CLOB: Communication Support for Efficient Replicated Database Recovery *

F. Castro-Company J. Esparza-Peidro M. I. Ruiz-Fuertes L. Irún-Briz H. Decker
F. D. Muñoz-Escoí
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
46022 Valencia, SPAIN
{fcastro,jesparza,miruifue,lirun,hendrik,fmunyoz}@iti.upv.es

Abstract

Replication protocols using an eager update propagation strategy commonly need a reliable broadcast service; i.e., a broadcast primitive with atomic delivery and, in some cases, also with total order. This communication service provides some appropriate features for the recovery tasks, although in some cases this will lead to partial blocking of the replica taken as the source in the updating process. CLOB is a framework for reliable broadcast protocols that log the missed update messages in case of failure, being able to automatically resend these updates when the faulty destinations recover. This behaviour is easily configurable and allows an efficient recovery mechanism in case of short-term failures, which can be combined with other version-based recovery protocols in order to manage long-term outages.

1. Introduction

Atomic broadcast protocols [7] have been widely used to provide communication support for database replication protocols based on the eager update propagation technique [6]. They lead to efficient solutions for guaranteeing one-copy serializability, because the transaction commit phase can be done without requiring a distributed voting phase [15], and with constant interaction; i.e., the write-sets of each transaction may be propagated at commit time, but they do not need to be transmitted before. As a result, the communication costs can be highly reduced with these techniques, at least when no failures arise.

When some replicas of the database have failed, the remaining active ones have to maintain some additional data in order to allow their future recovery. According to [12], one possible solution consists in the use of virtual synchrony [2], needing a state transfer during the view change of the replica group; i.e., in the interval that starts when the faulty replica requests its re-inclusion in the group and that finishes when it is considered a valid destination for new broadcasts. However, the support traditionally given in systems with virtual synchrony has needed a complete state transfer in this recovery phase. This principle cannot be easily migrated to the database field, since a complete database transfer needs too much effort, both in time and bandwidth terms. As a result, several database replication protocols log all missed updates during the failure period, in order to transmit them during the recovery phase (e.g., [10, 14]).

Log-based (synonymously, "missed-udpate") propagation is the best solution for short-term failures, but there are better approaches when the failure period is long. Again, in [12] several version-based solutions are described. Their aim is to reduce the amount of information to be logged and transferred. To do so, if a data item has been updated several times whilst a replica was faulty, only its latest version will be transferred. Thus, the active replicas need to log only the identifiers of the items being updated, and their current versions, but not their actual state. This minimizes the amount of log storage needed by this kind of solutions. On the other hand, this leads to lock in read mode –in the source replica– all data items to be propagated to the recovering replica, needing also additional time to read these items' states and to build the appropriate update message with them. However, this lock requirement can be removed in a DBMS with multi-version concurrency control, like PostgreSQL, and the extra time needed for reading the items' states is perfectly balanced with the smaller amount of data being transferred.

CLOB (*Configurable LOgging for Broadcast protocols*) is a framework for reliable broadcast protocols that are used as a basis for database replication protocols. Its aim is to manage the logging of missed messages in the broadcast protocol core, providing thus automatical recovery in short-

* This work has been partially supported by the Spanish MCYT grant TIC2003-09420-C02-01.

term failures, but discarding the log and notifying accordingly 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.

This framework is thoroughly described in the following sections. To this end, the rest of the paper is structured as follows. Section 2 describes the system model. Section 3 shows the overall architecture where CLOB may be included. Later, section 4 gives the main characteristics that CLOB introduces in the broadcast protocols pluggable in this framework. Section 5 provides some performance results. Finally, section 6 presents some related work and section 7 concludes the paper.

2. System Model

Database replicas are placed each one in a different node of a partially synchronous distributed system, where clocks are not synchronised but message transmission time can be bounded. The database is fully replicated in each node, i.e., each replica has a complete copy of the whole database.

