FOBr: A Version-Based Recovery Protocol For Replicated Databases*

Francisco Castro-Company Luis Irún-Briz Félix García-Neiva
Francesc D. Muñoz-Escoí
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
46022 Valencia (SPAIN)
{fcastro,lirun,fgarcia,fmunyoz}@iti.upv.es

Abstract

Within the field of databases that are deployed in distributed environments there is a need to guarantee consistency among replicas and availability among nodes despite of network disconnections and node crashes.

A recovery protocol, such as FOBr, manages the database update of a recovering node as it might have missed many transactions during its absence. FOBr does so without stopping data access, and minimizing its interference with the active nodes, their memory usage, and the network traffic.

The recovery protocol we propose here is very suitable when a fast recovery of the missed data is required. It balances the recovery issues among nodes very fairly and due to this, out-dated data will promptly be recovered. Thus accesses to data that is not out-dated are not interrupted or delayed in any way. These characteristics allow regular transactions to be performed during the recovery process.

1. Introduction

The adoption of Internet as a quotidian tool for data access and the spectacular growth of users impose the underlying databases to stretch the services they offer several steps beyond. These databases must maintain a huge amount of information and, furthermore, a huge load as the number of concurrent users increases highly. This is the point where need of database replication arises. Replication must provide some advantages such as availability, scalability and transparency. However, it implies some dangers that must be solved such as inconsistency or high retrieval latency.

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

While users access the information from very distant and geographically scattered locations, the database view we must be able to provide must be global, i.e it has to be the same as it would be if the data was centralised. Choosing a single server as the information provider doesn't supply any of the characteristics mentioned as replication advantages: the required bandwidth would be excessive and the service performance would be very poor.

Replication is achieved with the use of consistency protocols. A consistency protocol in such an environment takes care of several tasks such as data replication among nodes, load balance or accesses efficiency. In addition, it makes all these tasks invisible to the user. All the issues regarding consistency, availability and information retrieval or updates are internally managed while the user accesses the database as if it was not replicated at all.

Several consistency protocols have already been presented [8, 12]. They all benefit from the advances in the broadcast algorithms area [2, 7]. Communication primitives are used as the basis for consistency coordination among nodes. Notice that a wide area network (WAN) restricts communication capabilities and thus the consistency protocols design: communication among nodes must be carried only when strictly necessary and the information exchanged in each message mustn't exceed a reasonable size.

No database replication protocol can be considered complete if it does not provide recovery capabilities. Network partitions, node crashes and node failures are a common issue to be dealt with. Recovery protocols manage the node updates during failures and guarantee fault tolerance (we can use replication to guarantee that a crashed node that offered several services can be replaced by another one).

Traditionally, this is performed overloading one of the active nodes until a crashed node recovers. At best, a log is used to store all the transactions that any node might be missing and to be able to apply them later during its recovery.

We have designed and developed several consistency

protocols for our *COPLA* system. *COPLA* is the result of the *GlobData* [3] project: an architecture that provides an object–oriented view of a replicated database. This system includes the consistency protocol as a pluggable module so that it is a great basis for the implementation and testing of these protocols with real applications.

We are currently deploying two consistency protocols, each one with a different update propagation approach; eager in *FOB* [10], and lazy in *COLUP* [5]. The approach to provide *FOB* with recovery capabilities that we are introducing in this paper is based on versions management. When the system shows node failures, we keep a structure in the alive nodes with the object identifiers (this way we reduce the amount of memory needed to supply recovery) modified during the crashes. When a node recovers, these previously active nodes collect bunches of the identifiers states and send them to the recovering nodes. We only require reliable FIFO broadcast [2] guarantees for these messages which makes the delivery very fast.

This way we split the recovering steps between the rest of active nodes into small work packages that are performed simultaneously so that not only one of them is affected for a long time. Neither the recovering nodes nor the previously active nodes need to stop the users regular database accesses during recovery.

This paper is organized as follows. Section 2 introduces several details used through the rest of the paper. Section 3 describes the algorithm and section 4 studies the failures analysis and solutions. Later, performance results are provided in section 5, related work is discussed in section 6, and finally, section 7 concludes the paper.

