writing state to disk for durability. Identifying this dependency is one of the core contributions of this paper. In this section, we explain the problem and its underlying causes via an example, and discuss its impact on scalability.

**Centralized SI database.** The initial case is a centralized system to which we will contrast a replicated system. In a centralized system, if clients concurrently submit two update transactions  $T_4$  and  $T_9$  – whose writesets,  $W_4$  and  $W_9$ , do not conflict – the database can commit them in any order:  $T_4$  then  $T_9$ , or  $T_9$  then  $T_4$ . However, to guarantee durability, each commit requires a disk write, a disk write for  $T_4$  and another for  $T_9$ . The IO subsystem can group the commits of  $T_4$  and  $T_9$  into a single disk write, which greatly improves performance.

**One GSI replica.** Next, consider a replicated GSI database consisting of one replica and one certifier, both at version zero. If clients submit  $T_4$  and  $T_9$  concurrently to the replica, it executes them and sends their writesets,  $W_4$  and  $W_9$ , 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  $W_4$  at version 1 and  $W_9$  at version 2, and sends these results back to the replica.

Upon receiving the responses from the certifier, the replica must commit  $T_4$  first to reach version 1, then commit  $T_9$  to reach version 2, the same sequence that the certifier determined. Under GSI, the replica cannot change this order. Otherwise, a new transaction could receive a snapshot containing the effects of  $T_9$  but not  $T_4$ , a snapshot that never existed globally (i.e., at the certifier). A typical database offers no mechanism to specify a commit order externally. Yet the middleware must not allow the replica to swap the order in which the commits of  $T_4$  and  $T_9$  occur. Without an 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 to commit  $T_4$  and another for  $T_9$ , 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:  $T1, T2, T3, T4, T5, T6, T7, T8, T9$ . Under GSI, the replica must observe this total order. Not only must the replica commit transaction  $T4$  before  $T9$ , but the remote transactions  $T1, T2, T3$  must commit before  $T4$ , and  $T5, T6, T7, T8$  before  $T9$ . In a naïve implementation, the replica would go through the version sequence: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, which would be 9 synchronous disk writes, but this can be improved, as follows.

**Grouping remote writesets.** We batch the writesets of several remote transactions by combining their effects into one transaction  $T1\_2\_3$  with writeset  $\{W1, W2, W3\}$ , and similarly another transaction  $T5\_6\_7\_8$  with writeset  $\{W5, W6, W7, W8\}$ . The replica version follows the 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 the commits of  $T1\_2\_3, T4, T5\_6\_7\_8, T9$  concurrently to a standard database and force committing  $T1\_2\_3$  first,  $T4$  second,  $T5\_6\_7\_8$  third, then finally  $T9$ . Databases do not provide such low level mechanisms to the client interface.

The replication middleware cannot allow the database to commit the transactions in any order. The commits must follow the global order because 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. Furthermore, allowing transactions to commit in any order would require the middleware to block the start of all 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. Therefore, we propose two solutions for uniting ordering and durability.

**Solution 1: Move durability to the middleware.** One solution is to have the middleware, which decides the ordering, be responsible for making transaction modifications durable. The middleware can then batch all available writesets at certification. In the above example, the middleware batches all nine writesets into a single disk write. At the replicas, commits become in-memory actions and serializing them is not a performance issue.

**Solution 2: Pass the ordering information to the database.** The database API interface is extended such that the middleware can specify the commit order. With this API, the replication middleware submits all transactions concurrently to the database with their required commit order. In the above example, the database makes  $T1\_2\_3, T4, T5\_6\_7\_8,$  and  $T9$  durable in one disk write and commits them in the correct order.

The key insight is that in both solutions durability is united with ordering. Regardless of whether the two are united in the replication middleware or in the database, 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 the writesets of

update transactions, and (3) has the ability to enable/disable synchronous writes to disk.

## 4 Architecture and Design of Base

*Base* represents a traditional replicated database 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 unite ordering and durability. We discuss fault tolerance and recovery for each of the three replicated designs in Section 7.

### 4.1 Architecture

*Base* is a replicated database system that uses GSI for concurrency control. *Base* 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 certification. Replicas communicate only with the certifier component, not directly with each other. The certifier certifies update transactions from all replicas and orders them.

The *Base* design is a pure middleware solution since no modification to the database source code is required. Attached to each replica is a *transparent proxy* that intercepts database requests. The proxy appears as the database to clients, and appears as a client to the database. The proxies and certifiers are the *replication middleware*.

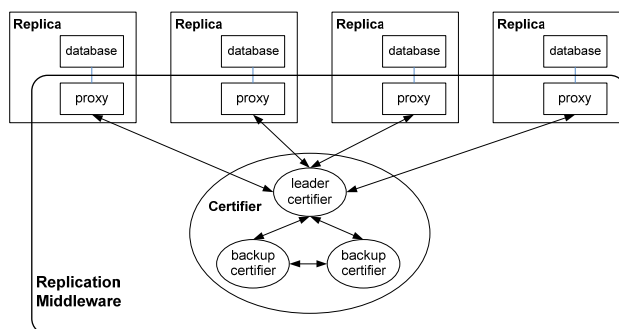


Figure 2 - Architecture of Base

Figure 2 shows the architecture of the *Base* system. The two main components, database replica and certifier, are replicated asymmetrically with different replication degrees. Database replicas are replicated mainly for performance, whereas the certifier is replicated mainly for availability. For this study, we simply assume a separate certifier component which is itself replicated, though our conclusions apply to other configurations. For example, the certifier component could be implemented via an atomic broadcast mechanism incorporated into the proxy at every replica.

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 Transactions

Each transaction is assigned a snapshot. The transaction operations are executed locally against the snapshot. The commit of a read-only transaction is simple as it always commits successfully. However, processing the commit of an update transaction is more complex and we informally present here the steps under GSI. We provide additional implementation details in Sections 6 and 8.

We use the following terminology. We use *version* to count database snapshots. The version at a particular database replica is called `replica_version`. The database starts at version zero, i.e., in 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 only for update transactions).

