The GlobData Fault-Tolerant Replicated Distributed Object Database *†

Luís Rodrigues Hugo Miranda Ricardo Almeida
João Martins Pedro Vicente

Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática
{ler,hmiranda,ralmeida,jmartins,pedrofrv}@di.fc.ul.pt

30th July 2002

Abstract

GlobData is a project that aims to design and implement a middleware tool offering the abstraction of a global object database repository. This tool, called Copla, supports transactional access to geographically distributed persistent objects independent of their location. Additionally, it supports replication of data according to different consistency criteria. For this purpose, Copla implements a number of consistency protocols offering different tradeoffs between performance and fault-tolerance. This paper presents the work on strong consistency protocols for the GlobData system. Two protocols are presented: a voting protocol and a non-voting protocol. Both these protocols rely on the use of atomic broadcast as a building block to serialize conflicting transactions. The paper also introduces the total order protocol being developed to support large-scale replication.

1 Introduction

GlobData [1] [1] is an European IST project started in November 2000 that aims to design and implement a middleware tool offering the abstraction of

*This work has been partially supported by the project IST-1999-20997, GlobData.

†Sections of this report where published in the Eurasia-ICT 2002 conference proceedings.

[1]The GlobData partners are: Instituto Tecnológico de Informática de Valencia (ITI), Spain; Faculdade de Ciências da Universidade de Lisboa (FCUL), Portugal; Universidad Pública de Navarra (UPNA), Spain; GFI Informatique (GFI), France; Investigación y Desarrollo Inforático (IDI EIKON), Spain.

a global object database repository. The tool, called COPLA, supports transactional access to geographically distributed persistent objects independent of their location. Application programmers have an object-oriented view of the data repository and do not need to be concerned of how the objects are stored, distributed or replicated. The COPLA middleware supports the replication of data according to different consistency criteria. Each consistency criteria is implemented by one or more consistency protocols, that offer different tradeoffs between performance and fault-tolerance.

This paper reports the work on strong consistency replication protocols for the GLOBDATA system that is being performed by the Distributed ALgorithms and Network Protocols (DIALNP) group at Universidade de Lisboa. Based on the previous work of [17, 16], two protocols are being implemented: a voting protocol and a non-voting protocol. Each of these protocols supports two variants, namely eager updates and deferred updates. The protocols are executed on top of an off-the-shelf relational database that is used to store the state of persistent objects and protocol control information. All protocols rely on the use of atomic broadcast as a building block to help serialize conflicting transactions. A specialized total order protocol is being implemented in the *Appia* system [14] to support replication in large-scale. The atomic protocol inherits ideas from the hybrid protocol of [19]. The paper introduces the GLOBDATA architecture, and resumes both the consistency protocols and the atomic multicast primitive that supports them.

This paper is organized as follows: Section 2 describes the general COPLA architecture. Section 3 presents the consistency protocols. Section 4 presents the atomic multicast primitive that supports the protocols. Section 5 presents some optimizations to the basic protocols. Section 6 describes the recovery procedure for failed nodes. Section 7 discusses related work that studies database replication based on atomic broadcast primitives. Section 8 concludes this paper.

2 Copla System Architecture

COPLA is a middleware tool that provides transparent access to a replicated repository of persistent objects. Replicas can be located on different nodes of a cluster, of a local area network, or spread across a wide are network spanning different geographic locations. To support a diversity of environments and workloads, COPLA provides a number of replica consistency protocols.

The main components of the COPLA architecture are depicted in Figure 1. The upper layer is a "client interface" module, that provides the functionality used by the COPLA applications programmer. The programmer has an object-oriented view of the persistent and distributed data: it uses a subset of Object Query Language [7] to obtain references to distributed objects. Objects can be concurrently accessed by different clients in the context of distributed transactions.

