Recovery Protocols for Replicated Databases - A Survey [*]

Luis H. García-Muñoz, J. Enrique Armendáriz-Íñigo, Hendrik Decker, Francesc D. Muñoz-Escoí
Instituto Tecnológico de Informática, Valencia, España
{lgarcia, armendariz, hendrik, fmunyoz}@iti.upv.es

Abstract

The main goal of replication is to increase dependability. Recovery protocols are a critical building block for realizing this goal. In this survey, we present an analysis of recovery protocols proposed in recent years. In particular, we relate these protocols to the replication protocols that use them, and discuss their main advantages and disadvantages. We classify replication and recovery protocols by several characteristics and point out interrelationships between them.

Keywords: Replication, Databases, Recovery, Dependability, Performance, Fault Tolerance, High Availability

1 Introduction

In replicated databases, identical copies of data items are stored on different computers at different, possibly very distant sites. As a subarea of database theory and practice, the field of replication is acquiring growing relevance. It is increasingly used for supporting dependability, i.e., performance, fault tolerance and high availability.

Among all available replicas, clients can improve their throughput by transparently accessing the server replica that is closest to them. Suitable protocols cater for mutual data consistency at each replica. Whenever a server site fails or the connection to it is broken, client transactions are redirected to other available servers. For maintaining high availability, a need for efficient recovery procedures arises, for bringing failed or temporarily disconnected nodes back into the network of active servers as fully functional peers.

The recovery task basically consists in transferring the updates lost during failure from one or more active nodes to one or more recovering sites, without impeding the overall system capability of providing normal application services. Since recovery must re-establish consistent data, the development of recovery protocols must take the idiosyncrasies of the used replication protocols into account. Under this premise, various recovery protocols have been proposed in the literature, among them [15, 11, 13, 12, 6, 7, 2].

Ideally, a good replication system should use mechanisms that are simple (so as to reduce overhead), cope well with network overload, maintain consistency, provide continuous service and avoid transaction rollbacks [11]. Similarly, a good recovery protocol should be simple, efficiently distribute the recovery work among available nodes, and seamlessly allow for simultaneous concurrent transactions. Additionally, both replication and recovery protocols must take into account the concurrency of transactions, which in many applications are required to comply with the ACID requirement [3], i.e., the atomicity, consistency, isolation and "durability" (a.k.a. persistence) of updates.

Typically, a synchronization mechanism for supporting the updating of alive and recovering replicas is deployed, since otherwise, recovery may become too complicated. A straightforward way to synchronize replicas would be to interrupt the ongoing application, but then, high availability is sacrificed. However, with a suitable Group Communication System (GCS) [8] and virtual synchrony [4], it is possible to generate synchrony points between failed and recovering sites, taking the set of messages delivered to non-failed sites into account. The GCS provides a membership service and a reliable multicast. Membership services maintain a list of available, i.e., currently active and connected sites, and implement the view concept [8], distinguishing between different states of the update history in which data items are seen by mutually isolated groups of servers.

This work is focused on replication and recovery strategies designed for the primary partition model [17, 8]. It enforces that, in case of network partitioning, only the subgroup that has a majority of the system replicas (if any) can continue processing transactions. Thus, consistency is easily maintained since no other group of active replicas can cause conflicts in the recovery procedures. Hence, there is always a group of replicas that maintains an up-to-date database state, and any other subgroup can recover by obtaining such state from some replica of the majority subgroup. This model is typically assumed in recent works about database replication and recovery. Partitionable mod-

[*]This work has been partially supported by the MCYT and MEC grants TIC2003-09420-C02 and TIN2006-14738-C02.

els have been assumed in the field of mobile databases, but we do not survey such kind of systems in this document.

The main goal of this work is to present a survey of alternative options and strategies employed by replication and recovery protocols developed in recent years. In that context, also different concepts for GCSs and virtual synchrony are reviewed. In sections 2 and 3, we characterize and classify different kinds of replication protocols in regard to their interplay with recovery. In section 4, we narrow the focus on replication protocols based on group communication and discuss them broadly. In section 5, we conclude. As an appendix, we present a tabled comparison that summarizes and highlights significant characteristics as distinguished in sections 2 - 4.

2 Basic Questions for Replication Protocols

Since recovery is usually embedded in the replication process, the following questions must be answered (cf. [10, 17, 18, 19]). The various concepts as mentioned and labeled with acronyms below, are going to be explained in subsequent subsections.

1. *Server Architecture* (A): are transactions executed in *p*rimary-copy (P) or *u*pdate-anywhere (U) mode?
2. *Server Interaction* (I): is interaction between replica servers *c*onstant (C) or *l*inear (L)?
3. *Transaction Termination* (T): do transactions terminate by *v*oting (V) or *n*on-voting (N)?
4. *Update Propagation* (U): is it *eager* (E) or *lazy* (L)?

Clearly, answers to some of the questions establish a distinction between various kinds of replication protocols. These are going to be used in 3 for classifying replication protocols that host the recovery protocols as addressed in section 4.

In general, protocols must take the objectives as enumerated below into account.

1. *Enable and optimize transaction concurrency*. Two basic kinds of concurrency control mechanisms are distinguished as follows.
   - *Optimistic Concurrency Control*. This assumes that transaction conflicts are unlikely to occur when shared data are accessed. In that case, remote server resources can remain largely untapped until transaction commit time. And if conflicts do occur, then transactions are aborted without further ado, so that they may be re-tried later.
   - *Pessimistic Concurrency Control*. Conflicts are expected to occur, and remote resources must be ready to be tapped on demand at any moment during transaction time. Unless a deadlock occurs, pessimistic concurrency control makes sure that transactions will terminate successfully. Implementations of this pessimistic policy are the well-known Two Phase Locking (2PL), Strict 2PL and Timestamping (which, however, is occasionally used with optimistic control as well).
2. *Minimize transaction abortions*. This depends on the used concurrency control (as indicated in the previous point) and on the type of transactions, in the sense that, the more write operations there are and the longer the transactions last, the more conflicts are likely.
3. *Maintain replica consistency*. This is strongly though not inextricably related to concurrency control. In general, applications differ in their requirements of consistency, so that the isolation level of transactions may vary.

