Consistency Protocols in Globdata[*]

Francesc D. Muñoz-Escoí, Luis Irún-Briz, Pablo Galdámez, José M. Bernabéu-Aubán, Jordi Bataller, and M.Carmen Bañuls

Instituto Tecnológico de Informática.
Polytechnic University of Valencia.
Camino de Vera, s/n. 46022. Valencia. Spain.
E-mail: {fmunyoz,lirun,pgaldam,josep,bataller,banuls}@iti.upv.es

Abstract Globdata is a software tool that provides an object-oriented view of a set of replicated relational databases. These databases may be distributed in a wide area environment. To ensure consistency among the independent databases that build this environment some protocols are needed. These protocols have to reach a high degree of transaction completions, aborting a minimal percentage of them, and ensuring the consistency of all the databases.
A system of this kind may be used in the development of applications for companies that have several branch offices, such as banks, hypermarkets, etc. Usually, these companies have a large amount of operations that can be solved with local information, but sometimes the information generated in other branches is also needed. The services provided by Globdata allow an efficient completion of both kinds of requests.

1 Introduction and Motivation

When an organization starts an internet project, a variety of objectives must be met. Many of the current internet applications (i.e., e-bank applications) manage huge amounts of information. This information is mainly accessed with a strong geographical locality. In addition, these types of application usually strive for a high degree of availability, since they offer services not only to external, but also to internal clients, which must be capable of accessing the information at any time. The locality of the accesses suggest that in many cases the database can be partitioned [12,4]. In many scenarios, it may be necessary to replicate the information in a set of servers, each one attending its local clients. The different replicas of the database must be then interconnected, a WAN being usually the best fit alternative.

Another example of this scenario can be found in telephony applications, managing large amount of information, where access patterns are highly local.

When the databases containing the information must be replicated, it becomes necessary to introduce protocols and algorithms that provide a minimal set of guarantees about the consistency of the data [1,13]. The traditional approaches for replicating databases are centered in the use of fast LAN's

[*] This work has been partially supported by European grant IST-1999-20997

[9,10,8,6,5], where network-intensive protocols are used. In internet applications, the network is a limited resource, and the problems introduced by the WAN must be appropriately dealt with [11,3,2].

The Globdata project [7] strives to provide a solution for these kinds of applications in which efficiency, availability and high volume data handling must be achieved. It does so by defining a specific architecture for a set of replicated databases, together with a programming API and a set of consistency modes for data access.

This paper presents a proposal of protocols capable of meeting the consistency requirements posed by Globdata's goals. Although several protocols with slightly different goals are presented, all of them share a characteristic: They cannot be classified as pessimistic, since transactions are allowed to proceed locally, being checked for consistency violations at commit time. When a consistency violation is found, the transaction is rolled back.

The rest of the paper is organized as follows. Section 2 provides some common concepts shared by all consistency protocols, and several characteristics of the Globdata environment where the protocols are used. Section 3 presents an overall version of all consistency protocols and discusses the algorithm choices that can be adopted and the resulting algorithms, once these options have been chosen. In section 4 we describe in detail one of the consistency protocols outlined in the previous section. Section 5 discusses how failures can be managed in all consistency protocols, since the failure handling procedures are common for all of them. Section 6 describes the actions taken when a faulty node recovers, and finally, in section 7 we provide some concluding remarks.

2 Common Concepts

Within Globdata, each replica of the database communicates with the other replicas through the local consistency managers. Consistency managers are, thus, in charge of implementing the consistency protocols which drive a Globdata system. Globdata's proposed architecture places them as mediators for every data access action the local sessions (i.e., transactions) perform.

Some of the protocols use lazy replication. Consequently, not all system nodes have the latest version of each object. As a result, considering a particular object, the set of nodes can be assigned one of the following roles:

- *Owner node*: It is the node where the object has been created. It is the manager for *access confirmation requests* (see below) for the object; i.e., it allows or denies these requests when the session initiators ask it about them.
- *Synchronous nodes*: They are the nodes preconfigured in the system to maintain the up-to-date replicas of the object. Their number for each object is preconfigured. They are needed for fault tolerance.
- *Deferred nodes*: These nodes do not usually maintain up-to-date replicas of the object, although they may have synchronous replicas sometimes (at least, when the session that has caused the latest object version has been initiated in one of them).

