© Copyright 2003 Bolour Computing.

Dynamic Primary Site Migration: An Approach to Update Conflict Avoidance in Replicated Databases

Azad Bolour

Abstract

An approach to conflict avoidance for update-anywhere transactions in a replicated database is outlined. This approach is based on dynamically changing the primary site of records to be updated by a transaction. Different variations of this approach provide a range of compromises between performance, consistency, and availability.

1. Symmetric Replication Based on Primary Site Migration

1.1. Symmetric Replication

In a typical replicated database, different *database sites* are connected by WAN connections. But application clients at each site connect to the database at that local site via LAN connections.

We use the term *symmetric replication* (*SR*) to mean replication in a database with the following characteristics:

- **Primary site model**. Each record has a unique primary site at which that record is allowed to be updated by application programs. We say that a site *owns* (or is the *owner* of) a record, if that site is the primary site of the record.

- **Asynchronous replication to other sites**. Updates are replicated asynchronously to the updating transaction by the underlying DBMS replication service to every other site.

- **Location independence**. Application programs are independent of the primary sites of the records they access. Because location independence can be difficult to achieve, a milder requirement, which we call *unrestricted updatability* is sometimes the best one can hope in SR. Unrestricted updatability means that an application is able to update any record but not necessarily transparently of the primary site of the record.

When a transaction is begun by a client connected to the replica of the database at a local site, some of the records to be accessed and updated by that transaction may be owned by remote sites. A central conflict in symmetric replication is how to access and update such remotely-owned records while preserving transaction semantics.

1.2. Conflict Avoidance

Database management systems generally support update-anywhere replication by allowing an application to update the copy of a record at the local database, and by providing support for after-the-fact *resolution* of any conflicts between that local update and concurrent updates of the copies of that same record at other sites. But there is no guarantee that an arbitrary application is amenable to conflict resolution: that simple procedures can be devised for it to resolve conflicting concurrent updates from different sites.

In this paper, we are concerned instead with conflict avoidance. A conflict-avoiding SR system provides an *application-level replication substrate*: a set of modules and database design rules (including requirements on update triggers) that make conflict avoidance and location independence as transparent as possible to application programs. Where total transparency cannot be provided by the substrate, the substrate may instead allow unrestricted updatability, by placing some minimal demands on application programs.

We take the replication mechanisms of commercial database systems as a given. Our aim is not to explore

system-level enhancements to DBMS replication services. Rather we have the more modest goal of enhancing the given replication services of commercial databases by additional functions in the application-level replication substrate. We use the term *replication substrate* to refer to the combination of the application-level substrate and the underlying DBMS replication service.

Figure 1 depicts the architecture of a symmetric replication system.

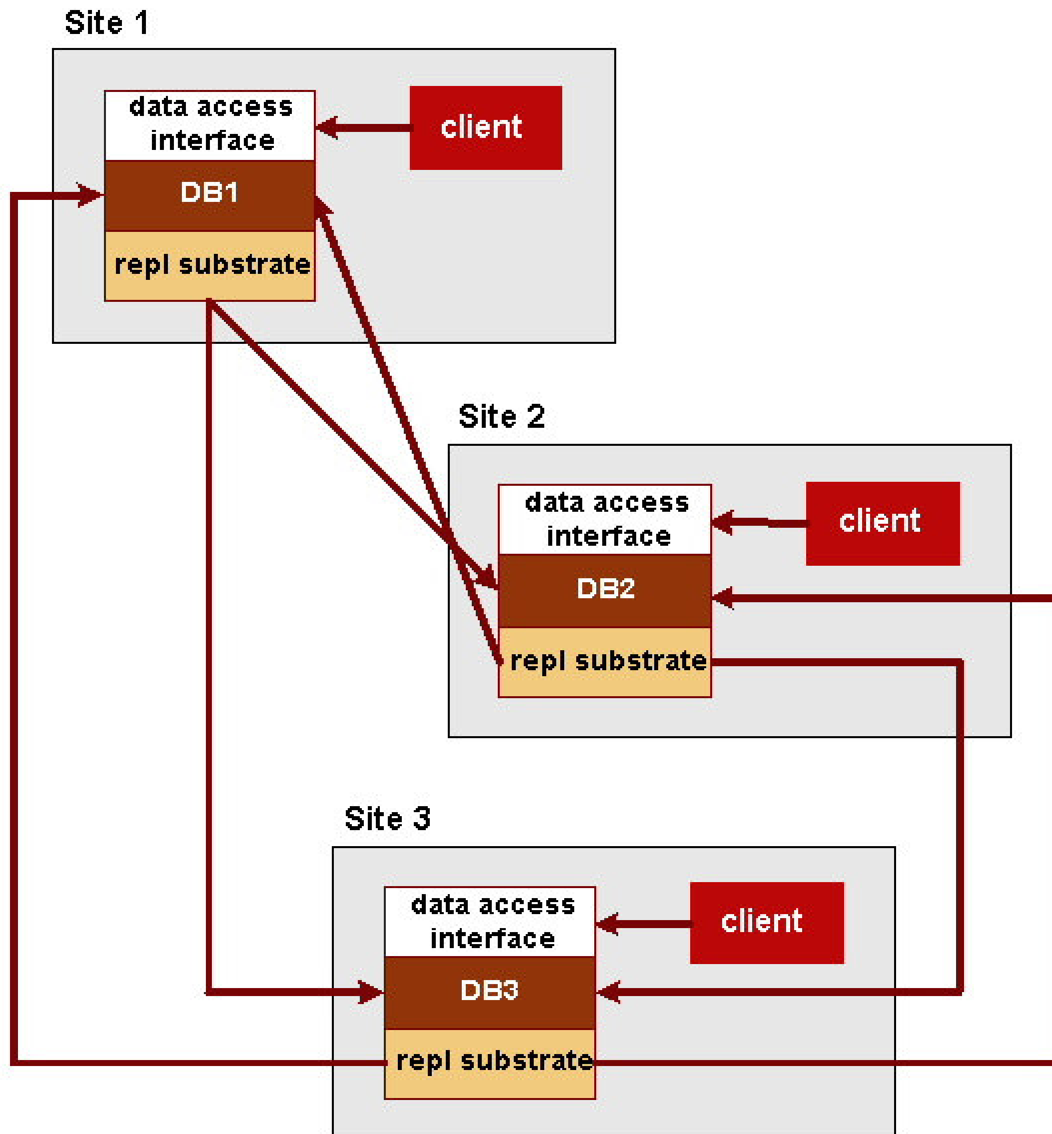

Figure-1. Symmetric Replication Architecture.

In this architecture, the data access interface includes a standard call-level SQL interface such as JDBC, and a specialized replication support interface used to support unrestricted modifiability. On the other hand, the replication substrate includes system level components to replicate updates to other sites, and application level components (triggers, and stored procedures) to support conflict avoidance.

The replication substrate provides certain consistency guarantees to application programs, provided that the substrate's database design rules are followed, and that application programs conform to the (minimal) requirements of the replication support interface.

The design of one such application-level replication substrate is outlined in this paper. Such a design tries to resolve the following forces: performance, consistency, availability, and location independence.

1.3. Consistency/Performance/Availability Trade-offs

For performance reasons, it is desirable to place the primary copy of each record at a site that minimizes WAN access to that copy to update it or to obtain its most recent version. Consider a transaction T that is initiated at a client node on site `S1`'s LAN. It would be ideal if each record accessed by T is owned by the database replica at site `S1`. That way, the client connects to the local database and completes the entire transaction via that connection. Because the connection is local, latency would be minimized. Also because the entire record set of the transaction is primary at the local site, there would be no need either to verify the transaction's reads by comparing them with corresponding primary copies at other sites, or to use the two-phase commit protocol to commit the transaction's writes. (A touted advantage of using replication is avoidance of distributed transactions and their associated performance and administrative costs.)

