FACULDADE · DE · CIÊNCIAS UNIVERSIDADE · DE · LISBOA

A REPLICA CONSISTENCY ALGORITHM FOR GLOBDATA

João Travassos Cabral Martins

Dissertação submetida para obtenção do grau de

MESTRE EM INFORMÁTICA

Orientador:

Luís Eduardo Teixeira Rodrigues

Júri:

Rui Carlos Mendes de Oliveira

João Pedro Guerreiro Neto

Março de 2003

A REPLICA CONSISTENCY ALGORITHM FOR GLOBDATA

João Travassos Cabral Martins

Dissertação submetida para obtenção do grau de

MESTRE EM INFORMÁTICA

pela

Faculdade de Ciências da Universidade de Lisboa

Departamento de Informática

Orientador:

Luís Eduardo Teixeira Rodrigues

Júri:

Rui Carlos Mendes de Oliveira

João Pedro Guerreiro Neto

Março de 2003

Abstract

This thesis addresses the problem of replica management in a distributed object-oriented database system. It presents a protocol to ensure data consistency across the different nodes of the system. This protocol relies on recent advances in group communication techniques, and on the use of atomic broadcast as a building block to help serialize conflicting transactions.

The protocol was implemented in the context of the GLOBDATA project. GLOBDATA was an European IST project designed and implemented a data management middleware tool, named COPLA. The tool offers the abstraction of a global object database repository, supporting transactional access to geographically distributed persistent objects independent of their location. COPLA supports the replication of data according to different consistency criteria. Each consistency criteria is implemented by one or more consistency protocols, that offer different trade-offs between performance and fault-tolerance.

A general description of the algorithm is given, followed by a thorough description of its implementation within the COPLA tool. This implementation is then evaluated against other COPLA consistency protocols.

KEY-WORDS: middleware, distributed systems, object-oriented databases, fault-tolerance, replication, group communication.

Resumo

Esta dissertação aborda o problema de gestão de réplicas num sistema de bases de dados orientado a objectos. Apresenta um protocolo que assegura a coerência de dados entre os diferentes nós de um sistema. Este protocolo baseia-se em desenvolvimentos recentes nas técnicas de comunicação em grupo, e no uso da difusão atómica como um bloco de construção fundamental para ordenar transacções em conflito.

Este protocolo foi concretizado no contexto do projecto GLOBDATA. O GLOBDATA é um projecto europeu no âmbito do qual foi concebida e desenvolvida uma plataforma de gestão de dados, denominada COPLA. Esta plataforma oferece a abstração de um repositório de objectos global, suportando o acesso transaccional a objectos persistentes geográficamente dispersos, independentemente da sua localização. O COPLA suporta a replicação de dados de acordo com diferentes critérios de coerência.

A dissertação apresenta primeiro uma descrição geral do algoritmo, seguida de uma descrição pormenorizada da sua concretização na plataforma COPLA. Esta concretização é de seguida avaliada e o seu desempenho comparado ao desempenho dos restantes protocolos de coerência do COPLA.

PALAVRAS-CHAVE: sistemas distribuídos, bases de dados orientadas a objectos, tolerância a faltas, replicação, comunicação em grupo.

Acknowledgments

First, I would like to thank my thesis supervisor, Luís Rodrigues. His guidance, both during the GLOBDATA project where I worked in the algorithm and its implementation, and during the actual writing of the thesis, were invaluable.

My colleagues in the GLOBDATA project, Pedro Vicente, Ricardo Almeida and Hugo Miranda, have made an enjoyable team to work in, and a source of inspiration, good ideas and even great fun.

The other members of the DIALNP research group, M. João Monteiro, Sandra Teixeira, Nuno Carvalho, Filipe Araújo and Alexandre Pinto, have made a great environment to do research in, providing all sorts of interesting ideas and concepts.

A word of appreciation goes to my fellow researchers of the LASIGE laboratory, and office companions, Norman Noronha, Daniel Gomes, Bruno Martins and Miguel Costa from the XLDB group, for being good friends, and always willing to discuss ideas completely outside their research topics.

Finally, a word to my parents, without their love and support I would have never been able to complete this undertaking.