The approach we take to concurrency control is influenced by the fact that data may be lazily replicated, voiding the possibility of using traditional locks. Each object is, thus, assigned an owner node which controls accesses to that object. This control is only enforced when sessions try to commit.

When a session initiates its commit phase, the owner nodes of the accessed objects are contacted to discover if the session has used the latest committed versions of the objects. In this case, the owners validate the accesses, and the session can be committed. If any of the objects read by the session is reported to have an outdated version, the session is flagged for abortion. We refer to these contact actions as "*access confirmation requests*", since they only ask the owners about the correctness of the object versions used by the session.

Globdata has been designed for distributed systems that may use wide area networks. Within WANs, nodes may fail and network partitions may happen. Globdata needs, thus, the services of an appropriate membership protocol. This membership service assigns static node identifiers to the nodes that form the system, reducing the length of a node identifier. Consistency protocols require node identifiers to record the object ownership, as well as the role assigned to each node when a particular object is considered.

Since we use *access confirmation request* management, session identifiers (or SID's, for short) must include in their fields the identifier of the node that has initiated them. Thus, in case of node failure, the consistency protocol knows which granted access requests belonged to the faulty node, being able to manage the situation appropriately.

Objects also use an identification structure similar to sessions. For instance, object identifiers (OIDs) hold the static identifier of the node that has created them. When a node fails, all its objects are inherited by a new manager. However, this change in the management has to be written down by all live nodes in some records that they hold. These tables are later used to find the manager for each object.

Moreover, we also need to hold an object version associated to each object. This helps when a node recovers, since it only needs to send to one of its neighbors the last object version known for each object maintained in the recovering database. When a node receives that message, it replies with all changes that must be applied to bring the recovering database up to date.

An object version number includes the SID of the session that has written its last value. This information is used to build the graph of causal precedent sessions in the consistency protocols based on session updates (see sections 3.3 and 4 for details).

In case of partition failures; i.e., when some network links fail and two or more node subgroups[1] remain isolated, we only allow the subgroup with a majority of up-to-date replicas of a given object to work with sessions where that object is

[1] We use the term "*subgroup*" for naming each of the sets of communicating nodes that have appeared after the network partition. Each subgroup is unable to communicate with any other.

involved. If a partition arises or if some node fails, the role of the faulty owners[2] is moved to one or more than one of the remaining live managers.

3 Overall Protocol Descriptions

All consistency protocols follow a common basic algorithm that consists of the following steps:

1. Each consistency manager maintains in the database a set of tables with meta-data, that is, data needed to implement a particular consistency protocol. In particular, all the protocols keep in meta-data tables information about the versions of the objects stored in the local database.
   When a local session tries to read an object in a database access, the local consistency manager checks its version in the local database. If the consistency manager finds that the local version is outdated, it sends a message to the owner node of the object requesting the latest updates. The local session is blocked until these updates have arrived and have been applied to the local database. In fact, the query that originated this read access still has not been executed in this database.
   Write accesses are not managed this way. They are directly applied to the database without any check. However, although no check is made some records are taken for all accesses. Actually, for each access made in a session the sets of objects read or written in those accesses are written in the meta-data tables to be able to build later the readset and writeset of each session.
2. When a session tries to commit, its local consistency manager retrieves the read and write sets of the session and sends an *access confirmation request* to each one of the owners of the objects involved in that session, including for each object, its OID, object version and access mode (read or write).
   When an owner receives one of the above messages, it compares the version of the object within the message with the latest versions for that object. If they are equal, the owner grants the requested access permission and sets this SID as the current owner of that access grant. The access grant is exclusive if the access mode was “write” or shared if the access mode was “read”. The grant is assigned to the requester until its session has propagated its updates to, at least, the synchronous set of replicas for the objects involved in the session. Once the session has been committed in this set of nodes, the session initiator releases the access grants after changing the version of their managed objects accessed in write mode within the requesting session.
   While a session holds access grants, other sessions that request the same grants in conflicting modes receive a deny reply to their requests. When such a reply is delivered, the consistency manager that requested that grant aborts the corresponding session.