In practice, this ideal partition of records into sites can sometimes be approximated, but is seldom achievable. The need for performance reasons to complete a transaction by a single connection to a local database, and the need for consistency reasons to connect to remote databases for verification and update of records not owned by the local site are thus opposing forces in the design of a replication substrate. In order to enhance performance, the designer may decide to forego certain consistency guarantees in the replication substrate.

A similar tension exists between the requirements of data availability and consistency.

Dynamic primary site migration is one approach to resolving the conflicting demands of performance, consistency, and availability. This approach is appropriate in applications that exhibit *site affinity* of accessed records over short time periods, but where site affinity may change, albeit slowly, over longer time periods. A record has *affinity* to a site during some time interval, if that site is the only site that updates the record or requires up-to-date access to it during that time interval.

This work is a continuation of earlier work reported in [1]. The dynamic primary site migration algorithm presented in [1] has a major consistency flaw with 3 or more replicas. In this paper, the issue of consistency is discussed in more detail, and the algorithm presented in [1] is further refined to address its shortcomings. The algorithm presented in [1] was implemented by using the Sybase replication server and used for in a replicated software lifecycle database.

Our refinements of that algorithm presented in this paper, we believe, can be added to our previous implementation relatively easily. However, such an implementation has not been attempted. Similarly, no implementation has yet been attempted with other database systems, though in principle, a similar implementation should be possible with Microsoft SQL Server.

The evolution of this approach has been greatly influenced by the work of Adya [2].

Historical note. The idea of using a unique site to control updates to a database object can be traced back to [3] and [4]. The idea of allowing the controlling site to change over time has also existed for some time; see, for example, [5].

1.4. Overview of the Basic Approach

The main idea presented in [1] for symmetric replication is **dynamic primary site migration** or **change of record ownership**: if a transaction is begun at a site `S1`, and a record that is to be updated within that transaction has a different primary site, say, `S2`, then the primary site of the record is first changed dynamically from `S1` to `S2`, and then the update is executed at `S2`. Ideally, the change in the primary site occurs transparently to the application program within the application-level replication substrate. In any case, the change in the primary site occurs via an independent transaction at the record's primary site, `S2`. In this manner, what would otherwise be a distributed transaction is converted to a local transaction, and the use of the two-phase commit protocol can be avoided.

In its simplest form, this approach ignores the verification of a transaction's reads in the application-level replication substrate.

To implement dynamic primary site migration, two additional pieces of information are kept for each record, the record's *primary_site*, and the record's *timestamp*: a version number for the record. When a record is updated, its timestamp is incremented transparently to the application in an update trigger for the record's table. If the primary site of the record is not the local site where the application has issued the update, then the record's primary site must be changed to the local site of the transaction before the update occurs (or before it is committed).

1.4.1. Verifying Up-to-Dateness at a New Primary Site

Since the primary site of the record must contain its most recent version, the primary site cannot be changed to the local site unless the version of the record at the local site is identical to its version at its current primary site. Because all updates are replicated to all sites in an SR regime, shortly after the transaction that created the latest version of a record commits, that version is replicated by the DBMS replication service from its current primary site to all other sites, including the local site of the transaction at hand. So it is likely that the migration algorithm finds this latest version of the record at the local site. To check that the version at the local site is the latest version, the timestamp of this version is compared to the timestamp of the version of the record at its current primary site.

If the latest version of the record is not found at the local site within a configured timeout, the migration algorithm fails, causing the transaction at hand to abort. The application program is therefore required to retry such a failed transaction. To the extent that an application exhibits site affinity, the probability of such failures will be low. And since the application must deal with similar failures caused by deadlocks, the need to retry a transaction subsequent to a migration failure can easily be accommodated in the application program.

This approach addresses conflicts between concurrent updates. If we treat deletes as the creation of special *deleted* versions of records, this approach can also be extended to deletes.

1.4.2. Insertions and Uniqueness Constraints

Insertions, on the other hand, present special issues for dynamic primary site migration. When inserting records with arbitrarily-defined surrogate keys, the key space may be partitioned into ranges belonging to each site, and an insertion initiated at a local site given a key in the range for that site. In this way, surrogate key conflicts may be avoided. Such an approach may work, for example, for creating new bank accounts in different regions of the world.

But avoiding conflicts in externally-supplied application-level primary keys (e.g., social security number) is a more challenging problem. To avoid such conflicts, we make use of two devices: *unborn primary site* and *unborn record version* (the latter terminology borrowed from Adya [2]) Each potential record is assigned a primary site, called its *unborn primary site*. The unborn primary site is determined by some application-defined mapping of the primary key space to the set of primary sites, e.g., via hashing or range partitioning. Before a record is inserted at the local site, an *unborn version* of it is created at its unborn primary site. The unborn version is specially timestamped to indicate that it is yet to be *inserted*. Then the unborn version's primary site is migrated to the local site where the insertion is to occur.

In this manner, all insertions of a record with the same primary key first attempt to create an unborn version at the record's unborn primary site. If there is a concurrent insertion, the attempt to create a second unborn version fails at the record's unborn primary site. (Crash resistance of this procedure is discussed later in this paper.)

Unfortunately, this approach does not generalize in a simple manner to multiple application-level uniqueness constraints on the same table. Insertion into such tables requires a more drastic approach such as having a unique unborn site for the table as a whole.

1.5. Missing Updates

When using dynamic primary site migration, unpredictable network delays may cause successive updates to a record made at different sites to reach a given replicate site out of order.

Example 1. Missing Updates

Suppose that a transaction `T1` at site `S1` updates a record `a` to version `a[1]`, and then the primary site of the record changes to site `S2`, where a transaction T2 updates it to version `a[2]`. Both `a[1]` and `a[2]` would be replicated to a third site, `S3`. But `a[2]` may reach `S3` before `a[1]`.

1.5.1. Ensuring Record-Level Monotonicity

The replication substrate includes procedures to ignore out-of-sequence updates, and thereby ensures record-level monotonicity. At the replicate site, the timestamp of an arriving replicated version is compared with the timestamp of the existing version at the site, and the newly arrived version ignored unless it has a higher timestamp.

This procedure is implemented in Sybase by using so-called *replication function strings*: procedures triggered by the arrival of replicated records to a replicate site. In other database systems, monotonicity of records can, in principle, be checked by using special triggers that fire when an update reaches a replicate site, and by undoing the effect of an out-of-sequence update. However, the details of using replicate site triggers for this purpose remain to be worked out.

1.5.2. The Missing Update Anomaly

Even though dynamic primary site migration makes it possible to avoid conflicts in application level updates, the fact that updates may occur at different sites, and they may reach replicate sites with varying amounts of delays, implies that inconsistencies at replicate sites cannot be ruled out under this regime (even if a transaction's reads are verified as current).

Example 2. Missing Update Anomaly

Continuing with Example 1, suppose transaction `T1` at site `S1` updates two records, `a`, and `b`, say, from versions `a[0]` and `b[0]` to versions `a[1]` and `b[1]`, that must be consistent with each other, and then transaction `T2` at site `S2` only updates `a` to version `a[2]` (consistently with `b[1]`), and consider again the case where `T2`'s updates reach a third site `S3` before `T1`'s. In this case, the replication of `T2`'s writes to `S3` does not include `b[1]`. Thus, `S3` now contains `a[2]` and `b[0]` with no guarantee of consistency.