3 Classification of Replication Protocols

In [10], the following two modes for propagating updates to replicas are distinguished:

1. *Eager*. All replicas are updated during transaction execution time, i.e., no transaction is committed before all network nodes are updated. This guarantees a very high degree of replication consistency but increases transaction response times and thus slows down performance, due to the multiplicity of updates and message rounds. Moreover, eager updating is not a viable solution in mobile networks where nodes may be disconnected for extended periods.
2. *Lazy*. The updates of a transaction generally are executed in a single dedicated replica, typically the nearest one or the owner node of updated items. Updates are propagated to all remaining replicas asynchronously, which in general amounts to a separate transaction per node. Thus, lazy update propagation permits a wide variety of synchronization points, which however are needed because temporary inconsistencies may easily arise, due to the lack of synchrony.

According to [19], eager replication protocols can be classified along to the following three dimensions.

1. *Server Architecture*: it determines where transactions are executed in the first place. According to [10], the two main options are

- *Primary copy* [13]. Transactions always are directed to a designated node, which holds the "primary copy" of updated items. It is the only one to actively process updates solicited by transactions.
- *Update anywhere* [14]. Any replica can directly process any transaction, i.e., transaction updates can be directed to and processed by any replica.

2. *Server Interaction.* The degree of communication among database servers at transaction time, i.e., the amount of network traffic generated by a given replication protocol, is measured by the number of interchanged messages. Two cases can be distinguished:
    - *Constant Interaction.* Independently of the number of operations in the transaction, a constant number of messages is used to synchronize the servers. Typically, protocols in this category groups all operations of the transaction into a single message [1, 2, 6, 7, 11, 12, 13, 15].
    - *Linear Interaction.* Here, each operation of a transaction is dealt with separately. Operations can be sent either as SQL statements or as log records that contain the results of executing operations in particular servers [4, 11].
3. *Transaction Termination.* This is related to how atomicity is guaranteed. Two cases can be distinguished:
    - *Voting Termination.* Replicas are coordinated by an extra round of messages. It can be as complex as an atomic commitment protocol, or as simple as a single confirmation message [2, 4, 14].
    - *Non-voting Termination.* Nodes can decide on their own to commit or abort a transaction [9, 16].

As indicated above, it is also important to take the network's partition model into account [4, 8, 17]. Mostly, the primary partition model is employed. It enforces that, in case of network partitioning, only the sub-group that has a majority of alive nodes can continue to process transactions. Thus, consistency is maintained easily, since no other group of active servers can then cause conflicts. Hence, there always is a group of sites that maintains an up-to-date state of replicas, and any other sub-group can recover by obtaining this state from some replica of the majority sub-group.

In general, eager update algorithms are preferable to lazy ones whenever replica consistency is achieved in primary copy mode. However, if performance is key and consistency can be compromised to some degree, better results are obtained with lazy update algorithms, no matter whether primary copy or update-anywhere algorithms are used. At first sight, this may seem to contradict statements in [3, 5]. However, GCS-based protocols were not yet considered in those older references, which only considered pessimistic and optimistic concurrency control mechanisms with voting for transaction termination.

4 Recovery Protocols

For recovering a failed site, the actual state of the database needs to be transferred to it. Only after that is accomplished, the recovering site can again accept requests from other sites or from clients. To transfer the current state, three options can be distinguished: either to copy over the whole database, or to only transfer incrementally the last versions of all data items that were modified during the failure period, or to resend the update messages that did not reach the failed node.

For classifying different kinds of recovery protocols, it is useful to answer the following questions. Answers to each of them are going to be addressed in more detail in subsequent subsections.

1. *Transfer Model* (TM): is it a *full database transfer* (FT), or a version-based *incremental transfer* (IT) or are *lost* messages *resent* (LR)?
2. *Concurrency control during recovery*: (a) Regarding optimism (O), is it *optimistic* (O) or *pessimistic* (P)? (b) Considering the number of managers (M), does it use a *single* (S) or *multiple* (M) managers? (c) Is it *multi-versioned (V)* (Y(es)/N(o))?
3. *Recovery-work distribution* (W): is it *centralized* (C) or *distributed* (D)?

4.1 Recovery Protocols from Kemme, Bartoli and Babaoglu

In [15] the authors propose solutions to transfer the state of the database to the recovering replicas without interrupting the transaction process in the rest of the system. To this end, they consider a replication model that applies eager update anywhere, with a constant interaction and non-voting transaction termination. Basically two ways for transferring the recovery information are discussed: (a) the GCS regular state-transfer when a view-change event arises, or (b) using a specially tailored recovery protocol. Option (a) is immediately discarded since the amount of state to be transferred is very big (the entire database) and this is impractical.

We describe on the sequel the five recovery alternatives presented in [15], plus the *enriched view synchrony* mechanism assumed in that paper and used in all its recovery protocols.

4.1.1 Full Database Transfer

Despite discarding a database transfer when it is initiated by the common GCS behavior (GCS are usually employed in the active replication model; when a replica is added, they transfer the current state of the replicated object to such recovering replica), transferring the entire database still has sense in a few cases. Indeed, this management is mandatory for new replicas, but also attractive if the size of the database is small or if most of data has been changed during the failure interval. In this case we have a pessimistic and centralized concurrency control with a unique manager. The advantages of this method are its simple implementation and that it does not fully suspend the execution of the application, since the write operations are only delayed on the objects that are not yet transferred, and read transactions are allowed. The disadvantages are that it is made under a data transfer transaction schema that sets a read lock, which is released when the data has been read, and transferred to the recovering replica. Additionally, this could be highly inefficient in cases where the failure time for a replica was short.