A group membership service is assumed. The set of possible system nodes is known in advance, and a *primary partition* model is used in case of network partitions; i.e., only the active subgroup with a majority of the preconfigured nodes, if any, is allowed to continue in case of a network partition. This characteristic will prevent communication when the system is started, until at least a majority of the system nodes are able to exchange messages among them. Thus, the needs for message logging are reduced in these initial phases. Moreover, the consistency of the database is trivially ensured in case of network partitions, since only the primary subgroup is able to continue, and it will be the only one who creates the log of missed updates, resending it when the subgroups are joined.

A *partial-amnesia crash* failure model [4] is assumed for processes, whilst links may produce *omission* failures. This failure model is more realistic than the *crash* or *fail-stop* failure models [7], where a node is assumed to stop once it has failed, forcing to recover with another node identifier and without any previous state. A strict adoption of such models implies a complete database transfer in each recovery and that action will be too costly.

The typical user of the CLOB support is a database replication system. Such a system has to use a *constant interaction* [15] approach in order to propagate its updates among replicas; i.e., the replica that has initially processed a given transaction propagates its updates to the rest of replicas only once. A *linear interaction* policy (i.e., one that propagates the updates each time the transaction updates the database, instead of grouping them in order to multicast a unique message) needs additional parameters to decide when the logged messages can be removed.

Note that in the *linear interaction* case all messages have to be logged, even if all the nodes are active, removing them when the transactions associated to them have been completed (either committing or aborting). When a node fails, all the transactions already started in the other nodes have to be tagged, and their currently logged messages have to be maintained in the log in order to provide a complete sequence of messages for each transaction needed in the recovery process of such a faulty node. The algorithm described in this paper does not consider this case, and the log only maintains the missed updates. However, this does not work in a linear interaction system, because all the transactions are propagated to all replicas when they start, and in the faulty replica, all initiated transactions are removed from the system when it restarts, aborting them. As a result, with the scheme outlined in the following sections, CLOB only sends the missed updates, and part of them correspond to some transactions that have been aborted due to the failure. For these transactions, we need to send all their messages, including also those that were sent to the faulty node immediately before its failure. These extensions are not described in this paper, but have to be applied if we use a system based on *linear interaction*.

3. Architecture Overview

The CLOB support needs to be on top of a membership service and below the database replication protocol, which is the main user of the CLOB services, as it is shown in figure 1. The membership service needs to use an unreliable transport; for instance, one based on UDP. Each time the system membership changes, either due to a node join, a node failure or a network partition, its notifications have to be delivered to both the CLOB broadcast protocol and to the database replication protocol placed on top of it, and in such order.

These four layers can be easily integrated in a more ellaborated architecture, as the one used in GlobData [9], where all these components build up the COPLA manager, placed on top of the database management layer called UDS [1]. Another sample could be the RJDBC middleware [5]. In

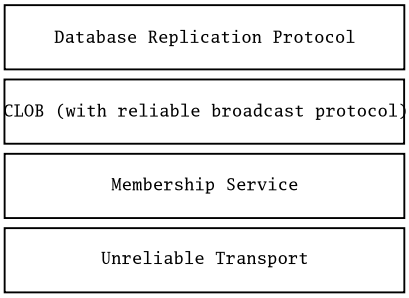

Figure 1. CLOB architecture.

this latter case, these layers have been integrated as a JDBC-compliant driver with database replication support which runs on top of the native JDBC driver for the DBMS being used.

On top of the database replication protocol several additional layers could be placed. For instance, one representing the client applications that use the database. Additionally, each node will be able to access a local database replica. This is done using a common DBMS API to be called by the database replication protocol.

4. CLOB Description

A broadcast protocol needs to comply with these requirements in order to be included in the CLOB layer:

- Logging of missed messages.
- Logging of received messages.
- Configurable removal of missed messages.

Let us see in the following sections why these requirements are needed.

4.1. Logging of Missed Messages

If a system node is placed in a non-partitioned group, or in a subgroup with a majority of the preconfigured nodes it will log all the messages it delivers if at least one of the preconfigured nodes has failed. These logged messages will be sent to the unavailable nodes when they rejoin the system, and will be automatically removed when all the configured nodes were available and once they would have received such messages.