More specifically, suppose `amount` is a field of these records, and there is a consistency constraint that `|a.amount - b.amount| <= 5`. Let the initial amounts for both records be 0, and suppose `T1` changes both to 10 (leading to a difference of `10 - 10 = 0` which preserves the consistency constraint), and `T2` changes `a.amount` to 15 (leading to a difference of `15 - 10 = 5` which also preserves the consistency constraint). When `T2`'s update is replicated to `S3`, `S3` will have `a.amount = 15` and `b.amount = 0`, leading to a difference of 15, which violates the consistency constraint. Figure 2 summarizes this example.

Operation	Site	a			b		
		amount	psite	ts	amount	psite	ts
initial state	**S1**	**0**	**S1**	**0**	**0**	**S1**	**0**
	S2	0	S1	0	0	S1	0
	S3	0	S1	0	0	S1	0
T1 @S1 a.amount = 10,	**S1**	**10**	**S1**	**1**	**10**	**S1**	**1**
	S2	0	S1	0	0	S1	0

b.amount = 10	S3	0	S1	0	0	S1	0
T1 replicated to S2	**S1**	**10**	**S1**	**1**	**10**	**S1**	**1**
	S2	10	S1	1	10	S1	1
	S3	0	S1	0	0	S1	0
T2 @S2 a.amount = 15	S1	10	S2	2	**10**	**S1**	**1**
	S2	**15**	**S2**	**3**	10	S1	1
	S3	0	S1	0	0	S1	0
T2 replicated to S1 and S3	S1	15	S2	3	**10**	**S1**	**1**
	S2	**15**	**S2**	**3**	10	S1	1
	S3	15	S2	3	0	S1	0

Figure-2. Primary Site Migration and the Missing Update Anomaly.

Legend. psite: primary site, ts: update timestamp;
bold: primary site, underline: inconsistency.

This anomaly will be called the *missing update anomaly*. And even though Example 2 was constructed to show an instance of this anomaly caused by primary site migration, the anomaly can also occur when the primary sites of records are fixed.

The issue is that only the *update set* of a transaction is replicated to other sites, whereas the transaction's *consistency set* (the set of all records used to determine the consistency of its updates) may include other records that are read but not updated. And when the update set reaches a replicate site, the latest versions of the other records in the transaction's consistency set may not have reached that site because of transmission delays. Therefore, at the replicate site the consistency set of the update would include mismatched versions of different records.

This anomaly can be exacerbated by dynamic primary site migration. Later in this paper, we explore ways of reducing the impact of this anomaly.

2. Specifics of the Primary Site Migration Algorithm

The primary site migration algorithm itself must serialize concurrent requests to migrate the primary site of a record, and must be resilient to failures. We first present the basic migration procedure, and then show how it deals with concurrent updates and failures.

Each table that is subject to replication includes a unique primary key field, and additional fields required by the replication substrate. Here is an example of such a table:

```
create table the_table (
    id int,                  -- Primary key.
    -- Other application columns omitted.

    -- Replication substrate columns.
    primary_site int,        -- Primary site.
    ts int,                  -- Update timestamp.
    migration_ts int         -- Migration timestamp.
)
```

Listing 2.1. Required structure of a replicated table.

The timestamp is initialized to -1 for a unborn version of a record, and then incremented by 1 on each update of the record, including the update of the unborn version of the record to its initially-inserted version, and the update of the last version of the record to its deleted version. The purpose of the update timestamp is to allow the identification of the versions of a record at different sites. The timestamp of a record is incremented in the update trigger for the record's table and transparently to the updating application.

The migration timestamp of a record is incremented only each time a record is migrated from one site to another. The migration timestamp is a device used to make it easier to reason about our algorithms.

When a record's primary site is to migrate from one site to another, the migration algorithm first applies the following update procedure to the record's version at its current primary site. [Note. Code samples in this paper appear in *Transact-SQL*-style pseudo-code. These pseudo-code samples do not necessarily represent running code.]

```
create procedure update_primary
    @id int,            -- Primary key.
    @primary_site int,  -- Primary site of caller's version.
    @new_site int,      -- Primary site to migrate to.
    @ts int,            -- Record timestamp of caller's version.
    @migration_ts int   -- Migration timestamp of caller's version.
as

update  the_table
set
        primary_site = @new_site,       -- Change primary site.
        ts = ts + 1,                    -- Change record version.
        migration_ts = migration_ts + 1 -- Change migration version.
where   id = @id                        -- For the given record.
    and primary_site = @primary_site    -- If site is still primary.
    and ts = @ts                        -- If record has not changed.
    and migration_ts = @migration_ts    -- Redundant test.

if @@error != 0 return -102
else if @@rowcount = 0 return -101
else return 0
```

Listing 2.2. Primary site update procedure.

The migration algorithm then applies a similar update to the version of the record at the local site. The first application of the update - applied at the remote current primary site of the record - is known as the *from half*, or the *from update* of the migration algorithm. The second application of the update - applied at the local (new) primary site of the record - is known as the *to half*, or the *to update* of the migration algorithm. The combined procedure is given by the following code fragment:

2.1. Algorithm Site Migration

```
-- At the local site.
select  @local_site     = this_site

-- Get latest know primary site and update and migration timestamps.
select  @primary_site   = primary_site,
        @ts             = ts,
        @migration_ts   = migration_ts
from    the_table
where   id  = @id
```

```
if @primary_site = this_site
    -- succeed
else begin
    -- "From" half of migration. Remote update step.
    exec @ret = primary_site.the_db.dbo.update_primary
        @id,
        @primary_site,
        @local_site,
        @ts,
        @migration_ts
    if @ret != 0 begin
        rollback
        -- fail
    end
    -- "To" half of migration. Local update step.
    exec @ret = update_primary
        @id,
        @primary_site,
        @local_site,
        @ts,
        @migration_ts
    if @ret != 0 begin
        rollback
        -- fail
    end
    -- succeed
end
```

Listing 2.3. Primary site migration algorithm.

Note that the qualification of the remote update ensures that site migration occurs only if the versions of the record at the local site and the remote site have the same timestamps (indicating that they are identical copies). If the primary version of the record was updated or migrated before the *from* update step of the migration procedure is attempted, the intended application-level update of the record will not occur or will not commit. The current application-level transaction will then be aborted as a result of this failure, and the application must be prepared to retry a transaction subsequent to such a failure (in the same way that the application retries transactions after a deadlock failure).

Note also that the *from* update of the algorithm uses a separate transaction across a connection to the current primary site. If that update commits, then it is propagated to all other sites. If the *from* update succeeds, therefore, whether or not the later local update commits, the fact of the site migration is replicated to all sites. including the local site. Thus, from the point of view of the system as a whole, the migration has occurred when the *from* update commits.

The local update of the primary site is still necessary so that subsequent local updates to the record (executed before the remote update reaches the local site via replication) find that the local site is in fact the primary site of the record.

Note that if the local update of the record also commits, then this update too is propagated to all other sites. The resulting record includes the combined effects of the original application-level update, and the substrate primary-site update. And this version of the record will supercede the version updated at the remote site because it will possess a higher timestamp: its timestamp will have been incremented twice: once for the application-level update, and once for the primary-site update.

2.1.1. Initiating Site Migration in Triggers

This algorithm must be initiated and must succeed for each record updated by a transaction before that

transaction commits. To do so in an application-transparent manner requires an update trigger to loop through the updated records, executing the primary site migration procedure for each.