[2] And this only can happen if we ensure that at least an up-to-date replica of the managed object remains in the live group.

Using the above approach, different consistency protocol variants can be designed providing support for different kinds of consistency modes. Next we describe the three consistency modes we care about within Globdata. Later we show how different alternatives in the overall approach lead to those modes, and finally we outline the protocols resulting from such alternatives.

3.1 Consistency Modes

Three different consistency modes are considered in Globdata. Each session that uses the Globdata services may only use one of these modes at a given time, although there are methods in the API for changing the consistency mode associated to the application sessions. These consistency modes are:

- *Plain*: This mode only allows isolated read requests and guarantees that all accessed objects respect session causal commit order. However, the versions of the accessed objects may not be the latest ones; i.e., they may be out of date.
- *Checkout*: This mode is a permissive variation of the transaction mode discussed below. In checkout mode, the isolation property is not guaranteed. Thus, if several sessions have read accesses on a given object, one of these sessions is allowed to promote its access to write mode (this can break serializability, thus, within a standard transactional setting this would lead to the blocking of the promoting transaction, or to the abort of a transaction). However if two of these sessions promote their read accesses to write mode for the same object, one of them is aborted.
- *Transaction*: The usual transaction guarantees (basically, the ACID properties) must be provided.

We have described the consistency modes in increasing order of restrictiveness; i.e., transaction mode allows less concurrency conflicts than checkout mode, and checkout mode less than plain mode. So, when two sessions that use different consistency modes conflict, some rules have to be adopted to resolve that conflict. In practice, we use the rules of the more restrictive mode.

3.2 Protocol Alternatives

The consistency algorithm shown at the beginning of section 3 admits various choices in several of its steps. We outline these options and their multiple alternatives now. The chosen characteristics generate different kinds of consistency protocols that we describe later.

These are the algorithm characteristics that admit multiple choices:

- *Update multicast when a session commits*: A protocol may broadcast the session changes when it commits. In this case, all object replicas are synchronous and if the commit procedure is careful with the session commit order, all consistency modes can be easily guaranteed.

On the other hand, the session changes can be multicast only to a reduced set of replicas (those placed in the preconfigured synchronous replicas for the objects involved in the session). This management leads to lazy replication. Since different objects may reside in different sets of synchronous nodes, special care has to be taken to guarantee all consistency modes when lazy replication is used.

– *Per object or per session update propagation*: If lazy replication is used, when a session commits we have to choose whether all session updates have to be propagated to all nodes where at least a synchronous replica of any updated object exists, or the updates are propagated per object. The second option implies that if two objects have been modified in a session and they do not reside in the same set of synchronous nodes, then we only propagate to each set of synchronous replicas the changes that involve one object, but not all session changes.
  The first option guarantees that plain mode consistency is satisfied, however the second option does not always guarantee that.

Additionally, depending on the consistency mode that a session uses, the actions that a consistency manager has to do when a session reads objects or commits are the following:

– *On read accesses*: If plain mode is being used, the object version stored in the local database need not be the latest one. However, causal commit consistency must be guaranteed, so the session changes have to be stored in a database only when all sessions that precede the one that is committing have been already stored in that database. So, plain mode is not problematic on read accesses if some care has been taken when commit operations were made using other consistency modes.
  For checkout and transaction modes, when a read access is requested, the consistency manager has to ensure that the latest object version exists in the local database to avoid the session being rolled back later. If this version is not present (and this may only happen when lazy replication is used), it has to be requested to the object owner, notifying it about the object version stored in the database of that session initiator. When this request is received by the owner, the requested version has to be returned to the session initiator. However, since plain mode must be usually guaranteed, not only the latest version is needed, but all object versions between that stored in the initiator database and the latest one. Moreover, not only these objects must be returned: the initiator also needs all other session changes for all the sessions that have caused these object updates.
– *On commit time*: There are no commit operations in plain consistency mode because only read operations are allowed in that mode.
  In checkout mode, the read accesses are treated the same way as in transaction mode, but the commit procedure differs a bit. In this mode, if a session has read an object version that at commit time is not the latest one, it is not aborted. However, objects in the write set have to be checked at commit