**Certification.** When an update transaction attempts to 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 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 commit order. The log is necessary to allow the certifier to recover because we use the crash-recovery model. The state of any replica is always a consistent prefix of the certifier's log. Writeset intersection is a quick operation and writing to the persistent log is efficient because multiple writesets can be batched. Thus, certification is lightweight.

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

**Durability.** The durability function is performed in the databases. Since the certifier determines the global order of commits, proxies commit update transactions and remote writesets serially to ensure the same global order is followed at each replica.

## 5 Tashkent-MW and Tashkent-API

Tashkent-MW and Tashkent-API are derived from *Base* to combine durability and commit ordering.

### 5.1 Tashkent-MW

To unite durability with ordering in the middleware, we disable synchronous writes that guarantee durability in the database replicas. Instead, the replication middleware logs the database modifications (i.e., writesets) for durability. However, the certifier already logs this information via its persistent log to allow certifier recovery. Thus, it is relatively simple to transform the *Base* design into Tashkent-MW, because enabling/disabling synchronous writes is a standard feature in many databases. Furthermore, since *Base* is a pure middleware solution, so then is Tashkent-MW.

The commit operations at each database replica are still serial, but they are now fast in-memory operations. System throughput and scalability are greatly improved because the durability function is performed at the certifier, which efficiently groups the writesets into a much smaller number of synchronous writes. There is, however, a cost to moving durability out of the databases and it appears in the recovery procedure. We address recovery in Section 7.

### 5.2 Tashkent-API

To unite ordering with durability in the database, the API available to the middleware is extended so that 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.

With the new API, upon receipt of the *commit decision* from the certifier, the Tashkent-API proxy forwards to the database both the commit command and the version number returned from the certifier. Similarly, remote writesets are applied in separate transactions with their appropriate commit sequence numbers. The proxy applies remote writesets and local commits to the database concurrently. Thus, the database can group the commit records as usual for efficient disk IO, but internally the database enforces the supplied commit ordering.

This extended API is a simple interface yet powerful; its use should be restricted to the replication middleware. 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, it is not always possible for the middleware to submit transaction commits concurrently. At these times, some commit operations must be serialized, and this reduces the performance gain. This section describes the condition for which a commit must be serialized and how to detect if this condition arises.

Conflicts may arise when committing two different local update transactions. Should their accompanying two remote writesets modify a shared element, they generate an "artificial" local write-write conflict at the database replica if the middleware submits these commits concurrently. Under snapshot isolation, a write-write conflict results in one of the transactions being aborted. Therefore, the middleware must avoid creating these "artificial" conflicts.

The root of the problem is that the artificially conflicting remote writesets, which were generated by remote transactions that *did not* run concurrently, now appear as concurrent to the database replica when applied. This situation can only occur in Tashkent-API, not in Tashkent-MW or *Base* as they serialize commits. We illustrate this case with a concrete example and present a detection scheme.

**Artificial Conflict Example.** Figure 3 shows two local *concurrent* transactions,  $T_{44}$  and  $T_{46}$  at a replica and two remote writesets,  $W_{43}$  and  $W_{45}$ , 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 is  $W_{44}$ .  $T_{44}$  commits as version 44 globally. But before  $T_{44}$  can commit, remote writeset  $W_{43}$  is received from the certifier and applied to the replica as local transaction  $T_{43}$ .  $T_{43}$  must commit before  $T_{44}$  commits as explained in Section 3 to follow the global commit order.

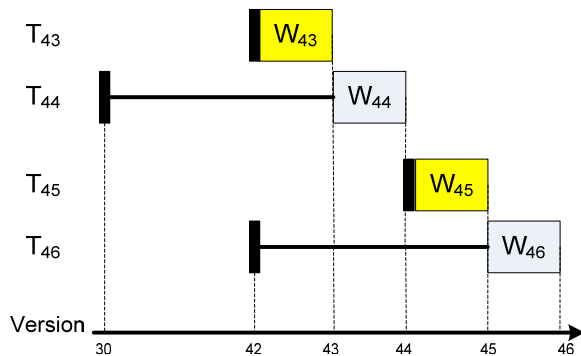


Figure 3: Example of concurrent commits

The proxy can safely submit the commits of transactions  $T_{44}$  and  $T_{46}$  concurrently at the database along with transactions  $T_{43}$  and  $T_{45}$  if none of the remote writesets ( $W_{43}$  and  $W_{45}$ ) 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 desired behavior.

If an artificial conflict exists between  $T_{43}$  and  $T_{45}$ , e.g.,  $W_{43}$  sets  $x=17$  and  $W_{45}$  sets  $x=39$ , then this artificial conflict is detected and  $T_{45}$  will be serialized waiting for  $T_{43}$  to commit. In this case, at least two disk writes are needed. For example, the proxy submits transaction  $T_{43}$  and the commit of  $T_{44}$  concurrently, waits for the acknowledgement from the database. Then the proxy submits transaction  $T_{45}$  and the commit of  $T_{46}$  concurrently. The best the database can do in this case 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. If artificial conflicts between remote writesets are frequent, then in the worst-case all commits are forced to be serialized and Tashkent-API degrades towards the performance of *Base*.

**Artificial Conflict Detection.** The proxy in Tashkent-API needs to know if it is safe to apply a remote writeset concurrently with previous remote writesets. Concurrency is safe if intersection testing is null between the remote writeset and the previous remote writesets (e.g.,  $W_{43}$  and  $W_{45}$  do not modify a shared item). The certifier may already have done this intersection testing in the normal certification process at their original commit time if the remote writesets were from concurrent transactions.

Our solution is to have the proxy tell the certifier how far back (i.e., what prior version) it would like the remote writesets to have been tested for write-write conflicts (i.e., intersection testing). If one or more of the remote writesets has not already been tested that far back, the certifier performs additional intersection testing for those writesets. With the commit/abort decision, the certifier also returns the version to which the remote writesets are conflict-free. The proxy uses this information to determine which remote writesets can be submitted concurrently. If two remote writesets conflict, then the proxy must commit the earlier writeset first before submitting the latter one.

The version the proxy submits to the certifier need only be the current version of the database as no remote writesets will be returned prior to this version (recall that remote writesets carry updates the replica has not seen yet). The proxy stores the current version of a replica in `replica_version` and it is this version the proxy submits with a transaction to the certifier.