Figure 3 outlines the sequence of activities triggered by a local update in a replicated database using dynamic primary site migration.

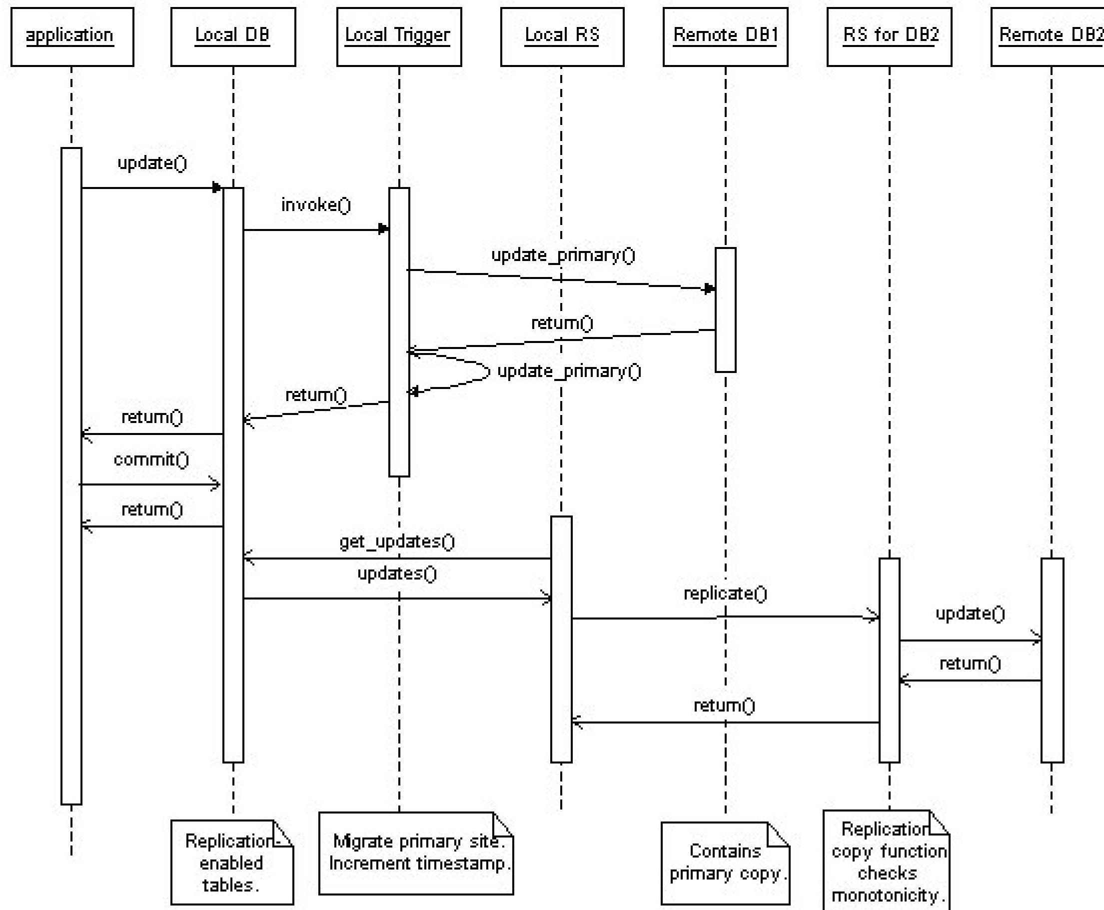

Figure-3. Sequence of replication actions in an SR system using primary site migration.

As shown in Figure 3, an update trigger at the local site migrates the primary site of an updated record to the local site. And after the transaction commits at the local site, the update is replicated by the DBMS replication service to all other sites (only one such replication shown in Figure 3). At a replicate site, an application-level replication *copy function* (see section 1.5.1) executes the update, ensuring the monotonicity of updates received from different sites.

In Sybase, triggers fire after an update is executed but before an update statement completes. Thus, in Sybase, a trigger implementation of the primary site migration procedure would migrate the primary sites of the affected records after an update statement is executed but before the update statement completes. Therefore, care must be taken to use the *before image* of a record's timestamp (the version in the special *deleted* trigger table) in the migration algorithm.

Otherwise, this reversal in the natural order of site migration and update at the local site is essentially immaterial, since they both occur within the same transaction, and are committed at the same time. The primary indication of site migration at the primary site will still be committed strictly prior to the local update transaction.

There are, however, a couple of nuances in the algorithm related to this ordering reversal. First, note that it is the responsibility of the update trigger to increment the timestamp of the record. But care must be taken to

perform this update *after* any required primary site migration in the trigger. That way, the migration algorithm's comparison of timestamps between the local and primary site versions of the record for the purpose of determining the record's currency remains valid (see Listing 2.2).

Second, because the update locks the record at the local site, after the first attempt at site migration fails as a result of the local site not being up-to-date with respect to the record, the current transaction must abort to release the update lock, thereby allowing the queued replications of more recent updates to the record to proceed.

Here is what a trigger might look like:

```
create trigger the_table_update
on the_table
for update
as

begin
    declare
        @id int,
        @ts int,
        @migration_ts int,
        @primary_site int,
        @this_site int,
        @site varchar(30),
        @proc varchar(100),
        @ret int

    select @this_site = 1   -- Change constant for different sites.

    declare updated_cursor cursor for
        select id, primary_site, ts, migration_ts
        from deleted

    open updated_cursor

    fetch updated_cursor
    into @id, @primary_site, @ts, @migration_ts
    while @@sqlstatus = 0
    begin
        print @id
        if not @primary_site = @this_site begin
            select @site = "site" + @primary_site
            select @proc = @site + ".the_db.dbo.update_primary"
            exec @ret = @proc
                @id,
                @primary_site,
                @this_site,
                @ts,
                @migration_ts
            if not @ret = 0 begin
                rollback transaction
                return
            end
            exec @ret = update_primary
                @id,
                @primary_site,
                @this_site,
                @ts,
```

```
                @migration_ts
            if not @ret = 0 begin
                rollback transaction
                return
            end
        end

        update test set ts = ts + 1 where id = @id

        fetch updated_cursor
        into @id, @primary_site, @ts, @migration_ts
    end

    close updated_cursor
    deallocate updated_cursor
end
```

Listing 2.4. Replicated table's trigger.

2.1.2. Concurrency Control and Crash Recovery

2.1.2.1. Crash Resistance

Consider the site migration algorithm of Listing 2.3. If the remote update in this algorithm fails, then the entire procedure is aborted, and it is as if no migration occurred. If the remote update in this algorithm succeeds, then as far as the system as a whole is concerned, the site migration will have occurred. If subsequent to the success of the remote update, the corresponding local update fails, the local transaction is aborted, but the remote update which occurred as a separate remote transaction will have committed. Because that remote update changes the primary site field of the record to the local site, that change is then replicated to all sites including the local site of the original application-level transaction.

Now a system crash during the primary site migration procedure can lead either to the remote update of the algorithm to fail, or to the remote update to succeed but the local update to fail. In the former case, no changes occur to the primary site assignment of the record. In the latter case, the primary site will have been changed, and this fact will be propagated to all sites.

Of course, it takes time for a new primary site assignment to reach a replicate site, and during this time, the version of the record at the replicate site would be pointing to a previous primary site for that record. If an update is attempted during this time at the replicate site, the *from* half of the migration procedure will find that the indicated site is no longer the primary site of the record, and will fail, causing retries until the new primary site assignment reaches the replicate site.