The broadcast protocol being used has to be at least a reliable one with FIFO order [7], as the one used in the FOB database replication protocol [13]. In such cases, the database replication protocol requires some additional mechanisms to decide which transactions can be committed. For instance, in the FOB case a voting phase is needed. On the other hand, when an atomic causal broadcast protocol [7] the database replication protocol may not need such a voting phase [14]. However, our basic log-based recovery support will be identical in both cases.

In order to achieve an easy management, the log is divided in several segments. Each segment stores all messages that were sent and delivered between two consecutive view change events, and it has a header that includes this information:

- A segment ID composed by the identifier of the new group view being installed.
- A header core with the identifiers of all the nodes that were faulty when the segment started.

Thus, when a node rejoins the system, the following algorithm is used to find and transmit the appropriate sequence of messages it has missed:

1. The history of log segments is scanned until the oldest segment which has the recovering node ID in its header core is located.
2. As a result, all the active nodes know which messages are needed by the recovering node –all messages in the sequence of log segments that starts in the one located in the previous step– and these messages are packed in a *recover message* and transmitted using point-to-point communication to such recovering node.

   To this end, a given criterion is used by the previously available nodes in order to decide which one of them will act as source of the data transfer. For instance, the node with greatest identifier lower than the recovering one.
3. Once this message transfer has concluded, the recovering node will broadcast an acknowledgement message.
4. Once the previously active nodes receive this acknowledgement message, they start to remove the recovering node ID from the headers of all log segments transmitted to it.
5. If in the previous step the headers of some log segments become empty, such segments are removed from the log.

The previous algorithm allows an easy finding of the set of messages to be transmitted, and it also provides a simple log removal criterion in its last step. Unfortunately, it is not fault-tolerant: what does it happen when one of the previously active nodes fails in or before step 3? At a glance, its log headers will not be updated, so its log of sent messages will not be accurate and becomes useless once it recovers.

Thus, the following extensions are needed to deal with failures in this recovery algorithm:

- The log of sent messages is filtered in a recovering node **before** it requests to be accepted again in the system; i.e., before its membership monitor sends its first message to the rest of system nodes.

  In this filtering process, the header cores are removed in all the log segments maintained by the recovering node.

- Step 2 of the previous algorithm does not consist in the transmission of only the sequence of messages missed by the recovering node, but also of the headers of all the older log segments (initially not intended for the recovering node) where its node ID did not appear, if any. Note that we only need to transmit the headers, since the message logs themselves are already in the logs of the recovering node, and have been filtered in the step described above.

  These additional headers are packed separately and placed in another field of the *recover message*. They will allow that the recovered node could play a source role in future recoveries of other currently faulty nodes.

- Note that the previous item leads to include in step 2 all the segment headers. As a result, the step 4 is also locally run in the recovering node, filtering thus all the received segment headers which contained its own node ID.

- It is also worth noting that if any of the headers of the recovering node remains empty once the transmitted headers of the *recover message* have been applied, such empty headers are removed and their associated log segments, too.

The latest item in the previous list avoids some problems in case of failure during the recovery process. For instance, let us assume that a node A is active in a given system view, with its log segments correctly maintained. Then, a node B rejoins the system, producing a view change event. As a result, all the log segments missed by B have to be transmitted to it. When B is terminating its recovery process, it broadcasts an acknowledgement message to all system nodes. If B were the unique member of some of the segment headers, these headers and their associated log segments would have been removed from the log once this acknowledgement was received. But, let us assume that A failed when it decided to remove such segments, but without updating their headers nor starting such removal. When A finally recovers, none of these segments nor their headers will be transmitted to A, and it has to remove them from the log. So, this sample justifies that all filtered and cleaned headers that have not been retransmitted in the *recover message* must be removed from the log.

4.2. Logging of Received Messages

All the messages broadcast and received by the system nodes are locally logged in their destination nodes until the target application –in our case, the database replication protocol that applies the received messages in its local database replica– is able to acknowledge that these messages have been processed. If the node fails before the target application has processed such message, when the node recovers the same message is locally delivered again, without needing any communication with its sender.