For fault-tolerance, and to improve locality of read-only transactions, an object database may be replicated at different locations. Several consistency

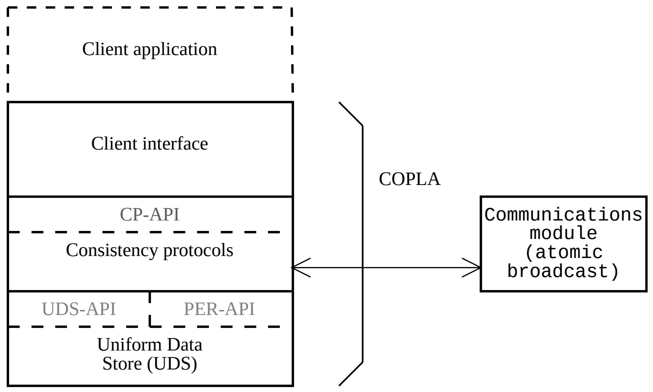

Figure 1: COPLA architecture

protocols are supported by COPLA; the choice of the best protocol depends on the topology of the network and of the application's workload. To maintain the user interface code independent of the actual protocol being used, all protocols adhere to a common protocol interface (labeled CP-API in the figure). This allows COPLA to be configured according to the characteristics of the environment where it runs.

The uniform data store (UDS) module (developed by the Universidad Pública de Navarra) is responsible for storing the state of the persistent objects in an off-the-shelf relational database management system (RDBMS). To perform this task, the UDS exports an interface, the UDS-API, through which objects can be stored and retrieved. It also converts all the queries posed by the application into normalized SQL queries. Finally, the UDS is used to store in a persistent way the control information required by the consistency protocols. This control information is stored and accessed through an dedicated interface, the PER-API.

Architectural challenges The GLOBDATA project is characterized by a unique combination of different requirements, that make the design of the consistency protocols a challenging task. Namely, the GLOBDATA aims to satisfy the following requirements:

- *Large-scale:* the consistency protocols must support replication of objects in a geographically dispersed system, in which the nodes communicate through the Internet. This prevents the use of protocols that make used of specific network properties (such as the low-latency or network-order preservation properties of local-area networks [18]).
- *RDBMS independence*: a variety of commercial databases should be supported as the underlying data storage technology. This prevents the use of solutions that require adaptations to the database kernel.

- *Protocol interchangeability*: COPLA must be flexible enough to adapt to changing environment conditions, like the scale of the system, availability of different communication facilities, and changes in the application's workload. Therefore it should allow the use of distinct consistency protocols, that can perform differently in several scenarios.
- *Object-orientation:* even if COPLA maps objects into a relational model, this operation must be isolated from the consistency protocols. In this way, the consistency algorithms are not tied to any specific object representation.

3 Strong Consistency Protocols

In GLOBDATA, the application programmer may trade fault-tolerance for performance. Therefore, a suite of protocols with different behavior in the presence of faults is being developed by different teams. Another project's partner, the ITI, is developing a suite of protocols based on the notion of object ownership [15]: Each node is the manager for the objects created in it, and is responsible for managing concurrent accesses to those objects. On the other hand, the DIALNP team at Universidade de Lisboa, is developing two protocols that enforce strong consistency even in the presence of faults. In fact, the two protocols reported here can also be configured to trade reliability for performance, by implementing a deferred updates scheme. The strong consistency protocols rely extensively on the availability of an uniform atomic broadcast primitive. The implementation of this primitive will be addressed later in the paper.

3.1 Interaction Among Components

We now describe the strong consistency protocols designed for COPLA. Both protocols cooperate with the Uniform Data Store to obtain information about which objects are read or updated by each transaction. This information, in the form of a list of unique object identifiers (OIDs), allows the protocols to have fine-grain information about which transaction conflict with each other. Since the consistency protocols only manipulate OIDs, they remain independent from the representation of objects in the database.

The Copla transactional model In COPLA, the execution of a transaction includes the following steps:

1. The programmer signals the system that a transaction is about to start.
2. The programmer makes a query to the database, using a subset of OQL. This query returns a collection of objects.
3. The returned objects are manipulated by the programmer using the functions exported by the client interface. These functions allow the application to update the values of object's attributes, and to read new objects

through object relations (object attributes that are references to other objects).

4. Steps 2-3 are repeated until the transaction is completed.

5. The programmer requests the system to commit the transaction.

Interaction with the consistency protocols The common protocol interface basically exports two functions: a function that must be called by the application every time new objects are read by a transaction, and a function that must be called in order to commit the transaction.

The first function, that we call UDSAccess(), serves two main purposes: to make sure that the local copies of the objects are up-to-date (when using deferred updates, the most recent version may not be available locally); and to extract the state of the objects by calling the UDS (the access to the underlying database is not performed by the consistency protocol itself; it is a function of the UDS component). It should be noted that in the actual implementation this function is unfolded in a collection of similar functions covering different requests (attribute read, relationship read, query, etc.). For clarity of exposition, we make no distinction among these functions in the paper.