4.1.2 Incremental Transfer Using Version Numbers

If the recovering replica was not active for a very short time, or the data updates were few, may be more advisable to determine which part of the database needs to be transferred. To do so, global identifiers for the transactions are used, so that the replica that will send the information for the recovery, can determine the last transaction that was correctly executed in the recovering replica and with this, the pending updates to send. It has the same replication model exposed before. For the recovery it uses the version-based transference model, with pessimistic concurrency control, a unique manager for the concurrency and the recovery work is not fully suspend the execution of the application, the write operations are only delayed on the objects that are not yet transferred, only the changed data are transferred to recovering replicas and the read lock can be released immediately on the not changed data. The disadvantages are that it is necessary to review the entire database to determine the objects to transfer, which can cause overload. An updated object is locked since the begin of the data transfer transaction until it is either transferred or considered non- relevant. Finally, note that not all the DBMSs can mark the objects with version numbers (i.e., with the identifiers of the transactions that generated their current values) as is required by this recovery protocol.

4.1.3 Reducing the Amount of Data to Check

Using a so-called "reconstruction table" can alleviate the disadvantages exposed in the previous subsection. It is a data structure to store information about recently updated data. A record in this table consists of a row identifier and a global identifier for the last transaction that updated the row. Each update is recorded in the reconstruction table, unless all sites have successfully performed the update.

In contrast to the row level locks of the previously discussed protocol options, this one only needs to set a single lock on the entire database. Once the incremental data set to be transferred is determined, that lock is replaced by fine-grained row level locks on the respective data items.

Replication is as before, and the incremental recovery is accomplished with pessimistic concurrency control and a unique centralized version-based manager site for controlling concurrency and the distribution of recovery tasks.

The main advantages are that version numbers are not needed. Hence, its implementation is more independent of the underlying DBMS, since labeling takes place modularly in a separate table that can be straightforwardly implemented as a relational table. Also a scan of the entire database is no longer needed, and non-relevant data are locked only for a very short period. The disadvantages are that the use of the reconstruction table demands additional space (that, however should be negligible even for very small devices, which nowadays also dispose of vast amounts of memory). And in spite of a relatively fast release of non-relevant data locks, the read-locking time span of relevant data could be considerable.

4.1.4 Filtering the Log

Up to now, sites have been supposed to set read locks for synchronizing the data transfer with concurrent transaction processing. In the previously discussed optimization, locking of non-relevant data is reduced, but locks on relevant data may still last long. To avoid locks, multiple versions of data can be used, i.e., the use of multi-version concurrency control, as in `PostgreSQL`, `Oracle` and optionally in `MS SQL Server`. In that case, transactions can continue to update the database while earlier versions that have been missed by the recovering site are transferred to it.

Recovery is version-based and incremental, concurrency control is pessimistic, and a unique centralized version-based manager site is used to distribute recovery tasks. Advantages are that transaction execution is not suspended, data transfer is not needed and locks are fully avoided. The disadvantage is that multiple data versions must be kept, but that can be left to the underlying DBMS, as in `PostgreSQL`, so that recovery is not burdened with that.

4.1.5 Lazy Data Transfer

Up to this point, all mentioned solutions use view changes as synchronization points. That is a simple approach but has several drawbacks:

1. The recovering site has to delay transaction processing on data that must be transferred (not necessary in version-based concurrency control).

2. If workload is high and data transfer takes long, then a recovering site might not be able to store all transaction messages delivered during data transfer, or it might not be able to apply these transactions fast enough to catch up with the rest of the system.

3. If the recoverer site fails, the recovering site needs to be reset so that recovery process can re-start all over again.

These drawbacks can be avoided if we decouple the synchronization point from the view change.

Initially the recovering site discards the messages delivered in the view change and the recoverer site starts the transfer. When the transfer is about to complete, the recoverer and the recovering sites determine a delimiter transaction to be delivered in the view change. The recoverer site then transfers all changes performed by transactions with an identifier that is smaller than the identifier of the delimiter transaction. The recovering site starts queuing transaction messages with identifier greater than the identifier of the delimiter transaction and finally applies these transactions once the data transfer is completed. The latter is done in several rounds. Only in the last round (when the delimiter transaction is determined), the transfer is synchronized with concurrent transaction processing by setting appropriate locks. The idea is to send in each round those data that were updated during the data transfer of the last round.

Again, the version-based incremental transfer mode is used for recovery, concurrency control is pessimistic, using a unique centralized multi-versioned manager to distribute recovery tasks. The significant variant of this protocol is that the transfer of the actual database state takes place in lazy mode. So, at least the failures at the recoverer site are handled more efficiently. The disadvantage is that this protocol requires a reconstruction table to maintain the information about recently modified data.

4.1.6 Enriched View Synchrony

EVS makes use of an online reconfiguration of recovering nodes. Hence, the following key problem of all solutions for recovery as discussed so far requires even more attention: view changes can happen before reconfiguration is completed. View changes that occur during reconfiguration can cause considerable complications. For example, suppose that a site X acts as the recovering site of a node that joins the primary view and that X leaves the present view before reconfiguration is completed. At this point, only the node X and the recoverer node knows that reconfiguration has not been completed. All other nodes do not know whether the recovering node is qualified to process transactions nor which nodes need to continue with the reconfiguration process. At worst, this could lead to a primary view in which no member would be able to process transactions.