Note also that in the log described in section 4.1, CLOB also logs those messages sent and delivered –indeed, if some node is faulty and both logs are maintained, both of them share their messages; i.e., messages are not stored twice–, but the rationale of both logs is different. The *missed log* stores all messages delivered by each node in a view where at least one of the preconfigured system nodes is unavailable. This is needed to build up the source of messages to be transmitted to those unavailable replicas when they recover. On the other hand, the *receiving log* always stores all delivered messages, independently of the current set of available nodes, and its function consists in guaranteeing that all delivered messages have been finally processed.

The aim of this second log is to avoid message losses. Note that in a system based on virtual synchrony, once a node failure is detected a view change event is started and completed. Thus, all system nodes may know which was the last message delivered in the faulty node. However, delivery of a message does not always imply its processing, since the node may have failed before such a processing was completed. Traditionally this has not been a problem in systems that use virtual synchrony, since in the recovery protocol the complete state of the replicated object is transferred to the recovering replica. So, there is no problem in such cases with missed messages. These assumptions are not valid in our CLOB support, since it tries to minimize the amount of information being transferred in the recovery process.

Thus, CLOB tries to send to the recovering replica only those messages that such replica was not able to deliver. So, if a message was delivered in a replica immediately before its failure, some mechanism is needed to guarantee that such message is finally processed. To this end, a message is logged in its destination nodes as soon as it is received and tagged as deliverable, once the broadcast protocol decides on that. This latter step depends on the broadcast protocol being used.

Messages are discarded from this log as soon as the destination component acknowledges their processing. In order to do so, the CLOB broadcast protocols need an additional operation for their users that initiates the message removal from the log. These protocols need to associate an identi-

fier to each message they have delivered, and this identifier is the one needed to request the removal of such message once it has been processed.

Note also that a node may fail once it has processed a message, but before it has requested its removal. In our target scenario this can be a problem, since this may lead to processing the same message twice. To prevent this, the database replication protocol has to ensure that it knows which has been the history of updates already applied. Thus, this database replication protocol needs to add a table (named DELIV_TABLE) to the database schema where the identifier of the latest applied update message is recorded each time an update is applied. To ensure that failures do not corrupt such information, both the update resulting from a writeset processing and the increment of the DELIV_TABLE contents are made in the same transaction, ensuring the atomicity of these changes. Additionally, the update on DELIV_TABLE will be made in the latest step before requesting the transaction commit. This ensures that the associated database lock will be held a minimal time, reducing thus the probability that a transaction becomes blocked in its access to DELIV_TABLE. Using this support, when a database replica is being recovered its local CLOB module will redeliver all messages that were not removed from the receiving log, and the database replication protocol will process all messages in this set whose identifiers were greater than the one recorded in the DELIV_TABLE.

4.3. Configurable Removal of Missed Messages

A simple recovery mechanism can be implemented combining the support given by the *missed* and *receiving logs*. Indeed, no special recovery protocol is needed to manage such replica recoveries, but only a minimal extension of the database consistency protocol in order to deal correctly with message ID updating in the DELIV_TABLE as described above.

This recovery mechanism –i.e., a log-based recovery one– is not optimal for all failure scenarios, as already stated in section 1. If a node remains failed during a long period of time, there are other version-based recovery mechanisms that may reduce a lot the amount of information to be transmitted, shortening thus the recovery time, since the number of required updates in the recovering database replica is minimized.

As a result of this, the automatic recovery process outlined in previous sections is highly recommended for short-term outages, but it is not adequate for long-term ones. On the other hand, CLOB is responsible for the log-based recovery strategy, whilst a version-based one can only be supported by the database replication protocol placed on top of CLOB. Both solutions can be easily combined, using the log-based solution for short-term failures and the version-based solution for medium- or long-term ones.

Thus, CLOB uses a configurable parameter that selects which recovery strategy has to be used depending on the size of the sending log that the available nodes maintain whilst some of the database replicas remain crashed or inaccessible. This parameter may have the following values:

- **-1**: Version-based recovery is not possible. The sending logs have to be always in use, independently of their size. This value will be used by those database replication protocols that are not based on item versioning, as [5, 10].
- **0**: Log-based recovery is disabled. The sending logs will not be used. This will only happen if the broadcast protocol being used does not comply with the CLOB requirements described in section 4. However, the *receiving log* management will still be active with this configuration.
- **Positive values**: These values express the affordable size of the sending log, in KB. Whilst the sending log remains smaller than such size, log-based recovery will be automatically used. Once such size is exceeded for a given faulty node, such node will be removed from the header cores of the sending log and it will be forced to recover using a version-based approach managed by the database replication protocol itself. Note that in these cases, version-based recovery has to be started as soon as a replica fails, but it will not be used if such replica recovers soon.