The second function, called commit(), is used by the application to commit the transaction. In response to this request the consistency protocols module has to coordinate with its remote peers to serialize conflicting transactions and to decide whether it is safe to commit the transaction or if it has to be aborted due to some conflict. In order to execute this phase, the consistency protocol request the UDS module to provide the list of all objects updated by the current transaction. Additionally, the UDS also provides the consistency protocols with an opaque structure containing the state of the updated objects. It is the responsibility of the consistency protocol to propagate these updates to the remote nodes.

Replication strategies Using the classification of database replication strategies introduced in [21], the strong consistency protocols of COPLA can be classified as belonging to the "update everywhere constant interaction" class. They are "update everywhere" because they perform the updates to the data items in all replicas of the system. This approach was chosen because it is easier to deal with failures (since all nodes maintain their own copy of the data) and it does not create bottleneck points like the primary copy approach. They are "constant interaction" because the number of messages exchanged by transaction is fixed, independently of the number of operations in the transaction. Given that the cost of communication in most GLOBDATA configurations is expected to be high, this approach is much more efficient than a linear interaction approach. The protocols described below explore the third degree of freedom: the way transactions terminate (voting or non-voting).

Interaction with the atomic broadcast primitive An atomic broadcast primitive broadcasts messages among a group of servers, guaranteeing atomic and ordered delivery of messages. Specifically, let m and m' be two messages sent by atomic broadcast to a group of servers g. *Atomic* delivery guarantees that if a member of g delivers m (resp. m'), then all correct members of g deliver m (resp. m'). *Ordered* delivery guarantees that if any two members of g deliver m and m', they deliver them in the same order. These two properties are used by both consistency protocols: the order property is used by the conflict resolution mechanism, and atomic delivery is used to simplify atomic commitment of transactions.

3.2 The Non-Voting Protocol

This protocol is a modification of the one described in [17], altered to use a version scheme for concurrency control [6], and adapted to the Copla transactional model.

The protocol uses the following control information for each object: a version number and a flag that states whether or not the local copy of this object is up-to-date. If an object is out-of-date, the identifier of the node that has the latest version of the object is also kept. Note that in the basic protocol, all replicas are up-to-date when a transaction commits. Only in the deferred updates mode, it is possible that some replicas remain temporarily out of date. All this information is maintained in a consistency table, which is stored in persistent storage, and is updated in the context of the same transaction that alters data (i.e., the consistency information is updated only if the transaction commits).

When an object is created, its version number is set to zero. Each time a transaction updates an object, and that transaction commits, the object's version number is incremented by one. This mechanism keeps version numbers synchronized across replicas, since the total order ensured by atomic broadcast causes all replicas to process transactions in the same order.

When enforcing serializability, two kinds of conflicts must be considered by the protocol: read/write conflicts and write/write conflicts. Read/write conflicts occur when one transactions reads an object, and another concurrent transactions writes on that same object. Write/write conflicts occur when two concurrent transactions write on the same object. In GlobData, all objects are read before they are written (as shown above in the Copla transactional model), so a write/write conflict is also a read/write conflict. Considering this definitions, in the version number concurrency control scheme, conflicting transactions are defined as follows:

> Two transactions t and t' conflict if t' has read an object with version v_o and when t' is about to commit, object o's version number in the local database, v'_o, is higher than v_o. That means that t' has read data that was later modified (by a transaction t that modified o and committed before t', thus increasing o's version number), and therefore t' should be aborted.

```
• UDSAccess(t,l):
    1. Add the list l of objects to list of objects read by transaction t.
• commit(t):
    1. Obtain from the UDS the list of objects read ($\mathrm{RS}_t$) and its version
       numbers, and the list of objects written ($\mathrm{WS}_t$) by this transaction.
    2. Send $< t, \mathrm{RS}_t, \mathrm{WS}_t >$ through the atomic broadcast primitive.
    3. When the message containing t is delivered by the atomic broadcast:
        (a) If t does conflict with some other transaction
            i. Abort.
        (b) else (consistent transaction)
            i. Abort all transactions conflicting with t
            ii. Commit the transaction.
```

Figure 2: Non-voting protocol

The general outline of the non-voting algorithm is now presented:

1. All the transaction's operations are executed locally on the node where the transaction was initiated (this node is called the delegate node).