In the example of Figure 3, to certify  $T_{46}$ ,  $W_{46}$  is certified back to version 42 ( $T_{46}.tx\_start\_version$ ) in normal certification. Next, assuming that the accompanying remote writeset  $W_{45}$  was only certified to version 44 at the time of its commit, then  $W_{45}$  is also additionally certified back to `replica_version=42` if it is to be committed concurrently with  $T_{43}$  and  $T_{44}$ . If there is a conflict then the proxy delays submitting  $W_{45}$  until the conflicting transaction  $T_{43}$  commits.

The overhead of the additional certification checks at the certifier is minimal. The certifier records for each writeset the point to where it has been (further) certified and avoids repeated checks for responses to other replicas.

## 6 Implementation of the Replication Middleware

In this section, we provide the pseudo-code of GSI and show how it is implemented as pure middleware. In Section 8 we discuss the mechanisms used by the proxy to interface to PostgreSQL.

### 6.1 Certifier

The certifier maintains (1) a persistent log recording tuples of each writeset and its commit version (`writeset`, `tx_commit_version`) for all committed update transactions and (2) the global system version (`system_version`). A certification request provides the certifier with the writeset and start version of the transaction (`T.writeset`, `T.tx_start_version`). The certifier returns the remote writesets, the certification result (commit or abort), and the commit version of the transaction (`tx_commit_version`). The pseudo code of actions at the certifier for a transaction  $T$  is the following:

```

On a certification request for transaction T having
(T.tx_start_version, T.writeset):
1. The input T.writeset is tested for intersection against entries in
LOG whose tx_commit_version is greater than
T.tx_start_version. An intersection occurs if 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 }
   ELSE decision ← "abort".
3. The output of the procedure contains:
   • the remote writesets that the replica has not received
     yet between tx_start_version and
     T.tx_commit_version,
   • decision (either "commit" or "abort"),
   • T.tx_commit_version.

```

We implement the certifier 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, which makes the certifier very efficient at batching all outstanding writesets to disk via a single *fsync* call.

## 6.2 Transparent Proxy

A proxy in front of each database intercepts 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 remote writesets. The pseudo code for actions at the proxy is the following:

**A- On proxy intercepting *T*'s BEGIN:**

1. *T* receives a snapshot of the database,
2.  $T.tx\_start\_version \leftarrow replica\_version$ .

**B- On *T* read and write operations:**

1. Read and write operations are executed on *T*'s snapshot.

**C- On proxy intercepting *T*'s COMMIT:**

1. *T.writeset* is extracted.
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. Receive three outputs from certification:
  - (a) remote writesets,
  - (b) "commit" or "abort",
  - (c)  $T.tx\_commit\_version$ .
4. Replica applies the remote writesets (a) in their own transaction.
5. **IF** the second output (b) is "commit", **THEN** { *T* commits, and  $replica\_version \leftarrow T.tx\_commit\_version$  (c)}. **ELSE** *T* aborts.

The proxy intercepts database requests without any modification to the database as explained below. The labels refer to the corresponding parts of the pseudo code.

**[A1] Intercepting BEGIN.** The proxy 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.** The proxy 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] Receiving certifier output.** The proxy receives remote writesets, the certification decision and  $T.tx\_commit\_version$ .

**[C4] Remote writesets.** Remote writesets are first stored in a memory-mapped file, called *proxy\_log*, a new transaction is started containing the remote writesets, the transaction is sent to the database, and 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*. If the certifier decision is commit, the proxy

forwards "COMMIT" to the database and awaits the reply. In Tashkent-API,  $T.tx\_commit\_version$  is added as a parameter to the "COMMIT". The database commits the transaction. Then, the proxy updates  $replica\_version = T.tx\_commit\_version$ .

Steps [C4] and [C5] are serialized both in *Base* and Tashkent-MW. The proxy applies the remote writesets, 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 a performance optimization where the proxy performs partial certification against a local copy of the remote writesets it has seen so far. By locally certifying a writeset, the proxy increments the effective start version of a transaction which reduces the work at the certifier. If local certification fails, the transaction is aborted, obviating unnecessary processing at the certifier.

**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 under GSI because certification is correct as long as the new transaction is labeled with a version that is the same or earlier than its actual version. The certifier still detects all write-write conflicts.

**Bounding Staleness.** Remote writesets are sent in response to update transaction commits. If a replica does not receive an update transaction for a period of time (e.g., a few seconds), its proxy proactively requests remote writesets from the certifier to bring the replica up-to-date.

**Proxy Implementation.** The proxy is implemented as a multi-threaded Java program. The proxy provides a JDBC interface to clients, and it uses a JDBC driver suitable for the database. All access to the replicated tables must go through the proxy, and not directly to the database.

## 7 Fault Tolerance

In all systems, 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 as 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.

### 7.1 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 (correctness of 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 the 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 the WAL.

Thus, the 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*). Databases typically offer one of two options for disabling WAL activity. This generates two cases on how recovery is performed.

**Case 1: Disable both integrity and durability.** 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. In SI databases taking a database copy is logically equivalent to a reading 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 remote writesets 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.

**Case 2: Disable only durability.** The second case is where databases offer the option of “enable WAL synchronous writes, but disable WAL synchronous writes on commits of update transactions”. This option guarantees data integrity but does not guarantee durability. In other words, 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-MW does not need to take database copies. The database uses its recovery mechanism and then the middleware applies the necessary remote writesets to bring the database up-to-date.

## 7.2 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. The middleware proxy re-applies 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.

If the set of uncommitted update transactions are not known (e.g., the proxy crashes with the database), the proxy can obtain the missing writesets from the certifier’s log and apply them. Reapplying writesets in the global order is always safe. At the end of this step, the proxy knows the version of the database. After the database recovers and the proxy updates the database, the replica resumes normal operations.

## 7.3 Certifier Replication and Recovery

The certifier is identical in all three systems. The certifier state is replicated for availability across a small set of nodes using Paxos [13]. The replication algorithm uses a *leader* elected from the set of certifiers. The leader 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. When a majority of certifiers reply, 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 an update via a state transfer from another up certifier, participates in electing a new leader (if necessary), and logs certification requests to disk.