2.1.2.2. Concurrency Control

The migration algorithm leads to a strict serial schedule of migrations of a given record's primary site from one site to another. Intuitively, the following conditions ensure the serialization of the migration for each record:

- **Unique initial site**. Each record has a unique primary site when it is created.
- **First migration wins at old primary site**. Only one of several concurrent site migration requests can succeed at a given primary site (since the migration timestamp is incremented at each such update).
- **Next migration is delayed at new primary site**. The next site migration request cannot succeed until the current local transaction completes (which must occur later in real time than the *from* half of the current migration algorithm, i.e., later in real time than the last site migration).

Informally, then, it follows that by using the primary site migration algorithm, each record will have a unique primary site at any given time.

These intuitive ideas may be formalized as follows:

Assertion 1.

Consider all instances of all versions of a record present at any site up to a given point in time in an SR system using the algorithm *Site Migration*, and suppose that the maximum migration timestamp of any such instance of the record up to this time is `T`.

Given a migration timestamp `t` (`<= T`), there is a unique primary site `S[t]` that is identically present in all existing and previous instances of the record (at any site) with migration timestamp `t`.

Proof of Assertion 1.

By induction on the largest migration timestamp of the record.

Basis. For an initially created (unborn version of a) record, the migration timestamp is assigned to be -1, and because the unborn version is only created at its unborn primary site, the primary site of the record having a migration timestamp of -1 is unique.

Induction step. Suppose that for each t ((<= T) there is a unique primary site for all instances of the record having a migration timestamp t, and consider an instance of the record migration timestamp `T`. This instance must have been created initially as a result of a migration from a primary site of the record with migration timestamp `T - 1`. This follows from the basic update statement of the migration algorithm, which is the only way the migration timestamp may be incremented.

But by the induction hypothesis, there is a unique primary site for this record with migration timestamp `T - 1`. Let this site be `S[T - 1]`. Then the *from* half of any primary site migration with a *to* migration timestamp of `T` must have occurred at site `S[T - 1]`. The corresponding update in the *from* half of such a migration would be:

```
update  the_table
set     primary_site     = @new_primary_site,
        ts               = ts + 1
        migration_ts     = T
where   id               = @id
    and primary_site     = @primary_site
    and ts               = @ts
    and migration_ts     = T - 1
```

Listing 3.1. *From* update of primary site migration resulting in migration timestamp of `T`.

If there are multiple concurrent such updates, they would be serialized by the DBMS at site `S[T - 1]`, and the first such update produces a migration timestamp of `T`. All other such updates would fail by finding that the migration timestamp has been incremented to `T` or higher. The *to* site for the first execution is therefore the only primary site for timestamp `T`, as required.

Corollary 1.1.

Algorithm Site Migration defines a linear sequence of primary sites for each record over time. This sequence of sites will be called the record's *migration path*.

Proof of Corollary 1.1.

The migration path of a record is the sequence of unique sites associated with each value of the migration timestamp, as given in Assertion 1. Clearly, successive values of the migration timestamp lead to temporally

successive primary sites for the record. This is because migration *from* a site must strictly succeed migration *to* a site in real time, since the *to* transaction of a migration is serialized by the DBMS at the given site with the *from* transaction of the next migration (the one with the next higher timestamp).

2.1.3. Insertions

Recall that the insertion of a record with a natural primary key requires a two-step procedure similar to an update of a record. First, an unborn version of the record is created at its primary site and marked as migrated to the local site at which the insertion is to take place. Then the actual insertion at the local site can take place (as if the unborn version were being updated, with the fields being inserted).

This procedure is split between a record's unborn primary site, and the local site of the insertion, and may be cut short after the unborn creation at the remote site (either because of a crash, or because the local transaction that includes the insert is aborted for some other reason).

In that case, the unborn record remains at the unborn primary site, and is eventually replicated to all other sites, including the local site at which the insertion failed. At that time, both the unborn primary site and the local site are consistent with respect to the status of the record. They both indicate the local site to be the primary site of the existing unborn version of the record. A subsequent insertion of the record is then reduced to an update of the unborn version.

Thus, when the insertion algorithm finds that an unborn version of the record exists at its unborn primary site, the algorithm aborts, causing retries until the unborn version is replicated to the local site. Then an insertion is reduced to the update of the unborn version to the inserted version.

Note that an unborn version of a record is most easily represented by using a special update and/or migration timestamp, e.g., -1. Similarly for a deleted version of a record (for example, a timestamp of -2 might be used for deleted records). The application code, on the other hand should be unaware of the existence of such special versions. By providing a view that filters out negative timestamps for each table, one can make the application code transparent to these special versions of records.

3. Cluster-Based Primary Site Migration

3.1. Islands of Consistency

By an *island of consistency* we mean a totally independent portion of a database. Each transaction, and each consistency constraint would span a unique island of consistency. We may call a primary site partitioning *disjoint* if each island of consistency belongs to a unique primary site. In a static and disjoint primary site regime, missing updates will not cause inconsistencies, since they would be unrelated to updates received from other sites.

When the replication partitioning is based on islands of consistency, the only consistency issue associated with replication is the (accepted) lack of currency of data at replicate sites. When an organization buys into the replication methodology, it has accepted this lack of currency as a price to pay for performance, simplified applications, and so on.

A degenerate partition that assigns a single primary site to all records, of course, creates a single monolithic island of consistency.

In general, it may not be possible to design a proper partitioning of a database into disjoint islands that match consistency sets and access patterns to the database. In that case, the designer may decide to compromise certain consistency guarantees for performance.

3.2. Clusters

In [1] it was suggested that when an application object is represented as multiple records, or when clusters of related records are generally manipulated together, the cluster as a whole could be considered to belong to a site where all of its constituent records are allowed be updated by user applications. Then the required migration algorithm would be applied just once to the cluster as a whole, as opposed to repeatedly to each individual record of the cluster that is to be updated.

A cluster of records acts as a small quasi-island of consistency. In some applications, it may be possible to partition a database into relatively small subsets of tightly-coupled records, for example, the records representing each complex high-level application object, where most transactions and most consistency constraints span unique clusters. Under a cluster-based replication regime, intra-cluster transactions would be guaranteed by the replication substrate to preserve intra-cluster consistency constraints. But inter-cluster transactions, and inter-cluster consistency constraints would require special intervention by the application program. In our original motivating application for which a cluster-based design was used (a software lifecycle database) inter-cluster transactions and consistency constraints were very rare. To the extent that other applications have similar characteristics, they may benefit from a cluster-based primary site regime.

In a cluster-based primary site regime, the primary site, and the migration timestamp are attributes associated only with the cluster as a whole. But both clusters and individual records would have update timestamps. Each time a record is updated, both its update timestamp and the update timestamp of the cluster to which it belongs are incremented. Clusters are represented in an auxiliary *cluster table*, which has the following general structure:

```
create table cluster (
    cluster_type    varchar(12),
    cluster_id      integer,
    primary_site    integer,
    ts              integer,
    migration_ts    integer
)
```

Listing 3.2. Cluster table.

Thus, for example, if a cluster represents an entire master-detail hierarchy rooted at a master *customer* record with an integral surrogate key, the surrogate key of each customer would act as the cluster identifier for all customer clusters. The key of a cluster table is `(cluster_type, cluster_id)`. However, for brevity in the remainder of this paper, we shall ignore *cluster_type*.

The cluster table is also replicated to all sites.

The monotonicity of updates to individual records from different primary sites reaching a replicate site would be checked as before by comparing an arriving record's timestamp with the timestamp of the version of the record at the receiving site.