This complication is due to the fact that a member of a primary view is not necessarily an up-to-date member. In order to handle such situations, an extension of the traditional group communication abstraction is proposed in [15], the *enriched view synchrony* (EVS). Instead of ordinary views, EVS deals with so-called "enriched views", also called *e-views*. An e-view is a view with additional structural information. Sites in an e-view are grouped into non-overlapping sub-views that in turn are grouped into non-overlapping sub-view-sets. A view change then notifies about a change in the composition of the e-view (sites that appear to be reachable); such changes are performed automatically by the EVS. Additionally, EVS introduces e-view change events that notify about changes in the structure of the e-view in terms of sub-views and sub-view-sets. In contrast to view changes, e-view changes are requested by the application through dedicated primitives.

The characteristics of EVS can be summarized as follows: It maintains the structure of e-views across view changes. 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 does not consider total order and uniform delivery. However, accommodating these properties will be simple since they are orthogonal to the properties of EVS.

EVS provides simpler algorithms in regard to virtual synchrony. In particular, it provides the subsequently explained characteristics with respect to the incorporation of a site into the primary view (even though it is the DBMS who decides when to start the database transfer). When a site joins a primary view, it is done locally, i.e., it does not matter whether an operational primary view exists or not. When a recoverer site fails before terminating the data transfer to the recovering site, the remaining sites in the primary sub-view know that the recovering site is not updated, so it still is a member of their set of sub-views, but not of their sub-view. When a site enters the primary sub-view, all sites in that view know that the site now is updated and operational.

In short, with EVS, we are able to encapsulate the reconfiguration process, and the database system receives a more realistic picture of what is going on.

4.2 Recovery Protocols from Holliday

In [11], protocols called *Broadcast Writes*, *Delayed Broadcast* and *Single Broadcast* (the latter already presented in [1]) for recovery and replication are discussed.

According to the classification in [18] as discussed in section 2, these are update-anywhere and non-voting protocols. Concurrency control is performed by the DBMS with strict two phase locking (*Strict 2PL*). These protocols use a GCS providing virtual synchrony. Virtual synchrony is used to ensure that messages are delivered in the same view in which they were broadcast and that two sites that pass to a new view have delivered the same set of messages in the previous view. These protocols use total order multicast primitives as provided by the GCS for controlling transactions. The explicitly stated objective of these recovery protocols is to minimize system downtime and disruption caused by failures.

4.2.1 Single Broadcast Recovery

When a site fails, the multicast subsystem detects the failure and the membership protocol creates a new view from which the failed site is excluded. The operational sites will receive a view change message. If a commit request message for a transaction T was delivered in the previous view, then it has been delivered to all sites that comprised that view and the transaction was committed or aborted by all the sites in the view. When the failed site that was excluded in the following view recovers locally, it will have obtained the effects of T and all transactions that committed in the view to which it belonged, but not later effects of any later view.

Some GCSs with virtual synchrony provide recovery mechanisms that log delivered messages, so that when a site recovers, missed messages can be executed at this site. That can be used when the Single Broadcast replication protocol is employed. If the GCS does not provide for global recovery, some sites can be assigned to be *loggers* for update messages. Note that messages can be logged intelligently. For example, none of the messages from aborted transactions need to be logged. Thus, the logger only stores view changes and operations of committed transactions. Also note that this change log is different from the recovery log maintained by the DBMS at each site and is used for *local* recovery. When a view change is indicated, the logger makes an entry in the change log, recording membership changes. Also delivered broadcast messages with transactional updates are added to the change log. If the transaction has read obsolete data, the corresponding entry is erased and the transaction aborted. Otherwise, the transaction is executed and committed.

The change log is used as follows. When the communication system detects a membership change and one or more sites are added to the view, no update transaction messages are delivered to it, or to any other site, until the new site has exchanged messages with one logger and the logger indicates that recovery is complete. The logger will then see a view change and a request of the new site to be updated, and will look for the last view in which the site to recover was present. If the site has been absent for a long time and the logger does not have registry of it or is a newly incorporated site, the full database must be transferred. Otherwise, the transactions that were committed after the last view in which the recovering site was a member, are sent to the site in their commit order.

Here, the transfer mode clearly is either FT (cf. section 4) or, based on the log, incremental. Concurrency control during recovery is not needed because no transaction is processed until the recovery is complete. Recovery is centralized in a site that also acts as logger. The advantage is that clear decision criteria can be applied for determining whether full database transfers are really necessary and when they can be avoided by sending only the messages lost since the last view to which the failed site belonged. The disadvantages are that no transaction is processed until recovery is completed. Moreover, data versioning is required so that write and commit messages can take notice of stale data reads, in which case the transaction is aborted.

4.2.2 Delayed Broadcast Recovery

The *delayed broadcast* replication protocol decouples the writeset broadcast from the commit broadcast for any transaction. This behavior raises some problems when recovery is being considered. It might happen that the recovering site was able to deliver the writeset for a particular transaction, but not its commit or rollback message. So, that writeset was lost when the site failed and should be retransmitted now by the recoverer site if its commit message was delivered whilst the recovering site was crashed; i.e., messages delivered in a view where the recovering site was up and running might have to be remembered and resent by the recoverer.

Two possible solutions for the problems caused by the writeset-commit decoupling are presented:

1. *Log Update Method*. The loggers must examine their logs or the state of the database to determine if there exists on progress transactions in the sites without failure. If there are, the logger should mark these transactions so that when the commit or abort message is delivered, if the commit was successful, the Logger will find the record containing the writes for that transaction and copy it to the view change record. So when a previously failed site rejoins to the group, the logger begins with the execution of the writeset of the transactions that were in progress when the site failed, following with the operations of the transactions that were originated and committed while the site was failed. The commit order is the same for all non-aborted transactions. The operations of the aborted transactions are

not included in the log since their effects are undone in the sites without failure.