Lisboa, March 2003
João Travassos Cabral Martins

To my parents.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Replication is often seen as a way to increase availability and performance in distributed databases. However, until recently, strongly consistent database replication was thought to be too heavy to use in production environments. Gray et. al. [13] argued that using conventional distributed locking for replication was not feasible, and proposed other alternatives.

However, recent advances in distributed message ordering algorithms have given rise to efficient total order broadcast algorithms. These algorithms guarantee that messages broadcast to all the nodes within a system are delivered in the same order in all processes. Replication algorithms that take advantage of this property have been developed, which show promising results.

1.1 Objectives

This thesis approaches the problem of replica management in a distributed object-oriented database system. It presents a protocol to ensure data consistency across the different nodes of the system. This protocol relies on recent advances in group communication techniques, and the use of atomic broadcast as a building block to help serialize conflicting transactions.

This protocol was implemented in the context of the GLOBDATA project. GLOBDATA is an European IST project that aimed to design and implement a middleware tool, named COPLA, offering the abstraction of a global object database repository, supporting transactional access to geographically distributed persistent objects independent of their location. COPLA supported the replication of data according to different consistency criteria. Each consistency criteria is implemented by one or more consistency protocols, that offer different trade-offs between performance and fault-tolerance.

1.2 Results

The work described in this dissertation achieved different results. Namely, we were able to:

- Contribute to the COPLA architecture, to enable it to use several different consistency protocols, and make their implementation as independent as possible from the other aspects of the database, like client program interface, or object to table mapping.

- Adapt and implement an efficient mechanism for replica consistency maintenance, within the defined architecture. This protocol is based on existing work [24], adapting it to meet the demands and constraints of COPLA.

- Evaluate the resulting implementation, by comparing it with other protocols implemented for COPLA.

1.3 Thesis structure

This thesis is composed by six chapters, whose structure is outlined below.

Chapter 2 begins with an overview of the more important concepts relating group communications, and then presents an overview of current state-of-the-art database replication algorithms, and a small comparative analysis of each system is made at the end.

Chapter 3 presents the overall architecture of the COPLA system. It explains each of its modular components, their roles within the system, and the general way they work and interact with each other.

Chapter 4 describes the NonVoting consistency algorithm for the COPLA tool. It begins by a general, abstract description of the algorithm, followed by an explanation of the COPLA module where it will be implemented, and finally the actual implementation is discussed. An optimization to the algorithm, taking advantage of a particular facility provided by a communication protocol, concludes the chapter.

Chapter 5 evaluates the NonVoting algorithm, by comparing its implementation with other protocols designed for the COPLA system. A thorough description of the testing procedure is given, as well as the test results. An analysis of these results is then performed.

Chapter 6 concludes this document, and lays some guidelines for future work.

Chapter 2

Related Work

In the database literature, one can find different alternatives to enforce replica consistency. Some authors 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* [12, 11]. 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 [13]. One of the main problems 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, that has been studied in recent works, is to use a replication scheme based on the use of efficient atomic multicast primitives. Systems such as [24, 16, 23] use the message order provided by atomic broadcast to aid in the serialization of conflicting transactions.

In this chapter, several database replication systems that are based on this principle will be surveyed. Before that, we make a brief survey of the main group communication primitives that are used to support the replication protocols.

2.1 Group communication primitives

The work presented in this thesis is a replication algorithm that uses the guarantees offered by a group communication system. As it will be shown, these properties simplify the task of replication considerably. Knowledge about group communications is required, in order to properly understand the systems presented below, as well as the subject of this thesis, and the reasoning behind its design.

2.1.1 System model and failure semantics

Before defining the communication primitives, we must give a clear description of the system model they assume. We will use the model presented in [32].

The system model described here is known as the *asynchronous model*, where there is no bound on message delay, clock drift, or the time necessary to execute a processing step. Its definition is patterned after the ones in [10, 8].

We consider a set of processes $p_i \in P$, connected through a set of communication channels $c_{ij} \in C$ (c_{ij} connects p_i to p_j). Communication is asynchronous, i.e. there is no bound on communication delays.

A process is a finite deterministic automaton (with possibly infinitely many states) [10, 8]. A process may either be correct or failed. A *correct* process executes steps from its automaton. When a correct process crashes, we say that it failed. A *failed* process stops executing (Byzantine errors [21] are not considered).

