WICE-SI: Pragmatic Inter-Cluster Replication

Technical report - July 4 2006

J. Grov[1], L. Soares[2], A. Correia Jr.[2], J. Pereira[2], R. Oliveira[2], and F. Pedone[3]

[1]University of Oslo
[2]University of Minho
[3]University of Lugano

July 4, 2006

Abstract

Multi-master update everywhere database replication, as achieved by protocols based on group communication such as DBSM and Postgres-R, addresses both performance and availability. By scaling it to wide area networks, one could save costly bandwidth and avoid large round-trips to a distant master server. Also, by ensuring that updates are safely stored at a remote site within transaction boundaries, disaster recovery is guaranteed. Unfortunately, scaling existing cluster based replication protocols is troublesome.

In this paper we present a database replication protocol based on group communication that targets interconnected clusters. In contrast with previous proposals, it uses a separate multicast group for each cluster and thus does not impose any additional requirements on group communication, easing implementation and deployment in a real setting. Nonetheless, the protocol ensures one-copy equivalence while allowing all sites to execute update transactions. Experimental evaluation using the workload of the industry standard TPC-C benchmark confirms the advantages of the approach.

This technical report is a reworked version of a paper submitted to PRDC 2006. But in that version, we present a general algorithm and focus the theoretical discussion on serializability. In addition, this version includes a consistency discussion arguing that our protocol do in fact provide snapshot isolation.

1 Introduction

Database replication is an attractive concept both to increase fault tolerance and to improve scalability by enabling several database sites to serve the same queries. The main challenge of such systems is that coordinating updates among the participating servers inevitably delays the execution of update-transactions. A particularly promising approach is taken by replication protocols based on group communication such as DBSM [22, 12] and Postgres-R [18, 32]. By taking advantage of optimistic concurrency control allowed by transactional semantics and of atomic multicast provided by group communication, it provides performance and scalability even in face of demanding workloads such as the industry standard TPC-C benchmark [28].

Unfortunately, scaling existing cluster based replication protocols to a wide area network is troublesome. Notably, the latency of uniform atomic (or safe) delivery required to ensure fault tolerance has a profound impact on optimistic concurrency protocols leading to increased abort rate [11]. This wastes resources and endangers the ability to commit long lived transactions in a busy server. Although optimistic delivery can mitigate this limitation [27], using it requires an in-depth rewrite of existing protocol implementations. In fact, the only generally available group communication toolkit supporting it is Appia [21, 24].

Furthermore, although research has been addressing group communication in wide area networks for a long time, industrial deployment is far more common in clusters. Therefore one should expect wide area features to be far less tested and optimized, if implemented at all. The overhead of maintaining automatic management

of membership spanning multiple geographically apart sites is also not negligible. Finally, the practicality of group communication over wide area networks is also compromised by network configuration and security issues, such as firewalls, tunnels and NAT gateways. In particular, using true multicast for efficiency is often not an option.

In contrast, optimization of high latency and high bandwidth TCP/IP connections has seen widespread deployment. For instance, numerous "TCP/IP-accelerator" products are commercially available to enhance transmission performance across satellite links [20].

In this paper we present WICE-SI, a protocol targeted at multiple clusters interconnected by a wide area network. In contrast with lazy replication protocols, such as Oracle Streams [31], WICE-SI ensures that no globally committed transaction (i.e. which has been acknowledged to clients) is lost. On the other hand, by allowing all replicas to be fully on-line and executing update transactions, it improves resource efficiency and performance when compared to volume replication [29], often the only choice for disaster recovery in mission critical applications. In detail, the contributions of this paper are:

- Introduces the protocol providing 1-copy equivalence of the native database consistency criterion, even in the presence of faults, while confining group communication within local area clusters and improving practicality.
- Takes advantage of directly implementing stabilization across wide-area directly on TCP/IP to greatly reduce the likelihood of a transaction being aborted during the certification phase, which is the single greatest obstacle to the scalability of previous proposals [11].
- Provides an experimental evaluation of the protocol applied to a multi-version database when running the workload of the industry standard TPC-C benchmark [30], thus verifying the previous claim.

The rest of this paper is organized as follows: In Section 2, we present background and related work. In Section 3, we introduce our system model and assumptions. Section 4 presents the protocol, while Section 5 gives the experimental results. Section 6 concludes the paper.

2 Background

In outline, the responsibilities of a replica control protocol are: (1) the overall concurrency control, ensuring that transactions executing concurrently on different sites do not violate consistency; (2) the efficient propagation of updates among the replicas; and (3) atomic commitment, that is, ensuring that agreement is reached among all replicas on the outcome of transactions.

In a wide-area setting, a major challenge is to minimize the number of messages in order to reduce overall latency. Previous studies [7, 3, 18, 22, 11, 32] advocate the appropriateness of *optimistic* concurrency control to such settings. An optimistic protocol only validates a transaction's execution against its concurrent transactions after it has executed all operations and just before it is allowed to commit. If the validation fails, the transaction is aborted and possibly restarted.[1] In a replicated database, optimistic validation usually means that a transaction is first executed locally at one site without any network interaction. Only when all operations are locally executed, the execution is globally validated against remote, concurrent transactions using some *certification* procedure. Recent replica control protocols using this approach include the optimistic protocol presented in [3], the Postgres-R protocol presented in [18] and the *Database State Machine* (DBSM) [22].

In replica-control protocols with optimistic validation, updates are commonly propagated as part of the validation protocol. Moreover, in these protocols, atomic commitment is handled through *dictatorial* commit [1]: No individual site can veto a commit decision as long as a remote transaction request is validated by the distributed validation procedure. It is then common [18, 22], to combine update propagation and dictatorial-commit using an atomic broadcast primitive [16] which ensures that all non faulty replicas receive updates in the same total order. This ordering is then used as a global timestamp [6], and is used to decide whether a transaction is allowed to commit or abort.

[1] For non-interactive transactions, the restart can be performed automatically.

One challenge arising in all protocols that combine dictatorial-commit protocols with failover guarantees is that before a transaction can commit, it must be ensured that the updates are present at *all* non faulty sites before the transaction is allowed to commit. This is necessary to avoid *dirty reads*, as illustrated by the following example: Assume a transaction T is allowed to commit at site A before T's updates are installed at site B, and assume A fails immediately after, such that T is never known at site B. This can clearly lead to inconsistencies, as any client who has already read T's updates from A will assume that the updates of T are also present in cluster B.

To handle this, we must ensure that the updates are *stable*, i.e., that they will be known by at least one remaining site after any kind of disaster from which we want to recover, before we allow the transaction to commit.

If in cluster settings the modular use of a primitive providing uniform atomic delivery [16] is advantageous by offloading all the group communication complexity and optimization to the communication toolkit, the high and uneven latencies in wide area networks require a differentiated treatment of local and remote replicas. This allows us to optimize the replication protocol by masking the high latency and make a conscious use of the bandwidth of wide area links.

While for wide area specific database replication we opt for the optimistic execution approach, two recent studies of group communication based replication recommend the use of pessimistic execution [2, 19] and follow the active replication approach [25, 15]. In [2], each update request is atomically broadcast and executed by every replica. In [19], transactions are broadcast at once greatly reducing the previous (per request) message overhead. To avoid executing transactions sequentially, in [19] it is assumed that transactions are a priori annotated with conflict classes so that coarse-grained locks can be acquired before starting execution and thus non-conflicting transactions are allowed to execute concurrently.