The transfer model for the database update used here is log-based, during the recovery a pessimistic concurrency control is used with a single manager, the recovery work distribution is centralized in a unique site. This protocol has the advantage that it does not need the data versioning used in the single broadcast protocol. The flow of messages is executed in the recovering sites in the original order, recreating with this the same conditions than in the non-failed sites. As a disadvantage we have that the loggers must maintain the logs of previous views whether or not a site fails in case of there were write messages from the terminated transactions in the those views. Additional work is done by the sites that behave as loggers.

2. *Augmented Broadcast Method*. This second method gives additional process to the sites of the on-going transactions and requires a change in the recovery lock manager algorithm. If a site S_j has any transaction in course when a new view is installed (assuming that such a view change implies that a replica has rejoined the system), it modifies the commit protocol in a way that the writesets are included in the commit messages for all transactions that broadcast their writesets in a previous view; the sites that have been operating through the change of view will ignore the writesets and will directly process the commit messages. The sites that are loggers will log the augmented message. This extension is only needed by on-going transactions; i.e., not for those that are started once the recovery process is finished.

   Similar to the previous method, the transfer model for the database update used here is log-based, during the recovery a pessimistic concurrency control is used with a unique manager, the recovery work distribution is centralized. The advantages of this protocol are that data versioning is not needed, the messages are executed in the recovering sites in the original order, recreating with this the same conditions than in the non-failed sites as the previous protocol, but tries to avoid the overload in the Loggers by distributing it towards the sites that have on-going transactions. As disadvantages we have that additional work is done by sites of transactions in course and requires a change to the recovery lock manager algorithm: the write requests are included in the commit request.

4.2.3 Broadcast Writes Recovery

When transactions are long, the Broadcast Writes protocol has a clear advantage over replica update protocols that do not use multicast. We can assume that when a view change occurs, there will be many on-going transactions and it is better not to abort all of them at each view change. Due to this, using database sites as Loggers instead of relying on the recovery mechanism provided by the multicast system could be of significant benefit. The Augmented Broadcast global recovery method presented for the Delayed Broadcast protocol could be used for Broadcast Writes. In Augmented Broadcast, only the final broadcast for a transaction, the commit request, is affected by the need to augment it with the writeset. The method then works as it does for Delayed Broadcast. When the Log Update method is used with Broadcast Writes, the Logger must be careful to remove messages from the log for a transaction that is aborted for any reason. In the case of Delayed Broadcast, only transactions that were aborted at the time of termination request had to be removed from the log. However, with the Broadcast Writes protocol, transactions can be aborted by sites because of deadlocks. If the write requests of two or more transactions cause a deadlock, all operational sites will abort one of the transactions (and the same transaction is aborted at each site). The writes of the aborted transaction are not included in the update portion of the view change record. However, the last write of the transaction to be aborted could be logged and replayed to the recovering site.

In these two last recovery protocols, the update transfer mode is log-based. During recovery, pessimistic concurrency control with a unique manager based on 2PL is used; the distribution of recovery tasks is centralized. The advantages of these protocols are that they are capable of supporting the most general transaction types in a distributed database, without the need of data versioning. Moreover, the second protocol tries to balance the work among *loggers* and the other sites. The disadvantages of the second protocol are the same as for *Delayed Broadcast*, i.e., additional work is burdened upon on-going transaction processing sites, and a change of the recovery lock manager algorithm is needed: write requests are included in the commit request message so that this information is entered into the log. The disadvantage for the case of *Log Update Method* is that loggers must take care of clearing messages from the log for a transaction that is aborted for whatever reason and not only those with an explicit abort message or whose commit message is rejected.

4.3 Parallel Recovery from Jiménez, Patiño and Alonso

The proposal for doing the recovery task in a parallel way exposed in [13] is based on a model that consists in a set of database replicas in an asynchronous system. This model is extended with a failure detector. Sites interchange messages through a reliable channel, and no Byzantine failures are considered.

The system is structured in two layers. The first layer has the replication middleware and relies on a GCS. In this middleware the replication and recovery protocols are implemented. Its GCS provides membership service, reliable multicast and the notion of view. The second layer contains the data being replicated, it is assumed that the data is divided into disjoint partitions (or classes) and each one has a master site. The transactions that access data in a given partition should be local to the partition master site; i.e., if a transaction requests processing in a non-master site, this site forwards the request to the partition master site. A site executes only its local transactions; for remote transactions only installs their updates. The transactional system supports strict two phase locking.

The aim of the recovery protocol is to identify the missed transactions in a failed but now recovering site, obtaining these transactions from an active site and applying them in such recovering site. Recovery is made on a partition basis, i.e., each partition is recovered independently from other ones. A partition can be in one of the next states: (a) *online*: those partitions that are working normally; (b) *crashed*: when its master site has failed; (c) *recovering*: when such master site is restarted; (d) *pre-online*, when the recovering has completed its first steps but is not yet *online*.

A partition can be elected as *recoverer* and it changes to that state. The recovery procedure terminates with a forwarding phase during which the partition is in *forwarding* state. A partition can not process transactions from clients during the crashed, recovering or pre-online states, in which only can process transactions associated with the recovery. When a recovering site joins to a working group a view change is performed. As part of this procedure, the recovering site indicates the *log sequence number* (LSN) of the last committed transaction. Once a site is elected as recoverer site, it sends the recovery information to the recovering site. The recoverer site is able to process transactions even in the recovery process.

This protocol can be extended to support parallel recovery in several sites. Thus, the same partition master site is able to multicast missed transactions to multiple recovering sites (if more than one site are restarted at once). Additionally, when a site is recovering, its missed transactions are sent to it from all the master sites that have any transaction to be recovered. So, recovery parallelism is improved from both of these sides.