## 8 Middleware Interface to PostgreSQL

### 8.1 Proxy-Database Interface

The interaction between the proxy and database is database-specific. We describe here how to interface to PostgreSQL. For another database, the specifics would change, but the underlying mechanisms are common in most databases.

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

In PostgreSQL, 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 to create the backup copy. 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 sequentially the remote writesets and the previously aborted transaction. If soft recovery fails, the proxy performs normal system recovery.

## 8.2 Deadlock Avoidance

Because PostgreSQL uses write locks, a deadlock may develop between the writeset of a local transaction and the writeset from a remote transaction. This deadlock scenario exists in all three designs *Base*, Tashkent-MW, and Tashkent-API.

**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 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  $T1$  holds a write-lock on  $x$  and  $T2$  holds a write-lock on  $y$ . Then,  $T2$  tries to update  $x$  and  $T1$  tries to update  $y$ . Neither  $T1$  nor  $T2$  can proceed as each is waiting for the other.

**Deadlock between a Local Writeset and a Remote Writeset.** In a replicated system, a similar deadlock situation can arise if  $T2$  is on a remote replica, has been certified, and is being applied as a remote writeset to the replica where  $T1$  is currently executing and holding a lock. Unlike the standalone system, in the replicated system  $T2$  must eventually be permitted to commit at the replica and  $T1$  will eventually be aborted either at the replica or at the certifier.

Some databases allow tagging transactions with priorities. If such a mechanism is available then avoiding a deadlock is straight forward: we mark remote writesets 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 remote writesets and local update transactions (we run soft recovery if PostgreSQL aborts a remote writeset), or (b) detect and prevent deadlocks eagerly in the middleware as an optimization using *eager pre-certification*.

**Eager Pre-certification.** The proxy can avoid deadlocks by eagerly detecting write-write conflicts between remote writesets and local writesets. Conceptually, we leverage the local certification functionality at the proxy (Section 6.2) to eagerly certify every write in a transaction as it occurs against the pending remote writesets. Similarly, remote writesets 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 remote writeset to be executed.

Both local certification and eager pre-certification do not increase the total number of low-level certification comparisons. They change *when* (earlier than the commit time) and *where* (on the replica rather than on the certifier) some certification comparisons are performed, so 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).

## 8.3 Extending the Database API

For Tashkent-API, the database must enforce a specified commit ordering. In PostgreSQL, 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 PostgreSQL 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. Since this behavior is identical to the original standalone PostgreSQL system, database recovery is not affected by this change.

## 9 Performance Evaluation

### 9.1 Methodology

Under GSI read-only transactions are executed locally on the receiving replica. The scalability of the replicated system is limited by the rate of updates and the overhead of maintaining consistency. Thus, we assess the performance of uniting durability with ordering by comparing Tashkent-MW and Tashkent-API to *Base* using three benchmarks that vary widely in their update rates.

**AllUpdates Benchmark.** We developed a benchmark, called AllUpdates, in which clients rapidly generate back-to-back short update transactions that do not conflict. The average writeset size is 54 bytes for each update transaction. AllUpdates represents a worst-case workload for a replicated system.

**TPC-B Benchmark.** TPC-B is a benchmark from the Transaction Processing Council [23] that uses transactions containing small writes and one read. The average writeset size is 158 bytes. In contrast to the AllUpdates benchmark, TPC-B transactions have both reads and writes, plus its workload contains write-write conflicts.

**TPC-W Benchmark.** TPC-W is a benchmark from the Transaction Processing Council designed to evaluate e-commerce systems. It implements an on-line bookstore and has three workload mixes. We report results from the most commonly used *shopping mix* workload (with 20% updates). The average writeset size is 275 bytes. In contrast to the other two benchmarks, the relatively heavy-weight transactions of TPC-W make CPU processing the bottleneck.

Thus, the three benchmarks AllUpdates, TPC-B, and TPC-W represent a spectrum of workloads differing in terms of fraction of writes, transactions complexity and conflict rate. We measure the performance of a single standalone database and determine the number of clients needed to generate 85% of the peak throughput. In the following experiments, each replica is driven at this load.

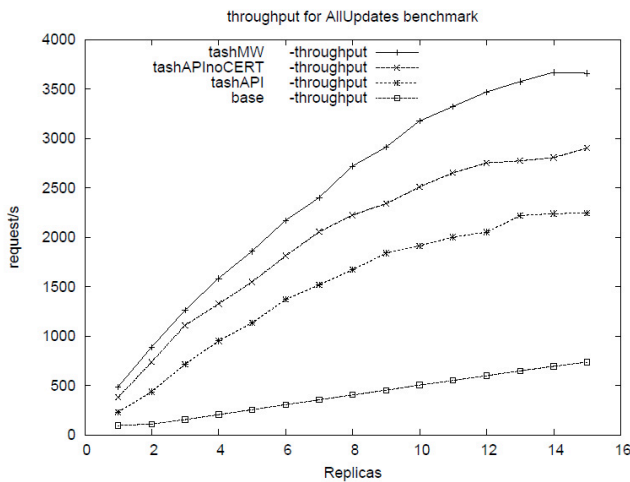


Figure 4: Throughput for AllUpdates (shared IO).

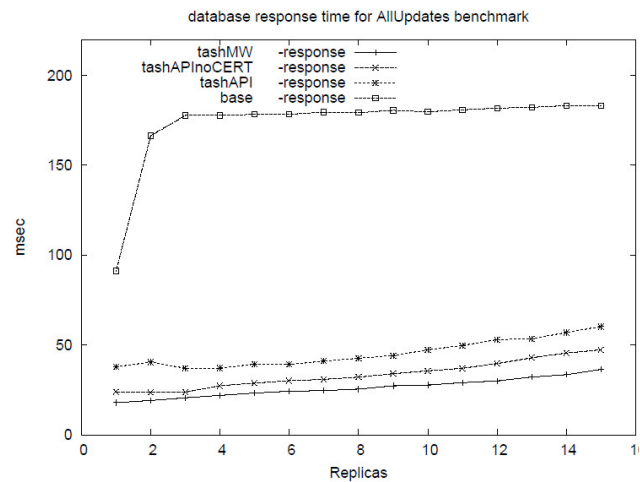


Figure 5: Resp. Time for AllUpdates (shared IO).