2. When the application requests a commit, the set of read objects and its version numbers, and the set of written objects is sent to all nodes using the atomic broadcast primitive.

3. When a transaction is delivered by the atomic broadcast protocol, all servers verify if the received transaction does not conflict with other local running transactions. There is no conflict if the versions of the objects read by the arriving transaction are greater or equal to the versions of those objects present in the local database. If no conflict is detected, then the transaction is committed, otherwise it is aborted. Since this procedure is deterministic and all nodes, including the delegate node, receive transactions by the same order, all nodes reach the same decision about the outcome of the transaction. The delegate node can now inform the client application about the final outcome of the transaction.

Note that the last step is executed by *all* nodes, including the one that initiated the transaction.

Depicted in Figure 2 is a more detailed description of the algorithm. It is divided in two functions, corresponding to the interface previously described. Both functions accept the parameter t, the transaction to act upon. UDSAccess() also accepts a parameter, l, which is the list of objects that t has read from the UDS. Note that step 3 of the commit() function is executed by all nodes, including the delegate node.

The algorithm uses the order given by atomic broadcast for serializing conflicting transactions: if a transaction is delivered and is consistent, it has priority over other running transactions. This implies that if there are two conflicting transactions, t_1 and t_2, and t_1 is delivered before t_2, then t_1 will proceed, and t_2 will be marked as conflicting (in step 3(a)), because it has read stale data.

The decision is taken in each node independently, but all nodes will reach the same decision, since it depends solely on the order of message delivery (which is guaranteed to be consistent at all replicas by the atomic broadcast protocol). When a commit is decided, the version number of the objects written by this transaction are incremented, and the UDS transaction is committed.

Note that to improve performance, local running transactions that conflict with a consistent transaction are aborted, in step 3(b). There is a conflict when the running transaction has read objects that the arriving transaction has written. This would cause the transaction to carry old versions of read objects on its read set, which would cause it to be aborted later on in step 3(a). This way an atomic broadcast message is spared[2].

Aborting a transaction does not involve any special step. In this case, the commit() function is never called, and all that has to be done is to release the local resources associated with that transaction.

3.3 The Voting Protocol

This protocol is an adaptation of the protocol described in [13] adapted to the Copla transactional model. It consists in two phases, a write set broadcast phase, and a voting phase.

The general outline of the algorithm is as follows:

1. All the transaction's operations are executed locally on the delegate node, obtaining (local) read locks on read objects (note that, in order to be written, an object must be previously read).

2. When the application requests a commit, the set of written objects is sent to all nodes using atomic broadcast.

3. When the write set of a transaction t is delivered by atomic broadcast, all nodes try to obtain local write locks on all objects in the set. If there is a transaction that holds a write lock on any object of the write set of t, t is placed on hold until that write lock is relinquished. Transactions holding read locks on any object of the write set of t are aborted (sending an abort message through atomic broadcast). When the delegate node has obtained all write locks, sends a commit message to all servers, through atomic broadcast.

4. Upon the reception of a confirmation message, a node applies the transaction's writes to the local database and subsequently releases all locks held

[2]This optimization may not be effective in all cases : if the running transaction has already sent its message, then there is no saving, the transaction is merely aborted sooner. When its message arrives, it will be discarded.

- *UDSAccess(t,l):*
  1. *For each object in the list l obtain a read lock. If any of those objects is write-locked, t is place on hold until that object's write lock is released.*
- *commit(t):*
  1. *Obtain from the UDS the list of objects written (WS_t) by t.*
  2. *Send $< t, \mathrm{WS}_t >$ through the atomic broadcast primitive.*
  3. *When the message containing t is delivered by atomic broadcast:*
     (a) *For each object o in WS_t, try to obtain a write lock on it, executing the following steps atomically:*
        i. *If there is one or more read locks on o, every t' that has that read lock is aborted (by sending an $a_{t'}$ message using atomic broadcast), and the write lock on o is granted to t.*
        ii. *If there is a write lock on o, or all the read locks on o are from transactions t_{recv} whose message $< t_{recv}, \mathrm{WS}_{t_{recv}} >$ has already been delivered, t will be placed on hold until those write locks are released.*
        iii. *If there is no other lock on o, grant the lock to t.*
     (b) *If this node is the delegate node for t, send c_t by atomic broadcast.*
  4. *When a c_t message is delivered: commit t, writing all its updates in the database and releasing all locks held by t. All transactions t'waiting to obtain write locks on an object written by t are aborted (a $a_{t'}$ message is sent through atomic broadcast).*
  5. *When a a_t is delivered: If t is a local transaction the message is ignored, otherwise abort t, releasing all its locks.*

Figure 3: Voting protocol

on behalf of that transaction. Upon the reception of an abort message, the delegate node aborts the transaction an releases all its locks (other nodes ignore that message).