This recovery protocol assumes a replication protocol based on a primary-copy server architecture, with constant server interaction, non-voting transaction termination, and eager update. The recovery protocol is log-based with a pessimistic concurrency control with a single manager, and with recovery work distribution.

The main advantages of this protocol are that when the recovery task is performed in a parallel form supposes an optimization in the transfer time and load balancing. The single period in which the transactions are not processed is during a view change, when the sites are blocked. This protocol presents the disadvantages of processing the transactions solely in the partition master site, and when failure periods are long the information to transfer may be abundant.

4.4 The COLUP Recovery Protocol

In [12] a configurable eager/lazy replication protocol with a lazy recovery protocol is proposed. This replication protocol, called *Cautious Optimistic Lazy Update Protocol* (COLUP), uses the concept of node role, given special importance to a node where a particular object is created. Such node is referred to as the *owner* for all objects created by its local applications. This owner node will be consulted during the voting phase performed at commit time. In this way the owner is the manager for the object accesses and is responsible for coordinating the propagation of the last versions of the object. An identifier for the owner node is included in the identifiers of the objects. For any object, a set of nodes will maintain synchronous copies; i.e., consistent replicas of its state. The other nodes that have a replica of the object constitute the set of asynchronous nodes. In these nodes the updates to the object will be eventually received, once the updates have been committed in synchronous replicas.

Conflicts between transactions are solved in an optimistic way, using object versions and reviewing them during the commit phase. As a result, a transaction is aborted if it has read obsolete values that were updated by other concurrently committed transactions. Thus, access to the objects is allowed with no need of locks. A disadvantage of the lazy updates is that the probability of aborting transactions gets increased. It is necessary then to establish a threshold for the probability of aborting a transaction accessing obsolete object values. Thus, when a transaction tries to access an object, this probability is calculated and compared with the established threshold. As result, the algorithm obtains an updated version for the objects that might be obsolete. Using a high threshold the number of requested updates is minimized, and the number of transactions executed in the system is increased since the used resources for update propagation are decreased. But this may cause an increase in the number of aborted transactions because the number of objects with obsolete values is also increased. This can degrade the system productivity, so it is convenient the use of an algorithm to dynamically adapt the threshold value to an optimal value.

The recovery protocol considers the existence of a membership monitor that is executed on each node. The monitor observes a preconfigured set of nodes, and notifies its local node about the changes in this set. When the membership monitor detects a failed node a notification is sent to each node that remains in the system. This causes an update in the "list of alive nodes". During the execution of a transaction a number of messages must be sent to the different owners of the objects. If a message must be sent to a failed owner, then it will be redirected to a new owner of that object. Each node sends a message with the *previous grants* conceded to the objects by the previous owner. The new owner can process the requests as if it was the original owner node of the object.

When an original owner node recovers from a failure, every alive node is notified by its membership monitor. Then, further messages must be sent to the original owner node. In addition, the recovering node sends a message to the node that managed its owned objects and with this, synchronizes the activity in both nodes. A recovering node may receive requests for objects that were updated during the failure interval. In order to handle this situation, the recovering node must consider each object of which it is owner like an asynchronous replica until it is updated by a synchronous replica. To ensure that a recovering node achieves a correct state for its owned objects, an asynchronous low priority process is executed. This process sends an update request for all non-synchronized objects including the new objects created during the failure period.

The replication protocol is eager update-anywhere, with constant server interaction and voting transaction termination. Recovery uses a version-based transfer model. The concurrency control is optimistic with a distributed manager and multiversioned.

As advantages offered by the recovery protocol we can find that the recovery task is totally supported by the hybrid replication protocol, so the recovery is part of the basic algorithm and it is not necessary to add more code. Another advantage is that the updates are deferred until the recovering node accesses obsolete data. With object versioning is not necessary the use of locks and the rate of aborted transactions is reduced. As disadvantage we found that the time of COLUP for processing transactions is usually greater than in pure lazy replication protocols.

4.5 CLOB: Short-Term Failure Recovery

CLOB (Configurable LOgging for Broadcast protocols) described in [6] is defined as a framework for reliable broadcast protocols that are used as a basis for database replication. Its aim is to manage the logging of missed messages in the broadcast protocol core, providing with this automatic recovery for short-term failures, but discarding the log and notifying the database replication protocol modules in case of long-term outages. This kind of support can be easily combined with version-based recovery protocols. To this end, once a failure is detected the database replication protocol must follow its traditional version-based management for recovery purposes, but it will be discarded if the replica is able to rejoin the system soon. In this case, CLOB automatically propagates the missed update messages to the recovering replica, which receives and applies them avoiding any additional waiting time both in the source and destination replicas. On the other hand, if the outage period exceeds a given threshold, the reliable broadcast service will notify the replication protocol about that, discarding the message logs maintained by CLOB and delegating the recovery management to the upper-layer components.

The replication protocol is eager update-anywhere, with constant server interaction but would have to consider some additional parameters to decide when the logged messages can be eliminated. This protocol applies voting transaction termination. The basic support for the recovery based on logs will be identical if the transaction termination is voting or non-voting. During the recovery, the transfer of the state of the database is version-based in long-term failures, but is log-based in short-term failures.

As advantages we can mention that it combines version-based and log-based transfer of information for the recovery depending on which is more advisable, without restricting to a single model of transfer and being able to take advantage of each one in its case. A minimum blocking time for replicas that participate in the recovery is also obtained when the log-based recovery is used. As disadvantages, it is necessary to maintain the related information to both recovery methods, and the transaction service time is increased even with no failures because all messages must be saved in persistent storage.

4.6 The FOBr Recovery Protocol

The recovery protocol explained in [7] FOBr, is designed as a complement for the replication protocol FOB (Full Object Broadcast), which is an optimistic eager update-anywhere protocol and makes use of a GCS [8] membership service. In this protocol the concept of replica role is used and it can be:

1. Owner node: Initially it is the node where the object was created; this is what we call physical ownership. However, the node where a set of objects was created might have crashed and the ownership migrates (logical ownership). This owner node is the manager of *access confirmation requests* (ACR) for that object.