time and their version must be equal to the latest one. If this does not happen, the session is aborted. To sum up, read objects may change while the session is executing, but written objects must not be overwritten by other concurrent checkout sessions.
In transaction mode, both read and written objects must have their latest version when the session is committed. If a subsequent change has been made by another concurrent session, the one that is terminating must be aborted.

3.3 Consistency Protocols

We present three consistency protocols that have different characteristics. They are the following:

- *Full object broadcasting*: This protocol uses immediate updates in all system databases, so it does not use lazy replication. Thus, the writeset of a committed session is broadcast to all system nodes and it is immediately applied. Of course, not all sessions are committed, since object owners must grant the access confirmation to do so. These access permissions depend on the consistency mode used by the session.
  It supports the three consistency modes and read accesses do not need any additional action (they can be locally performed without any special handling).
- *Simple object update*: This protocol uses lazy replication and object updates, instead of session updates. As a result, although this protocol complies with all consistency modes, plain mode requires more effort, since the way sessions that use transaction or checkout mode propagate their updates does not provide the guarantees needed by plain mode accesses.
  Note that at commit time, the updates are only propagated to the preconfigured synchronous replicas of each modified object. Thus, it is possible that the full effects of a session are not reflected at all nodes that have received an update message from it: a node can have a synchronous replica for one of the involved objects, and a deferred replica for one of the other objects.
  When a read operation needs a more recent version than the one stored in the local database, only the latest version is requested (and obtained) from the object owner. No other contents need to be transferred.
- *Session set update*: This protocol uses lazy replication and session updates; i.e., when the updates are transferred to other nodes, not only the object changes are transmitted to their synchronous replicas, but all session updates (i.e., the session writeset) are transferred to all nodes that have at least a synchronous replica of one of the changed objects.
  To support plain mode, an additional problem appears: before the effects of a session can be applied, all sessions preceding it in causal order need to have been applied before to the same database. Sometimes, however, this has not yet done. For instance, when an object has a deferred replica in a given node that has not received any update for a long period of time and

some of the objects that maintain a synchronous replica in that node have been modified in the same session.

The sequence of steps needed to get a group of missing sessions is the following:

1. A request is sent to the object owner, asking for the session that has made the latest change on that object and all its precedent sessions using causal commit order. All nodes maintain a log of committed sessions, until they have been applied to all system nodes (when this happens, the session is removed from the log).

   The request also carries the local (and out-of-date) object version number for the requested object. So, the object owner, following the SID's stored in the object versions and scanning the logs, is able to build the graph of precedent sessions.
2. The object owner replies with the graph of sessions. This graph has as its root the session that has caused the latest update on the requested object and that also includes all its precedent sessions following causal commit order. This causal commit order is easy to find using the SID's stored in the object versions. For each session, the log also maintains its writeset and readset. So, with this information, all the graph can be built. To add more layers to the graph, only the readsets of the current leaves of the graph have to be inspected, and all the sessions that appear in the object versions of those readset objects are included in that layer. When a session does not appear in the log, it can not be added to the graph since this means that all its changes have been applied in all system nodes.

   The graph built in this step only maintains the SID's of the sessions, not their readsets or writesets.
3. The requester checks the received graph and scans it in depth order, starting at the leaves and removing from the graph all sessions that have already been applied to the local database. When the scanning arrives to a level where no session has been removed, this procedure terminates. The resulting graph is returned to the object owner node.
4. The object owner receives the returned graph and replies with the readsets and writesets of all the sessions found in the graph. These data is stored in the requester node when it is received, terminating thus the retrieval of the precedent sessions.

