Online Reconfiguration in Replicated Databases Based on Group Communication

Bettina Kemme
School of Computer Science
McGill University, Montreal
kemme@cs.mcgill.ca

Alberto Bartoli
Facoltà di Ingegneria
Università di Trieste
bartolia@univ.trieste.it

Özalp Babaoğlu
Department of Computer Science
University of Bologna
babaoglu@cs.unibo.it

Abstract

Over the last years, many replica control protocols have been developed that take advantage of the ordering and reliability semantics of group communication primitives to simplify database system design and to improve performance. Although current solutions are able to mask site failures effectively, many of them are unable to cope with recovery of failed sites, merging of partitions, or joining of new sites. This paper addresses this important issue. It proposes efficient solutions for online system reconfiguration providing new sites with a current state of the database without interrupting transaction processing in the rest of the system. Furthermore, the paper analyzes the impact of cascading reconfigurations, and argues that they can be handled in an elegant way by extended forms of group communication.

1. Introduction and Motivation

Replicating data across several sites is a well-known technique for increasing availability and performance in distributed databases but introduces the problem of keeping all copies consistent. Replica control mechanisms can be classified as being either *eager*, i.e., updates are coordinated before transactions commit [9], or *lazy*, i.e., updates are propagated only after transactions commit (e.g., [20]). Although eager replication can easily guarantee 1-copy-serializability and can be made fault-tolerant in a straightforward way, we believe it is fair to say that eager solutions have had very limited practical impact. Database designers believe that eager replication is too complex, has poor performance and does not scale. Instead, commercial database systems are based primarily on lazy strategies favoring performance over correctness: most of their solutions guarantee neither data consistency nor fault-tolerance [11].

Motivated by this gap between theory and practice, recent proposals for replicated databases [1, 2, 21, 18, 12, 15, 14, 17] propose new approaches that exploit the rich semantics of group communication systems [10, 19] to implement an eager-style replica control. Most of these solutions propagate the updates of the transactions using a total order multicast that delivers all messages at all sites in the same order. The database uses this order as a pattern to follow in the case of conflicts, i.e., conflicting updates are serialized in the order that the group communication system delivered them. Several simulation studies [18, 15, 12] and a real implementation [14] have proven the superior performance of such an approach compared to traditional eager replica control mechanisms. The proposed solutions are able to handle effectively site and communication failures [2, 15]. This is accomplished primarily through the virtual synchrony properties of the underlying group communication system, which notifies about failures in such a way that surviving sites receive exactly the same set of messages before being informed about the failure.

What is often missing from the various proposals is how failed sites can rejoin the system after recovery, how partitions can merge after repairs or how new sites can be added to a running system. Reconfiguration that is necessary when the number of sites increases is a far more complex task than that necessary when the number of sites decreases. In particular, before a joining site can execute transactions, an up-to-date site has to provide the current state of the data to the joining site. One possible solution is to require suspending transaction processing during this data transfer, an approach taken, e.g., by [2]. This option, however, may violate the availability requirements of many critical systems if the amount of data to be copied is extremely large. Instead, all reconfigurations should be handled *online* whereby transaction processing continues and is interfered as little as possible by reconfiguration. We are not aware of any existing reconfiguration mechanism that fulfills this requirement.

This paper proposes efficient and elegant solutions to online reconfiguration in replicated databases. We discuss various alternatives for data transfer to joining sites, all of them allowing concurrent transaction processing. Given that it is impractical to perform data transfer as a single "atomic step", we pursue solutions that admit cascaded reconfigurations during the data transfer itself. An important contribution of our approach is a clear separation between the tasks of the group communication system and the tasks of the database system: while the former is responsible for communication and group management, the latter is charged with handling data transfer and coordination of transactions. Such separation is important in practice because it simplifies the integration of existing database technology.

We present our approach in two steps. First, we propose reconfiguration algorithms based on the basic virtual synchrony paradigm offered by most group communication systems. Within this context, various data transfer strategies are discussed, ranging from transferring the entire database to more sophisticated schemes admitting piecewise reconfiguration. Relying on basic virtual synchrony, however, results in complex reconfiguration protocols if further failures may occur during the data transfer. For this reason, we show in a second step how to modify the previous algorithms using an enriched virtual synchrony model, called EVS [4]. The resulting framework enables simpler solutions that admit various failure scenarios.

We note that this paper pursues a database perspective to reconfiguration focusing on database issues. As a result, our treatment of group communication in general, and virtual synchrony in particular, is necessarily abbreviated.

The paper is structured as follows. The next section provides a brief overview of virtual synchrony and replica control based on group communication. Section 3 outlines the principal problems that need to be solved for online reconfiguration. Section 4 presents various alternatives for online reconfiguration based on the virtual synchrony model. Section 5 refines the previous solutions by using enriched form of virtual synchrony to appropriately encapsulate reconfiguration. Section 6 concludes the paper.

2. Basic Concepts

2.1. Virtual Synchrony

We assume an asynchronous system where neither message delays nor computing speeds can be bounded with certainty. Messages may be lost and sites may fail by crashing (we exclude Byzantine failures). Crashed sites may recover. Sites are equipped with a group communication system supporting *virtual synchrony* [10, 19]. Virtual synchrony provides applications with the notion of *group membership* and with a *reliable multicast* communication primitive (a message is sent to all members of the group). Virtual synchrony provides consistent information about the set of group members that appear to be currently reachable. This information takes the form of *views*. The system determines a new view as a result of crashes, recoveries, network partitions and merges, or explicit group joins and leaves.