3 System model

We assume the page model for a database [6]: A collection of named data items which have a value. The combined values of the data items at any given moment is the database state. We do not make any assumptions on the granularity of data items.

We assume a distributed database where each site is modeled as a sequential process. In detail, the execution of each site is modeled as a sequence of steps that may change the site's state. Namely, the database state is manipulated by executing READ(x) and WRITE(x) steps. A transaction T is a sequence of read and write operations followed by a COMMIT(T) or ABORT(T) operation.

Each site is assumed to contain a complete copy of the database and provides snapshot isolation[5]: Each write-step creates a new copy of the updated item. Read-steps are allowed to access older versions of data items, and at any time, beginning transactions are assigned a snapshot defined by a version counter lts. The version counter at site s is incremented as soon as an update transaction T has applied all its updates and is finished executing at s.[2].

Write-steps are ordered using *write-tokens* to implement using *first-commit wins*: As soon as a transaction T is finished executing and ready to commit, all other transactions requesting tokens held by T must be aborted. Thus, if a transaction is blocked awaiting the write-token, it can only commit if the current holder is aborted. When transaction T has successfully finished execution and applied all updates, lts is incremented to allow other transactions to see its updates.

We consider a finite set of database sites that communicate through a fully connected network. Both computation and communication are asynchronous. Sites may fail only by crashing and do not recover, thus stopping to execute database operations, or send or deliver further messages.

Database sites are organized in clusters. Within a cluster we assume a primary component group membership service that provides current and consistent views of the sites believed to be up [9]. This service is intended to allow, at any moment, the deterministic identification of a distinguished site as the cluster's *delegate* (given

[2]In practice, this is the *commit*-point of T. But this is the *local* commit, and since we consider distributed databases we say that T commits only when its updates are applied at all nodes

by a function delegate()) as well as supporting the provision of a view-synchronous multicast primitive (Section 3.2). The availability of a primary component group membership service implicitly assumes that consensus is solvable in our system model [14]. We do not however explicitly explain how this is achieved or otherwise make use of any assumptions besides an asynchronous system model.

Among clusters, we assume that the failure of an entire cluster is reliably detected at the other sites. That is, if all sites in a cluster fail then the fact is eventually noticed by the other clusters' delegates. Otherwise, the cluster is never suspected to have failed.[3] At each cluster, the set of clusters believed to be up is given by a function remoteClusters().

3.1 Database interface

The replication protocol presented in Section 4 uses a replication interface with the database engine that is part of the API being defined in the context of the GORDA project [10]. The interface has been implemented in a number of DBMS, notably in PostgreSQL [17] and Derby [4]. The interested reader can find its detailed definition in [23]. Basically, it allows the inspection of a transaction's execution at three specific points: (1) on the beginning of the transaction's execution, (2) at the end of the transaction's execution, just before it start committing updates or rolls back, and (3) when the local database system has committed the transaction and is ready to reply to the client. Furthermore, the database engine provides an update method executed with priority over any other running transactions that allows to update the values of a given set of items.

More precisely, we assume that the replicated database engine allows to register four callback methods as follows:

onExecuting(tid) invoked before a transaction is about to enter the executing state, i.e., before it starts execution. The transaction is identified by tid.

onCommitting(tid, ws, wv) invoked when the transaction tid has finished its local execution, is validated by the local concurrency control, and is about to enter the commit phase. The database provides the set of tuples written (ws) by the transaction, as well as the written values (wv). At this point the transaction has all its updates buffered and all write locks still acquired.

onAborting(tid) invoked when the transaction tid fails and is about to abort.

onCommitted(tid) invoked after the transaction has completed making all updates persistent, updated the local version-counter, entered the committed state and is ready to reply to the client. Updating the local version-counter means that the updates of transaction tid become part of the snapshot of subsequent transaction.

When it invokes any of the above methods, the database engine suspends the execution of the transaction until the protocol replies by invoking the database methods continueExecuting(tid), continueCommitting(tid), continueAborting(tid) and continueCommitted (tid), respectively.

Replica updates are submitted to the database using the db_update (tid, ws, wv) method which applies the values in wv to the tuples in ws by means of a high priority transaction. A transaction submitted through db_update only triggers the onCommitted(tid) events. High priority means that any regular (i.e. non-high priority) transaction holding locks on any item in ws will be aborted. Moreover, high priority transactions are serialized when requesting locks and then executed concurrently.

3.2 Communication primitives

3.2.1 Intra-cluster communication

Within clusters a group communication toolkit is available providing reliable point-to-point communication and FIFO uniform view-synchronous multicast [9]. These primitives rely on the existence of a (primary component)

[3] This assumption is equivalent to have a perfect failure detector among the clusters [8]. In a wide area setting, its provision would require the use of a specially dedicated communication infrastructure among the clusters or rely on human intervention to declare the unavailability of all cluster sites.

group membership service that tracks the membership of the cluster through a sequence of *views* satisfying the following property:

Agreement on the view history Let v_i^p denote the i^{th} view at site p. If p installs v_i^p and if q installs v_i^q, then we have $v_i^p = v_i^q$.

The agreement property allows us to denote a view simply by v_i without mentioning a particular site. The specification of a group membership service includes additional properties that, for the sake of simplicity, we intentionally omit here.

Point-to-point reliable communication is defined by two primitives r_send and r_deliver, and satisfies the following properties:

Delivery Integrity A message m is delivered at most once and only if m has been previously sent.

Reliable Delivery: If sites p and q both belong to $v_i \cap v_{i+1}$ and p sends m to q in v_i then q delivers m before v_{i+1}.

Uniform view-synchronous multicast is defined through primitives u_vscast and u_vsdeliver and satisfies the following two properties:

Delivery Integrity A message m is delivered at most once and only if m has been previously vscast.

Sending View Delivery A message m vscast in view v_i is delivered in view v_i.

Uniform View Synchrony If site p belongs to view v_i and delivers m before v_{i+1}, then every site q in $v_i \cap v_{i+1}$ delivers m before v_{i+1}.

FIFO uniform view-synchronous multicast is invoked through primitive fifo_u_vscast and in addition to the above properties satisfies:

FIFO Delivery If site p vscasts two consecutive messages m_1 and m_2, then no site delivers m_2 before m_1.

3.2.2 Inter-cluster communication

Among them, clusters exchange messages through a point-to-point FIFO reliable channel. They use primitives fifo_r_send and fifo_r_deliver satisfying the following properties:

Delivery Integrity A message m is delivered at most once and only if m has been previously sent.

Reliable Delivery: If cluster p and q are both correct and p sends m to q then q eventually delivers m.

FIFO Delivery If cluster p sends two consecutive messages m_1 and m_2 to cluster q, then q does not deliver m_2 before m_1.

A cluster is said to be correct if it does not fail entirely.

4 The WICE-SI protocol

The WICE-SI protocol adopts an optimistic concurrency control policy. Transactions are executed optimistically at any site and then, just before commit, certified against concurrent transactions. WICE-SI borrows from protocols such as Postgres-R [18] and DBSM [22] often called *certification based* protocols. These protocols share two fundamental characteristics: (1) each database site is assumed to store the whole database and transactions can be executed at any site, and (2) all update transactions are certified and, if valid, committed in the same order at all sites.