2. Common Concepts

We assume a distributed system where database replicas are placed each one in a different node. This system is partially synchronous; i.e., 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 *partial-amnesia crash* failure model [1] is assumed for processes, whilst links may produce *omission* failures. This failure model is more realistic than the *crash* or *fail-stop* failure models [2], 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. However, it will be needed in case of a database replica corruption. In such a case, the faulty replica has to notify all the others about the crash of its copy, requesting a complete transfer of its database.

In our *COPLA* replication support each replica of the database communicates with the other replicas through the local consistency managers. *COPLA* isolates the user as well as the database manager from several tasks such as replication and a relational to object–oriented conversion.

As the consistency protocol can be changed as a plug–in, multiple consistency protocols may be used to achieve consistency but only one is allowed at a time. The consistency protocol uses a communication service to coordinate the network replicas. These replicas are preconfigured so that the network group and the node identifiers are known as soon as the system starts.

In our protocols, we deal with the concept of ownership. A node is owner of a set of objects and an object belongs to a node. The ownership allows us to identify which node is in charge to maintain the most updated version of every object.

Regarding ownership of a given object, a node can be classified into the following roles:

- 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 (or *ACR*, see section 2.4 for details on this) for that object.
- Synchronous nodes: These nodes did not create the object but are considered up-to-date replicas of it. They provide us with fault tolerance.

As our consistency manager has been designed for distributed systems that may use both local and wide area networks and where network errors may occur (nodes may crash and network partitions may happen), a membership service in charge to detect these situations is needed.

2.1. Membership Service

The membership service detects and informs about network changes. When a node crashes or rejoins the system, the membership service informs about the new system *view* to the consistency manager.

This service is in charge to assign *view numbers* when some node or group of nodes fail or recover. If a network partition arises, the view number is incremented and consensuated as a result of a view change in a primary subgroup. Minor subgroups can not increase their view number until they rejoin the primary one. Thus, this view number allows us to identify partitions and order them sequentially.

The membership service also detects when the group of nodes belongs to a *primary* or to a *minor* subgroup. Our

system uses the *primary partition* [11] model: only the subgroup with more than half of the network nodes is allowed to proceed and commit transactions. A *minor* subgroup will always be identified by a view number lower than the one of the *primary* subgroup in spite of the configuration changes the *minor* one might suffer.

2.2. Session Identifier

A session groups a number of transactions. Since we use *ACR* management, the session identifiers (or *SIDs*) include information about the node identifier where it was initiated.

In case of a node failure, the consistency protocol is able to know the owner of the granted *ACR* and if faulty, it will be able to manage the situation appropriately. The *SID* is useful for a number of issues such as transaction abortions or message identification and delivery.

2.3. Object Identifier

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 information is called *metadata* and will also need to be transferred when a recovering node receives its updated information.

2.4. Consistency protocol: *FOB*

The recovery protocol explained in section 3 is designed to be a complement for our *FOB* consistency protocol. *FOB* –which stands for “Full Object Broadcast”– is an eager and optimistic protocol. It is eager because all updates are broadcast before the transaction commits. It is optimistic because no distributed concurrency control operations are taken before a session tries to commit.

When the user calls *commit*, the protocol performs several operations before it is effectively applied into the database:

1. It collects the transaction *WriteSet* (i.e., the set of objects inserted, updated, or deleted in a given transaction) and groups its *OIDs* by their owner node.
2. An access confirmation request (or *ACR*, for short) 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:
   (a) 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.
   (b) 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.
      iii. It releases the *ACRs* granted to the finished transaction.

3. Algorithm Specification

The recovery protocol has two different phases that will be described separately. The first one comprises all the events happened from the instant when a node failure was detected until the moment when it rejoins the system. During this phase some records have to be taken about all the transactions that the faulty node has missed. On the sequel, we refer to this phase as the *collection phase*.

The second phase comprises the steps followed by all the system nodes when a previously faulty node starts again. This is the true recovery protocol; however, as it depends on the information collected during the *collection phase*, both phases must be described to define the complete recovery process. We refer to this last part of the protocol as the *recovery phase*.