New views are communicated to sites through *view change* events. A site that delivers a view change event $vchg(V)$ is informed that the new view is V. In this case we say that the site *installed* V. We say that an event (e.g., the delivery of a message) occurs in view V at a given site if and only if the last view to be installed at the site before the event was V. Given two views V and W, we say that V and W are *consecutive* if and only if there is a site for which W is the next view to be installed after V. V and W are *concurrent* if and only if there is no site that installed both V and W. Intuitively, concurrent views reflect different perceptions of the group membership, typically as a result of partitions.

A fundamental property of virtual synchrony is that view changes are globally ordered with respect to message deliveries: given two consecutive views V and W, any two sites that install both views must have delivered the same set of multicast messages in view V.

We say that any view with a majority of sites is a *primary view* (the number of sites is assumed to be static and known to all sites). As clarified in the next sections, our algorithms allow transaction processing only at sites in the primary view. Extending our discussion to dynamic groups or other definitions of primary view (e.g., a view containing a majority of the *previous* primary view) is straightforward. We assume that the composition of concurrent views do not overlap. If this is not provided by the group communication system [5, 6], we assume a thin software layer on top of the virtual synchrony layer, that hides from the application primary views that are not installed by a majority of sites (as, e.g., in [22]).

The replica control protocols of the next section use a total order multicast: any two sites that deliver two multicast messages deliver them in the same order [10, 19]. Furthermore, we shall assume a *uniform reliable multicast* with the following guarantee. Let V and W be two consecutive primary views and let S_1 and S_2 be sites that installed V. If S_1 is a member of W but S_2 is not (S_2 crashes or installs some non-primary view W' as the next view after V), and S_2 delivers message m in V, then S_1 also delivers m before installing W. In other words, messages delivered by S_2 in V constitute a subset of those delivered by S_1 in V.

Note that we do not specify which messages have to be delivered in minority views. As we discuss in Section 2.3, members of minority views behave as if they had failed by ignoring delivered messages and refraining from executing transactions. The above adaptation of "uniformity" to partitionable systems happens to suffice for the replica control protocols of the next section and can easily be implemented with minimal changes to existing group communication systems (e.g. [16]).

2.2. Replica Control

The replicated database system consists of a set of sites. Each site runs an instance of the database management system and maintains a copy of the database. Each site is a group member. We assume for the time being that all sites are in the same view. In the next sections we shall extend the discussion to accommodate failures and recoveries.

We use the transaction model of [9]. A transaction is a sequence of read and write operations on objects. As for replication, we use the *Read-One-Write-All* (ROWA) strategy: a transaction performs read operations on the local copy while write operations are applied to all copies. Concurrent transactions with conflicting operations (accessing the same object and at least one of them is a write) must be isolated from each other. We use 1-copy-serializability as correctness criteria: all sites execute conflicting operations in the same order and there exists a serial execution of the transactions with the same order of conflicting operations.

Various ROWA protocols based on group communication primitives have been proposed [1, 2, 21, 18, 12, 15, 14, 17]. They vary in the number of messages per transaction, the ordering mechanisms used (FIFO order, total order etc.), and their concurrency control. For simplicity, we describe reconfiguration in the context of only one of these protocols [1]. We have chosen this protocol because it is simple to describe and there exist various protocols that follow similar execution and communication patterns (only one message per transaction using the total order multicast) [12, 15, 14], and these protocols have shown good performance. However, reconfiguration associated with other replica or concurrency control schemes will be very similar.

The replica control protocol that we consider is described in the following. We assume that objects are tagged with version numbers. A transaction T is submitted to some site S in the system and executed in several phases. For now, we assume that either all read operations precede the write operations, or all write operations are delayed until all read operations have been performed. The first two phases are local to S, while all other phases are executed at all sites:

I. **Local Read Phase**: For each read operation $r(X)$ on object X acquire a shared read lock and execute the operation on the local copy.

II. **Send Phase**: Once all read operations are executed bundle a single *transaction message* m_T containing all write operations and the identifiers of the objects read by T along with the respective version numbers. Send m_T using the total order multicast.

III. **Serialization Phase**: Upon delivery of a transaction message m_T perform in an atomic step:

1. **Assign global identifier**: Assign a globally unique identifier $gid(T)$ to T. $gid(T)$ is the sequence number of its transaction message m_T, i.e., the position of m_T in the total order of all delivered transactions.
2. **Version Check**: For each object X read by T, if the local version number of X (after applying all updates of transactions delivered before T) is greater than the version number read by T, then abort and terminate T.
3. **Lock Phase**: Request all write locks for T. If there is a local transaction T' in its local phase with a conflicting read lock, abort T'.

IV. **Write Phase**: As soon as the lock for a write operation $w(X)$ is granted, perform the corresponding write operation. Assign the version number $gid(T)$ to the object X.

V. **Commit Phase**: As soon as all write operations have been performed, release all locks and commit T.

Only phase III is executed serially for all incoming transactions. The execution of write operations (phase IV) is only serial in the case of conflicts. Non-conflicting operations can be executed concurrently and different sites may even commit transactions in different orders as long as they do not conflict. This is important since processing messages serially as assumed for most applications deployed over group communication (including, e.g., [2]) would result in significantly lower throughput rates. Our protocol uses the sequence number of the transaction message m_T as the global identifier $gid(T)$ of T. This has important advantages. First, $gid(T)$ can be determined independently at each site. Second, it represents the serialization order of transactions. Last, by using the gid for tagging objects, we have the guarantee that all sites have the same version number for an object at a given logical time point. Notice that version numbers are necessary to detect whether read operations have read stale data. The protocol is serializable since read/write conflicts are handled by aborting the reading transaction, write/write conflicts are ordered in the same way at all sites according to the total order of the multicasts, and write/read conflicts are handled by traditional 2-phase-locking (the read waits until the write releases the lock).