4 Algorithm Specification

This section provides additional details for one of the proposed consistency protocol. In this protocol, the message transport is assumed reliable (in the sense of TCP/IP reliability) and a membership service exists, which notifies the system nodes about the failures and recoveries of other system nodes.

4.1 Session Set Update Protocol

This protocol transfers the whole set of session updates each time a session is committed. This set of updates is sent to all nodes that have at least a synchronous replica for one of the objects updated in that session. As a result, plain mode can be easily supported, without needing any special action; i.e., the reads can be locally completed without needing any further message exchange.

Each consistency manager executes the following algorithm:

- Every node maintains a log containing every session applied in its local database. A process is run asynchronously in every node in order to update each node in the system. When this asynchronous process can ensure that a session has been applied in every node, this fact is indicated and the session can be eliminated from the logs.
- When a node detects an out-of-date object in a read request, it locates the owner node of the object (N_o), and sends a request message to it in order to update its object copy. This request message contains the identifier and version of the out-of-date object.
- The owner node receives the request message, and looks at its meta-data for the set of causal dependent sessions, needed to update the requested object from the given version to the version held in the local database of the owner node. This process is performed by the following algorithm:
  - The owner node looks at the meta-data for the last session that modified the requested object (T_o). In this session, other objects have been read (the readset of T_o or $R(T_o)$).
  - For each object o_i contained in $R(T_o)$, the node should search its log for every session T_j having o_i in its writeset (T_j causally precedes T_o).
  - The node takes then T_o and every T_j and composes a graph representing the causal dependencies of T_o and every T_j.
  - For the last sessions added to the composed graph, the algorithm iterates in order to include every causal dependency in the causal graph. The iteration ends when all the logged sessions with causal precedence have been included in the causal graph.
- The composed graph (that is actually containing statements of sessions) is sent to the requesting node, which analyzes the graph in order to eliminate already applied sessions, and to determine whether each session in the graph can be applied in its local database.
  The graph can be cut out at a session T when this session T has been already applied in the requesting node. This occurs when every object in the writeset of T has a lower version than the version held in the local database.
  In addition, a session in the graph cannot be applied to the local database when an object contained in its readset has a higher version than the version held in the local database (that is, there exists causal precedent session yet unknown for the requesting node), and this out-of-date object must be updated before.

- When the cutting out process is completed, the requesting node sends a new message to the owner node, requesting the complete writeset (values and versions) of the meta-session resulting from compacting the graph.
- The owner replies with a message that holds the writesets of each session included in the previous request. This information can be extracted from the meta-data tables, since all these sessions have been locally applied by this owner node and it knows about all these writesets.

Note that "plain" consistency mode is directly implemented, granting that each update preserves the causal consistency of the local view of the database.

The amount of information needed to provide this functionality consists of:

- A log of every applied session for each node. This implies redundancy in the logs.
- A session is kept in the log until an asynchronous process determines that the session has been applied in every node.

5 Failure Analysis

Several failure scenarios must be considered to ensure that these protocols work when failures arise. We consider two kinds of situations here. The first one deals with the completion of the steps given in the protocols specification. The second one deals with the migration of the managing role between nodes. We discuss both of them next.

5.1 Algorithm Completion

When a node fails, our membership monitor detects this failure and notifies all of the remaining nodes about this event. Partition failures are notified the same way, but in this second case the set of faulty nodes may be bigger.

Let us see what happens with the sessions initiated by a faulty node. We can distinguish the following cases, according to the step at which the failure happens:

- If one of these sessions has not surpassed its access confirmation granting step of the algorithm, no record of that session exists in any of the live nodes. So, that session can be discarded. In fact, when its host node recovers, it must abort that session, forcing its application to repeat the work.
- If the session fails once it has obtained the remote access grants, but before it has multicast any update a similar situation arises. No record of the session updates can be found in any of the live nodes, so the session cannot be terminated in the remaining nodes. As a result, that session must be aborted when its host node recovers. However, the faulty node has obtained some access grants and this may prevent other sessions to work.