3.1. Collection Phase

As soon as the node is notified about a node failure by the membership protocol, two steps are taken:

- The remaining alive nodes decide which ones of them inherit the ownership of the faulty node objects. The criterion used to migrate this inheritance is deterministic and it can be as simple as choosing the following alive node (in ascending order) as the inheriting node.
- A structure is created in each alive node in order to hold the *OIDs* of all the objects that will be updated while these nodes are not present. The structure needs to be stored persistently in order to allow node failures where the node does not simply disconnect, but it also crashes. This ensures that we will be able to provide recovery after total system failures; i.e., when all the system nodes crash. Notice also that the system is not allowed to perfom operations in a minor subgroup in case of network partitions. In that case, the structure is not necessary.

This structure will hold a *recovery list*. This list stores the updated *OIDs* that any faulty node is missing during any network view.

As multiple network view changes may occur, while there exist faulty nodes in the system the structure needs to be maintained. For the same reason, every alive node has to maintain every updated object in its recovery list, even if it is not their owner.

3.2. Recovery Phase

During the recovery phase two different roles are distinguished among the set of participating nodes:

- *Recovering node*: The node, or set of nodes, that is trying to join the system and that needs to bring its database up-to-date.
- *Previously-active node*: The nodes that hold information to help the joining ones join the system. These nodes have their databases updated and they provide the changes needed by the recovering nodes.

A node chooses its role when a network view change is notified. At this time, it has to compare the view number of its group with the view number of the joining group to decide which of the two groups is the updated one.

While this recovery protocol is run, all previously-active nodes go on processing local transactions and multicasting their updates as usual.

If any object accessed by any transaction started during the recovery period belongs to any of the recovering nodes, the inheriting node is still in charge of the *ACR* messages service. The recovering nodes are also able to proceed as usual, except for the fact that they are still not owners of any objects.

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:

1. The previously-active nodes build a **JOIN_UPDATE** message to update the currently owned objects that they know the recovering nodes have missed. Additionally, the transactions started in the previously-active nodes include the recovering node in the destination set of their update broadcasts. This way, the recovering node participates in the group regular operation as soon as possible. The recovering node receives these updates, but postpones their application until it applies all the **JOIN_UPDATE** messages.

   This **JOIN_UPDATE** message is built following this procedure:

   (a) The recovery list is checked to obtain the set of updated *OIDs* that the node currently owns.

   (b) 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.

   (c) The set of missed *OIDs* and network views is included too 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.

   As every previously-active node is in charge only of the set of missed *OIDs* that it currently owns, the recovery work is distributed among the active replicas so that none of them is overloaded and the work is performed in parallel.

   Once this information is collected, the nodes send the **JOIN_UPDATE** messages to the recovering nodes. The message needs to be sent even if the recovery list is empty or the previously-active node is not owner of any missed object, because the recovering node does not know this fact and it is expecting a message from all active nodes.

   Moreover, if failures arise while this protocol is running at this step, all objects owned by the previously-active node that fails will be inherited by one of the remaining previously-active nodes. This new owner node has to send an additional **JOIN_UPDATE** message with the information that has just inherited.

   Once sent, the previously-active nodes will be expecting **MERGED** responses from the recovering node.

2. 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. All these **JOIN_UPDATE** messages have to be applied in the same transactional context. Otherwise, as information is split into pieces, some integrity constraints might be broken. When all the **JOIN_UPDATE** messages are received, this transaction is committed. Notice that our *FOB* protocol is optimistic and regular transactions might have performed write accesses that intersect with the **JOIN_UPDATE** messages *OIDs*. To avoid this situation, the **JOIN_UPDATE** application must be preceded by the abortion of the locally conflicting transactions.

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.

We can take advantage of the lapse of time occurred from the **MERGED** sending to the **NO_ACT** reception to apply the postponed updates in a background process. This way, the ownership migration is not delayed.

3. 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.

   Once the **NO_ACT** is sent, the previously-active node is not responsible for the objects of the recovered node anymore. If this previously inheriting node receives any *ACR* for an object whose ownership is lost, it will be forwarded to the real owner.

   From the previously-active point of view, the ownership is not returned until the recovering node has completely recovered.

4. Finally, when the recovering node receives the **NO_ACT** message, will be able to manage its objects and the recovery is completed.