2.3. Failures

In [15], we have shown that the replica control protocols described in the previous section can easily be extended to cope with site and communication failures by (i) sending messages using uniform reliable multicast as defined in Section 2.1 and by (ii) restricting transaction processing to the sites of the primary view. If a site leaves the primary view and installs a non-primary view, it simply stops processing transactions and ignores all incoming messages. Members of a consecutive primary view simply continue as before. They do not need to perform any further coordination to handle the failure. We have shown that (i) and (ii) guarantee transaction atomicity. That is, whenever a site commits a transaction T, T will be committed at all sites that are member of a primary view for a sufficiently long time. Moreover, no other site aborts T (a site might fail before committing T, but then, T was still active or had not started executing at the time of the failure). The corresponding behavior holds in the case a site aborts a transaction T.

With weaker forms of message delivery (e.g., reliable delivery), transaction atomicity can be violated: a failed site S might have committed a transaction T shortly before the failure even though m_T was not delivered at the sites that continue in a primary view [15]. Upon recovery of S, T must be reconciled (see, e.g., [13]).

3. Reconfiguration in Replicated Databases

Before transaction processing can begin, the system "bootstraps" as follows. An initial set of sites that defines a majority and that has a copy of the database start by joining the group. Transaction processing begins as soon as they have installed a view that includes all of them, defining the first primary view. At this point other sites may start and join the group. These "new" sites may or may not have the initial copy of the database. A site that crashes and recovers will perform some clean-up on its copy of the database (see below) and then will join the group again.

In the following, we shall describe the actions necessary to enable joining sites to resume transaction processing when they installs a primary view. S_r depicts a site that is recovering after a failure (S_r had crashed or a network partition had occurred), S_n depicts a new site, S_j depicts either type of joining site. Furthermore, S_p denotes a peer site transferring the current database state to S_j. Several tasks and issues for online reconfiguration can be identified:

- **Single Site Recovery** As in centralized systems, S_r first needs to bring its own database into a consistent state. This requires the redoing of updates of committed transactions that were not yet reflected in the database when the failure occurred and the undoing of updates of transactions that were aborted or still active at the time of the failure. For this, each site usually maintains a log during normal processing such that for each write operation on object X the before- and after-images of X are appended to the log. Since single site recovery is a standard database technology performed before S_r rejoins the group, we refer to [9] for details.
- **Data Transfer** A peer site S_p of the primary view must provide S_j with the current state of the database. The simplest solution is to send an entire copy (this is the only solution in the case of a new site). Alternatively, S_p only sends the data that was updated after S_r failed.
- **Determination of a Synchronization Point:** Care has to be taken in the case transaction processing is not suspended during the data transfer. We must guarantee that for each transaction in the system, either the updates of the transaction are already reflected in the data transfered to S_j or S_j is able to process the transaction after the transfer has successfully terminated. Determining the synchronization point depends strongly on the data transfer technique that is employed.

In the following, we shall assume that a primary view always exists. In the case the primary view disappears (i.e., a total failure), transaction processing may resume only after execution of a *creation protocol* for determining the set of all transactions committed in the system and for applying the changes of such transactions at all participating sites. In the context of the replica control protocols presented in section 2.2, the creation protocol, in general, requires the involvement of all sites. This is ultimately due to the asynchrony between the database system and the group communication system: a site actually commits a transaction T some time after the delivery of the transaction message m_T; during this time the site could install a new view and/or fail. For example, suppose there are three sites S_1, S_2, S_3 all members of the primary view V. S_1 sends the transaction message m_T that is delivered in V at all sites. It might happen that S_1 commits T and then fails, whereas both S_2 and S_3 first install a new primary view W excluding S_1, and then fail before committing T. In that case, only the log of S_1 will contain the commit record of T. Therefore it is neither enough that the creation protocol involves a majority of sites nor that it involves all sites of the last primary view W. Instead, the logs of all sites in the system have to be considered. We omit the details of the creation protocol (it merely requires comparing the highest transaction identifier gid_S of transactions applied by each site S). Note that if S_2 and S_3 had not failed they would have committed T guaranteeing that the members of the primary view commit all transactions that are committed by any site in the system.

4. A Suite of Reconfiguration Algorithms

Efficiency of a given reconfiguration strategy depends on a number of parameters: the size of the database, the transaction throughput, the read/write ratio within the workload, the system utilization and so on. As a consequence, one should be able to adapt the strategy to the specific scenario, in particular, with respect to the data transfer task. In the following we will discuss stepwise redefined transfer strategies aiming to solve three issues. First, the data transfer should interfere as little as possible with ongoing transaction processing. Second, it should require as little CPU and network overhead as possible. And third, the solutions should be easy to implement with existing database technology.

4.1 Data Transfer within the Group Communication System

Some group communication systems are able to perform data transfer during the view change (e.g. [10, 3, 22]). The essential aspects of this feature are as follows. Let V and

W be two consecutive primary views and let W be the first view including the joining site S_j. The application defines the state that has to be transferred to S_j (by providing marshalling/unmarshalling routines that will be invoked by the system when needed). During the view change protocol, the group communication system fetches the state from the application layer of some member of V, then incorporates this state at the application of S_j and finally delivers the view change event $vchg(W)$. During the view change protocol the activity of the application is suspended and the system does not deliver application-originated messages. As a result, all sites that install W do so with the same state.