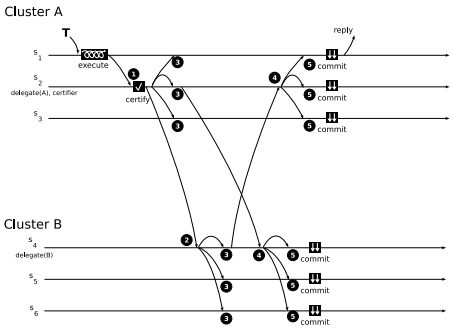

Figure 1: WICE-SI: example of handling of transaction T

The protocol given here provides snapshot isolation, similar to Postgres-R(SI)[32] and a variant of DBSM supporting snapshots (abbrev DBSM-SI)[12]. By assuming multi-version concurrency control at each site, read-operations are not considered for certification as each transaction T is assumed to read from a *snapshot* defined by all committed transaction when T entered execution[5].

The fundamental difference between Postgres-R(SI), DBSM-SI and WICE-SI is when and where the certification is performed. The two former protocols use a total order broadcast primitive and certify each transaction once the totally ordered message is delivered. In Postgres-R(SI), each transaction is certified at the site that executed it and the outcome of the certification is then sent to all the other sites. In the DBSM-SI, the write set of the transaction is sent to all sites allowing the certification to be carried at all sites avoiding the last communication step of Postgres-R(SI).

WICE-SI does not make use of a total order primitive, instead ordering is explicitly handled by the protocol. In WICE-SI, one of the sites plays the role of certifier, it totally orders and certifies all transactions. Each valid transaction is then broadcast together with its ordering timestamp and updates. This allows to leverage the knowledge about the system's topology and make optimizations that would not be possible otherwise.

The WICE-SI algorithm is exemplified in Figure 1. In a nutshell, the handling of a transaction proceeds as follows. Consider a system consisting of two clusters A and B. Each cluster has a designated delegate responsible for handling the communication with the other cluster. The delegate of cluster A, site s_2 is also responsible for certifying all executed transactions. When an update transaction T is submitted to site s_1 (T's initiator), it is readily executed and sent to the certifier. If it succeeds, then the certifier propagates T's updates and ordering-timestamp, both locally and to cluster's B delegate. The latter, in turn, propagates T locally. Once a delegate is certain that all sites in its cluster delivered T's data it acknowledges the fact to the other cluster's delegate. This acknowledgement is multicast locally by each delegate. Once a database site knows T's data has been delivered everywhere and all previous transactions had been committed, then it commits T. The initiator of T can then reply to its client.

Note that the algorithm discussed here only applies to update transactions, as read-only transactions do not need validation as such. Nevertheless, because we increment the versions of any object written by an update transaction T_i before T_i becomes stable, the commit of any read-only transactions must be restricted to ensure that no updates are read and exposed before T_i is stable (see the discussion of "dirty reads" in Section 2). For clarity, we omit this from the protocol and assume it to be handled by the local DBMS by blocking the commit of a transaction T_r until all updaters from which it has read from are committed. Note that this requirement apply to update transactions as well, but since a transaction T_i can only read from transactions that were certified when T_i began execution, the certification procedure guarantees that T_i will not become stable unless all preceding transactions are stable as long as it is certified

4.1 Algorithm

We now consider the protocol algorithm in detail (Figure 2). It is composed by a set of handlers that deal with events triggered by the database engine (”Events at the initiator” and ”Transaction commit”) and with message delivery. We assume that every database site knows the current system's certifier through a function certifier() and that each cluster delegate can find the other participating clusters through a function remoteClusters() as well as identifying some delegate's cluster through function cluster(). Further, the function delegate() is used to determine whether the current site is the delegate of its cluster or not.

Global site variables Each database site manages four sets containing transactions known to be certified, locally updated, locally commited and remotely stable. It keeps track of the number of locally executed transactions in variable lts. The certifier keeps track of the number of certified transactions in variable gts.

Events at the initiator When a transaction tid is submitted to the database engine, the callback procedure onExecuting(tid) is invoked prior to tid's execution. Here the initiator replica saves the current database version (given by the number of transactions committed locally) for tid and allows the database to proceed. Should the transaction abort locally, onAborting() is invoked and the transaction is simply forgotten by the protocol.

On the contrary, if tid succeeds then onCommitting() is invoked and its write set and written values (ws and wv) provided by the database are reliably sent to the certifier along with the version of the database on which the transaction executed. The transaction's execution is left suspended until it is certified and its outcome known. If tid ends up committing then continueCommitting(tid) will be called, otherwise the initiator receives a (ABORT, tid) message from the certifier and forces the transaction to abort locally.

Certification Upon delivering an update transaction to certify — (CERTIFY, tid, ts, ws, wv) — from some initiator site the certifier performs the certification of tid against its concurrent transactions. For every certified transaction (but not necessarily committed yet) ctid with timestamp greater than tid's version timestamp, the write sets of ctid and tid are compared. If there is a non empty intersection then the certification fails and an abort message is sent back to tid's initiator.

When tid passes the certification test then the certifier's sequence number is incremented and tid added to its set of certified transactions. The transaction's id, ordering timestamp, write set and written values are then sent to all other replicas. Locally, tid is sent using the FIFO uniform view-synchronous multicast primitive as a (UPDATE_LOC, tid, gts, ws, wv) message. Remotely, it is sent using the FIFO reliable point-to-point primitive to each remote cluster as a (UPDATE_REM, tid, gts, ws, wv) message.

Remote delivery of updates Once a cluster delegate delivers a transaction from the certifier it simply forwards the message to the local replicas using the FIFO uniform view-synchronous multicast primitive.

Local delivery of updates When a replica delivers a transaction tid it signals the fact adding it to its set of updated transactions. The use of a uniform primitive ensures that once the transaction is delivered at the current replica it is eventually delivered at all non faulty replicas in the cluster. Therefore, if the replica is a cluster delegate it acknowledges the fact that tid became stable at the cluster to all clusters. The just delivered updates are applied. If the replica is the tid's initiator then it just needs to proceed with continueCommitting(tid). This can be regarded as a local commit-operation. Although tid does not hold high priority locks at the initiator, the fact that it passed certification means that between its execution and the local commit, no other certified transaction conflicted with it, and consequently, tid will not be aborted by another transaction requesting high-priority locks at tid's initiator. For all other sites, db_update is invoked.

Delivery of remote acks Each time a delegate delivers a stability acknowledgment for transaction tid from some cluster, the pair (tid, cluster) is added to its acks set. When tid has been acknowledged by all remote clusters, then the delegate locally declares the transaction remotely stable using the (non- uniform) view-synchronous multicast primitive — (STABLE_REM, tid). When this message is delivered each replica adds tid to its remotestable set.