3.3. The Issue of Cluster-Wide Currency

Consider now the procedure for changing the primary site of a cluster from one site to another.

Before a cluster's primary site can be changed to the local site of an executing transaction, the entire cluster must become current at the local site.

It was stated erroneously in [1] that the currency of an entire cluster of records at a site may be checked by comparing the cluster's timestamp at the local site with the cluster's timestamp at its current primary site. The missing update anomaly shows up the error in this assertion, as illustrated in the following example.

Example 3. Cluster Currency

Continuing with Example 2, suppose that a cluster `C` contains the records `a` and `b`, and that transaction `T1` at site `S1` updates the cluster timestamp from 0 to 1 for its update of `a`, and to 2 for its update of `b`, then the primary site of the cluster migrates to site `S2` (changing the cluster timestamp to 3), where transaction `T2` updates the cluster timestamp to 4 for its update of `a`. When the results of transaction `T2` are replicated and reach `S3` before the results of transaction `T1`, the current cluster timestamp of 4 reaches `S3` together with version `a[2]` created by transaction `T2`.

At this time, even though site `S3` has the latest version of the cluster record (timestamp 4), it does not have the latest version of the cluster as a whole, it has `{a[2], b[0]}`, whereas the latest version of the cluster is `{a[2], b[1]}`. All parts of the cluster that were updated by `T1` and not by `T2` would be missing from `S3` if `T2` is replicated to `S3` before `T1`.

Because `S3` is not current with respect to the cluster, it cannot act as the primary site for it. So if some transaction T3 initiated at site `S3` requires to update, say both `a` and `b`, it should not be allowed to migrate the primary site of the cluster to `S3`, which is a pre-requisite to updatability at site `S3`. If the transaction is allowed to proceed, it acts on an inconsistent version of the cluster, and the basic design goals of conflict avoidance and consistency preservation are violated: its update of `b` will conflict with `T1`'s initial update of `b`, and it updates a possibly inconsistent state of the cluster.

With only two sites, this scenario cannot occur.

3.4. Levels of Consistency in Cluster-Based Replication

In order to describe the trade-offs of consistency guarantees for performance in cluster-based replication, consistency guarantees may be differentiated by using the following categories:

1. **Currency**. It is generally assumed that data at the primary site must be current (reflect the latest committed transactions), while data at replicate sites may lag the latest truth. An application which may lead to out-of-date data at the primary site is considered incorrect, and so the replication substrate must guarantee currency at the primary site.

2. **Valid Read/Write Sets**. An application may rely on the replication substrate to guarantee that the read/write set of each transaction is current, or just that the write set of a transaction is current. Because reads in a replicated database are allowed to be out-of-date, certain transactions may be able to tolerate out-of-date reads as long as their write sets are current and consistency constraints applicable to their write sets are satisfied.

3. **Intra-cluster consistency/inter-cluster consistency**. An application may rely on the replication substrate to guarantee general consistency, or only intra-cluster consistency. Or an application may not rely on the replication substrate for consistency.

4. **Primary site consistency/replicate site consistency**. An application may rely on the replication substrate to guarantee a certain type of consistency at the primary site only, or at both the primary and the replicate sites. At replicate sites, a relaxation of consistency in terms of a lack of currency is an accepted norm, and it may well be that additional relaxations in consistency within reason may be tolerable. For example, if a replicate site is used for statistical report generation, a consistency level equivalent to the read-committed isolation level might be sufficient for many applications.

4. Cluster-Based Primary Site Migration Algorithms

Several approaches are outlined below for cluster-based replication, with different degrees of compromise between the requirements of performance/site affinity, consistency guarantees, and availability.

4.1. Disjoint Islands

This degenerate case does not require primary site migration. Each disjoint island of consistency belongs to a unique site. The application program would be required to furnish the ID of the island to be worked on to a special database connection factory, and to obtain a database connection to the primary site of that island. The replication substrate would determine the primary site of the island, and would return a database connection to that site. The entire transaction then would use that connection.

For those clients whose database connections run over a WAN, performance suffers. Consistency is guaranteed by the DBMS transaction system at the primary site, and by the atomic sequence-preserving replication of transactions to replicate sites. Here, performance and availability are sacrificed for consistency.

4.2. Primary Site Migration for Updated Records

At another extreme, read currency may be sacrificed for performance and availability.

In this case, each record involved in an update has its primary site migrate to the local site of the transaction at hand (if the site is not primary for the record, already). This scheme guarantees write-set validity. But it does not guarantee general consistency at the primary site, since a transaction's reads may be out-of-date.

To provide general consistency guarantees at the primary site (only), a transaction's reads must be guaranteed as current. To migrate the primary sites of a transaction's entire read set for the purpose of read-set verification would be costly, and contrary to a major design compromise in using replication: trading of read currency for improved performance. But a transaction's reads could be verified by using optimistic concurrency control techniques (see [2] for a comprehensive treatise on optimistic concurrency control techniques).

However, it is difficult to make this approach transparent to the application. The replication substrate would require cooperation from the application to inform it of the identity of the records read (*select triggers* would allow transparency here, but such triggers are absent from the major commercial database systems). Also, an application's commit request must be intercepted to launch the verification of reads.

4.3. Using Timestamp Collections of Cluster Records

In this approach, the primary site is associated with a cluster as a whole, but each record retains an individual timestamp. The migration algorithm for a cluster verifies the currency of each record in the cluster by comparing that record's timestamp with the corresponding timestamp at the current primary site.

The algorithm requires the transmission of a timestamp collection (indexed by, say, record identifiers) for an entire cluster to the primary site, for the purpose of verification of currency. Therefore, this approach is suitable in an application where clusters are known to be small. In this case, the record timestamp collection for each cluster may be kept track of in a single variable-length field associated with the cluster, so that it is readily and efficiently available for transmittal to verify the currency of the collection.

However, this approach does not scale well with the size of clusters.

4.4. Acknowledged Replication

This approach preserves intra-cluster consistency, and sacrifices availability and inter-cluster consistency for performance.

To preserve the consistency of an entire cluster at each site, we can insure that all updates of a cluster have reached all sites before allowing the primary site of the cluster to be changed to a different site. One way to do this is to have each site acknowledge the receipt of a replicated transaction to the primary site at which the transaction is executed. The primary site can then keep track of which sites are current with respect to it for each cluster, and when a request is received to migrate the primary site of a cluster, only allow the request if all sites are current for the cluster.

Here is an outline of an implementation of acknowledged replication.

4.4.1. Algorithm *Acknowledged Replication*

4.4.1.1. Obsolescence Relationships

Acknowledged replication relies on keeping track of the obsolescence relationships between sites, i.e., that an update at one site obsoletes versions of records and clusters at other sites.

An *obsolescence record* is a tuple `(cluster_id, primary_site, stale_replicate_site, ts)` which represents the fact that a cluster identified by *cluster_id* was changed at site *primary_site* and obtained timestamp *ts*, but no acknowledgement of this change was received from site *stale_replicate_site_id*. In other words, for all we know, the replicate site may not have received the corresponding update, and is therefore obsolete with respect to this cluster.

4.4.1.2. The Algorithm

The algorithm requires each site to keep two tables of obsolescence records, called, *transmitted*, and *received*. The *transmitted* table of a site stores, for each primary update to each cluster at that site, and for each replicate site, the corresponding obsolescence record, for as long as that update remains unacknowledged from the given replicate site. The *received* table of a site stores, for each update replicated to that site, the corresponding obsolescence record that is to be acknowledged to the primary site at which the update originated.