Such an approach has important disadvantages. First, the group communication system can only send the entire database, because the system does not know which data has actually been changed since S_j's failure. This may be unacceptable if S_j is recovering from a short down-time. Second, the database would have to remain unchanged for the entire data transfer. Considering the enormous size of current databases, this can clearly violate the common 24-hour/7-day availability requirement. Last, group communication systems usually assume that they have control over the common group state. The database, however, is controlled by the database system, and if the group communication wants to access the data, it has to do so through the traditional interface, i.e., it has to submit transactions. While this might be feasible, it will be highly inefficient. Similar remarks apply to approaches in which the system relays to S_j all messages delivered during S_j's down-time, rather than the application-defined state (e.g., [2]). In this case S_j might have to apply thousands of transactions and not be able to catch up with the rest of the system.

4.2. Data Transfer within the Database System

As a result of the previous considerations, we believe that the data transfer should be performed by the database system using the appropriate database techniques. The group communication system should only provide the appropriate semantics to coordinate the data transfer.

We shall consider the following framework for all alternatives depicted in the following sections: Let W be the primary view installed when S_j joins the group. During the view change no database related state is transferred. Upon the delivery of $vchg(W)$, sites that were members of the previous primary view V elect one of them to act as peer site S_p for S_j. Election can be performed without message exchange, based on the compositions of V and W. Transaction processing continues unhindered at all sites in W, except for S_p and S_j. S_p transfers the data to S_j and S_j installs it. The data transfer need not occur through the group communication platform but could, e.g., be performed via TCP between S_p and S_j. For all but the last of the following data transfer strategies, the synchronization point in regard to concurrent transaction processing will be as follows: S_p transfers a database state including the updates of all transactions which were delivered *before* the view change $vchg(W)$. However, the data does not include the updates of transactions delivered *after* $vchg(W)$. Instead, S_j enqueues all transaction messages delivered after $vchg(W)$ and processes them once the data transfer is completed. After that, S_j can start executing its own transactions. We first assume that no further view changes occur during reconfiguration. We relax this requirement in Section 5.

4.3. Transferring the Entire Database

A simple option for data transfer is to transfer the entire database. This is mandatory for new sites but also attractive for recovering sites if the database is small or if most of the data has been updated since S_r failed. In order to synchronize with concurrent transactions, S_p transfers the data within the boundaries of a "data transfer transaction" DT:

I. *Lock Phase:* Upon delivery of $vchg(w)$, create transaction DT and request in an atomic step read locks for all objects in the database. Order these read locks directly after all write locks associated with transaction messages delivered before $vchg(w)$. Successive transactions requesting a write lock for an object locked by DT must wait until DT releases the lock.

II. *Data Transfer Phase:* Whenever a lock on object X is granted, read X and transfer it to S_j (which incorporates X into its database and sends an acknowledgment back). As soon as the acknowledgment is received, release the lock and normal processing can continue on X. Of course, both S_p and S_j can pack multiple objects and acknowledgments, respectively, in a single message.

Notice, that read operations can continue unhindered on S_p. Write operations are only delayed on objects that are not yet transferred. Also note that in order to reduce the number of locks, DT can request course granularity locks (e.g., on relations) instead of fine granularity locks on individual objects. The normal transactions can still request locks on a per object basis. The most important aspect in here is that DT's read locks must cover the entire database.

4.4. Checking Version Numbers

While transferring the entire database is simple to implement, it will often be highly inefficient, e.g., when S_r has been down for a very short time or when big parts of the database are seldomly updated. In such cases it may be more efficient to determine which part of the database actually needs to be transferred, i.e, which objects have been changed since S_r's failure, and to transfer only this part.

To do so, S_p must know up to when S_r has executed transactions. For this, S_r informs S_p about its *cover* transaction. The cover transaction for a site S is the transaction with the highest global identifier $gidmax_S$ such that S has successfully terminated all transactions with $gid \leq gidmax_S$. S_r can easily determine $gidmax_{S_r}$ by scanning its single site recovery log (details are omitted for space reasons). Now recall that: (i) all sites have the same global identifiers for transactions (namely the sequence numbers of the corresponding transaction messages), and (ii) conflicting transactions are serialized according to their gids at all sites. Accordingly, if S_p sends the objects that were updated by committed transactions with $gid > gidmax_{S_r}$, then S_r will receive all changed data.

Since the replica control protocol of Section 2.2 tags each object with the transaction that was the last one to update it, it is easy to determine the objects that have been changed since $T_{gidmax_{S_r}}$. The data transfer phase of the protocol of Section 4.3 can be modified as follows:

I. *Data Transfer Phase:* Whenever a lock on object X is granted, check whether the version is labeled with a transaction T for which $gid(T) > gidmax_{S_r}$. If this is the case, transfer X as in the previous section, otherwise ignore X and release the lock immediately.

4.5. Restricting the Set of Objects to Check

The optimization of the previous section still needs to scan the entire database, which may involve a considerable overhead. Moreover, an object is locked from the start of DT until it is either transferred or considered non-relevant. Thus, transaction processing on S_p can be delayed for quite some time. Finally, not all replica control protocols tag objects with version numbers as required by the protocol. In this section, we present an alternative that avoids these problems: (i) it does not rely on the use of version numbers as object tags, thus it can be applied to any database system; (ii) it does not require scanning all the objects in the database; and (iii) it unlocks non-relevant objects sooner.