A detailed description of the algorithm is shown in Figure 3. The algorithm uses the order given by atomic broadcast to serialize conflicting transactions. The final transaction order is given by the order of the $< t, \mathrm{WS}_t >$ messages. Conflict detection is done using locks.

Write/write conflicts, that occur when two concurrent transactions try to write over the same object, are detected by the lock system (two transactions try to obtain a write lock on the same object). Since write locks are obtained upon reception of $< t, \mathrm{WS}_t >$, the order of these messages determines the lock acquisition order. As seen in Figure 3, if a transaction t obtains a write lock, it will force a later transaction t' to wait when it tries to obtain its lock. If t commits it will force t' to abort.

Read/write conflicts, that occur when two concurrent transactions access the same object, one for reading and the other for writing, are solved by giving priority to writing transactions. When a $< t, \mathrm{WS}_t >$ message is delivered, write locks are obtained, causing transactions that have read locks on objects in WS_t to abort. This rule does not apply to transactions whose write set has already been delivered (step 3(a)ii): in this case t will be placed on hold until the decision is taken regarding the transaction(s) that own the read lock.

All nodes obtain the same write locks in the same order, because the order of the $< t, \mathrm{WS}_t >$ messages is the same in all nodes, and the lock procedure is deterministic. As such, all nodes will be able to respect the decision issued by the delegate node.

Optimization This protocol can be further improved, to avoid aborting unnecessary number of transactions. In the lock acquisition phase (after $< t, \mathrm{WS}_t >$ is delivered), instead of immediately aborting transactions that hold read locks on objects in WS_t, they can be placed on an alternative state, called executing_abort. This is to consider situations where t is latter aborted, which means that the other transactions that interfered with t were needlessly aborted.

Transactions in the executing_abort state can proceed executing, but cannot commit. If they attempt to, they will be placed on hold. If t commits, then all transactions in executing_abort because of t will be aborted. If t aborts, then the transactions in executing_abort will return to normal execution state (if there is no other transaction t' that is placing t in executing_abort).

Read-only transactions that try to commit and are in executing_abort state do not need to be put on hold - they can commit immediately. The final serialization order is as these transactions executed before the transaction that placed them in executing_abort.

4 The Atomic Broadcast Protocol

The two strong consistency protocols implemented in the COPLA middleware make extensive use of the properties of an atomic multicast protocol. To efficiently support the consistency protocols, a protocol designed for large-scale operation has been implemented [20].

The protocol is an adaptation of the hybrid total order protocol presented in [19]. The hybrid protocol combines two very known solutions for total order: sequencer based and logical clocks. A process may be active or passive: if it is active then it orders messages for itself and others; if it is passive then it has an active process that orders his messages. If more that one active process exists, then the order is established using logical clocks. The processes can change is role depending on the number of messages transmitted and the network delay between themselves and the other processes. These characteristics optimize the protocol behavior in large-scale networks.

Unfortunately, the original protocol as presented in [19] supports only a non-uniform version of atomic multicast, i.e., the order of messages delivered to crashed processes may differ from the order of messages delivered to correct processes. In the database context, this may lead to the state preserved in the database of a crashed process to be inconsistent. Therefore, in Copla, one needs an uniform total order protocol, i.e. a protocol that ensures that if two messages are delivered by a given order to a process (even if this process crashes), they are delivered in that order to all correct processes.

The new protocol also supports the optimistic delivery of (tentative) total order indications [8, 18]. Given that the order established by the (non-uniform) total order protocol is the same as the final uniform total order in most cases (these two orders only differ when crashes occur at particular points in the protocol execution), this order can be provided to the consistency layer as a tentative ordering information. The consistency protocols may optimistically perform some tasks that are later committed when the final order is delivered.