  To avoid this, a solution is provided. Since the multicasts are atomic and reliable, if an object update has been received by one of its synchronous replicas, all of them have received this update. So, when the membership service notifies the failure of a node, all object owners scan their grants lists. If some access has been granted to a session initiated in the faulty node, the access granter (that is also the owner of the object protected by that grant) has to check if some update multicast has been received associated to the SID that requested that access. If no such an update was received, the grant can be released, otherwise the following point has to be considered.

– If the session has at least initiated the update multicast, its updates may have arrived to other nodes. In this case, we need to perform the same actions as in the previous case. The grants held by this session have to be released. Since the updates have been received, and this may only arise if all grants were obtained and all changes made (but still not committed) in the original node, no additional access grant will be needed. Since the session initiator node has failed, the grants are not needed by any other node that replaces the faulty one, because this hypothetical replacer node already has committed this session. As a result, the access grants have already been used correctly and they must be released now.

  No other case needs consideration. Perhaps the session had not been completed yet, but this only means that it held some grants that have been released as a consequence of the steps explained above. So, the session has been completed now.
– If the node has failed once the update multicast was initiated but before the "updatever" message[3] has been sent (assuming that this kind of message is needed for that session), then another problem arises because the node that eventually inherits the object ownership does not have the version number information needed to send that message.

  When the node promoted as object owner starts its new role, some actions must be taken. It already knows that it maintains a synchronous replica. It may decide that another node (or more than one node) has to be promoted to hold a synchronous replica. This fact depends on the type of configuration needed. Moreover, it has to broadcast a message to all accessible nodes with deferred replicas of objects whose management it has inherited. This message contains the version number of these inherited objects. The deferred nodes will reply this message indicating the promoted node whether their current replica is actually out-of-date or not.

5.2 Role Migration

When a node fails, all the objects it managed have to be managed by one of the live nodes. As a result, the object ownership initially managed in that node has to be migrated to one of the other nodes. So, we have to discuss two tasks in this section. The first one deals with the criterion followed to elect the node

[3] This message is needed to notify all deferred replicas that hold the latest version number of the object until now, that this version has changed and that they do not have an up-to-date version in their local database.

that will replace the faulty one. The second task deals with the migration of the access granting management.

Let us see how these tasks may be carried out:

- The election of the replacer node is based on the static identifiers associated to all preconfigured members of the system. Since each node has an identifier of this kind, we only need to choose the live node with a synchronous replica that has the lowest identifier among all those that are greater than that of the faulty node (or the lowest one, if the faulty node had the greatest identifier). To be able to elect the new manager, a majority of synchronous replicas must still exist in the system.
  In case of using an even number of synchronous replicas, some criterion is needed to break the tie in case of network partitions. For instance, the subgroup that holds the node with lowest identifier among the previous set of synchronous replicas will maintain the new manager.
  We assume that the number of synchronous replicas is known in advance. In case of network partitions, it may arise that the owner for a given object remains alive, but the greater part of its synchronous replicas remain unavailable. If such a situation arises, the current manager must give up its role. A problem appears here. If the majority of synchronous replicas are in another subgroup after the network partition, one of them will take up the management role, according to the procedure described in the previous paragraph. However, if those replicas have failed, no other session in the whole system will be able to use that object again.
- The node that inherits the object ownership has to ask all the others about the access grants they have; i.e., it has to know which of the grants it manages has an owner and who is that owner. To this end, each node that holds a grant that was managed by the faulty node, sends an ownership message to the new manager. This message contains the identifier of the object associated to the grant, the access mode, and the SID of the session that holds it. If a node has no grants, it sends an empty message.
  A timeout is set by the new manager to receive all these messages. If some message has not been received in this period, an explicit request is sent to that node.
  Note that the criterion used to select the replacer node is known by all managers, so no message is needed to make public the identity of the new manager.
  In case of network partitions, this is also true. If one subgroup loses its manager for a given object, no session that accesses that object will be allowed in that subgroup. Moreover, all grants maintained in that subgroup have to be released, and all sessions that had those access grants must be aborted.