We propose to maintain a *reconstruction table RecTable* at each site keeping information about recently changed data. That is, a record of $RecTable$ consists of an object identifier $id(X)$ and a global identifier gid indicating that T_{gid} was the last transaction to update X. $RecTable$ should hold a record for each object X updated by transaction T if there is at least one site S that might not have yet executed T (e.g., S is not in the primary view when T commits). Only in the case of a data transfer $RecTable$ must be completely up-to-date (see below). Otherwise, it can be maintained by a background process whenever the system is idle:

I. *Registration of updates:* Let object X be last updated by committed transaction T. If $RecTable$ has already a record for X, then set the transaction identifier of this record to $gid(T)$, otherwise insert a new record $(id(X), gid(T))$.

II. *Deleting records:* Let $gidmax_S$ be the global identifier of the cover transaction for site S. Let $gidmax_{min}$ be the minimum of all $gidmax_S$. Sites maintain a conservative estimate of $gidmax_{min}$ through regular exchange of $gidmax$ values (in particular, by using for site S not in the primary view the last $gidmax_S$ announced by S while in the primary view). When a site increases its estimate of $gidmax_{min}$, it deletes from $RecTable$ each record with $gid(T)$ such that $gid(T) \leq gidmax_{min}$.

Based on $RecTable$, the data transfer protocol of section 4.3 can be modified by changing the lock phase as follows:

I. *Lock Phase and Determining the Data Transfer Set:* Upon delivery of $vchg(w)$, create transaction DT, request a single read lock on the entire database and wait until all transactions delivered before $vchg(W)$ have terminated and their updates are registered in $RecTable$.
Let $DID = \{id(X) | (id(X), gid) \in RecTable$ and $gid > gidmax_{S_r}\}$. Request read locks on objects X, with $id(X) \in DID$ and release the lock on the database. At this point, proceed with the data transfer phase as usual for the objects whose identifiers are in DID.

In contrast to the previous protocols we now set only a single lock on the entire database. Once the data set to be transferred is determined, which can be done easily with $RecTable$, this lock is replaced by the fine granularity locks on the individual objects. Hence, non-relevant data is locked for only a very short time.

$RecTable$ may be implemented as a traditional table in a relational database system. In this case, DID can be constructed with the simple SQL statement "SELECT $id(X)$ from $RecTable$ where $gid > gidmax_{S_r}$". Fast response to this query may be obtained by having an index on $RecTable$ with the global identifier being the search key. In the same way, an index on the object identifier will fasten the registration of new updates to $RecTable$. Notice that maintenance of $RecTable$ is mostly asynchronous and changes to $RecTable$ do not need to be forced to disk; thus we believe that its impact during normal transaction processing be small. Finally, notice that $gidmax_{min}$ can only increase and that when all sites are up and in the primary view records that are no longer needed are continuously deleted. In the case of relational databases we estimate the maximum additional space overhead to be the same as for two additional indices for each relation in the database.

4.6. Filtering the Log

So far, S_p has had to set read locks to synchronize the data transfer with concurrent transaction processing. Although the previous optimizations, amongst other things, shortened the time such locks are set on non-relevant data, locks on relevant data are still long.

We can avoid setting locks on the current database if the database system maintains multiple object versions. In this case, transactions can update the objects of the database unhindered while S_p will simply transfer the versions of the objects that were current when $vchg(W)$ was delivered. No data transfer transaction needs to be created and transactions at S_p can access the current database objects unhindered. Multiple versions are given, for instance, if the log maintained for single site recovery stores for each update of object X the entire physical after-image of X. The details of such a protocol can be found in [8].

4.7. Lazy Data Transfer

All the previous solutions use the view change as a synchronization point in that S_j enqueues all transaction messages delivered after $vchg(W)$ and eventually applies them to its local (up-to-date) copy of the database. While this is a simple and intuitive approach, it has several disadvantages. First, the peer node S_p has to delay transaction processing on data that must be transferred (unless there exist multiple object versions). Second, if the workload is high and the data transfer takes a long time, then the joining site S_j might not be able to store all transaction messages delivered during the data transfer, or it might not be able to apply these transactions fast enough to catch up with the rest of the system. Finally, a failure of S_p requires S_j to leave and rejoin the system and to restart the data transfer from scratch: since the sites that could take over after S_p's failure have continued transaction processing, they might not be able to provide S_j with the state of the database that was current upon the delivery of $vchg(W)$. These drawbacks can be avoided if we decouple the synchronization point from the delivery of $vchg(W)$ [7]:

- S_j initially discards transaction messages delivered in W and S_p starts a data transfer as described below. When the transfer is nearly completed, S_p and S_j will determine a specific *delimiter transaction* T_d, delivered in W. S_p transfers all changes performed by transactions with $gid \leq d$. S_j starts enqueueing transaction messages with gid such that $gid > d$ and will apply these transactions once the data transfer is completed.
- S_p transfers the data in several rounds. Only in the last round (when T_d is determined), the transfer is synchronized with concurrent processing by setting appropriate locks. The idea is to send in each round the objects that were updated during the data transfer of the last round. The last round is started either when the number of objects that are left to be transferred does not exceed a given threshold k_{max} or a maximum number $R_{max} > 1$ of rounds has been reached.