3.2.1. Considerations Let us consider a case where two minority partitions with recovery lists are joined (This is possible because a network disconnection may occur during a recovery period). The role of previously-active partition is also chosen using the partition with the greatest number view.

Note also that all multicasts used by this protocol need FIFO ordering guarantees. They have to be delivered to all the alive nodes, but that delivery may occur at different times.

4. Failure Analysis

The failure analysis has two key points that will be described separately.

4.1. Crash of a node during the recovery phase

A recovering node waits the **JOIN_UPDATE** messages from all the alive nodes. If a previously-active node crashes during this step, an inheriting node is chosen when the membership monitor notifies the network view change. This inheriting node sends the **JOIN_UPDATE** message that the faulty node should have sent because at this point, it is not sure that the sending was complete. In case the recovering node had already received this message, it discards the following copies and goes on. In case the faulty node is the inheritor of the recovering node objects, the transactions accessing its objects will be aborted. This process is repeated for the cases of multiple crashes.

If the faulty node is one that is recovering, no extra actions need to be performed if we take some care. As soon as a previously-active node receives the **MERGED** message, it removes the recovery list if the system is complete. This step must be performed before the **NO_ACT** is sent. Otherwise the non inheriting nodes would have removed the recovery list while the inheriting nodes might have not.

4.2. Joining of a node during the recovery phase

The incorporation of a member while a node recovers, may occur during two stages of recovery: before or after the **MERGED** message:

- *Before*: The previously recovering node waits for an answer from all the previously-active nodes but the newly recovering node must not be considered as previously-active because it does not have enough information to participate in this recovery. The newly recovery node inheritor already assumes this role.

  For the newly recovering node it is indifferent where the update messages come from, since the rest of nodes are able to update it.
- *After*: At this moment, the previously recovering node will be waiting for a **NO_ACT** confirmation but this message arrival will be delayed a little bit: The inheriting node needs to send the **JOIN_UPDATE** message to the newly recovering node with information from the previously recovering node.

5. Performance Results

In order to check the performance gains that may introduce the FOBr algorithm, we have implemented a similar algorithm based on update-logging. Once a node fails, the log-based recovery algorithm begins to save all the missed update messages in secondary storage. If the faulty node recovers, these missed messages are resent to it in the recovery process.

In these tests, a database with 6000 objects has been used. Their ownership is balanced among the four nodes that compose the system; i.e., each node has created 1500 objects. The tests are started with a first phase (the collection phase described in section 3.1) where one of the four nodes is not accessible, simulating a node failure. In the second phase (the proper recovery phase described in section

3.2), we measure the time needed to complete the recovery of this isolated node.

The system nodes are arranged in a cluster and use a high-speed network to interconnect them, so communication latency is negligible.

Two series of tests have been designed. In the first one, each transaction updates a set of 10 different objects randomly distributed among the 6000 ones that compose the replicated database. Different number of transactions have been used in this first series, but in all cases the probability of accessing and updating more than once each object is minimal. As a result, this first series corresponds in all the cases to a short-term failure, and the behaviour of FOBr is always worse than that of a log-based solution, as we can see in figure 1. This can be easily explained, since FOBr needs additional effort to get the state of each updated object: it has to read its OID from its log and later it has to retrieve its state. A log-based solution only needs to read the log and transfer it to the recovering replica. Since each logged update corresponds to a different object, no gain can be achieved with FOBr.

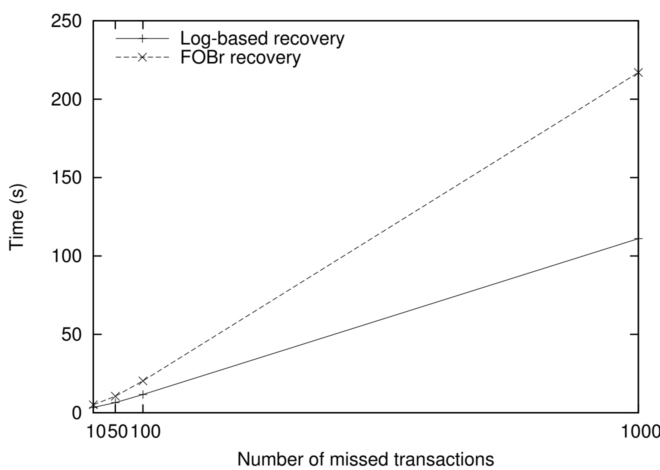

Figure 1. Recovery time with random access to objects.