2. Synchronous nodes: These nodes did not create the object but are considered up-to-date replicas of it. They provide us with fault tolerance.

Several transactions can be grouped in a session. Since an ACR management is used, the session identifiers (SIDs) include information about the node identifier where it was initiated. The objects are identified in a similar way to the sessions.

Objects are identified similarly as Sessions with object identifiers (OIDs). These OIDs hold several information, including the owner node, that identify them univocally through all the nodes. Besides the OID, the consistency protocol may need (FOB does) to maintain extra information associated to each OID such as version numbers, timestamps,...

This metadata information will also need to be transferred when a recovering node receives its updated information. When the user initiates a commit, the protocol performs several operations before it is effectively applied into the database:

1. It collects the transaction writeset and groups its OIDs by their owner node.

2. An ACR is sent to each owner node of these writeset objects. The owner nodes decide then whether to grant or revoke the access to these OIDs. This sending is performed sequentially and in ascending order of node identifiers in order to avoid multiple abortions.

3. The node receives the ACR responses and:

    - If any ACR is revoked, the transaction must abort. If any of the other ACRs was granted, a message must be sent to that node in order to release the grants.
    - If they are all granted, the transaction is propagated with a reliable broadcast and when delivered, it is committed in all the nodes. When a node receives this broadcast: (i) It aborts other locally conflicting sessions that are still in early phases of operation; (ii) It applies the changes into the database; and, (iii) It releases the ACRs granted to the finished transaction.

The recovery protocol has two phases:

1. Collection phase: It includes all the events that happen since the moment a node fails until the moment it joins to the system. Two steps are taken:

    - The remaining alive nodes decide in a deterministic way, which ones of them inherit the ownership of the faulty node objects.
    - A structure is created in each alive node in order to hold the *OIDs* of all objects that will be updated while these nodes are not present. The structure needs to be stored persistently in order to allow recovery in a total system failure. In this structure a recovery list is also stored, and it saves the updated *OIDs* that any site lost during a view change.

2. Recovery phase: it includes all the steps followed by the nodes of the system when a failed node initiates operations again. In the recovery phase we distinguished two roles for the participant nodes: the recovery node that is the node that is trying to join the system and needs to update its database, and the previously active node that is the node that has the information to help the recovering nodes to join the system.

    The recovery phase begins when the previously-active nodes receive a notification, by the membership service, about the recovery of some previously considered faulty node. This notification has two parameters, the recovering nodes list and the actual view number. Then, the following steps are taken:

    - The previously-active nodes build a `JOIN_UPDATE` message to update the currently owned objects that they know the recovering nodes have missed. This `JOIN_UPDATE` message is built following this procedure: (i) The recovery list is checked to obtain the set of updated *OIDs* that the node currently owns. (ii) The set of *OIDs'* states is retrieved from the database and it is included in the message in order to update the recovering node database. (iii) The set of missed *OIDs* and network views is also included, because the recovering node needs to hold recovery information until the system is complete. However, this information is not transferred if the currently recovering node is the latest one; i.e., no other faulty node exists when it has finished its recovery.
    - The recovering node waits until it has received the `JOIN_UPDATE` message from all previously-active nodes. As soon as a `JOIN_UPDATE` message arrives, the recovery list is reconstructed with the information provided by the message. The recovering node will have created a transaction to apply all the updates that it had to receive. Once committed, the recovering node sends a **MERGED** message to all the previously-active nodes and waits for a **NO ACT**(**NO ACT** stands for "node active") response.
    - When the previously-active nodes receive the **MERGED** message they know that the recovering node has applied all the remaining updates. If the **MERGED** receiver was not the inheritor of the recovering node objects, it simply assumes

that the recovering node has recovered the ownership. If the receiver is the inheritor, it has to migrate this ownership packing the *ACR* granted locks into a **NO ACT** message and send it to the recovering node.

- Finally, when the recovering node receives the NO ACT message, will be able to manage its objects and the recovery is completed.

According to this, the replication protocol is eager update-anywhere, with constant server interaction and with voting transaction termination. The recovery protocol has a version-based transfer model, the concurrency control is optimistic with a distributed manager and with multiversion. The recovery work is distributed.

The advantages offered by the recovery protocol are that minimizes the amount of data to transfer, balances the recovery work, and allows the execution of transactions during recovery time. It has low space requirements. Its disadvantages are: (a) it complements a replication protocol (FOB) with a non-standardized isolation level; (b) for each transaction that commits, we must explore its writeset (and save the *OIDs* contained in it) if there was any failed node.

4.7 Recovery Protocols from Armendáriz

In [2] three eager update replication protocols are considered, and a recovery protocol that can be applied to all of them is proposed. The first replication protocol called Basic Replication Protocol (BRP) is based on the optimistic 2PL (O2PL). As a result of the addition of improvements and variations to protocol BRP, is presented the second protocol called Enhanced Replication Protocol (ERP). This replication protocol reduces response times and transaction abortion rates by removing the Two Phase Commit (2PC) rule and the use of queues. Finally, the third replication protocol, named Total Order Replication Protocol with Enhancements (TORPE), that makes use of the total order multicast primitive provided by the GCS to ordering the transactions executed by the system. The main idea for the recovery proposed in [2] once a node re-joins the network after failure, an alive recoverer node is appointed. It informs the joining node about the updates it has missed during its failure. Thus a dynamic database partition (hereafter DB-partition) of missed data items, grouped by missed views, is established, in recovering and recoverer nodes, merely by some standard SQL statements. The recoverer will hold each DB-partition as long as the data transfer of that DB-partition is going on. Previously alive nodes may continue to access data belonging to the DB-partition. Once the DB-partitions are set in the recovering node, it will start processing transactions, which are, however, blocked when trying to access a DB-partition. Once the partitions are set in the recoverer, it continues processing local and remote transactions as before. It will only block for update operations over the DB-partition.