Before discussing this solution in more detail, we highlight its advantages. First, S_p has a better control of when to perform the data transfer and at which speed. Second, S_j has to enqueue and apply far less transactions. Third, the approach allows for much better concurrency as transaction processing at S_p is delayed only in the last round which we expect to be fast, since it will only transfer little data. Finally, failures of S_p before reconfiguration is completed can be handled more efficiently. As we shall see below, in each round i, the updates up to a certain transaction T_{gid_i} are transferred. S_j only has to inform the new peer site S'_p up to which T_{gid_i} it received the updates from S_p, and S'_p can continue the data transfer starting from that transaction. The actions at S_p are as follows:

Round $i, 1 \leq i \leq (n-1)$

1. *Determine the delimiter transaction* T_{gid_i} *of this round:* If $i = 1$ then let gid_i be the identifier of the last transaction delivered before $vchg(W)$. Otherwise, let gid_i be the identifier of the last transaction that was delivered before the round started. Wait until all updates of transactions with $gid \leq gid_i$ are included in $RecTable$ (i.e., at least the updates of all transactions up to T_{gid_i} will be transferred).
2. *Determine the data to be transferred:* If $i = 1$ then let $gid_{st} = gidmax_{S_j}$, otherwise $gid_{st} = gid_{i-1}$. Let $DID = \{id(X)|(id(X), gid) \in RecTable$ and $gid > gid_{st}\}$
3. *Data transfer:* For each $id(X) \in DID$ acquire a short read lock on X, read X, release the lock and then transfer X to S_j (the short read lock is only used to guarantee that only committed data is read). Furthermore, inform S_j about gid_i (for fail-over).
4. *Termin. Check I:* If $i = R_{max} - 1$ then go to Round n.
5. *Termin. Check II:* Let $DID_{next} = \{id(X)|(id(X), gid) \in RecTable$ and $gid > gid_i\}$. If $\#DID_{next} \leq k_{max}$ then go to Round n. Otherwise, increase i and repeat.

Round n:

1. *Determine the delimiter transaction* T_d*:* Inform S_j that this is the last round, and wait for a response (upon reception of this message, S_j starts enqueueing transactions and responds with the identifier $gidpropose$ of the first enqueued transaction). Upon reception of the response, let gid_n be the identifier of the last transaction delivered at S_p that already requested its write locks: $d = max(gid_n, gidpropose - 1)$.
2. *Final data transfer:* The data transfer of the last round is performed by a transaction DT ordered as follows: DT transfers all changes of transactions with $gid \leq d$ but no changes from transactions with $gid > d$ (they will be applied by S_j). DT now follows any of the protocols described in the previous sections. For instance:
   a. *Lock Phase and Determining the Data Transfer Set:* If $d = gid_n$, request immediately a read lock on the entire database. Otherwise, wait until $T_{gidpropose-1}$ has requested its write locks and then request the read lock on the entire database. Wait until all transactions with $gid \leq d$ are included in $RecTable$. Let $DID = \{id(X)|(id(X), gid) \in RecTable$ and $gid > gid_i\}$. Request a read lock for each object X with $id(X) \in DID$ and release the lock on the database.
   b. *Data Transfer:* As in section 4.3.

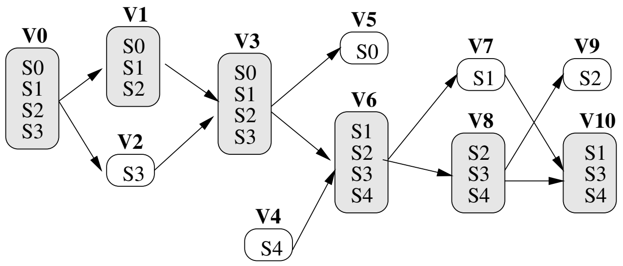

Figure 1. Example of virtual synchrony

Note that the set DID_{next} constructed within the termination check II is an estimate of the DID that will be constructed in the next round, because transaction processing is not suspended. We suggest that in the first round data are transferred per data partition (e.g., per relation). In case of failures during this round, the new peer site does not need to restart but simply continue the transfer for those partitions that S_j has not yet received.

5. Cascading Reconfigurations

A key problem for all solutions presented so far is that further view changes may occur before reconfiguration is completed. The problem is that reconfiguration is not an atomic step but may take a long time. The possibility of view changes during reconfiguration may greatly complicate the details. As an example, consider Figure 1 (ovals indicate views; grey-shaded ovals indicate primary views; consecutive views are connected by an arrow). Suppose that S_0 acts as peer site when S_3 joins the primary view $V3$ and suppose that S_0 leaves the primary view ($V6$) before reconfiguration has completed. Only S_3 and S_0 know that reconfiguration has not yet completed. It follows that S_1 and S_2 in $V6$ do not know whether S_3 is able to process transactions or whether one of them has to resume the data transfer. Similarly, S_4 does not know which of the other sites can act as a peer site for its reconfiguration. In fact, S_4 cannot even tell that there will indeed be an up-to-date peer site: from its point of view, it might be the case that no predecessor of this primary view was primary. The following example shows a further complication: S_1 and S_2 start reconfiguration in view $V6$ for S_3 and S_4. Then, a partition excludes S_1, leading to the primary view $V8$. Finally, S_2 partitions and S_1 reenters the new primary view $V10$. If $vchg(V10)$ is delivered before reconfiguration for S_3 and S_4 is completed, then there would be a primary view in which *no* member is able to process transactions. A further sub-protocol capable of discovering this situation is necessary.

This complexity is induced because a member of a primary view is not necessarily an "up-to-date member". Only if the data transfer is done as part of the view change protocol or if the application is able to perform the data transfer very quickly, the application-dependent notion of "up-to-date member" is essentially the same as "member of the primary view". However, as previously pointed out, such forms of data transfer are unsuitable for database systems.

The next sections outline an extension of the traditional group communication abstraction, called *enriched view synchrony (EVS)* [4], and its possible use in the context of database applications [7]. EVS will allow us to easily handle the failure scenarios described in this section.