Global site variables

```
1  local = []
2  certified = updated = pending = ()
3  commited = remotestable = acks = {}
4  gts = lts = 0
```

Events at the initator

```
5  upon onExecuting(tid)
6        local[tid]=lts
7        continueExecuting(tid)
8  end

9  upon onComitting(tid, ws, wv, type)
10       rsend(CERTIFY, tid, local[tid], ws, wv) to certifier()
11 end

12 upon onAborting(tid)
13       continueAborting(tid)
14 end

15 upon rdeliver(ABORT, tid) from i
16       db_abort(tid)
17 end
```

(1) Certification

```
18 upon rdeliver(CERTIFY, tid, ts, ws, wv) from initiator
19       foreach (ctid, cts, cws, cwv) in certified do
20             if cts < ts and (cws ∩ ws ≠ ∅)) then
21                   r_send(ABORT, tid) to initiator
22                   return
23       gts = gts + 1
24       enqueue (tid, gts, ws, wv) to certified
25       fifo_u_vscast(UPDATE_LOC, tid, gts, ws, wv)
26       foreach cluster in remoteClusters() do
27             fifo_r_send(UPDATE_REM, tid, gts, ws, wv) to cluster
28 end
```

(2) Remote delivery of updates

```
29 upon fifo_r_deliver(UPDATE_REM, tid, ts, ws, wv) from certifier
30       fifo_u_vscast(UPDATE_LOC, tid, ts, ws, wv)
31 end
```

(3) Local delivery of updates

```
32 upon fifo_u_vsdeliver(UPDATE_LOC, tid, ts, ws, wv)
33       enqueue (tid, ts, ws, wv) to updated
34       if delegate() then
35             foreach cluster in remoteClusters() do
36                   r_send(ACK_REM, tid) to cluster
37       if local[tid] then
38             continueCommitting(tid)
39       else
40             db_update(tid, ws, wv)
41       enqueue (tid, ts, False) to pending
42 end
```

(4 and 5) Delivery of remote acks

```
43 upon r_deliver(ACK_REM, tid) from cluster
44       acked = {}
45       add (tid, cluster) to acks
46       foreach (tid, c) in acks do
47             add c to acked
48       if remoteClusters() ⊆ acked then
49             u_vscast(STABLE_REM, tid)
50 end

51 upon vsdeliver(STABLE_REM, tid)
52       add (tid) to remotestable
53 end
```

Transaction commit

```
54 upon onCommitted(tid, ts) and lts = ts + 1
55       lts = ts; add tid to commited
56 end

57 upon (tid) in commited and (tid) in remotestable
58       continueCommitted(tid)
59 end
```

Figure 2: WICE-SI protocol

Transaction commit Here, each site handles the onCommitted callback. When onCommitted (tid) is invoked, all write-locks held by tid are already released. But the updates might still not be exposed to readers, since we must ensure that updates are available in certification order to guarantee one-copy equivalence.

Consequently, the onCommit-handler checks all pending transactions. The pending-queue is traversed from beginning to end. The record representing the current tid is marked to show it is ready to be committed locally. A transaction is *applied* only as long as there is no previous transaction in the pending-queue which is not ready to commit. Observe that applying a transaction tid means that its updates are made available to other transactions by setting the lts-counter equal to tid's order-timestamp, adding the transaction to the committed-set.

As soon as transaction tid is known to be commited locally and stable everywhere the database is then allowed to perform a *global commit* (i.e. reply to the client), which happens after continueCommitted(tid).

4.2 Failure handling

The WICE-SI algorithm tolerates both the failure of single database sites as well as the failure of whole clusters. In this section we present and explain the recovery procedures in both cases.

Locally, each cluster is governed by a group membership service and local communication rests on view-synchronous multicast primitives. This definitely eases failure handling locally. In the event of a site been expelled from the group (because it was taken down, has fail, became unreachable, etc.) every other site in the group eventually becomes aware of the fact by installing a new view of the group. This allows each site to deterministically determine the cluster's delegate should the former failed. Moreover, view-synchrony

ensures that all sites surviving the previous view delivered the same set of messages, thus not requiring any synchronization among them. As a result, no particular procedure is required on the failure or an ordinary site. In the next two sections we examine the failures of a cluster's delegate and of the system's certifier. Then, we consider the failure of an entire cluster. For the sake of simplicity and lack of space, we assume that no sites are added to a cluster and that once a site is expelled from the group, whatever was the reason for this, it is no longer readmitted.

4.2.1 Delegate failover

In Figure 3a, we sketch a protocol for recovering from a site failure when this site was the cluster's delegate. On a view change, site d becomes aware it is the new cluster's delegate. To ensure that no transactions are blocked, d must rerun all transaction updates and acknowledgements received from remote clusters that may have been incompletely processed by the previous delegate.

New delegate: Synchronization request When initialized, the new delegate d sends a message (DELEGATE_SYNC, lts) to the certifier in order to ensure that all transactions certified since lts are delivered in its local cluster. The lts value corresponds to the latest transactions updated in d's cluster. The new delegate also contacts each remote cluster with (ACK_SYNC, lts, TRUE) acknowledging the local stability of all transactions up to lts, requesting similar action from the recipients (argument TRUE of the message).

Certifier: Handle synchronization request When delivering this message, the certifier resends (in order) each certified transaction with a certification timestamp larger then d's lts value.

All delegates: Synchronize ACK's When the message (ACK _SYNC, clts, reply) from cluster is delivered in a remote cluster C, the delegate of C regards all its updated transactions with $ts <= clts$ as acknowledged by cluster. It then just checks whether these transactions became stable in every cluster and proceeds accordingly. If reply was set to TRUE a similar message (now with reply set to FALSE) is sent back to the initializing delegate (just elected) so it can also update the respective acknowledgements.

4.2.2 Certifier failover

The most serious single server failure is when the current system's certifier becomes unavailable. When initialized, the new certifier advertises itself to all delegates. There may be previously certified transactions not yet known to new certifier so a state synchronization is due. Figure 3b shows our synchronization protocol in pseudocode. The code assumes two existing functions, blockCertification() and unblockCertification(). Their implementation is not shown, but they state whether all arriving certification requests should be buffered, awaiting the synchronization protocol to finish.

New certifier: Synchronization request The new certifier c starts by invoking blockCertification() and requesting from each cluster all the transactions they might have delivered and updated after the last one updated by c.

Each delegate: Send missing transactions When a (CERTSYNC _REQUEST, clts) is received by the delegate of a cluster C, it replies with a sublist of its updated transactions (tid, ts, ws, wv) such that ts $>$ clts, that is, not yet seen by the new certifier.

Certifier: Missing updates When processing a (CERTSYNC _REPLY, clts, missing) from remote cluster C, the new certifier c then checks each member of the missing list whether it has already received this transaction from another cluster. This will happen if two or more remote clusters both know about a transaction which is unknown by c. If not, the transaction is enqueued in the c's certified queue. As soon as all replies from remoteCluster() are delivered, c sets the certifiers counter gts to lts and starts distributing from its certified queue (1) locally transactions with $ts > lts$ and (2) remotely according to each cluster's last updated transaction. The certifier's gts counter is updated for each transaction distributed locally. Finished the update, certification is unblocked.

Figure 3a: Delegate failover

New delegate: Synchronization request

```
1  upon site is initialized as new delegate
2        rsend(DELEGATE_SYNC, lts) to certifier()
3        foreach cluster in remoteClusters() do
4              rsend(ACK_SYNC, lts, TRUE) to cluster
5  end
```

Certifier: Handle synchronization request

```
6  upon rdeliver(DELEGATE_SYNC, clts) from cluster
7        foreach (ctid, cts, cws, cwv) in certified do
8              if cts > clts then
9                    fifo_rsend(UPDATE_REM, ctid, cts, cws, cwv) to cluster
10 end
```

All delegates: Synchronize ACK's

```
11 upon rdeliver(ACK_SYNC, clts, reply) from cluster
12       foreach (utid, uts, uws, uwv) in updated do
13             acked = {}
14             if clts ≥ uts then
15                   add (utid, cluster) to acks
16                   foreach (utid, c) in acks do
17                         add c to acked
18                   if remoteClusters() ⊆ acked then
19                         u_vscast(STABLE_REM, utid)
20       if reply == TRUE then
21             rsend(ACK_SYNC, lts, FALSE) to cluster
22 end
```

Figure 3c:

All delegates: On failure of remote cluster

```
1  upon failure notification of cluster C
2        foreach (tid, ts, ws, wv) in updated do
3              acked = {}
4              foreach (tid, c) in acks do
5                    add c to acked
6              if remoteClusters() ⊆ acked then
7                    u_vscast(STABLE_REM, utid)
8  end
```

Figure 3b: Certifier failover

Global site variables

```
1  synch = []
```

New certifier: Synchronization request

```
2  upon site is initialized as the new certifier
3        blockCertification()
4        foreach cluster in remoteClusters() do
5              rsend(CERTSYNC_REQUEST, lts) to cluster
6  end
```

All delegates: Send missing transactions

```
7  upon rdeliver(CERTSYNC_REQUEST, clts) from certifier
8        missing = []
9        foreach (tid, ts, ws, wv) in updated do
10             if ts > clts then
11                   enqueue (tid, ts, ws, wv) to missing
12       rsend(CERTSYNC_REPLY, lts, missing) to certifier
13 end
```

Certifier: Missing updates

```
14 upon rdeliver(CERTSYNC_REPLY, clts, missing) from cluster
15       synched = {}
16       foreach (tid, ts, ws, wv) in missing do
17             if (tid, ts, ws, wv) ∉ certified then
18                   enqueue (tid, ts, ws, wv) to certified
19       add (cluster, clts) to synch;
20       foreach (c, ts) in synch do
21             add c to synched;
22       if remoteClusters() ⊆ synched then
23             gts = lts
24             foreach (tid, ts, ws, wv) in certified do
25                   if (ts > lts) then
26                         gts = gts + 1
27                         fifo_u_vscast(UPDATE_LOC, tid, ts, ws, wv)
28                   foreach (cluster, clts) in synch do
29                         if ts > clts then
30                               fifo_r_send(UPDATE_REM, tid, ts, ws, wv) to cluster
31             unblockCertification()
32 end
```

Figure 3: Failover handlers

4.2.3 Multiple failures

The WICE-SI protocol shall tolerate situations where multiple servers or entire clusters can fail abruptly. Most failure scenarios can be handled using a combination of the procedure for single servers. To avoid blocking during synchronization, we assume that all running synchronization routines are restarted if a delegate fails.

The only scenario which require special treatment is the loss of an entire cluster. In that case, the other clusters must be informed as soon as possible to allow blocking current and future transactions to become stable. A handler for this event is illustrated in Figure 3c.

To the reviewers: A correctness argument for the algorithm is given in the Appendix.

5 Evaluation

In replication protocols that rely on a system-wide uniform atomic broadcast, updates cannot be delivered before the message has been delivered (and acknowledged) by all sites. This means that a full round-trip to the most distant site $2 \cdot t_{max}$ is required before updates can be installed, regardless of the location of the initiator. As the probability of two transactions conflicting depends on the latency, this has a profound impact in the abort rate of DBSM and Postgres-R [11].

In WICE-SI, and considering two clusters C_A and C_B, total ordering of messages is performed using a sequencer, sited in cluster C_A, also referred to as the primary cluster. Each update transaction T's updates are installed as soon as the certification result is known, although visible to clients only after stabilization. Thus, it makes sense to distinguish between *install-interval* and *commit-interval*. Commit-interval denotes the time

Transaction Name	Empirical Distribution	Estimators	
Delivery	normal	mean=143.70	sd=2.33
Neworder	uniform	min=6.45	max=16.83
Orderstatus	normal	mean=1.66	sd=0.83
Stocklevel	uniform	min=1.85	max=2.33
Payment	normal	mean=2.26	sd=0.21

Table 1: CPU Times distributions (milliseconds).

from the end of execution until the transaction gets committed at the originating site and is still lower bounded by $2 \cdot t_{max}$. The install-interval is the time elapsed from the moment T finishes its optimistic execution until a given site installs the incoming updates. Ignoring intra-cluster latency, and considering transactions originated at C_A, the install-interval is negligible for servers in cluster C_A and close to t_{max} milliseconds in cluster C_B. On the other hand, for transactions originating in cluster C_B, the install-interval will be close t_{max} and $2 \cdot t_{max}$ milliseconds, for C_A and C_B respectively.

The most significant advantage of the WICE-SI protocol when compared to DBSM in a wide area network should therefore be its impact on the abort rate due to early delivery of updates. In this section, we experimentally verify this claim.

5.1 Experimental Environment

Experimental evaluation is conducted by running an actual implementation of the protocol within a simulated environment. By profiling real components with CPU cycle counters, the technique captures the actual overhead introduced by protocols [26]. By fine tuning the simulation components to accurately reproduce real components, it realistically reproduces results of real distributed systems [28]. When compared to testing in a real setting, this allows a tight control over experimental conditions, with advantages in repeatability and observability. The approach has been previously used to evaluate database replication protocols both in LANs and WANs [11]. In detail, we use simulated database clients, database engines and networks, and real implementations of replication and group communication protocols.

The workload generator is configured according to the industry standard on-line transaction processing benchmark TPC-C [30]. Briefly, a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. This workload is update intensive, as 92% of the transactions perform updates. It is also varied, as the *delivery* transaction takes a considerable amount of CPU time and has a very large read-set. The *payment* transaction is likely to produce Write-Write conflicts. The *neworder* transaction is short-lived and with higher locality.

The results thus vary according to the platform used for calibration of the simulated environment [28]. Results presented in this paper are therefore referred to the following hardware configuration: Each server has a single CPU AMD Opteron 250 running at 2.4GHz, 4Gb RAM and a RAID 5 SATA disk array with fibre attachment. Transaction processing engines and overheads are configured according to PostgreSQL 8.0. Storage throughput as measured at the transaction log is 40MBytes/s. CPU overheads are presented in Table 1 along with the corresponding generator distribution and estimators parameters. With properly configured indexes and within the range of presented results, it was verified that these are independent of the size of the database, as dictated by TPC-C scaling rules. Note also that these values do not include contention, as when blocked waiting for a resource, processes are not scheduled. Also according to PostgreSQL 8.0, transaction processing engines use a multi-version concurrency control approach.

In our target scenario, 3 database servers are positioned at each of two different sites, as shown in Figure 4. The network simulator is configured as a pair of switched 1Gbps Ethernet local area networks, connected by a dedicated T3 link (45Mbps) with 400ms round-trip latency, representative of an inter-continental satellite link. As a baseline, we present also results obtained when configuring all 6 servers within the same local area network.

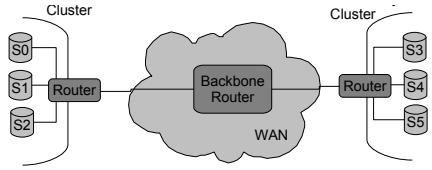

Figure 4: Network Topology.