The second series of tests corresponds 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.

In this series, a sequence of transactions accessing objects in a set of 15 or 150 objects is repeated multiple times. As a result, each accessed object has been accessed multiple times and it needs to be transferred only once using FOBr. This experiment will provide us a guide on the number of updates on each object during the failure period that are needed to make FOBr more convenient than a log-based solution.

Thus, in figure 2, the set of accessed objects corresponds only to 15 in each of the active nodes. As a result, in FOBr each of the source nodes need only to transfer the state of these 15 objects. As we can see, a log-based solution is still better than FOBr if a low number of transactions has been executed whilst the recovering node was faulty. But FOBr becomes the best one if each object has been updated more than 7 times.

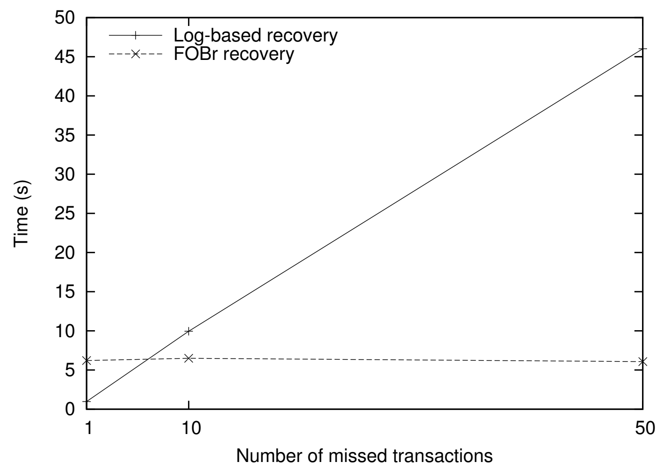

Figure 2. Recovery time with 15 objects/trans.

In figure 3 the same experiment is repeated again, but now each transaction accesses 150 objects in each of the active nodes. In this second experiment of the second series, the update messages are bigger than in the previous one. As a result, the recovery times of a log-based solution are worse than those of FOBr. In order to reduce this penalty, the log-based solution used in these experiments does not transfer the state of the updated objects, but the SQL sentence needed to complete such updates. This reduces the message size, but not the time needed to apply the updates in the recovering node.

Anyway, as we can see in figure 3, FOBr provides initially a longer recovery time than a log-based solution (when only one update has been applied to each object during the failure period), but becomes the best option if more than two updates are applied on average to the set objects to be recovered.

To sum up, a log-based solution is better than FOBr if we can ensure that the objects will not be updated multiple times during the time a node remains faulty. The exact number of updates needed to ensure that FOBr is the best

option depends on the whole number of objects to be transferred in the recovery process, but both needed numbers are quite low in all cases.

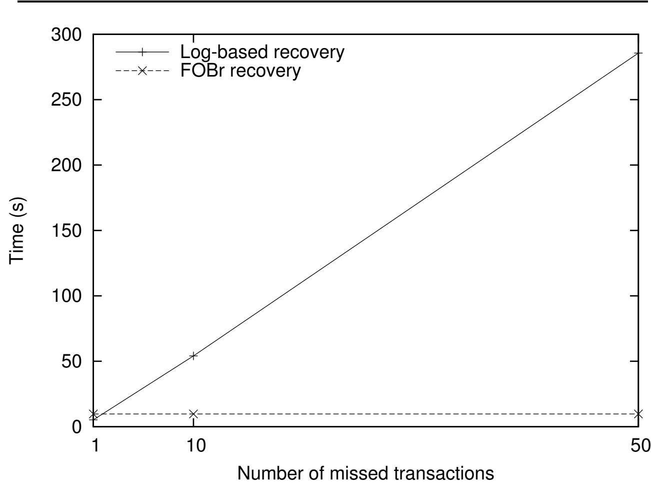

Figure 3. Recovery time with 150 objects/trans.