Let us see what happens when positive values are used, since in such situations both recovery techniques have to be prepared in parallel, but only one of them is finally applied depending on the size of the sending log. So, we have two cases:

- **Sending log smaller than the configured size**. In this case the log-based recovery solution will be automatically applied. Once the recovery is completed, all replicas have to notify (using a CLEAR_STATE notification or message) their upper layer, stating that the recovery has been completed. When the database replication protocol receives such notification, it removes all the information kept for the recovered replica in its version-based recovery algorithm.

  In practice, this CLEAR_STATE notification is transferred to the upper layer once the step 4 has been completed in the algorithm described in section 4.1; i.e., no additional broadcast is needed, since the CLEAR_STATE notification is locally propagated –the broadcast was already done to complete the acknowledgement of step 3.

- **Sending log greater than the configured size**. When the log stored for a given node (say N_f) is greater than the configured maximum size, a VERSION_RECOVERY notification is internally propagated to the database consistency protocol, since the log has the same size in all replicas and all of them notice that its size is greater than the configured value. Once such a message has been transferred to the upper layer, the identifier of the node N_f is removed from all the log segment headers where it was stored. If any of such headers is emptied in this filtering process, such segment is removed from the log.

  The VERSION_RECOVERY message implies that the database consistency protocol has to manage the recovery, without any help from the logging mechanism of the broadcast protocol.

As a result, in both cases an additional local notification is needed to synchronise the adoption of the appropriate recovery technique, guaranteeing that the same decision is taken in all replicas. This is a minimal cost, since no additional broadcast has been needed to complete this step.

5. Performance Results

In order to check the performance gains that may introduce the CLOB algorithm, we have simulated its use as a supporting tool for the FOBr [3] version-based recovery protocol. In these tests, a database with 6000 objects and four replicas has been used. The nodes are set in a cluster with a high-speed interconnection network, so communication latency is negligible.

The tests correspond to the best case environment for FOBr; i.e., one where the transactions executed during the failure period have accessed multiple times the same objects. In such a case, FOBr only needs to transfer the latest state version of these objects, whilst a log-based solution still has to transfer all the logged messages.

During the time a node remains failed, a sequence of transactions accessing objects in a set of 15 objects is repeated multiple times. As a result, each accessed object has been updated multiple times and it needs to be transferred only once using FOBr.

Figure 2 shows three different configurations of the CLOB support for the FOBr algorithm, depending on the size of the parameter discussed in section 4.3. This parameter takes the following values in our tests:

- **-1**: It is shown as CLOB(-1) in the figure, and it corresponds to the log-based recovery solution. Remember that this value prevents the use of the version-based protocol.
- **0**: This value does not use the log-based recovery, using only the log to ensure the delivery of the received messages. The recovery tasks are completed using only the version-based protocol.
- **100K**: When the log size exceeds 100KB, the version-based recovery algorithm is used, otherwise the log-based solution is taken. In this test, this corresponds to approximately 12 missed transactions.

  Note that this value does not provide optimal results in this example, but it is difficult to find out in advance the optimal value for this parameter. This optimal value depends on the application access pattern. In this test we can see that such optimal value corresponds to the size of the log when the updates of 8 missed transactions have to be transferred, but we have used a fixed access pattern for all transactions and this will not happen in a real environment.

Thus, in figure 2, the set of accessed objects corresponds only to 15 in each one of the active nodes. As a result, in CLOB(0) (i.e., the version-based configuration) each of the source nodes need only to transfer the state of these 15 objects. As we can see, the CLOB(-1) (i.e., the log-based configuration) solution is still better than CLOB(0) if a low number of transactions has been executed whilst the recovering node was faulty.