For the purposes of group communication, a crashed process does not recover. A recovering process is given a new identifier, and is treated by the group communication system as a new process. This "new" process may have access to the state of the "old" process (if part or all of its state is saved on permanent storage), but synchronizing this state with the rest of the system is handled at the application level.

Communication channels c_{ij} are assumed to be reliable (a message m sent by process p_i to process p_j will eventually reach p_j if neither p_i nor p_j fail) and to deliver messages in FIFO order.

It is a well know result that in this model, it is impossible to distinguish a crashed process from a slow process or a process connected through a slow channel. This observation has led to the impossibility result [10] on distributed consensus[1], and to the definition of the synchronous model with bounded delays. The synchronous model has some attractive features, but has the drawback to lead to poor performances because worst case assumptions must be made on communication delays.

The model we shall consider in this thesis is called *virtually synchronous* [4, 5]. It can be defined as the asynchronous model completed with an *imperfect failure detector* [8], $\diamond S$, that has the following properties:

- Strong Completeness: Eventually every process that crashes is suspected by every correct process.

- Eventually Weak Accuracy: There is a time after which some correct process is never suspected by any correct process.

With this failure detector, we can now construct interesting services, that form our group communication system: a group membership service, that summarizes the failure detection information into *views*, and communication primitives.

[1]Informally, in the consensus problem, a set of processes must agree on a common value, chosen from the values each participant initially proposes. The problem is solved when every non-faulty participant reaches a decision.

2.1.2 Group membership service

Consider processes structured into groups g used in the context of multicasts: each multicast m is addressed to a group g, i.e. to every process $p_i \in g$. For a group g of processes, the group membership service GMS is assumed to construct a sequence of *views* $v_0(g), v_1(g), ..., v_i(g), ...(v_i(g) \subseteq P)$ corresponding to the successive composition of the group g as perceived by GMS, and to deliver each view $v_i(g)$ to the members of the view. The sequence of views satisfies the following conditions:

- if process $p \in g$ has crashed, then GMS will eventually detect it and define a new view from which p is excluded (if p recovers after the crash, it comes back with a different process id, and is thus considered as a new process);

- if a process p joins the group g, FD will eventually detect it, and define a new view including p;

- if a view $v_i(g)$ is defined and $p \in v_i(g)$, then either p eventually receives view $v_i(g)$, or $\exists k > 0$, such that $p \notin v_{i+k}(g)$;

- $\forall p, q \in g$, if p and q receive views $v_i(g)$ and $v_j(g)(i \neq j)$, then p and q both receive $v_i(g)$ and $v_j(g)$ in the same order.

2.1.3 View synchronous multicast

In the context of the model defined above, the semantics of view synchronous multicast are defined by three properties [36], listed below.

Consider a group g, a view $v_i(g)$ and a message m multicast to g.

- Delivery Integrity: If a process delivers m in $v_i(g)$, then some process has multicast m.

- No Duplication: For any message m multicast by a process, m is delivered at most once at each process in $v_i(g)$.

- Regular Virtual Synchrony: If $\exists p \in v_i(g)$ which has delivered m in $v_i(g)$ and has installed a view $v_{i+1}(g)$, then all processes $q \in v_i(g)$ which have installed $v_{i+1}(g)$ have delivered m before installing $v_{i+1}(g)$.

Intuitively, these properties state that, when a view change occurs, we can be sure that all processes that have passed from the previous view to the current one (i.e., all correct processes) have received the same set of messages. This is why the system is named *view synchronous*: a view change defines a *synchronization point* between all processes.

2.1.4 View synchronous atomic multicast

View synchronous multicast guarantees that all correct processes will receive the same set of messages. However, it says nothing about the order of such messages. This primitive builds upon view synchronous multicast, defined above, guaranteeing that all processes will see the same message order.

View synchronous atomic multicast (also called atomic multicast, or ABCast) semantics can be defined by an additional property [36] (in addition to the three defined above), *total order*.

Let m and m' be two messages that are ABcast in view $v_i(g)$.

- Regular Total Order: If any two correct processes $p, q \in v_i(g)$ deliver both m and m', they deliver them in the same order.