Typically, the most efficient total order algorithms do not provide uniform delivery and assume the availability of a perfect failure detector. Such algorithms may provide inconsistent results if the system assumptions do not hold. On the other hand, algorithms that assume an unreliable failure detector always provide consistent results but exhibit higher costs. The most interesting feature of the protocol derived for Copla is that it combines the advantages of both approaches. On good periods, when the system is stable and processes are not suspected, the algorithm operates as if a perfect failure detector is assumed. Yet, the algorithm is indulgent, since it never violates consistency, even in runs where processes are suspected.

5 Optimizations to the Basic Protocols

The basic protocols described in Section 3 can be optimized in two different ways. One consists in delaying the propagation of updates, the *deferred updates* mode. Other consists in exploiting the optimistic delivery of the atomic multicast algorithm.

5.1 Deferred Updates

Both algorithms presented before can be configured to operate on a mode called *deferred updates*. This mode consists in postponing the transfer of updates until such data is required by a remote transaction, trading fault-tolerance for performance. Note that, when using deferred updates, the outcome of a transaction is no longer immediately propagated to all replicas: it is stored only at the delegate node. If this node crashes, transactions that access this data must wait for the delegate node to recover. On the other hand, network communication is saved because updates are only propagated when needed.

The changes to the protocol required to implement the *deferred updates* mode are encapsulated in the getNewVersions(t,l) function, which is depicted in Figure 4. In both protocols, the function is called after step one, i.e., it becomes step two of UDSAccess().

- *For each OID in l:*
    1. *Check if the object's copy in the local database is up-to-date.*
    2. *If the object is out-of-date, get the latest version from the node that holds it.*

Figure 4: getNewVersions(t, l)

Associated with each OID, there is a field, called *owner*, that contains the identifier of the node holding the latest version of that object's data. If that field is empty, then the current node holds the latest version.

When deferred updates mode is not used, modified data is written to the database at the end of the commit procedure. This step is modified to implement deferred updates: only the delegate node writes the altered data on its database, setting the owner field to empty. All the other nodes write the identifier of the delegate node in their databases. The only information that is sent across the network is merely a list of changed OIDs (instead of that list plus the data itself).

5.2 Exploiting Optimistic Atomic Delivery

As described above, the atomic broadcast primitive developed in the project has the possibility of delivering a message optimistically (opt-deliver), i.e., the message is delivered in a tentative order, which is likely to be the same as the final order (u-deliver). This can be exploited by both consistency protocols. The tentative order allows the protocols to send the transaction's updates to the database earlier. Instead of waiting for the final uniform order to perform the writes, they are sent to the database as soon as the tentative order is know. When the final order arrives, all that is required is to commit the transaction. This hides the cost of witting data behind the cost of uniform delivery, effectively doing both things in parallel.

Non-voting protocol Upon reception of an opt-deliver message, all steps in the commit() function are executed, with the following modifications: in step 3(a), conflicting transactions are not aborted, but placed on hold (transactions on hold can execute normally, but are suspended when they request a commit, and can only proceed when they return to normal state); in step 3(b-ii), the data is sent to the UDS, but the transaction is not committed.

When the message is u-delivered, and its order is the same as the tentative one, all transactions marked on hold on behalf of the current one are aborted, and the transaction is committed. If the order is not the same, then the open UDS transaction is aborted, all transactions placed on hold on behalf of this one are returned to normal state, and the message is reprocessed as if it arrived at that moment.

Voting protocol (This modification is for the optimized version of the protocol) Upon reception of an opt-deliver message for transaction t, all steps in the commit() function are executed, with the following modifications: in step 3(b), the c_t message is not sent; instead the transaction is placed in a waiting state, and the updates are sent to the database. When the same message t is u-delivered, if the order was maintained, the transaction is placed in normal state, the c_t message is sent, and the rest of the procedure is the same. If the order differs, then all transactions waiting for the final message ordering opt-delivered before t, and not yet u-delivered, are aborted.

6 Node Recovery

To support recovery of nodes, the COPLA consistency algorithms store information about every committed transaction in a persistent log (the log is maintained by the PER-API of the UDS module). Writes to the log are performed in the context of the corresponding transaction. Therefore, the UDS ensures that if the transaction commits the log update is also committed. The log consists of a table with three columns: the transaction's sequence number, the transaction's unique identifier and the set of changes performed by the transaction. Each line corresponds to a transaction log record.