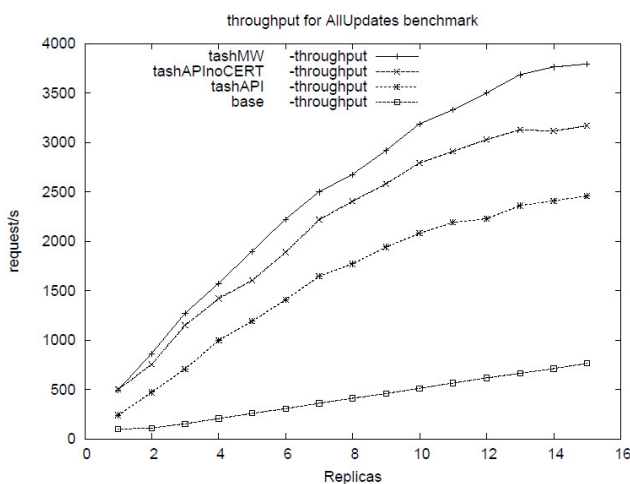


Figure 6: Throughput for AllUpdates (dedicated IO).

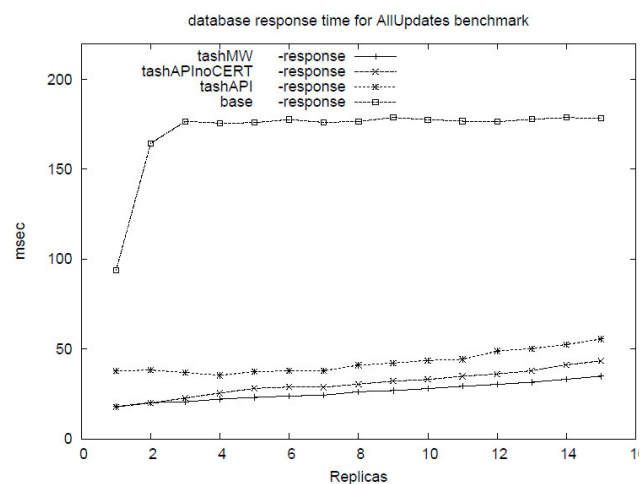


Figure 7: Resp. Time for AllUpdates (dedicated IO).

**System specification.** Each machine in our cluster runs the 2.6.11 Linux kernel on a single Intel Xeon 2.4GHz CPU with 1GB ECC SDRAM, and a 120GB 7200rpm 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 7.3). We use the PostgreSQL 8.0.3 database configured to run transactions at the snapshot isolation level (which is the strictest isolation level in PostgreSQL where it is called the “*serializable transaction isolation level*”).

**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 system `fsync` takes about 8ms, but the actual time varies depending on where the data resides on disk (6ms-12ms).

## 9.2 AllUpdates

Figure 4 shows the throughput results for *Base*, Tashkent-MW, and Tashkent-API running AllUpdates. The x-axis is the number of replicas and the y-axis is the requests per second

(i.e., transactions per second). The corresponding response times are shown in Figure 5.

In Figure 4, the throughput curve for *Base* is at the bottom and grows linearly at the rate of just over 49 req/sec, up to 735 req/sec at 15 replicas. This matches the capacity of each replica’s IO channel to process synchronous writes. At 8 ms per `fsync` call, when commits are serialized the upper limit on throughput is 125 req/sec. If remote writesets are 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 remote writesets and one for the local transaction, for an expected limit of 50-60 local transaction commits per second depending on the time to complete the `fsync`.

In contrast, Tashkent-MW achieves a total throughput of 3657 req/sec at 15 replicas, or five times that of *Base*. This high throughput is achieved because with durability united with ordering at the certifier, it is able to group an average of 29 writesets per `fsync`.

Tashkent-API also performs much better than *Base*, achieving 2240 req/sec, or three times that of *Base*. While its throughput is also very high, Tashkent-API does not achieve the

performance of Tashkent-MW, for three reasons. First, Tashkent-API has an inherently longer round-trip latency for each client request than Tashkent-MW. There are 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 commit serialization at the database is by far its bottleneck). Second, each of our servers has only one disk. With a single disk, the IO channel is shared between three different IO streams: reading the database pages, writing back dirty database pages, and durability logging. Third, Tashkent-MW and Tashkent-API write somewhat different log information in different ways.

To explore the effects of these factors on Tashkent-API, we first eliminate the extra latency at the certifier. While durability in the certifier is necessary for middleware recovery, we run with it off and show the resulting throughput in the fourth curve *tashAPIInoCERT* in Figure 4. In this case the certifier performs certification as usual, but it does not write information to disk. Without the IO delay at the certifier, the throughput increases to 2901 req/sec, but still less than Tashkent-MW.

Next, since logging IO is in the critical path when durability is in the database, we create a dedicated logging channel by putting the database in ramdisk. Therefore, the IO channel is dedicated to logging, and reading and writing database pages occurs in the main memory. We show in Figures 6 and 7 the results of using ramdisk. The relative throughput behavior in Figure 6 is similar to that in Figure 4, but all the curves adjust up somewhat because less contention improves throughput. However, 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.

We attribute the remaining performance difference between Tashkent-MW and Tashkent-API (i.e., between curves *tashMW* and *tashAPIInoCERT* in Figure 6) to differences between PostgreSQL and the certifier. The certifier logs only the writeset, whereas PostgreSQL logs before/after images of data pages. In addition, PostgreSQL has a heavier-weight multiprocess design compared to the multithreaded design of the certifier for Tashkent-MW. We expect the differences would disappear with proper changes to PostgreSQL.

The response times in Figures 5 and 7 both show a jump in the response time for *Base* between one replica and two replicas. At one replica there is only one fsync to commit each local transaction (there are no remote writesets). At two or more replicas an additional fsync call per local transaction is necessary for the remote writesets.