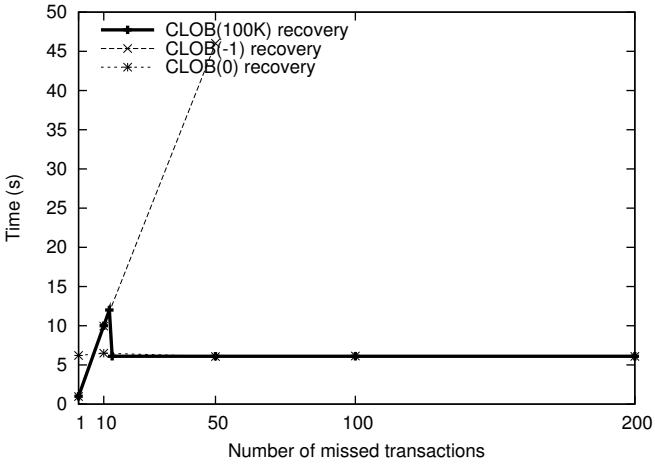

Figure 2. Recovery time of three CLOB configurations.

The CLOB(100K) configuration provides the same values than the CLOB(-1) one if the size of the log does not arrive to 100KB. This has been the optimal value when less than 8 transactions have updated the database objects. Once the log size exceeds this 100K threshold, the recovery times provided by CLOB(100K) are similar to those of the CLOB(0) configuration for this sample.

As we can see, CLOB cannot always guarantee the optimal results, but its overall performance is good if a correct threshold configuration is made. Take also into account, that the tests shown in figure 2 are specially tailored to the version-based recovery algorithm. Other access patterns will provide different behaviours for the lob-based and version-based recoveries, and this may produce a longer initial interval where the log-based solution is better than the version-based one. In such cases, the value of the CLOB threshold has less significance than in the sample shown in this test.

We can design another test more appropriate to the CLOB behaviour. For instance, we may use a sequence of four different transactions, accessing each one of them to 15 different objects; i.e., the set of accessed objects is four times greater than in the previous test. Its cost for a log-based recovery algorithm will be identical to the one shown above, but the time needed to recover using a version-based algorithm is four times greater than in the previous case.The results of this second test, without modifying the CLOB threshold are shown in figure 3. In this case, CLOB has produced an optimal recovery time in the interval that includes 1 to 12 missed transactions, but used the version-based approach when more than 12 transactions have been missed. It does not use the optimal approach for the interval between 12 and 22 missed transactions, but again this depends on the correct configuration of the threshold.

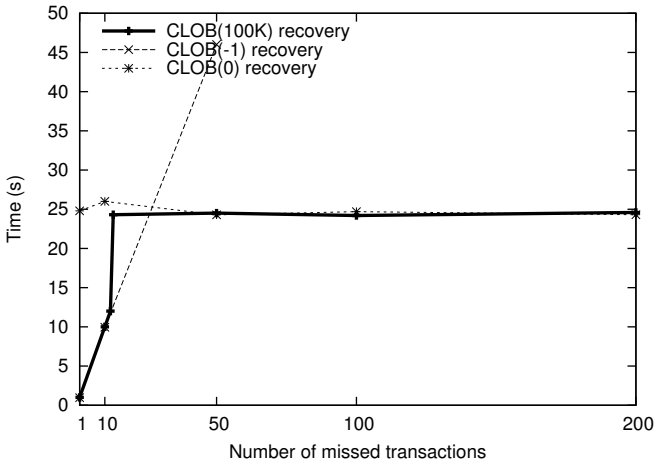

Figure 3. Recovery time of three CLOB configurations (2nd test).

In spite of all the *problems* outlined above to find out the optimal threshold, we have shown that the overall results of these two samples are better than those obtained using only one of the traditional recovery approaches (version-based or log-based).

6. Related Work

Usually, the best recovery protocols for replicated databases have been based on a logging technique [10, 11, 12, 14], but this implies that the amount of information to be transmitted may be high in long-term outages. On the other hand, log-based solutions do not need to lock in read mode the database to transfer the database state in the source replica, and this ensures that the source replicas will not block during the state transfer of the recovery process.