There are two complementary methods to perform recovery that are used by the COPLA consistency algorithms. The less efficient but simpler method consists in transferring to the recovery nodes a complete, up-to-date, copy of the entire database. This method is only used for nodes that have been failed or disconnected for a long period. The other approach consists in sending to the recovery node the transaction it missed during the period it was crashed or disconnected. The information required to replay these transactions is stored the COPLA log. Note that the log has finite size. That is why a complete copy of the database may be required after long crashes.

Both the voting and the non-voting algorithms operate as described in the previous sections as long as the local node remains in a majority partition. If due to failures or due to a network partition a node finds itself in a minority partition it stops processing write transactions. It is possible to configure the system to let read-only transaction to read (possibly stale) data in a minority partition. The total number of servers in the system and their locations is a configuration parameter that is provided by the system manager. As such, establishing whether or not a given group of nodes is a minority (if there are n servers, a minority group is one with $n/2$ members or less) is trivial.

The recovery procedure consists of the protocols that allow a node bring its state up-to-date with the state of the nodes in the majority partition (or to form a new majority partition after a total failure and recovery). To support the recovery procedure, the consistency protocol maintain three complementary *views* of the server membership. These views are the following:

- The *up-to-date process group view* $P(t)$ contains all nodes that are capable of processing transactions at time t.

- The *recovery group view* $R(t)$ contains all nodes that are connected with the nodes on $P(t)$ at time t. It therefore contains the process group view and all the nodes that are updating their state but noy yet capable of processing transactions.

- The *static group view* S contains all the nodes in the system and is specified in the initial node configuration. This view is static and does not change with time.

A majority (or minority) partition is always defined compairing the up-to-date process group view $P(t)$ with the static group view S, *i.e,* we consider that $P(t)$ has become a minority group if $\#P(t) \leq \frac{\#S}{2}$ Note that, by definition, $P(t) \subseteq R(t)$ at any given time t. In the normal case, $R(t) = P(t) = S$, i.e., all nodes are up and can process transactions.

The recovery method outlined below assumes that is possible to assign a unique sequence number to every transaction processed in the system. This sequence number is used during recovery to identify which transactions have been missed by the recovering nodes. This number is derived from the total order algorithm which is used by both consistency protocols: when the system starts up, all nodes will set the initial transaction number to the number of the last committed transaction. Whenever a new total order message containing a transaction arrives, the counter is increased, and its value is the number of the arriving transaction. Since all nodes receive the same messages in the same order, the number given to each transaction is the same in all nodes.

When a node recovers, that node must synchronize its database with the nodes that belong to $P(t)$. This is done in the following manner:

1. The recovering member r joins the group of replicas, and is placed in $R(t)$.

2. While recovering is taking place, processes that belong to the up-to-date process group view may continue to accept and process new transactions. Therefore, the recovering node must immediately start collecting all transactions it receives. Since it does not have its state up-to-date, it does not process these updates: they are kept in a pending state. The unique identifier (not the sequence number) of the first transaction added to the pending state is stored in a variable $pending_r$.

3. A node $p \in P(t)$ is selected to be the *peer* node for the recovering node. The peer node will be responsible for helping the recovering node to synchronize its database.

4. The recovering node r sends to p the sequence number of the last transaction that it processed successfully ($last_r$) and the identifier of the first transaction added to pending ($pending_r$). The transactions in this interval are all the transactions the recovery node has missed.

5. The peer node p then checks if all the transactions missed by the recovering node are registered in p's log. If yes, the outcome of these transactions are

transfered to the recovering node. If not, this means that the recovering node has been crashed or disconnect for too much time. In this case, a copy of the complete database is initiated.

6. After receiving all missed transaction, the recovering nodes processes the transactions that have been stored in pending until its state is fully synchronized with the group of up-to-date replicas. At this point, r joins the $P(t)$ and start processing transactions normally.

7 Related Work

In the database literature, one can find different alternatives to enforce replica consistency. Some authors [10, 9] suggested voting schemes, where a certain number of *votes* is given to each node, and a transaction can only proceed if there are enough replicas to form a sufficient *quorum*. This quorum must be defined in such a way that at least one replica detects conflicting transactions. The scalability problems of this technique (and of other related replication techniques) are identified in [11]. One of the main problem consists on the large number of deadlocks that may occur in the face of concurrent access to the same data: the number of deadlocks grows in the proportion of n^3 for n nodes. It has been suggested that a technique to circumvent this problem is to implement some sort of master-slave approach: each object belongs to a master node, and to avoid reconciliation problems, nodes that do not own an object make tentative updates, and then contact that object's master node to confirm those updates.