More specifically, for *Base* in Figure 7 where the logging channel is dedicated, there are 10 clients at each replica which results in approximately 10 concurrent local requests at each replica. Thus, the response time at one replica is about 90 msec (each client request waits for the other 9 fsync calls plus its own fsync). At two replicas and onwards, the response time rises to nearly 180 msec. This delay includes the 10 fsyncs to commit the local transactions, plus another 10 fsyncs to commit the grouped remote writeset for each local transaction. For the Tashkent systems, the delay increases slowly reflecting the overhead of processing more remote writesets at each replica.

The certifier is lightweight. For example, at 15 replicas in the Tashkent-MW system, the certifier certifies 3657 req/sec while its disk is less than 50% utilized and its CPU utilization is below 20%.

The replication middleware does not add significant overhead. The throughput of a 1-replica Tashkent-MW system running the full replication protocol is within 5% of a standalone system. For example, under the AllUpdates benchmark using a shared IO channel, a standalone database gives a throughput of 517 req/sec with 17 msec average response time compared to 490 req/sec with 18 msec for 1-replica Tashkent-MW system. The performance is similar when using a dedicated IO channel (515 req/sec with 17 msec for standalone versus 491 req/sec with 18 msec for 1-replica Tashkent-MW). Moreover, we see the same behavior under TPC-B and TPC-W; a 1-replica Tashkent-MW gives a performance close to that of a standalone database.

In summary, under this worst-case update-intensive workload, both Tashkent-MW and Tashkent-API show impressive scalability improvements. Tashkent-MW and Tashkent-API have improvements of 5.0x/3.0x (shared IO) and 5.0x/3.2x (dedicated IO) over *Base*.

### 9.3 TPC-B

TPC-B is also an update-intensive benchmark, but the transactions include reads from the database as well. The throughput and response times are shown in Figures 8 and 9, in the case of a shared IO channel. We see the same relative performance of the systems: Tashkent-MW gives highest performance (2.6x *Base*), *Base* the lowest, and Tashkent-API (1.3x *Base*) performs in between the other two.

Again, to explore the difference between Tashkent-MW and Tashkent-API, we remove the fsync in the certifier and show its improvement in the curve *tashAPIInoCERT*, resulting in a somewhat higher throughput.

We also plot in Figures 10 and 11 the throughput and response time results in the case of a dedicated IO channel (i.e., the database is in ramdisk). Here all curves are higher, but there is still a significant difference between Tashkent-MW and Tashkent-API. This difference is not due only to the additional latency at the certifier as the curve *tashAPIInoCERT* shows little additional throughput with the dedicated logging channel.

Unlike AllUpdates, TPC-B generates “artificial” conflicts among remote writesets (Section 5.2.1). An artificial conflict among remote writesets prevents Tashkent-API from submitting them concurrently to the database. This reduces opportunities to unite durability and ordering. When such an artificial conflict is detected, some remote writesets must be submitted serially in separate fsync calls. These additional fsyncs degrade Tashkent-API’s performance. The artificial conflict rate between remote writeset groups is 35% in TPC-B.

Finally, the response times in Figures 9 and 11 increase steadily as replicas are added. This reflects the overhead of applying writesets at the replicas.

### 9.4 TPC-W

In Figure 12 we show the throughput of TPC-W with the shopping mix (20% updates) for a shared IO channel. There is no difference between Tashkent-API and *Base* because the update rate is very low. 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 3\*2 fsync() calls per second at each replica), much lower than what is needed to saturate the local logging channel. Thus, Tashkent-API has no opportunity to group commits to disk. At very low update rates, separating ordering and durability does not create a bottleneck.

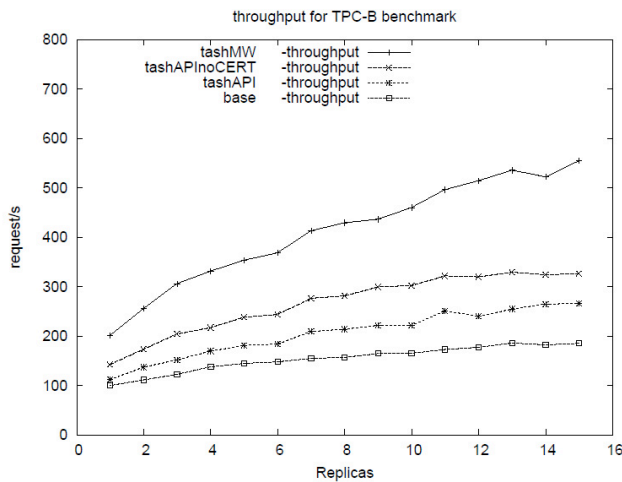


Figure 8: Throughput for TPC-B (shared IO).

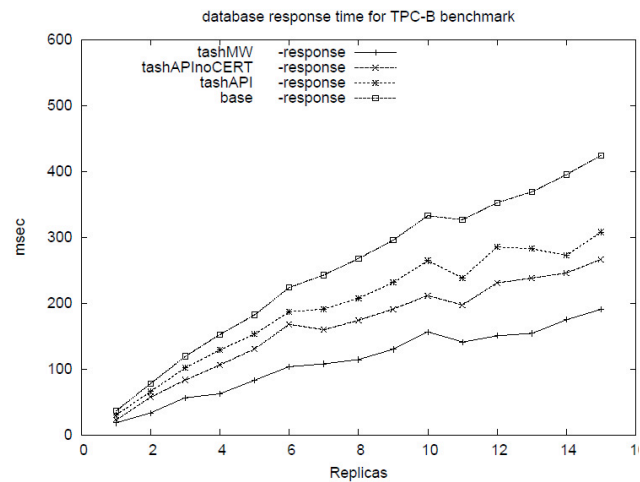


Figure 9: Resp. Time for TPC-B (shared IO).