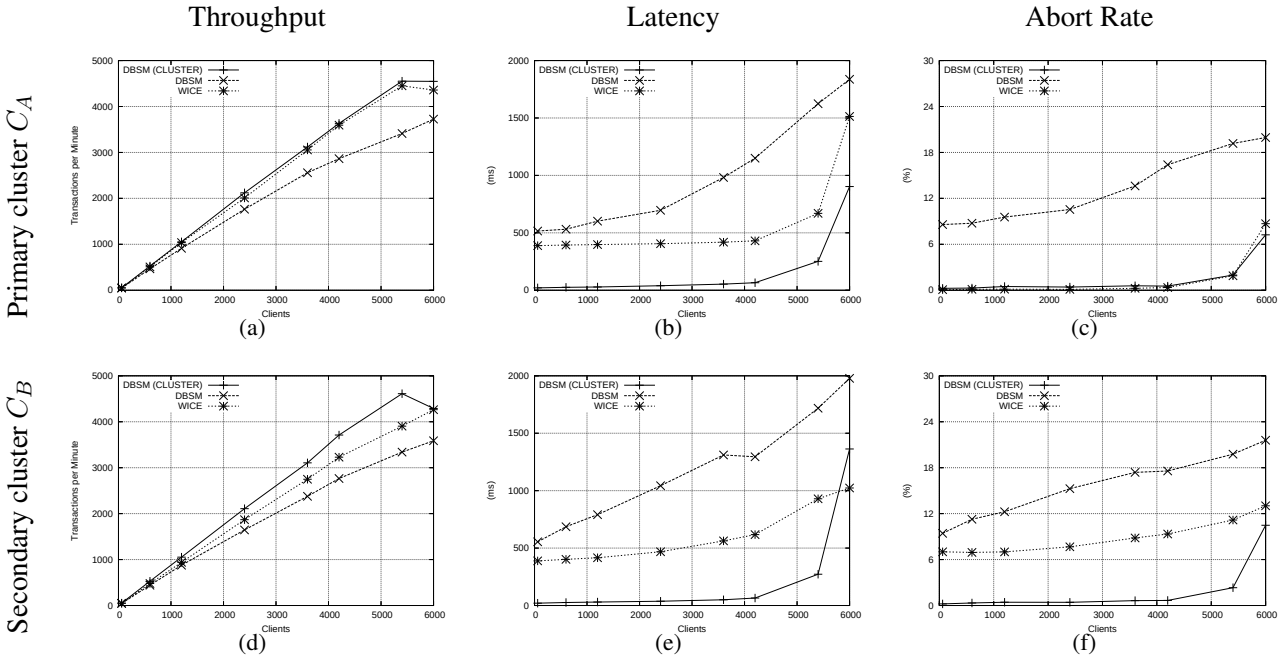

Figure 5: Performance results with 1-SI.

In all scenarios, we vary the number of simulated clients from 60 to 6000, equally spread by all servers. We also take advantage of the locality in TPC-C: Clients associated with same warehouse are connected with the same server to exploit locality, as suggested by the TPC-C specification. Note however, that with a small probability any client updates records associated with any warehouse.

5.2 Performance Results

The performance of the WICE-SI protocol is evaluated by observing the throughput, latency and abort rate achieved when compared with plain DBSM. As a baseline, we present results obtained by grouping all 6 servers in the same cluster (DBSM CLUSTER). The results are obtained with Write-Write conflict certification, thus achieving 1-SI, are presented in Figure 5. Results are presented separately for each cluster.

The first interesting observation from the baseline protocol (DBSM CLUSTER) is that the capacity of the system is exhausted with 6000 clients. This shows up as throughput peaking (Figure 5(a)), increasing latency due to queuing (Figure 5(b)), and abort rate due to increased concurrency (Figure 5(c)). By examining resource usage logs one concludes that this is due to saturation of available CPU time. We should thus focus on system behavior up to 4000 clients, as a properly configured system will perform flow control to ensure operation in that range. Namely, throughput grows linearly, latency is approximately constant and the abort rate negligible.

Then, we turn our attention to DBSM in the target scenario. Although throughput scalability is apparently close to linear, it is misleading as it corresponds to a high abort rate and a linearly increasing latency, in particular in cluster C_B (Figures 5(e) and 5(f)). Both are explained by the same phenomenon: As locks are withheld during wide area stabilization, queuing delays arise, thus proportionally increasing the probability of

later being aborted. Aborted transactions have to be resubmitted by the application, thus further loading the system. It is also important to underline that latency and abort rate impact both clusters equally, as expected, as both suffer with the same $2 \cdot t_{max}$ commit-interval.

As expected, the WICE-SI protocol improves the performance at the primary cluster without negatively impacting secondary clusters. Namely, in the primary cluster the abort rate is negligible (Figure 5(c)), comparable only with the DBSM CLUSTER scenario. The latency is also approximately constant in the safe operating range (i.e. up to 4000 clients), although impacted by the round-trip over the wide area link (Figure 5(b)). Note however that such impact is very close to the absolute minimum of $2 \cdot t_{max}$ at 400 ms.

Also as expected, the abort rate of transactions initiated in the second cluster, which are impacted by a t_{max} to $2 \cdot t_{max}$ commit-interval, is not negligible although still offering a substantial improvement on DBSM. In the next section, we discuss the impact of this in the expected usage scenario of WICE-SI.

5.3 Discussion

The workload assignment used in the previous section deserves some additional comments. The WICE-SI protocol targets the global enterprise where the goal of replication is twofold. First, by providing a cluster for each region of the globe one avoids having route all queries to a central location and thus avoid imposing the large latency on clients when no updates are performed, while at the same time balancing the load. Second, it improves availability as even catastrophic disasters can only impact the computing or communication infrastructure at a single location. One has therefore to consider clusters located in different timezones, having distinct peak utilization periods.

This means that the evaluation scenario in the previous section, in which traffic in both clusters is exactly the same, is the worst case scenario for the proposed protocol. In reality, one should be able to migrate the centralized sequencer to the currently most loaded cluster. The additional abort rate at other locations can then be easily solved by resubmission, as these clusters are off peak and thus with underutilized resources.

We also have not assumed that resubmission can be done automatically by the database management system. However, this is true for many workloads, especially in current multi-tiered applications. By taking advantage of such option one could thus completely mask the abort rate at secondary clusters.

6 Conclusion

Eager update-everywhere update database replication optimized for interconnected clusters in wide area networks is a valuable contribution to the infrastructure of the global enterprise. By providing the ability to locally serve clients it improves performance and by allowing failover ensures disaster recovery with no data loss. This is a hard problem, which existing commercial solutions address either by admitting some data loss or by centralizing update processing.

The proposed WICE-SI protocol shows how to scale replication protocols based on group communication to a wide area setting with increased performance, while at the same time increasing their practicality. This is achieved by restricting group communication within clusters and using a simple peer protocol over long distance links. The evaluation performed in a realistic platform illustrates the advantages of the approach, namely, linear throughput scalability, up to 2 times less latency and a negligible abort rate at the cluster supporting the region currently generating the most traffic.

The protocol can relatively easily be generalized to offer full serializability if needed, but at the cost of higher abort rate. Our next target is to present this generalized version together with a performance evaluation.

References

[1] M. Abdallah, R. Guerraoui, and P. Pucheral. Dictatorial transaction processing: Atomic commitment without veto right. *Distributed Parallel Databases*, 2002.

[2] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. Practical wide area database replication. Technical report, Center for Networking and Distributed Systems, Computer Science Department, Johns Hopkins University, 2002.

[3] T. Anderson, Y. Breitbart, H. F. Korth, and A. Woo. Replication, consistency, and practicality: are these mutually exclusive? In *Proceedings of ACM SIGMOD international conference on Management of data*, 1998.

[4] Apache. Apache derby. http://db.apache.org/derby.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, 1995.

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

[7] M. J. Carey and M. Livny. Conflict detection tradeoffs for replicated data. *ACM Transactions on Database Systems*, 1991.

[8] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2), March 1996.

[9] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 2001.

[10] GORDA Consortium. Gorda - open replication of databases. http://gorda.di.uminho.pt/consortium, October 2004.

[11] A. Correia Jr., A. Sousa, L. Soares, J. Pereira, R. Oliveira, and F. Moura. Group-based replication of on-line transaction processing servers. In *Dependable Computing: Second Latin-American Symposium*, 2005.