5.1. EVS

EVS replaces the notion of a view by the notion of enriched view, also called *e-view*. An e-view is a view with additional structural information. Sites in an e-view are grouped into non-overlapping *subviews* and subviews are grouped into non-overlapping *subview-sets*. Figure 2 depicts examples where the outer ovals denote views, the dashed ovals indicate subview-sets and the inner squares denote subviews. As before, a view change notifies about a change in the composition of the e-view (sites that appear to be reachable) and such changes are performed automatically by the system. Additionally, EVS introduces e-view change events that notify about a change in the *structure* of the e-view in terms of subviews and subview-sets (dashed arrows in the figure indicate e-view changes). In contrast to view changes, e-view changes are requested by the application through dedicated primitives. These primitives will allow us to encapsulate reconfiguration: `Subview-SetMerge(subview-set-list)` creates a new subview-set that is the union of the subview-sets given in subview-set-list (e.g., the e-view $EV4$ is installed as a result of a Subview-SetMerge(); note that this e-view differs from the previous one in structure but not in composition). `SubviewMerge(subview-list)` creates a new subview that is the union of the subviews given in subview-list. The subviews in subview-list must belong to the same subview-set and the resulting subview belongs to the subview-set containing the input subviews (e.g., the e-view $EV5$ is installed as a result of a Subview-Merge()).

The characteristics of EVS are summarized as follows: The system maintains the structure of e-views across view changes (in $EV3$, S_3 and S_0, S_1, S_2, respectively, are still in their own subviews and subview-sets). E-view changes between two consecutive view changes are totally ordered by all sites in the view. Finally, if a site installs an e-view ev and then sends a message m, then any site that delivers m delivers it after installing ev. Note, that the original definition of EVS [4] does not consider total order and uniform delivery. However, accommodating these properties will be simple since they are orthogonal to the properties of EVS.

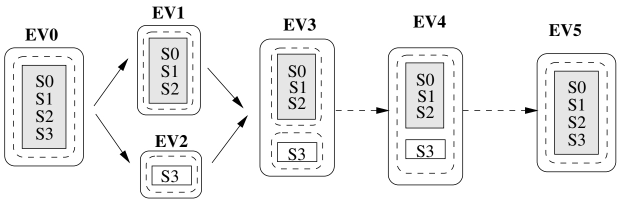

Figure 2. Example of EVS

5.2. Reconfiguration Using EVS

Transaction processing will be allowed only within a "primary subview", i.e., the subview with a majority of sites. Sites whose current subview is not primary do not process transactions (but might enqueue transaction messages if they are currently in a reconfiguration process). Figure 2 illustrates the main idea. At EV0 we assume all sites to have identical copies of the database, they are in the same subview and this subview is a primary subview. Then S_3 leaves the primary view because of a partition or failure and re-enters in $EV3$. Note that S_3 is not a member of the primary subview but remains in its own subview and subview-set. Reconfiguration is initiated by the peer site, say S_0, in the primary subview which submits the `Subview-SetMerge` message. When the corresponding e-view change is delivered ($EV4$), each site knows that reconfiguration has started. S_0 starts transferring data to S_3, following any of the solutions presented previously. When the transfer is completed and S_3 is able to process transactions autonomously, S_0 submits the `SubviewMerge`. The delivery of the corresponding e-view change ($EV5$) , which includes S_3 in the primary subview, represents the final synchronization point, i.e., all sites know that S_3 has the correct database state and executes transactions by itself. In other words, reconfiguration is encapsulated within the e-view changes $EV4$ and $EV5$.

In short, a primary view is represented differently depending on whether transaction processing is enabled in that view or not, the former being the case only when the view contains a primary subview. Moreover, the representation indicates which sites can process transactions (those in the primary subview) and which sites are being brought up-to-date (those whose subview is not primary but whose subview-set contains a primary subview). The resulting framework enables simpler algorithms with respect to virtual synchrony as it was depicted in Figure 1:

- A joining node enters the primary view as a result of a decision taken by the system, but it is the database system that decides when to issue the SubviewSet-Merge() for starting the data transfer. This can be any time after the view change, e.g., when the workload is low.
- When a site joins a primary view, it realizes *locally*, whether there is an operational primary subview or not. In the first case, it can remain quiet waiting for a peer site. In the latter case a creation protocol has to be run.
- When a peer site S_p fails (i.e., leaves the primary view and primary subview) before the data transfer to a joining site S_j has completed, the sites remaining in the primary subview will realize *locally* that S_j is not yet up-to-date: S_j will be member of the their subview-set but not of their subview.
- When a site S_j enters the primary subview, all sites in the view know that S_j is up-to-date and operational.
- When a peer site S_p fails and the view excluding S_p is still primary but there is no longer a primary subview, all sites in the primary subview realize *locally* that transaction processing must be suspended.

The handling of view changes and e-view changes by any site S in the primary subview can be summarized as follows (MySubview-Set and MySubview refer to the subview-set and subview of S):

I. *View change excluding or including new sites:*
   1. *New sites:* for each new subview-set $sv\text{-}s$ in the view: (i) choose deterministically a peer site S_p in the primary subview (S_p will be the peer site for all sites in $sv\text{-}s$); (ii) if $S = S_p$, then issue whenever appropriate a `Subview-SetMerge(MySubview-Set,sv-s)`.
   2. *Site S_p left the view and S_p was the peer for a site S_j:* Determine deterministically the new peer site S'_p. If $S = S'_p$ then: If S and S_j are in different subview-sets, then issue a `Subview-SetMerge(MySubview-Set,Subview-Set-of-`S_j`)` when appropriate (S_p left the primary subview before initiating the merge); otherwise, resume the data transfer (S and S_j are already in the same subview-set but not yet in the same subview).
   3. *A site S_j for which S was the peer site left the view:* stop the data transfer to S_j.
   4. *S has left the primary subview (thus the primary view):* stop processing transactions and stop any data transfer to recovering sites for which S is the peer site (this may occur as a result of partitions).