6. Related Work

Recent works on replicated database recovery algorithms exist [4, 6, 9] and all of them try to minimize the time required to complete such recovery, and the blocking time in the replicas taken as the source of such information transfer.

In order to do so, in [9] several approaches are described, and finally one with a multiple-phase (or lazy) state transfer is used, requiring *enriched view synchrony* and atomic broadcasts. Both features demand higher communication costs than the uniform reliable FIFO broadcast used in FOBr. This lazy transfer algorithm minimizes the blocking time in the active replicas since only one of them is partially blocked during the recovery, but requiring a longer period to get the recovering replica in a serving state. However, the basic approach being followed in each of these transfer phases is similar to ours: to collapse multiple updates on each object, transferring only the latest version of them. Additionally, since FOBr divides the source work among all the previously active replicas, our blocking time could be lower than that required in [9].

A similar approach is provided in [6] where all the active replicas are able to collaborate in the recovering tasks as sources of the state transfer. Additionally, it also places its support in the middleware layer (as FOBr does), instead of in the core of the DBMS as in [9]. However, the algorithm described in [6] also requires a total order broadcast protocol and strong virtual synchrony; i.e., it requires a communication support more expensive than FOBr, needing additional rounds to deliver a message. Additionally, this recovery protocol uses a missed-updates log, in order to transfer all writeset broadcasts missed by the faulty node. A version-based propagation technique like the one used in FOBr might provide better results for long-term outages, requiring also a lower amount of secondary storage, since *OIDs* are smaller than missed updates. On the other hand, in [6] an additional solution is described to reduce the amount of information to be logged and transferred, using checkpointing to install a recent copy of the database and later applying only the updates lost since the latest checkpoint. As a result, we cannot ensure that the performance of FOBr is better or worse than that provided by [9] and [6].

Finally, the solution described in [4] needs a shorter recovering time than FOBr, but it is integrated into a hybrid replication protocol that generally produces a higher abortion rate than *FOB*.

7. Conclusions

We have discussed in this paper a solution to provide recovery capabilities to consistency protocols for replicated databases minimizing the amount of information being transferred. This is particularly suitable in wide area networks.

Our approach ensures that the integrity constraints will hold and the fact that the recovery work is balanced among the system alive nodes guarantees low interference with the database regular operation.

The recovery process design allows transactions to be executed even while the recovery protocol has not finished so that users will not notice an extended interruption of service but only minimum delays for the transactions execution.

The use of node identifiers allows the protocol to deal with the concept of object ownership. This is the key of load balancing: Each node contributes with a subset of the whole bunch of objects to be recovered thus it is free to serve other user requests.

The use of object identifiers (*OIDs*) and versions imposes very low requirements of space to keep the missed objects. This fact makes information access faster and more selective: Only the last version of an object is migrated, as oppossed to the techniques where a log of transactions needs to be maintained and each object is transferred as many times as updates where applied on it. This advantage is very desirable for large scale environments, reducing thus the communication traffic.

Another task that has been considered carefully is the maximum maintenance of data availability even during the recovery process. In our protocol, a recovering node is in-

cluded as destination of transaction update broadcasts immediately. Moreover, the objects owned by the recovering node are completely accesible as they are managed by an inheriting node.

References

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

[2] 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.

[3] Instituto Tecnológico de Informática. GlobData Web Site, 2004. Accessible in URL: *http://globdata.iti.es*.

[4] 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.

[5] L. Irún-Briz, F. D. Muñoz-Escoí, and J. M. Bernabéu-Aubán. An improved optimistic and fault-tolerant replication protocol. In *Proc. of 3rd Workshop on Databases in Networked Information Systems*, volume 2822 of *Lecture Notes in Computer Science*, pages 188–200, Aizu, Japan, Sept. 2003. Springer.

[6] 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.

[7] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *International Conference on Distributed Computing Systems*, pages 156–163, 1998.

[8] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, Sept. 2000.

[9] 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.

[10] 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.

[11] A. Ricciardi, A. Schiper, and K. Birman. Understanding partitions and the "no partition" assumption. In *Proc. of the 4th IEEE Workshop on Future Trends in Distributed Computing Systems*, pages 354–360, Lisbon, Portugal, Sept. 1993.

[12] 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.