Intuitively, the total order property states that all processes see the same order of messages.

2.1.5 Uniformity

Both the view synchronous and the atomic multicast primitives can have two versions: uniform and non-uniform (the definitions given above are non-uniform).

From the model definition above, we recall that a correct process is one who "survives" a view change, i.e., $p \in v_i(g) \wedge p \in v_{i+1}(g) \Rightarrow p$ is correct in $v_i(g)$. If we examine the definitions given in sections 2.1.3 and 2.1.4, they apply their properties only to correct processes. As such, we say they are *non-uniform*.

Uniform versions of these primitives contemplate in their definitions both correct and failed processes. The uniform versions of view synchronous multicast and atomic multicast are given below (only the properties that changed are shown, the others remain the same).

Uniform view synchronous multicast:

- Uniform Virtual Synchrony: if $\exists p \in v_i(g)$ which has delivered m in $v_i(g)$, then all processes $q \in v_i(g)$ which have installed $v_{i+1}(g)$ have delivered m before installing $v_{i+1}(g)$.

Uniform atomic multicast:

- Uniform Total Order: if any two processes $p, q \in v_i(g)$ deliver both m and m', they deliver them in the same order.

What is the practical difference between regular and uniform versions of the primitives? With regular ABCast, the following situation can occur: a process $p \in v_i(g)$ receives a message m, processes it, stores the result on disk and then crashes. Meanwhile, all the other processes had not yet received m, and upon detecting the failure of p, install a new view $v_{i+1}(g)$, and decide a different order for messages (they deliver a message m' before m). This means that the information stored by p before crashing will be inconsistent with the rest of the system, making any subsequent reintegration process extremely difficult.

As such, in GLOBDATA, we will use the uniform versions of the primitives.

2.1.6 Atomic commit protocol

Some systems use a specialized communication primitive to decide if a transaction should commit or abort. This primitive is called atomic commit, and works by collecting a *vote* (yes or no) of each participant regarding the fate of a transaction. An unanimous *yes* vote will commit the transaction, a site voting *no* will abort the transaction.

More formally, an atomic commit protocol can be defined by the following properties [14]:

- Agreement: For all $p,q \in v_i(g)$, p and q reach the same decision.

- Termination: Every process $p \in v_i(g)$ that also installs $v_{i+1}(g)$ eventually decides.

- Validity: If a process $p \in v_i(g)$ decides *commit*, then all processes $q \in v_i(g)$ have voted *yes*.

- Non-Triviality: If all processes $p \in v_i(g)$ vote *yes* and also install view $v_{i+1}(g)$, then all processes p eventually decide *commit*.

Informally, the agreement property states that all processes will reach the same decision, the termination property that all correct processes will eventually decide, the validity property that a process will only decide commit if every process voted yes, and the non-triviality property that if all correct processes voted yes, then all processes will eventually decide commit.

2.2 Postgres-R

The Postgres-R [20] system is a modification to the PostgreSQL [27] database kernel, that adds a transparent replication layer within the database. This layer uses Ensemble [15] as the underlying group communication system, making use of its total order primitive.

Transaction messages are multicast using the total order primitive, and the order of their delivery is used to determine the serialization order of conflicting transactions. The authors presented several such protocols in [18], each of them guaranteeing different transaction isolation semantics. The protocol that ensures transaction serializability can be briefly described as follows:

1. **Local read phase:** Perform all read operations locally, acquiring the appropriate local read locks. Execute write operations on shadow copies (no write locks are acquired).

2. **Send phase:** If T_i is read only, then commit. Else bundle all writes into write set WS_i and ABCast it to all sites including the sending site (same delivery order at all sites).

3. **Lock phase:** Upon delivery of WS_i, request all local locks for WS_i in an atomic step:

    (a) For each operation $w_i(X)$ on item X in WS_i:

        i. Perform a conflict test: if a local transaction T_j has a granted lock on X and T_j is still in its read or send phase, abort T_j. If T_j is in its send phase, then multicast the decision message *abort* (decision messages are not ordered).

        ii. If there is no lock on X, grant the lock to T_i. Otherwise enqueue the lock request directly after all locks from transactions that are beyond their lock phase.