[12] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *Proceedings of The 24th IEEE Symposium on Reliable Distributed Systems*, 2005.

[13] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 2005.

[14] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 1985.

[15] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 1997.

[16] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

[17] PostgreSQL Inc. Postgresql. http://www.postgresql.org.

[18] B. Kemme and G. 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*, 2000.

[19] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Consistent data replication: Is it feasible in wans? In *Euro-Par*, 2005.

[20] C. Metz. Tcp over satellite... the final frontier. *IEEE Internet Computing*, 1999.

[21] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of The IEEE 21st International Conference on Distributed Computing Systems*, 2001.

[22] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distributed Parallel Databases*, 2003.

[23] J. Pereira, A. Correia Jr., N. Carvalho, S. Guedes, R. Oliveira, and L. Rodrigues. Database interfaces for replication support. Technical report, Universidade do Minho/Faculdade de Ciências da Universidade de Lisboa, 2006.

[24] L. Rodrigues, J. Mocito, and N. Carvalho. From spontaneous total order to uniform total order: different degrees of optimistic delivery. In *In Proceedings of the 21st ACM Symposium on Applied Computing*, 2006.

[25] F. Schneider. Replication management using the state-machine approach. In *Distributed Systems*. ACM Press/Addison-Wesley Publishing Co., 1993.

[26] L. Soares and J. Pereira. Experimental performability evaluation of middleware for large-scale distributed systems. In *7th International Workshop on Performability Modeling of Computer and Communication Systems*, 2005.

[27] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic total order in wide area networks. In *Proceedings of The 21st Symposium on Reliable Distributed Systems*, 2002.

[28] A. Sousa, J. Pereira, L. Soares, A. Correia Jr., L. Rocha, R. Oliveira, and F. Moura. Testing the dependability and performance of group communication based database replication protocols. In *IEEE International Conference on Dependable Systems and Networks - Performance and Dependability Symposium*, 2005.

[29] Symantec. Veritas backup software. http://www.symantec.com/enterprise/veritas/index.jsp.

[30] Transaction Processing Performance Council (TPC). TPC benchmark^{TM} C standard specification revision 5.0, February 2001.

[31] M. Tumma. *Oracle Streams - High Speed Replication and Data Sharing*. Rampant TechPress, 2004.

[32] S. Wu and B. Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *International Conference on Data Engineering*, 2005.

A Consistency Discussion

In this appendix, we discuss how our protocol provides consistency. Our arguments are presented through four *properties*. The accompanying discussion is not to be regarded as a correctness proof, but the basic ideas are sketched.

Property 1 If a transaction T is successfully certified, it will be executed by any correct server that delivers T's updates.

Correctness argument Updates are processed as soon as they are delivered. Thus, it is sufficient to show that when delivering the updates of a certified transaction, no correct site will abort T.

We first consider the case where T is aborted at the initiator s while awaiting the certification result. When T is sent for certification, it has finished execution, is validated by local concurrency control and the user has submitted a commit-request. Consequently, it can only be aborted by a remote, conflicting transaction. But as any transaction with the potential to abort T by breaking its locks must be certified before T, we are assured that T will fail certification if there is any chance of it being aborted at the initiator. Consequently, the property holds.

Property 2 If T is certified and T's updates are executed by some correct server s, T is eventually executed by all correct servers.

Correctness argument From Property 1, we have that T will be executed at all correct servers where the updates are delivered. Consequently, we must show that T's updates will eventually be delivered by all correct servers.

In a failure-free run, this is trivial: All certified transactions are sent to all delegates immediately after certification, and each delegate unconditionally forwards this update-message within the cluster.

In the case where either a delegate or the certifier fails during the distribution of T's updates, the distribution of updates may be incomplete. First, consider the case where the delegate of a cluster C fails. A new delegate is elected, and courtesy of the uniform vs_cast used to distribute updates within a cluster, we know that either none or all servers in C know about T. It is thus sufficient for the delegate to ask the certifier to resend any missing transactions (see Figure 3a).

Now, consider the situation where the certifier fails. In this case, there might be some certified transaction T' that is executed in some cluster, but which is not known by the new certifier. To ensure that T' is eventually delivered at all servers, the new certifier first requests any missing transactions from all cluster. When all correct clusters have responded, any updates known to be missing somewhere are resubmitted (see Figure 3b).

From this, we claim that Property 2 holds.

Property 3 If transaction T_i is applied before transaction T_k at site s, no site applies T_i after T_k.

Correctness argument Recall from the paragraph "Transaction Commit" of Section 4.1 that a transaction T is applied when the local database-version is set equal to T's order-timestamp. This can be proved by showing that if $w_i(x)$ of T_i is executed before $w_k(x)$ of T_k at some site s, T_i has been certified before T_k.

First, we consider the failure free runs. The certification handler is atomically executed, and as update-messages are distributed within the handler using FIFO-channels, the 'UPDATE_LOC'-message of T_i is received before the 'UPDATE_LOC'-message of T_k at s if and only if T_i was certified before T_k. Thus, transactions are added to the pending-queue in certification order.

From line 56-65 in Figure 2, it should be clear that the updates of a transaction can be applied only if all previous transactions are applied as well, and that this always happens in pending-queue order.

In the case where a delegate or the certifier fails during the distribution of either T_i or T_k, we must show that the respective synchronization protocols preserve the correct ordering of updates. Since a failure can only delay distribution, we focus on the the case where a failure occurs during the distribution of T_i, i.e. where there's a chance for T_k to be delivered before T_i if this is not handled correctly. Since FIFO-channels are used to distribute updates, T_k can never be delivered before T_i at any server s if the certifier sends T_i before T_k. To verify that this is indeed the case, we note the following:

- Whenever the message 'UPDATE_REM' is sent from the certifier, it is *either* sent upon certification or while traversing the certified-queue.
- As long as the certifier does not fail, the *certified*-queue holds previously certified transactions in the order of certification, by property of the enqueue-operation (see line 24, figure 2).
- Similarly, the updated-queue of each site stores transactions in the order of delivery. This means that as long as the certifier has never previously failed, this queue stores transactions in certification order.
- When the certifier requests missing transactions, these are added to the missing-queue by traversing the updated-queue.
- Finally, when a missing-queue is received from a delegate at the certifier, each entry is added to the certified-queue by in-order traversal.

Consequently, the certified-queue obeys the certification order after certifier-failover. When the new certifier is synchronized, missing transactions are submitted in the correct order. Based on this, we claim that updates are always distributed in certification order, and consequently, that Property 3 holds.

Property 4 WICE-SI provides snapshot isolation.

Correctness argument Snapshot isolation (SI) is an isolation level which does not guarantee serializability, but instead represents a pragmatic compromise. In [5], SI is defined as a follows: (1) Each read operation $r_i(x)$ executed by a transaction T_i reads from the most recent updater of x which was committed when T_i began execution; and (2) For any pair of concurrent update transactions T_i and T_j, either $ws(T_i) \cap ws(T_j) = \emptyset$ or one of the transactions are aborted. T_i and T_j are concurrent unless we have $commit(T_i) < begin(T_j)$ or $commit(T_j) < +begin(T_i)$.

According to this definition, our protocol does not guarantee SI since transactions may not *commit* until they are uniform. But in [5], SI is also defined in terms of the set of concurrency phenomena it precludes. The complete list of phenomena identified by [5] are *dirty write*, *dirty read*, *cursor lost update*, *lost update*, *fuzzy read*, *phantoms*, *read skew* and *write skew*. Only *write skew* and some forms of *phantoms* are allowed by snapshot isolation, all the others are forbidden.