The three replication protocol are eager update-anywhere with constant server interaction. BRP has voting transaction termination, whilst ERP and TORPE have non-voting termination. The recovery protocol is version-based, the concurrency control during the recovery is optimistic with a distributed manager and with multiversion. The recovery work is distributed.

The main advantages are that recovery is distributed, the DB-partition in a recoverer site can be released even when the recovery process is not concluded, and that transactions can be accepted and committed in recoverer sites if they do not interfere with the DB-partitions being recovered. The disadvantage is that if DB-partitions are defined on the basis of each view modified items, an object may be transferred several times, to avoid this we must "compact" the DB-partitioning.

5 Conclusion

Once this set of protocols has been surveyed, as a concluding remark we advise to consider recovery algorithms that use version-based management and that distribute the recovery work among available sites to balance the workload during the recovery process. Very few replicated database recovery systems are capable to combine these techniques to reduce recovery times. When it has been partially possible (as in [12, 13]), it was because replication protocols had some special characteristic (the use of a primary copy schema in [13], that reduces flexibility and compromises fault tolerance; and the use of lazy updates in [12], that compromises consistency). The work presented in [2] could be a good exception, but it has not presented performance measurements that confirm its good theoretical performance. This analysis will be used as a basis for designing new recovery protocols, trying to combine the advantages of all surveyed protocols.

References

[1] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. *LNCS*, 1300:496–503, 1997.

[2] J. E. Armendáriz. *Design and Implementation of Database Replication Protocols in the MADIS Architecture*. PhD thesis, Universidad Pública de Navarra, Pamplona Spain, Febrero 2006.

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

[4] K. P. Birman. *Reliable Distributed Systems Technologies, Web Services, and Applications*. Springer, 2005.

Table 1. Classification of replication and recovery protocols

		Replication				Recovery				
		A	I	T	U	TM	O	M	V	W
[15]	Full DB Transfer	U	C	N	E	FT	P	S	N	C
	Version number	U	C	N	E	IT	P	S	N	C
	Restrict set of objs.	U	C	N	E	IT	P	S	N	C
	Log Filter	U	C	N	E	IT	P	S	Y	C
	Lazy data Transf.	U	C	N	L	IT	P	S	N	C
[11]	Bcast writes Log upd.	U	L	N	E	LR	P	S	N	C
	Bcast writes Augm. bcast	U	L	N	E	LR	P	S	N	C
	Delayed bcast Log upd.	U	C	N	E	LR	P	S	N	C
	Delayed bcast Augm. bcast	U	C	N	E	LR	P	S	N	C
	Single bcast	U	C	N	E	1	P	S	N	C
[13]		P	C	N	E	LR	P	S	N	D
[12]		U	C	V	2	IT	O	M	Y	D
[6]	CLOB	U	C	N	E	3	O	M	Y	C
[7]	FOBr	U	C	V	E	IT	O	M	Y	D
[2]	BRP	U	C	V	E	IT	O	M	Y	D
	ERP	U	C	N	E	IT	O	M	Y	D
	TORPE	U	C	N	E	IT	O	M	Y	D

1. Considers full database transfer, is needed if a site is new or if it is not in the record of views in the logger.
2. It is configurable, and may be hybrid.
3. IT for long-term failures, and LR for short-term ones.

[5] M. J. Carey and M. Livny. Conflict detection tradeoffs for replicated data. *ACM Trans. Database Syst.*, 16(4):703–746, 1991.

[6] F. Castro, J. Esparza, M. I. Ruiz, L. Irún, H. Decker, and F. D. Muñoz. Clob: Communication support for efficient replicated database recovery. In *PDP*, pages 314–321, 2005.

[7] F. Castro, L. Irún, F. García, and F. D. Muñoz. Fobr: A version-based recovery protocol for replicated databases. In *PDP*, pages 306–313, 2005.

[8] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. In *ACM Computing Surveys 33(4)*, pages 1–43, 2001.

[9] S. Elnikety, F. Pedone, and W. Zwaenopoel. Database replication using generalized snapshot isolation. In *SRDS*, 2005.

[10] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD Conference*, pages 173–182, 1996.

[11] J. Holliday. Replicated database recovery using multicast communication. In *NCA*. IEEE-CS Press, 2001.

[12] L. Irún, F. Castro, F. García, A. Calero, and F. Muñoz. Lazy recovery in a hybrid database replication protocol. In *XII Jornadas de Concurrencia y Sistemas Distribuidos*, 2004.

[13] R. Jiménez, M. Patiño, and G. Alonso. An algorithm for non-intrusive, parallel recovery of replicated data and its correctness. In *SRDS*, pages 150–159. IEEE-CS Press, 2002.

[14] B. Kemme. *Database Replication for Clusters of Workstations (ETH Nr. 13864)*. PhD thesis, Swiss Federal Institute of Technology, Zurich,Switzerland, 2000.

[15] B. Kemme, A. Bartoli, and Ö. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *DSN*, pages 117–130. IEEE-CS Press, 2001.

[16] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD Conference*, 2005.

[17] A. Ricciardi, A. Schiper, and K. Birman. Understanding partitions and the 'no partition' assumption. In *4th IEEE Workshop on future trends in Distributed Computing Systems*, pages 354–360, Lisbon, Portugal, Sept. 1993. IEEE-CS.

[18] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *SRDS*, pages 206–215. IEEE-CS Press, 2000.

[19] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replcation in databases and distributed systems. In *ICDCS*, pages 464–474, 2000.