(b) If T_i is a local transaction, multicast the decision message *commit* (no order requirement).

4. **Write phase:** Whenever a write lock is granted apply the corresponding update. A local transaction can commit and release all locks once all updates have been applied to the database. A remote transaction[2] must wait until the decision message arrives and terminate accordingly.

In this protocol, the total order is used to serialize write/write conflicts at all sites. Read/write conflicts are also detected during the lock phase (3.a.i). Since read operations are only seen at the local site, local readers are aborted when a conflicting writer arrives, thus avoiding deadlocks and inconsistent solutions.

Due to the choice of implementing the replication layer at the core of the database, the authors had the chance (and burden) of inspecting closely the transaction execution and locking strategies of PostgreSQL, and to enhance / modify them where suitable. For instance, traditional PostgreSQL uses logical locking at the relation level, but a more efficient approach was taken by Postgres-R, tuple-based locking. This kind of locking is more fine grained, and allows for greater parallelism between transactions. Index locking was also modified, to avoid deadlocks with remote transactions.

2.3 University of Minho's partial replication

This work, described in [33], presents a replication algorithm that models the database as a state machine [25] (from now on refered to as DBSM approach). This model assumes that a transaction is first executed locally at a site, and interaction with other sites occurs only when the client requests for the transaction

[2]A remote transaction is a transaction that was initiated on another node.

commit. At that time, the transaction's updates and some control structures are propagated to all database sites.

The novelty of this work is that, unlike previous works, it considers *partial replication*, that is, each node within the system possesses a partial copy of the database.

The authors base their algorithm on two communication primitives: a fast atomic broadcast protocol, that provides optimistic message delivery [26], and a Resilient Atomic Commit (RAC) protocol. This protocol provides the same guarantees as an atomic commit protocol [14], but allows for *a)* nodes having only partial copies of the database, and *b)* allows that participants in the protocol decide commit even if some nodes are suspected to have failed or crashed. Briefly, the properties of RAC can be described as their difference to the properties offered by regular atomic commit protocols (see **??**).

The RAC differs in its validity and non-triviality properties, considering that $Items(t)$ is the set of objects read by a transaction t, and $Sites(x)$ is the set of sites that have a copy of object x:

Validity If a site declares *commit* for t, then for each $x \in Items(t)$, there is at least a site in $Sites(x)$ that voted *yes* for t.

Non-Triviality If for each $x \in Items(t)$ there is a site $s \in Sites(x)$ that votes *yes* for t and it is not suspected, then every correct site eventually decides *commit* for t.

The replication protocol itself is a modification of the one employed in the DBSM approach. In DBSM, a transaction executes locally at a site, and when the commit is requested, an atomic broadcast message is sent, containing the transactions read and write sets. When the message is delivered from total order, a certification test is performed, to ensure that the transaction is valid. If the transaction passes the certification test, it can be committed. Since the test is deterministic, and all sites receive transaction messages by the same order, all site will reach the same decision.

This approach does not work with partial replication. The certification test gives different results at each site, precisely because sites hold only partial copies of the database. As such, the certification step is modified. When a transaction is delivered from total order, the certification test is performed on it. Its outcome will then be the input to the RAC protocol. If the outcome of the RAC protocol is *commit*, then the transaction may be committed, otherwise it is aborted. This final commit step is in effect enforcing the certification step against the full database, in a distributed fashion. The use of RAC instead of a traditional atomic commit protocol is crucial: if a traditional protocol was used, the system would become less resilient to failures, since the suspicion of a single node would cause a transaction to abort.

The addition of an extra communication step at the end of each transaction naturally incurs a penalty in performance. To overcome this an optimistic atomic broadcast protocol is used. When a transaction message is optimistically delivered, the RAC protocol is initiated, therefore overlapping its execution with the atomic broadcast. If the final order confirms the tentative one, this overlapping incurs in substantial time savings. Should the ordering of the two deliveries mismatch, both the certification and the atomic commit executions are discarded and the process is repeated for the final order.

2.4 Scalable middleware architecture

This system, presented in [17], proposes a middleware replication architecture, which sits between clients and a conventional database system, and is composed of several modules.