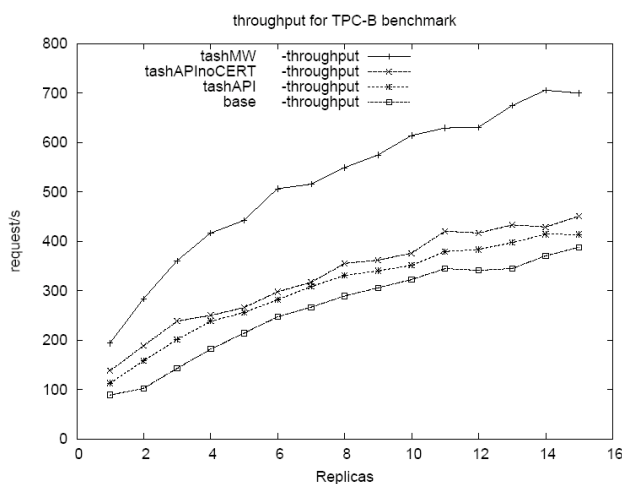


Figure 10: Throughput for TPC-B (dedicated IO).

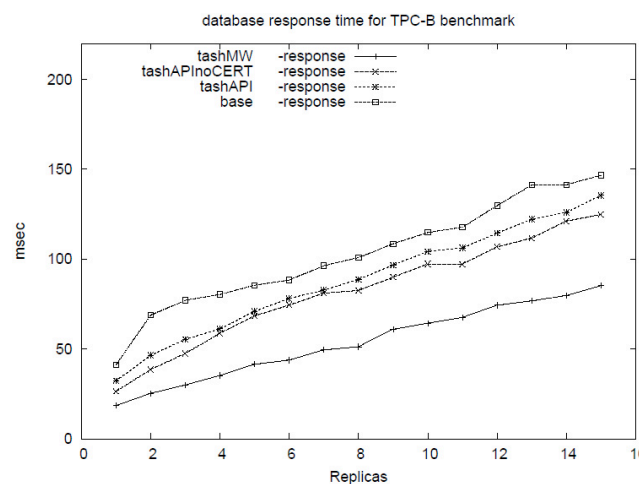


Figure 11: Resp. Time for TPC-B (dedicated IO).

The response times of read-only and update transactions are shown in Figure 13. Update transaction response times are similar for Tashkent-API and *Base*. Furthermore, the response times for read-only transactions are similar for all systems as read-only transactions are handled in the same way in all three systems.

However, the performance of Tashkent-MW is better than both Tashkent-API and *Base*. The reason is the shared IO channel for 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, but the large TPC-W environment does not fit in ramdisk on our machines. We expect that both Tashkent-API and *Base* would match the performance of Tashkent-MW if the logging channel is dedicated and the update rate remains low.

## 9.5 The effect of aborts

One may wonder if the benefits of Tashkent remain present even under workloads with a substantial number of aborts. In Figure 14 we demonstrate that this is the case. We show the

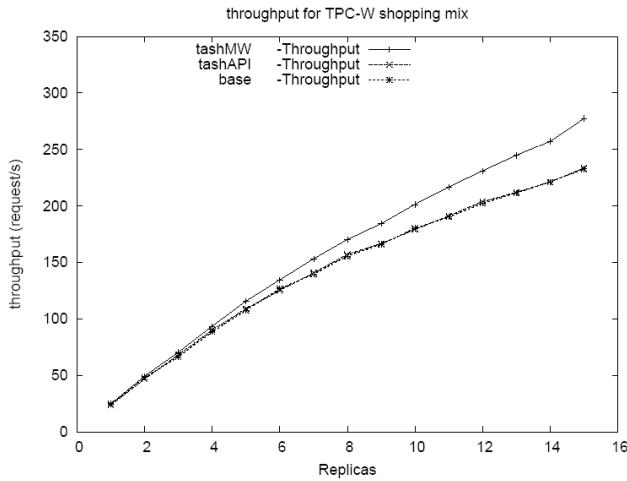
effects of aborts using the AllUpdates benchmark because TPC-B and TPC-W have very few (non-artificial) conflicts and subsequently low abort rates.

In AllUpdates, we force a system-wide abort rate by having the certifier randomly abort requests at that 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.

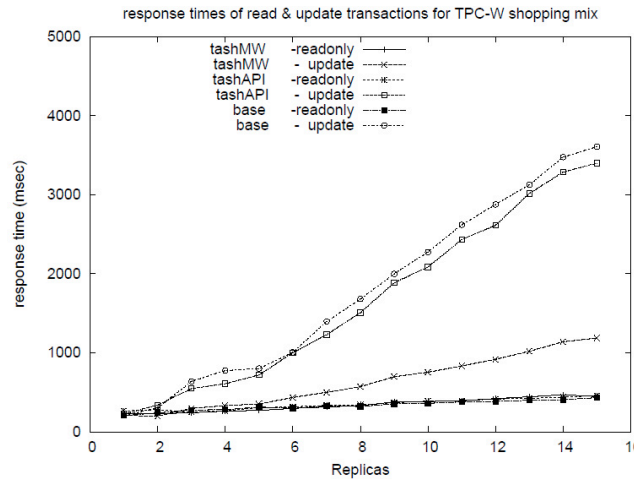
In Figure 14 we use *exaggerated* abort rates to demonstrate how the “goodput” (the throughput of non-aborted requests) is affected. We force three different abort rates: 0%, 20%, and 40%. The top three lines are for Tashkent-MW, the middle three are for Tashkent-API, and the bottom three are for *Base*. Thus, even under high abort rates the Tashkent systems have significantly higher performance than *Base*.

## 9.6 Recovery time

We report the recovery times for TPC-W because it is the largest of the three benchmarks (the database size is ~700 MB) and the most costly to recover.



**Figure 12: Throughput for TPC-W shopping mix (shared IO).**



**Figure 13: Resp. Time for TPC-W shopping mix (shared IO).**

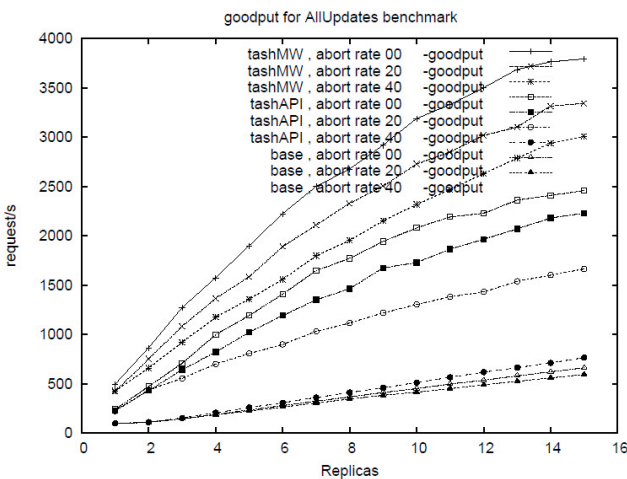
**Tashkent-MW database recovery.** Tashkent-MW requires taking periodic copies of the database for its recovery. In a 15-replica system, a replica requires about 230 seconds to dump a complete copy of the database while it continues processing transactions. That replica's throughput degrades by 13% during the 230 seconds. After a crash, a replica requires 140 seconds to restore the database from the dump file.