In version-based replication protocols [3, 8, 12, 13] only those data items that have been changed during the failure period are transferred to the recovering replica, reducing thus the amount of information to be transmitted. This is the best technique in long-term outages.

Our CLOB approach allows an easy transition between both techniques. Moreover, it automatises the logging tasks, freeing the database replication algorithm from these concerns. Currently, there are no systems with complete support for both recovering approaches. The work described in [12] studied both of them, but it finally proposes as the best approach one based on log-based transfers with lazy propagation in several phases. This may ensure, as the solution presented in [10], a minimal blocking time in the non-recovering replicas, but such a characteristic is also achievable in CLOB. Moreover, CLOB ensures a minimal effort in the database replication protocol in order to achieve this, at least when the log-based case is used.

7. Conclusions

The framework described in this paper allows an easy combination of automatic log-based and protocol-specific version-based recovery techniques. Such a combination provides a recovery solution with good overall performance, but requiring some additional storage space that has to maintain both the log of missed messages and the version-based information needed to complete the recovery. However, version-based information is only a little percentage of the usual log-based one, as it can be seen in several examples [8, 3], and this extra space requirement can be balanced with the performance gains.

References

[1] J. E. Armendáriz, J. J. Astrain, A. Córdoba, J. R. González de Mendívil, E. Martínez, and J. Bataller. A persistent storing service for use by consistency protocols. In *IASTED International Conference on Applied Informatics*, pages 1–6, Innsbruck, Austria, Feb. 2002.

[2] K. Birman and R. van Renesse, editors. *Reliable Distributed Computing with the Isis Toolkit*. IEEE-CS Press, Los Alamitos, CA, USA, 1994.

[3] F. Castro-Company, L. Irún-Briz, F. García-Neiva, and F. D. Muñoz-Escoí. FOBr: A version-based recovery protocol for replicated databases. In *Proc. of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, Lugano, Switzerland, Feb. 2005. IEEE-CS Press.

[4] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, Feb. 1991.

[5] J. Esparza-Peidro, F. D. Muñoz-Escoí, L. Irún-Briz, and J. M. Bernabéu-Aubán. RJDBC: A simple database replication engine. In *6th International Conference on Enterprise Information Systems*, pages 587–590, Porto, Portugal, Apr. 2004.

[6] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, Canada, 1996.

[7] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. ACM Press, 2nd edition, 1993. ISBN 0-201-62427-3.

[8] L. Irún-Briz, F. Castro-Company, F. García-Neiva, A. Calero-Monteagudo, and F. D. Muñoz-Escoí. Lazy recovery in a hybrid database replication protocol. In *Proc. of XII Jornadas de Concurrencia y Sistemas Distribuidos*, Las Navas del Marqués, Ávila, Spain, June 2004.

[9] L. Irún-Briz, F. D. Muñoz-Escoí, H. Decker, and J. M. Bernabéu-Aubán. COPLA: A platform for eager and lazy replication in networked databases. *Proc. of the 5th International Conference on Enterprise Information Systems*, pages 189–206, Apr. 2003.

[10] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. Non-intrusive, parallel recovery of replicated data. In *Proc. of 21st Symposium on Reliable Distributed Systems*, pages 150–159, Osaka Univ., Suita, Japan, Oct. 2002. IEEE-CS Press.

[11] B. Kemme. *Database Replication for Clusters of Workstations*. PhD thesis, Swiss Federal Institute of Technology Zurich, 2000. 155 pgs.

[12] B. Kemme, A. Bartoli, and O. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *Proc. of the IEEE Int. Conf. on Dependable Systems and Networks*, pages 117–130, Göteborg, Sweden, July 2001.

[13] F. Muñoz-Escoí, L. Irún-Briz, P. Galdámez, J. Bernabéu-Aubán, J. Bataller, and M. Bañuls. GlobData: Consistency protocols for replicated databases. In *Proc. of the IEEE-YUFORIC'2001*, pages 97–104, Valencia, Spain, Nov. 2001.

[14] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong replication in the GlobData middleware. In *Proc. of Workshop on Dependable Middleware-Based Systems (in DSN 2002)*, pages G96–G104, Washington D.C., USA, 2002.

[15] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, pages 206–217, Oct. 2000.