The transaction manager implements the replication protocol. It coordinates with the other sites through the exchange of messages, interacts with client applications, and submits transactions to the local database. The communication manager is the interface between the transaction manager and the group communication system, Ensemble [15]. The transaction manager interacts with the database through the connection manager. In this system the database used is PostgreSQL [27].

The replication protocol, described in detail in [23] uses the concept of *conflict classes* for load partitioning. The initially available data is divided into disjoint conflict classes, which can be as small as a tuple, or as large as an entire table. These basic conflict classes are then grouped into *compound conflict classes*. These compound conflict classes do not need to be disjoint, but they are required to be distinct. The load is divided based on compound conflict classes. Each compound conflict class has a *master* or primary site. A transaction T can access any compound conflict class, and it is assumed that the accessed class is know in advance (C_T).

A transaction T is considered to be *local* to the master site of C_T, and *remote* everywhere else. For example, consider two sites N and N', and two conflict classes C_x and C_y, with N being the master site of the compound conflict class C_x and N' the master site of C_y and C_x, C_y. A transaction updating C_x is local to N and remote at N', a transaction updating both C_x and C_y is local at N' and remote at N. A query over any of the three basic conflict classes can be local to either N or N'. For concurrency purposes, a simple locking table is used: each site has a queue CQ_x associated to each conflict class C_x. Upon delivery of a transaction T that accesses a compound conflict class C_T, each site adds T to the queues of the basic conflict classes contained in C_T.

The replication scheme makes use of an advanced total order algorithm [26] that provides optimistic message delivery. This means that the communication protocol first makes a tentative deliver of the message, which can be considered a good estimate on its final, definitive order. The protocol later delivers the same message in its final order. This tentative delivery is used by the replication layer to begin executing transactions earlier, building on the assumption that the ordering estimate is accurate most of the time.

A transaction is processed as follows. At its start, T is broadcast to all sites. Only Ts master site executes T. After completing the execution, the master site broadcasts a commit message to all other sites and piggy-backs to this message the result of all modifications performed by T (i.e., the write set of T). Upon receiving these modifications, a remote site proceeds to install the changes directly without having to execute the transaction.

The replication algorithm employed can then be described in terms of the events that occur during the lifetime of a transaction T – T is optimistically delivered (opt-delivered), T is delivered in its final order (to-delivered), T completes execution, and T commits.

When a transaction T is opt-delivered, it is queued in all the basic conflict classes it belongs to. This is done at all sites. At the master site of T, once T is the first transaction in all queues, it is submitted for execution.

When T finishes execution (this happens only at the master site of T), the to-delivery of T might have taken place. If it has, then T can be committed, since it is the first transaction in all its queues, and there cannot be a conflicting transaction ordered before T neither in the tentative order or the definitive order. The commit message is then broadcast to all sites. If the transaction has not been to-delivered, it is marked as executed. Waiting for the to-delivery before committing the transaction is necessary to avoid conflicting serialization orders at different sites.

When the to-delivery of T is processed at T's master site, T may or may not have been executed. If it is executed, then T is the first transaction in all of its queues, and there is no mismatch between its tentative and final order. As such, T can be committed, and the commit message is broadcast to all sites. If the transaction has not yet executed or is not local, the protocol checks for mismatches between the tentative and final orders of T, which would lead to incorrect executions. If any conflicting transaction T' has been opt-delivered before T but not yet to-delivered, then it is incorrectly ordered before T in the queues they have in common. Hence T and T' must be reordered, such that T is scheduled before T'. This would happen if T was already executing or had already completed its execution and was waiting to be committed. However, note that the abort only occurs at the site where T' is local (on all other sites T' is remote; thus, reordering only requires to switch the transactions in the queues). Moreover, the probability for this situation to occur is quite low as it requires that messages get out of order and that the messages that are out of order correspond to transactions that conflict and one of the transactions is being or has been executed at that site.

A local transaction commits by submitting a commit to the database. At remote sites, the updates received in the commit message are applied after the transaction has been to-delivered to ensure that the updates are applied following the total (serialization) order. When the updates are applied and the commit has been submitted to the database, the transaction is removed from the queues. This protocol guarantees 1-copy serializability.

2.5 The CNDS large-scale system

This system, developed at CNDS [7] aims to provide a software layer that sits between client applications and the database, providing replication in a transparent manner, both to clients and to the database. It is described in [2].