At the primary site, triggers update the *transmitted* table whenever a cluster timestamp is updated. At each primary site of a cluster, it is sufficient to keep track only of the highest unacknowledged timestamp of the cluster from each replicate site.

At a replicate site, special replication functions (or special replicate site triggers) update the *received* table to indicate the highest timestamp received for a cluster from a primary site. The *received* table entries for each primary site are then replicated to that site, and cause the *transmitted* table at the corresponding site to be updated. The *received* table may be replicated either by using the DBMS replication service, or (for better performance) periodically by an application-level procedure.

Before the primary site of a cluster can be changed, the algorithm checks that the current version of the cluster at its primary site has been acknowledged by all sites, by inspecting the *transmitted* table at the current primary site of the cluster.

In case the cluster's replication has not been acknowledged by all sites, the migration algorithm stalls, and eventually times out, causing the transaction to be aborted, and availability to suffer. Therefore, this algorithm is not suitable where the number of sites is large, or networks are relatively unreliable.

4.4.2. Proof of Correctness of Algorithm *Acknowledged Replication*

The proof of Assertion 1 and Corollary 1.1 for record-level migration may be retraced for cluster-level migration to show that a unique migration path exists for each cluster up to each point in time. The migration path of a cluster represents the sequential history of the sites that were primary for the cluster.

Algorithm *Acknowledged Replication* must then be shown to ensure the existence of the current versions of records in an entire cluster at each primary site of the cluster.

Assertion 2.

At the moment of a primary site migration for a cluster `C` by using the algorithm *Acknowledged Replication*, no site is missing an update for cluster `C`.

Proof of Assertion 2.

The proof of the non-existence of missing updates for acknowledged replication proceeds by induction on the length of the migration path of a cluster.

Basis.

A new unborn version of a cluster includes no updates and so cannot be missing any updates. After the unborn version of a cluster is created, the actual insertion of the record can be treated as an update in the induction step.

Induction step.

Suppose that in a migration path for cluster `C` of length `n`, `(S1, S2, ..., Sn)`, no site is missing any updates to `C` at each moment of migration. We need to prove that at the moment of the next migration from `Sn` to `S(n+1)` no site is missing any of `C`'s updates.

By induction we know that the updates from `S1` to `S(n-1)` must have reached all sites at the moment of migration from `S(n-1)` to `Sn`. So we need only prove that all updates at site Sn have reached all other sites at the moment of migration from Sn to `S(n+1)`.

This migration is only allowed if the replication of the last transaction at Sn affecting `C` (the one with the highest timestamp) has been acknowledged by all sites. But because the underlying DBMS replication service propagates transactions in the order of their occurrence, all previous transactions at Sn affecting `C` must also have been replicated to all other sites. Hence no site can be missing an update for `C` from any site in `C`'s migration path, as required.

Of course, while the new site remains the primary site of the cluster, all updates to the cluster occur there, and the site will contain the most recent versions of the cluster's records.

Hence, the algorithm yields, at any moment of time, a unique site that is primary for `C`, where all updates to `C` occur, and where all of `C`'s records are current.

Corollary 2.1.

Algorithm *Acknowledged Replication* produces strictly monotonic updates to each cluster at every site (independently of the site's primary/replicate status at different times).

Proof of Corollary 2.1.

The induction proof of Assertion 2 may be extended to prove the monotonicity of updates to each cluster at each site.

4.5. Obsolescence Multistamps

It is possible to elaborate on acknowledged replication to enhance availability at the expense of intra-cluster consistency at replicate sites. The approach allows a primary site migration to proceed between two sites that have a working communication link, as long as there are no missing updates for the cluster at the new primary site. Other sites may not be current with respect to the cluster at the moment of migration. So these other replicate sites can only be guaranteed a read-committed and record-monotonic level of consistency.

4.5.1. Cluster Multistamps

In [2] Adya presents a scheme for ensuring the consistency of data caches in client programs by using a device known as a *multistamp*. A similar device may be used to verify cluster consistency when changing the primary site of a cluster.

Definition. Acknowledgement Resolution.

An acknowledgement of an obsolescence record (cluster_id, primary_site, stale_replicate_site_id, acknowledged_ts) cancels an unacknowledged obsolescence record (cluster_id, primary_site, stale_replicate_site_id, unacknowledged_ts) if and only if acknowledged_ts >= unacknowledged_ts. The act of attempting such a cancellation will be called a *resolution*. An obsolescence record is *resolved*, if it can be ascertained that the replicated update specified by that obsolescence record has reached its target replicate site.

Definition. Unresolved Multistamp.

For each pair of primary and replicate sites, it is sufficient for our algorithm to keep track of a cluster's highest as-yet-unresolved timestamp for replications between these two sites. The set of all such as-yet-unresolved obsolescence records for a given cluster is called the *unresolved multistamp of the cluster*. So the unresolved multistamp may also be considered as a matrix of timestamps, where the row index designates a primary site, and the column index designates a replicate site, and where a matrix cell value at position [PS, RS] is the largest timestamp of a committed transaction at PS which is not yet known to have been replicated to site RS.

The multistamp is associated with a cluster at its current primary site, and reflects this primary site's knowledge of the current disposition of replications for the cluster from any primary site to any replicate site. We will refer to the unresolved multistamp matrix simply as the *unresolved array* of a cluster, and to a particular cell in this array as unresolved[PS, RS], where PS is a primary site and RS is a replicate site. A column indexed by site RS in this matrix represents, for each primary site PS, the timestamp that RS needs to receive from PS to validate that RS is current with respect to PS.

When a resolved obsolescence record is removed from the multistamp associated with a cluster, the corresponding unresolved array position becomes empty.

4.5.1.1. Algorithm *Cluster Multistamp*

The multistamp is maintained by update triggers at the primary site of each cluster. And as the cluster's primary site migrates from one site to another, the cluster's multistamp tags along.

When migrating to a new site, the cluster's received obsolescence records for the new primary site are resolved against the cluster's multistamp. Consider the migration of the primary site PS of a cluster to a new site RS in a multistamp regime. Here site RS sends to site PS the highest timestamps it has received for the cluster from each other site OPS. This is an array of timestamps indexed by primary site OPS, which we may call the *received array* of *RS* for the cluster. PS then validates its *unresolved* column for RS against the received array. If there is an unresolved timestamp for [OPS, RS], then the received timestamp for site OPS must match this unresolved timestamp. If all unresolved timestamps are thus resolved, RS is current with respect to all previous primary site updates, and the migration algorithm can proceed.

The DBMS replication service insures that transactions are replicated in the order of their commitment from one site to another. Therefore, when an unresolved timestamp *ts* for replication from one site to another is resolved or acknowledged, it may be assumed that all updates to the cluster committed at the given primary site up to this timestamp have been received by the given replicate site. In other words, to determine obsolescence at a replicate site, it is sufficient to keep track of the timestamp of the latest update to the cluster at each primary site.

Here are some specifics of the algorithm.

- When a cluster timestamp is updated at primary site `S1`, a trigger adds corresponding obsolescence records for all replicate sites to the multistamp of the cluster at that site. Further updates of the same cluster advance the timestamps of the corresponding obsolescence records.
- When acknowledgement of the receipt of a replication to a site `S2` reaches the primary site `S1`, the corresponding obsolescence record can be removed if its timestamp has not advanced.
- The migration of the primary site of a cluster from site `S1` to site `S2` can succeed only if all obsolescence records in the cluster's multistamp for replications to site `S2` can be resolved. The data required for the resolution of these obsolescence records must remain available in `S2`'s *received* table. Thus entries in the *received* table are not removed, but their timestamps may be advanced as new updates of a cluster are recieved from the same primary site.