An alternative approach followed in COPLA consists in using an active replication scheme based on the use of efficient atomic multicast primitives. Systems such as [17, 12, 16], use the message order provided by atomic broadcast to aid in the serialization of conflicting transactions. An example of such a system is the Dragon [2] project, that uses extensively replicated algorithms that take advantage of the ordering properties of atomic broadcast in order to avoid deadlocks [21, 22, 13]. However, unlike our approach, the Dragon protocols are implemented at the database-kernel level, and cannot be used with off-the-shelf database systems.

Concurrently with our work, the CNDS group [3] has developed a system [4, 5] similar to GLOBDATA. Their approach also separates the consistency protocol from the database module. However, unlike our protocols, their system does not provide fine grain conflict detection. In COPLA, because a node knows the read and write set of the transactions that is executing, it can: a) Detect if a given query (read-only transaction) can be applied immediately (i.e., if it does not conflict with pending update transactions); b) abort transactions earlier, saving an expensive atomic broadcast message.

8 Conclusion

This paper presented the strong consistency protocols supported by the COPLA middleware, a tool that provides transactional access to persistent transparently replicated objects. These protocols are based on the use of atomic broadcast primitives to serialize conflicting transaction and to enforce the consistency in the transaction commit phase. The protocols satisfy a set of challenging requirements imposed by the GLOBDATA architecture, namely: large-scale operation, RDBMS independence, protocol interchangeability and support for an object-oriented access to data. The paper also introduced the atomic broadcast algorithm that has been designed to support the execution of the consistency protocols.

Currently we are completing the implementation of both consistency protocols and the atomic broadcast protocol. We then plan to evaluate the impact of different loads on both algorithms under varying conditions, as well as the performance of the atomic broadcast primitive in the those varying conditions.

References

[1] http://globdata.iti.es/.

[2] http://www.inf.ethz.ch/department/IS/iks/research/dragon.html.

[3] Center for Networking and Distributed Systems, John Hopkins University, Baltimore, USA.

[4] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. Practical wide area database replication. Technical Report CNDS-2002-1, Center for Networking and Distributed Systems, February 2002.

[5] Y. Amir and C. Tutu. From total order to database replication. Technical Report CNDS-2001-6, Center for Networking and Distributed Systems, November 2001. Accepted to the IEEE International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, July 2002.

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

[7] R. G. G. Cattell. *The Object Data Standard: ODMG3.0.* Morgan Kauffmann Publishers, 2000.

[8] Pascal Felber and André Schiper. Optimistic active replication. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS'2001)*, Phoenix, Arizona, USA, April 2001. IEEE Computer Society.

[9] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, October 1985.

[10] D. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating System Principles*, pages 150–162, Pacific Grove, California, USA, December 1979.

[11] J. Gray, P. Helland, P. O'Neal, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Quebec, Canada, June 1996.

[12] J. Holliday, D. Agrawal, and A. El Abbadi. Using multicast communication to reduce deadlock in replicated databases. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, Nürnberg, Germany, October 2000.

[13] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*, Amsterdam, The Netherlands, May 1998.

[14] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 707–710, Phoenix, Arizona, USA, April 2001.

[15] F. D. Muñoz, L. Irún, P. Galdámez, J. M. Bernabéu, J. Bataller, and M. C. Bañuls. Globdata: Consistency protocols for replicated databases. In *Proc. of the IEEE YUFORIC'2001*, pages 97–104, Valencia, Spain, November 2001. ISBN 84-9705-097-5.

[16] M. Patiño Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC)*, Toledo, Spain, October 2000.

[17] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'98)*, Southampton, UK, September 1998.

[18] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC'98)*, Andros, Greece, September 1998.

[19] L. Rodrigues, H. Fonseca, and P. Veríssimo. Totally ordered multicast in large-scale systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 503–510, Hong Kong, May 1996. IEEE.

[20] P. Vicente and L. Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. Technical report, Departamento de Informática, Faculdade de Ciência, Universidade de Lisboa, 2002.

[21] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, Nürnberg, Germany, October 2000.

[22] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proc. of the 20th International Conference on Distributed Computing Systems (ICDCS)*, Taipei, Taiwan, Republic of China, April 2000.