The system is separated in two layers: a replication server and a group communication toolkit, named Spread [1].

Each of the replication servers maintains a copy of the database. When a client requests an action from the database, the replication servers agree on the order of the actions to be applied to the databases. As soon as the final order of an action is known, it is applied to the database. The replication server that received the request returns the database reply to the client. The replication servers use the group communication layer to disseminate the actions among the servers group and to help reach an agreement on the global final order of the set of actions.

The group communications toolkit offers primitives according to the Extended Virtual Synchrony model [22]. The replication algorithm relies particularly on the Safe Delivery[3] property.

The replication server is composed of three modules:

- A *replication engine*, that includes all of the replication logic, an can be applied to any database or application. It is based on the algorithm presented in [2].

- A *semantics optimizer*, that decides whether to replicate transactions and when to apply them based on the required semantics and the actual content of the transaction.

[3]Informally, this property ensures that, if a process delivers a message, then that message will be received and delivered by every other process in the same view, unless that process fails. The reader should refer to [22] for a more formal description.

- A *database specific interceptor*, that interfaces the replication engine with the DBMS client-server protocol. This interceptor allows existing PostgreSQL applications to be used seamlessly. Neither the applications nor the database need to be changed. A similar interceptor could be built to support other databases.

The replication engine uses the group communication toolkit to provide global persistent consistent order of actions in a partitionable environment. This is accomplished using a synchronous disk write per action at the originating replica, the minimum operation required to cope with a crash-recovery failure model.

In the presence of network partitions, the replications servers identify at most a single component of the servers group as the primary component. When a server belongs to a primary component partition, it can apply actions immediately to the database, upon their delivery by the group communications system. When in a non-primary component, the actions are applied according to the semantics optimizer component, described below. Updates generated in non-primary components will be propagated as network connectivity changes. These updates will be ordered in the final persistent consistent order upon the formation of the first primary component that includes them.

The semantics optimizer provides the ability to support various consistency models, according to varying application requirements. In the strictest model of consistency, updates can be applied to the database only while in a primary component. When the global persistent order of actions has been determined. However, read-only actions do not need to be replicated, thus they can be answered immediately.

The replication algorithm takes into account the existence of network partitions. As such, at any given time it considers a set of sites that form a majority partition, called the primary component. Only the sites in this primary component are able to perform transactions.

The algorithm divides actions (transactions) to be performed in *colors*. These colors are:

Red action An action that has been ordered within the local component by the group communication layer, but for which the server cannot, as yet, determine the global order.

Green Action An action for which the server has determined the global order.

White Action An action for which the server knows that all of the servers have already marked it as green. These actions can be discarded since no other server will need them subsequently.

Transactions are marked with a certain color according to the algorithm, that can be described in terms of the state that a server is in. There are four possible states.

Prim State. The server belongs to the primary component. When a client submits a request, it is multicast using the group communication to all the servers in the component. When a message is delivered by the group communication system to the replication layer, the action is immediately marked green and is applied to the database.

NonPrim State. The server belongs to a non-primary component. Client actions are ordered within the component using the group communication system. When a message containing an action is delivered by the group communication system, it is immediately marked red.

Exchange State. A server switches to this state upon delivery of a view change notification from the group communication system. All the servers in the

System	Architecture	Scale	Total Order Algorithm
Postgres-R	Integrated in DB	LAN	Conventional TO
UM	Integrated in DB	LAN/Wide	Optimistic + RAC
SMA	Middleware layer	LAN	Optimistic TO
CNDS	Middleware layer	Wide	Custom

Table 2.1: Summary of system characteristics

new view will exchange information allowing them to define the set of actions that are known by some of them but not by all. These actions are subsequently exchanged and each server will apply to the database the green actions that it gained knowledge of. After this exchange is finished each server can check whether the current view has a quorum to form the next primary component. This check can be done locally, without additional exchange of messages, based on the information collected in the initial stage of this state. If the view can form the next primary component the server will move to the Construct state, otherwise it will return to the NonPrim state.