  The number of entries in the *received* table of each site for a given cluster is bounded by the total number of sites, and can be expected to be quite small.

A formal proof that the algorithm avoids missing updates at the primary sites of clusters can be provided along the same lines as that of the algorithm *Acknowledged Replication*.

Note that the algorithm requires the received array to be transmitted to the current primary site, and the unresolved matrix to be transmitted back to the new primary site. In the worst case, the received array has size proportional to the number of sites, and the unresolved matrix has size proportional to the square of the number of sites. For a working system with rare disconnects, the average sizes of the transmitted received arrays and unresolved matrices can be expected to be considerably smaller than their worst case upper bounds. But even in the worst case, typical installations rarely include more than a dozen or so sites, and the sizes of the messages required in this migration algorithm will be relatively inconsequential.

Example 4. Primary Site Migration with Unacknowledged Replications

Here is how our earlier examples of updates at two different sites would unfold when using cluster multistamps. Suppose site `S1` is the current primary site of a cluster `C`, and that it is the only primary site `C` has ever had.

1. A transaction `T1` updates a member of `C` causing the cluster timestamp to change to 1.
2. `S1` receives a replication acknowledgement for `T1`'s updates from `S2` but not from `S3` (for example, because the network between `S1` and `S3` is down). Thus, cluster `C` has a single outstanding obsolescence record in its multistamp, namely, `(C, S1, S3, 1)`.
3. At this point, a transaction `T2` at site `S2` needs to update some member of the cluster and requests a migration for the cluster from site `S1` to site `S2`. The migration is allowed to proceed because the multistamp for `C` contains no records for replicate site `S2` indicating that all replications of `C`'s updates to `S2` have been acknowledged, and therefore `S2` is current with respect to `C`.
4. Now `T2` can updates some record of the cluster at `S2`. As shown in earlier examples, missing updates may occur at replicate sites in such a scenario. When `S2` updates the cluster, its updates may reach `S3` before `S1`'s updates, causing the version of the cluster at `S3` to be inconsistent.

   But a replicate site inconsistency of this type may well be tolerable as the price to pay for mitigating update stalls caused by network failures unrelated to the two immediate sites involved in updating a cluster.

   `S2`'s update adds two new obsolescence records to the cluster's multistamp resulting in: `{(C, S1, S3, 1), (C, S2, S1, 3), (C, S2, S3, 3)}`, where the timestamp of the update at `S2` is assumed to be 2.

Figure 4 shows the disposition of the cluster C at each site as the update and replication events of the example unfold. Presented in this Figure are the multistamps of the cluster at its primary site, and the contents of the *received* table at each site. The ack flag in a received record indicates that the receipt of the corresponding replication has been acknowledged.

Event	Multistamp	Site	Timestamp	Received
initial state	{}	S1	0	{}
		S2	0	{(C, S1, S2, 0, ack)}
		S3	0	{(C, S1, S3, 0, ack)}
T1 @S1	{(C, S1, S2, 1), (C, S1, S3, 1)}	S1	1	{}
		S2	0	{(C, S1, S2, 0, ack)}
		S3	0	{(C, S1, S3, 0, ack)}
T1 replicated to S2	{(C, S1, S2, 1), (C, S1, S3, 1)}	S1	1	{}
		S2	1	{(C, S1, S2, 0, ack), (C, S1, S2, 1)}
		S3	0	{(C, S1, S3, 0, ack)}
T1 replication acknowledgement received from S2	{(C, S1, S3, 1)}	S1	1	{}
		S2	1	{(C, S1, S2, 1, ack)}
		S3	0	{(C, S1, S3, 0, ack)}
T2 @S2	{(C, S1, S3, 1), (C, S2, S1, 3), (C, S2, S3, 3)}	S1	2	{}
		S2	3	{(C, S1, S2, 1, ack)}
		S3	0	{(C, S1, S3, 0, ack)}
T2 replicated to S1 and S3	{(C, S1, S3, 1), (C, S2, S1, 3), (C, S2, S3, 3)}	S1	3	{(C, S2, S1, 3)}
		S2	3	{(C, S1, S2, 1, ack)}
		S3	3	{(C, S1, S3, 0, ack), (C, S2, S3, 3)}
T2 replication acknowledgement received from S1 and S3	{(C, S1, S3, 1)}	S1	3	{(C, S2, S1, 3, ack)}
		S2	3	{(C, S1, S2, 1, ack)}
		S3	3	{(C, S1, S3, 0, ack), (C, S2, S3, 3, ack)}

Figure-4. Tracking the disposition of replications in multistamp-based primary site migration.

An obsolescence record (C, S1, S2, TS) in Figure 4 indicates that a primary transaction at site S1 affected cluster C and changed its timestamp to TS and was replicated, or was schedlued for replication, to site S2.

Example 5. Effect of Unresolved Obsolescence Records

Suppose that after T2 is replicated to S3 but before an acknowledgement of its receipt is received by site S2, S3 requires to update some member of the cluster. Suppose further that the network between S1 and S3 remains out of service.

In this case, the migration algorithm attempts to resolve the multistamp of `C` against `S3`. At this time, the multistamp of `C` includes two entries for replications to `S3`, namely, `(C, S1, S3, 1)`, and `(C, S2, S3, 3)`. Assuming that the update at `S2` is successfully replicated to `S3`, the obsolescence record `(C, S2, S3, 3)` can be resolved by checking the *received* table at `S3`. But the obsolescence record for the update at `S1` cannot be resolved, since `S3` has not received the replicated update made at `S1` at time 1. In this case, the migration fails, and the update times out and may be retried.

5. Conclusions

The algorithms presented provide a range of options for trading consistency guarantees for availability for performance. These algorithms are best suited to applications with the following characteristics:

- Clustering. The database may be partitioned into a set of relatively small clusters, where most transactions and most consistency constraints span a single cluster. Also, the cluster associated with each record must be easily identifiable, e.g., the cluster key may be the value of a well-known column (or set of columns of a record).
- Site affinity. Access patterns to clusters exhibit site affinity, that is, periods of short-term bursts of activity on the cluster at one site to the exclusion of other sites, where the activity site may change over time.
- Tolerance for lower levels of consistency. Especially at replicate sites.

Acknowledgements

This work is a continuation of work performed collaboratively with Moshe Bitton and Ron Chen. My thanks to Moshe and Ron for their continued interest and discussions. I would also like to thank Eric Lundblad for discussions on alternative primary site migration algorithms.

References

1. Bolour, A., Bitton, M., and Chen R., Multi-site concurrency control with the Sybase replication server, SQL Forum, Vol. 3, No. 4 (July/August 1994), pp. 17 - 22.

2. Adya, A., Weak Consistency, a Generalized Theory of Optimistic Implementations for Distributed Transactions, Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science, 1999.

3. Alsberg, P.A., and J. D. Day. A principle of resilient sharing of distributed resources. Proceedings of the Second International Conference on Software Engineering, October 1976, pp. 562-570.

4. Stonebraker, Michael. Concurrency control and consistency of multiple copies of data in distributed INGRES. IEEE Transactions of Software Engineering, Vol. SE-5, No. 3, May 1979; pp. 188-194.

5. Minoura, T., and G. Wiederhold. Resilient extended true-copy token scheme for a distributed database system. IEEE Transactions on Software Engineering, Vol. SE-8, No 3, May 1982, pp. 173-189.