We claim that our protocol provides the same protection against these phenomena as the definition quoted above. In outline, our protocol maintains concurrency control as follows: (1) Each read operation $r_i(x)$ executed by a transaction T_i reads from the most recent updater of x which was applied at T_i's initiator when it began execution; (2) No transaction is allowed to commit unless all transactions it has read from are already committed; and (3) For any pair of concurrent update transactions T_i and T_j, either $ws(T_i) \cap ws(T_j) = \emptyset$ or one of the transactions are aborted. Let $vts(T_i)$ denote the version-timestamp given to T_i by the initiator, and let $cts(T_i)$ denote the order-timestamp assigned by the certifier. Then, two transactions T_i and T_j are concurrent unless we have $cts(T_i) <= vts(T_j)$ or $cts(T_j) <= vts(T_i)$.

Property (1) and (2) are both guaranteed by our assumptions. (1) follows from the rule for updating lts, as described in Section 3. Property (2) is discussed in the introduction of Section 4. Finally, property (3) is ensured by our certification procedure. Assume, without loss of generality, that the write set of transaction T_j intersects with a previously certified transaction T_i. From our validation test (line 20, Figure 2, T_i will be aborted unless $cts(T_i) <= vts(T_j)$.

Now we show below that our system disallows the same set of phenomena as the original definition of snapshot isolation. When reading the following discussion, recall from Section 4.1 that the final *commit* of a transaction T_i only happens when its result is allowed to be externalized, i.e. only when the transaction is *committed* and *stable* (see Figure 1 and line 66 to 68 in Figure 2).

A **dirty write** is in [5] defined as
$w_1(x)...w_2(x)...(c_1 \text{ or } a_1)$. In other words, a dirty write occurs if some transaction T_1 first updates an object x, and another transaction T_2 is allowed to write x before T_1 is terminated.

Strictly according to this definition, WICE-SI does not protect against dirty writes: A transaction T_2 can update an object previously written by T_1 at some site s as soon as T_1 is known as T_i's updates are applied at s.

But when studying the possible harmful consequences of dirty writes given in [5], we see that none of them really apply to our case: *(1) Dirty writes complicates undo-logging*. Consider the history $w_1(x)w_2(x)a_1$. Here, restoring T_1's before-image would delete T_2's updates. But this problem does not apply to our setting, as we have the following assertion: As soon as a transaction T_1 is certified, it can only abort in the case that all sites hosting T_1 fails. Consequently, T_1 could not fail unless T_2 fails as well. And as we assume that sites do not recover after failures, this is not an issue with our protocol. *(2) Two transactions might write two objects in different order, i.e.* $w_1(x)w_2(x)w_1(y)c_2w_1(y)c_1$. But since updates are applied in certification order, this situation cannot occur here. Consequently, we claim that WICE-SI provides protection against dirty writes without explicitly forbidding them.

A **dirty read** is described by the sequence $w_1(x)...r_2(x)...(c_1 \, or \, a_1)$[5]. Again, this is not strictly forbidden by our protocol, as the updates of T_1 may be available to other transaction long before it is stable and allowed to commit. But as noted in Section 4, T_2 is not allowed to commit until T_1 is stable. Thus, we ensure consistency by requiring that if T_1 aborts, T_2 must also be aborted. This is usually denoted *cascading aborts*, and is regarded as a problem. But T_1 can only be aborted by a site failure, in which T_2 would be aborted as well. Thus, we claim that WICE-SI protects against dirty reads.[4]

Cursor lost update and **lost update** are executions on the form $r1(x)...r2(x)...w_1(x)...w_2(x)...c_1$[5]. In this case, the updates performed by T_1 are unknown by T_2, and are consequently "lost". This is avoided if we require that for any pair of transactions updating a common object, one of them must see the other's updates.

In both modes, this is ensured by the validation procedure: Without loss of generality, assume that T_2 is submitted for certification after T_1. If T_2's write-set intersects with T_1, one requirement for validating T_2 is that it has seen T_1's updates (and this is easily determined by the version-timestamp assigned by the initiator of T_2, see line 6 of Figure2). Consequently, lost updates can never happen.

A **fuzzy read** occurs when a transaction T_1 reads an object x which is updated by another transaction T_2 before T_1 has finished execution. In [5], this is described by the following pattern: $r1(x)...w_2(x)...(c_1 \, or \, a_1)$. From the description and example given in [5], it is clear that what matters is whether T_2 is allowed to update T_1 before the latter has executed its operations, regardless of when it commits. This distinction may not be important in a centralized database, but in our setting it is significant, as we want to allow concurrent transactions access before commit.

In either case, T_1 is only allowed to read from *certified* transactions by the local DBMS, and any updates applied after T_1 is initiated are hidden. Consequently, WICE-SI protects against fuzzy read.

A **phantom read** can happen if transaction T_1 executes a statement containing a query with a predicate P, and another transaction T_2 executes an insert, update or delete-operation such that the set of objects matching P is changed. This can lead to inconsistency, for instance if T_1 later executes a similar query on P.

In [5], the pattern $r_1(P)...w_2(y \, in \, P)...(c1 \, or \, a1)$ is used to describe this scenario.

It is clear that some phantom-problems are handled by the local DBMS at each site, since if T_1 re-executes a query on P at the initiator, it will read from the snapshot and thus return the same set of tuples. But as pointed out in [5], another potential problem is if two transactions T_1 and T_2 both run predicate reads and update distinct objects. Given an execution on the form $r_1(P)...r_2(P)...w_1(x \, in \, P)...w_2(y \, in \, P)$, inconsistency could arise if the values written to x and y depend on the set of tuples returned by the previous queries. This is not handled by our protocol (and can happen regardless of whether T_1 and T_2 has the same initiator). But as stated by [5], this concurrency phenomenon is allowed by snapshot isolation.[5].

Read skew is described by [5] as executions on the form $r1(x)...w_2(x)...w_2(y)...c_2...r1(y)...(c_1 or a_2)$.

[4] One can argue that this approach will cause more cascading aborts among read-only transactions, as these could avoid blocking by reading from an older snapshot, and thus might commit before the failure occurs. If supported by the local DBMS, read-only transaction might then be assigned a *committed-snapshot*, as opposed to *certified-snapshot* to avoid blocking queries unneccessary.

[5] As stated by [13], this phenomenon is an example of a *predicate-based write skew*. Thus, one can say that snapshot isolation protects against all phenomena except write-skew and predicate-based write-skew

The problem pointed out is that the state returned by T_1 may be inconsistent, as it reads one of T_2's updates but not the other.

Note that this can only happen if T_1 and T_2 have the same initiator (as we assume full replication, read-operations are only issued at the initator). As snapshot isolation dictates that T_1 can only read from transactions that where certified when it began executing, this situation can never occur.

A **write-skew** may occur if two transaction T_1 and T_2 reads two objects x and y, and subsequently updates one each. The general definition, as given by [5], is $r_1(x)...r_2(y)...w_1(y)...w_2(x)...(c1\,and\,c2\,in\,any\,order)$. It is well known that snapshot isolation does not protect against this phenomenon, and consequently, neither does WICE-SI.

Based on the discussion above, we claim that WICE-SI provides snapshot isolation as it protects against all phenomena except write skew and phantoms, i.e. the same isolation as in one-copy SI.