II. *E-view change notifying about the merging of subview-sets:* for each new subview sv in the subview-set of S: if S is the peer site S_p (determined in step I.1), then start data transfer to all sites in sv.

III. *E-view change notifying about the merging of subviews:* Recovery of the merged sites is completed.

Moreover, when the data transfer for all sites of a subview sv for which S acts as peer site is completed, S issues `Subview-Merge(MySubview,sv)`.

The behavior of the joining site S_j depends on the specific reconfiguration algorithm. With all proposed options except for lazy data transfer, S_j discards transactions until it is in the same subview-set as the primary subview (start of reconfiguration). Then it starts enqueueing transactions and applies them after its database is up-to-date. With lazy data transfer, the delimiter transaction T_d will be a transaction delivered between the Subview-SetMerge and the SubviewMerge event. The protocol guarantees that joining sites will receive the current database state and join the primary subview as long as there exists a primary subview for sufficiently long time (given by I.1, I.2, II, and the SubviewMerge). Also, only sites of the primary subview execute transactions and their databases are always up-to-date.

6. Conclusions

This paper provides an in-detail discussion of online reconfiguration in replicated database systems using group communication. The focus on the paper is on two important issues: efficient data transfer and fault-tolerance.

In regard to data transfer we propose several protocols with which we depict the main alternatives for a database supported data transfer and the most important issues to consider: determining the data that must be transferred, exploiting information available within the database (e.g., version numbers, log), maintaining additional information to fasten recovery, allowing for high concurrency, etc.

We do not provide a final statement which of the data transfer solutions to choose. First, for some of them specific characteristics of the underlying database system must be given. If these features are not provided they have to be implemented and integrated into the database system. This can be very difficult and might not pay off. But the efficiency of the solutions also depends on parameters like the size of the database, the percentage of data items updated since the recovering site failed, etc. We are planning to explore these issues by a real implementation based on Postgres-R [14].

Making the data transfer a task of the database introduces problems in regard to fault-tolerance. Since reconfiguration is not an atomic operation, simple virtual synchrony does not reflect sufficiently the state of the different sites in the system. EVS, in contrast, promotes a programming style in which the notion of "up-to-date" member depends on the membership of the primary subview, not of the primary view. Using EVS we are able to encapsulate the reconfiguration process, and the database system receives a more realistic picture of what is happening in the system.

References

[1] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proc. of Euro-Par*, Passau, Germany, 1997.

[2] Y. Amir. *Replication Using Group Communication over a Partioned Network*. PhD thesis, Hebrew University of Jerusalem, 1995.

[3] Y. Amir, G. V. Chockler, D. Dolev, and R. Vitenberg. Efficient state transfer in partitionable environments. In *Proc. of the ERSADS Seminar*, Zinal, Switzerland, 1997.

[4] Ö. Babaoğlu, A. Bartoli, and G. Dini. Enriched view synchrony: A programming paradigm for partitionable asynchronous distributed systems. *IEEE Transactions on Computers*, 46(6):642–658, June 1997.

[5] Ö. Babaoğlu, R. Davoli, L. Giachini, and P. Sabattini. The inherent cost of strong-partial view-synchronous communication. In J.-M. Hélary and M. Raynal, editors, *Distributed Algorithms*, Lecture Notes in Computer Science, pages 72–86. Springer Verlag, 1995.

[6] Ö. Babaoğlu, R. Davoli, and A. Montresor. Group communication in partitionable systems: Specification and algorithms. Technical Report UBLCS-98-1, Dept. of Computer Science, University of Bologna, Apr. 1998. To appear in IEEE Transactions for Software Engineering.

[7] A. Bartoli. Handling membership changes in replicated databases based on group communication. Technical report, Facoltà di Ingegneria, Università di Trieste, 2000.

[8] A. Bartoli, B. Kemme, and Ö. Babaoğlu. Online reconfiguration in replicated databases. Technical report, University de Bologna, Italy, 2000.

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

[10] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. The ISIS - system manual, Version 2.1. Technical report, Dept. of Computer Science, Cornell University, Sept. 1993.

[11] M. Buretta. *Data Replication*. Wiley Computer Publ., 1997.

[12] J. Holliday, D. Agrawal, and A. E. Abbadi. The performance of database replication with group multicast. In *Proc. of FTCS*, Madison, Wisconsin, 1999.

[13] B. Kemme. *Database Replication for Clusters of Workstations*. PhD thesis, ETH Zürich, 2000.

[14] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proc. of VLDB*, Cairo, Egypt, 2000.

[15] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, September 2000.

[16] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, 1996.

[17] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proc. of DISC*, Toledo, Spain, 2000.

[18] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proc. of Euro-Par*, Southampton, England, 1998.

[19] D. Powell and other. Group communication (special issue). *Communications of the ACM*, 39(4):50–97, April 1996.

[20] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proc. of SIGMOD*, Denver, Colorado, 1991.

[21] I. Stanoi, D. Agrawal, and A. El Abbadi. Using broadcast primitives in replicated databases. In *Proc. of ICDCS*, Amsterdam, Holland, 1998.

[22] A. Vaysburd. *Building Reliable Interoperable Distributed Objects with the Maestro Tools*. PhD thesis, Cornell University, 1998.