**Tashkent-API and Base database recovery.** In Tashkent-API and *Base*, a database replica recovers exactly as a standalone database. The database reaches a consistent state using its own internal recovery procedure in a few seconds (2-4 seconds).

**Certifier recovery.** A certifier recovers much faster than a database. Its recovery time is dominated by the time required to update its log. The growth rate of the certifier log at 15 replicas is 201,600 writesets per hour at an average of 275 bytes per writeset, or 56 MB per hour. The log is transferred from one of the up certifiers and is essentially a file transfer that takes about 1 second in our LAN for each hour of down time.

## 9.7 Summary

Under high update workloads, separating durability from ordering creates a significant scalability bottleneck. Combining these two functions, either in the middleware or in the database, alleviates this bottleneck and greatly improves performance. The Tashkent-MW solution is more robust than Tashkent-API if the workload has a significant fraction of artificial conflicts. Such a workload forces Tashkent-API to serialize some commits and behave more like *Base*.



**Figure 14. Certifier goodput under different abort rates (dedicated IO).**

## 10 Related Work

### 10.1 SI, GSI, and Serializability

Serializability [17, 2] is the most common 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 the TPC-A, TPC-B, TPC-C, and TPC-W benchmarks). Even if these conditions do not hold for an application that uses a fixed set of transaction templates, these transaction templates can be modified automatically to be serializable under SI. In practice, database developers understand SI and are capable of using it.

**Applying writesets (all systems).** After a database recovers, the proxy applies the remote writesets that occurred during the down time and recovery time of the database. The proxy batches the remote writesets and applies them to the database at a rate of 900 writesets per second. At 15 replicas, the rate of updates is 56 writesets per second (20% update of 280 req/sec, in Tashkent-MW). For H hours of down time, it requires approximately 222\*H seconds.

The notion of generalized snapshot isolation (GSI) is introduced in Elnikety et al. [3]. Generalized snapshot isolation preserves SI isolation guarantees. In particular, the authors prove that if an application runs serializably under SI, it also runs serializably under GSI. 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.

Tashkent has both eager and lazy features. Tashkent uses GSI and avoids global write-write conflicts eagerly during certification. It appears as a single snapshot isolated database to clients. It does not need update reconciliation and the effects of committed update transactions are not lost in the system due to a replica failure. However, replicas are not updated in lockstep. If a replica does not receive update transactions, it may become stale. Tashkent bounds staleness using a simple mechanism to fetch updates from the certifier (Section 6.2).

## 10.3 Replication in SI Databases

Kemme et al. [12] discuss how to implement different isolation levels (including serializability and snapshot isolation) in replicated databases using group communication primitives. In addition, they implement Postgres-R [11], and integrate 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 Tashkent-MW is a *pure* middleware solution. Postgres-R(SI) commits update transactions sequentially. This limits scalability if durability is guaranteed by the database but ordering is determined in the middleware. The database needs one fsync call for each commit. The Tashkent systems remove this fsync bottleneck.

In Postgres-R(SI) the replication protocol is tightly coupled with the concurrency control. For example in one of the prototypes, the 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 replica components are 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) and makes the certifier compete on resources with the database at each replica, possibly reducing replica performance.

Plattner et al. [18] present 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. During update propagation, transactions commit serially at the slaves as well. 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 the receiving replica and only certify the core writesets at the certifier. Certifying the writesets is an order of magnitude less work than executing the transaction. Thus, the Tashkent designs effectively distribute much of the update transaction workload.

## 11 Conclusions

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

We present Tashkent-MW and Tashkent-API as two example solutions that unite durability and ordering. In Tashkent-MW, durability is united with commit ordering in the replication middleware. Tashkent-MW is a *pure* middleware solution for high-performance replicated databases.

Tashkent-API unites durability and commit ordering inside the database. The replication middleware passes the commit order information to the database. In this solution, additional care must be taken to ensure the middleware proxy does not generate artificial conflicts between remote writesets that are submitted concurrently. This constraint prevents some opportunities to combine durability and ordering.

We also present *Base*, a replicated database system where durability and commit ordering are separate. We experimentally compare Tashkent-MW and Tashkent-API to *Base* to assess the benefits of uniting durability and commit ordering.

We implement the *Base*, Tashkent-MW and Tashkent-API systems on top of PostgreSQL and analyze their performance using our AllUpdates benchmark as well as the standard benchmarks TPC-B and TPC-W. Under high update transaction loads the advantages of uniting durability and ordering become significant. We show that both versions of Tashkent greatly improve scalability under high update loads and outperform *Base* by factors of 5 and 3 times, respectively, in throughput while also providing lower response times.

## 12 Acknowledgments

We thank Willy Zwaenepoel from EPFL for his insightful suggestions and valuable feedback. We also thank the anonymous reviewers for their constructive feedback. Finally, we thank Anastassia Ailamaki for her guidance in preparing the final draft. This research was partially supported by the Swiss National Science Foundation grant number 200021-107824.

## 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] Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.
- [3] Sameh Elnikety, Fernando Pedone, and Willy 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] Lars 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] Jim Gray, Pat Helland, Patrick O’Neil, and Dennis 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] Leslie Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2):133-169, May 1998.
- [14] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo 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. July 1986.
- [18] Christian Plattner and 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] Calton Pu and Avraham 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), San Francisco, California, December 2004.
- [22] Fred B. Schneider. 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] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In proceedings of 20th International Conference on Distributed Computing Systems (ICDCS’2000), Taipei, Taiwan, April 2000.