6 Recovery Analysis

When a node recovers, the membership monitor notifies all live nodes about it. So, during the reconfiguration of the system state, two tasks must be performed related to the new node: recovery of the object ownership (for all the objects initially created by that node, if any) and updates of its local database to make it consistent with those of the other nodes.

In order to recover the object ownership no special action must be taken by the new node. One of the previously active nodes had inherited the object management for all objects of the currently recovered node. This node must send to the original owner all its information regarding access grants. Until this message is received by the recovered node, it can't manage its incoming access grant requests or releases. It has to hold them temporarily. Once the message and the object ownership have been transferred to the recovered node, it resumes the access grant management. It may happen that an access request or release arrives later to the node that has returned the object ownership to the recovered node. If that happens, all these messages must be forwarded to the current manager. No special action is needed by the sender of such messages. The forwarding is done by the message receiver, who knows which is the current manager for the object involved in that request or release.

Another change is needed if the recovered process belongs to the class of nodes that must maintain synchronous replicas of several objects. The owners of those objects have to reinclude it in the set of synchronous replicas, and possibly one of those replicas has to be degraded to the deferred category (although it initially maintains an up-to-date copy, but eventually it will become obsolete if it is degraded). No message exchange is needed to do so.

Note also that all the tasks explained above are made during the reconfiguration steps. In these steps no new sessions are allowed and the session management is temporarily disabled.

In this recovery procedure, the new node sends to one of its neighbors (for instance, the one that has the greatest probability of having inherited its object ownership, i.e., the one that has an identifier immediately greater than its) a request of the updates. In this request it includes the OIDs and object versions that it has in its local database. As a reply, its neighbor will send a message with all updates that have to be made to make consistent its database to those of the other nodes.

7 Concluding Remarks

Global data access is increasingly important to a large number of Internet-based applications. Such access has to be provided with guarantees of data availability and consistency. This work proposes an approach to build such applications more easily and reliably by means of replicating commercial grade, reliable database engines, and running adequate protocols to keep their data consistent.

Concurrency control in these protocols is quite optimistic, since access confirmation requests may be made when local updates have concluded. This behavior is suitable for distributed replicated databases whose applications usually work only with "local" data, i.e., data that has been created by other local applications, but where replication is needed to improve the access over "remote" data when the greater part of the sessions use "read only" access.

The algorithms discussed are being implemented within the Globdata Project, where they will be accesed by different kinds of applications, with different access patterns, which will give us in the future, the oportunity to verify the best performing approach for each kind of application.

References

1. P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley, 1987.
2. Fabrizio Ferrandina, Thorsten Meyer, and Roberto Zicari. Correctness of lazy database updates for object database systems. In *POS*, pages 284–301, 1994.
3. Fabrizio Ferrandina, Thorsten Meyer, and Roberto Zicari. Implementing Lazy Database Updates for an Object Database System. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 261–272, Santiago, Chile, 1994.
4. Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. pages 173–182, 1996.
5. Maurice Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Transactions on Database Systems*, 12(2):170–194, June 1987.
6. Maurice Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. on Database Sys.*, 15(1):96–124, March 1990.
7. ITI, UPNA, and FFCUL. COPLA programming interface, deliverable 04 (workpackage 01). Technical report, Globdata, IST Programme, project number: IST-1999-20997, June 2001.
8. Sushil Jajodia and David Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Trans. on Database Sys.*, 15(2):230–280, June 1990.
9. Bettina Kemme and Gustavo Alonso. A suite of database replication protocols based on group communication primitives. In *International Conference on Distributed Computing Systems*, pages 156–163, 1998.
10. Narayanan Krishnakumar and Arthur J. Bernstein. Bounded ignorance: A technique for increasing concurrency in a replicated system. *ACM Trans. on Database Sys.*, 19(4):586–625, December 1994.
11. Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. on Comp. Sys.*, 10(4):360–391, November 1992.
12. Erhard Rahm. Empirical performance evaluation of concurrency and coherency control protocols for database sharing systems. *ACM Trans. on Database Sys.*, 18(2):333–377, June 1993.
13. Ouri Wolfson and Sushil Jajodia. Distributed algorithms for dynamic replication of data. pages 149–163, 1992.