Construct State. In this state, all the servers in the component have the same set of actions (they synchronized in the Exchange state) and can attempt to install the next primary component. For that they will send a Create Primary Component (CPC) message. When a server has received CPC messages from all the members of the current component it will transform all its red messages into green, apply them to the database and then switch to the Prim state. If a view change occurs before receiving all CPC messages, the server returns to the Exchange state.

2.6 Comparative analysis

Despite the fact that all of these systems are similar to each other, since they base their replication mechanism on the properties of virtual synchrony and total order, they also have a number of significant differences. Table 2.1 summarizes each of the systems main characteristics.

The first system, Postgres-R, is built into the core of the database. This feature allows it to take advantage of the knowledge of the inner workings of the database to achieve greater parallelism in transaction processing. It also enables to use more efficient message mechanisms (propagating actual table changes instead of the transaction request, which would have to be re-executed).

This option comes at a cost, however. It's positioning means that it will be tied to the particular database used, making it impossible to change. Also, implementations using commercial databases become very difficult.

Continuing in the line of replication layers integrated within the database, the system proposed by the Univ. of Minho proposes an enhancement to the work of Postgres-R: it supports partial replication, that is, instead of all nodes having a copy of the full database, they maintain solely a partial copy. This brings performance benefits, since a lot of applications exhibit a high locality access pattern (i.e., they access only a portion of the database). By partitioning the database carefully, performance gains can be achieved.

This facility comes at a cost, however: an additional communication step is required to validate a transaction, because each node does not possess enough information to do this on it's own. The authors offset this by using an optimistic total order algorithm, which enables the system to start the validation step sooner.

The work of Peris et. al. proposes a different architecture from the previous two. Its replication layer is placed outside the database, between the client applications and the database. Neither of these components realize that replication exists. This approach brings a new degree of freedom to the system, but at a cost: there is much less knowledge of the effects of transaction, and as such the ordering of messages must be much more strict (Postgres-R uses its “inside knowledge” to reorder some transactions). Because of this, they also focused on the use of efficient total order algorithms, which provide optimistic delivery, to enhance the performance of the system.

The final system described here, by the CNDS group, also has a middleware architecture, sitting transparently between the database and client applications. The novelty of this work relies in their consideration for wide scale applications, and the problem of network partitions. This has lead the authors to use a significantly different total order algorithm, and greater efforts must be made to ensure the system does not become incoherent.

In summary, despite the common idea of resolving transaction conflicts with the use of total order, the actual implementations can be quite different. However, all of these systems share a common trait: their performance depends as much on the replication algorithm itself as on the communication protocol used. This can be seen, for instance, comparing the first and second systems presented: both Postgres-R and the system of Peris et. al. can benefit from similar communication protocol improvements (in fact, the authors of both these systems have cooperated with each other), despite the apparent difference in architectures. However, the difference between the CNDS system and the Peris et. al. system is patent, in terms of communication protocols and replication strategy employed, despite the similar architecture.

We can also study these systems according to the classification presented in [37]. This classification proposes three parameters:

Server Architecture: Where transactions are executed. The possibilities are *update everywhere*, when updates are performed on all servers, and *primary*

System	Server Architecture	Server Interaction	Tx. Termination
Postgres-R	update everywhere	constant	voting
UM	update everywhere	constant	voting
SMA	update everywhere	constant	non-voting
CNDS	update everywhere	constant	non-voting

Table 2.2: Replication strategy classification

copy, when updates are performed on a server which owns the modified data, which will subsequently propagate changes to the replicas.

Server Interaction: The degree of communication between replication servers during transaction execution. The alternatives are *constant interaction*, when a constant number of messages is exchanged, independently of the number of operations performed by each transaction, and *linear interaction*, when the number of messages exchanged grows linearly with the number of operations a transaction performs.

Transaction Termination: The way the transactions terminate, i.e., how atomicity is guaranteed. The options are *voting termination*, when there is an additional round of messages to coordinate the replicas, and *non-voting termination*, when each site can decide on their own whether to commit or abort a transaction.

In light of these parameters, we can classify each of the systems presented according to Table 2.2.

The first conclusion we can draw from this table is that all systems use a constant interaction scheme. This is clearly, in the general case, the most efficient design, because the number of messages used will be independent of the number of operations performed by a transaction.

Secondly, all systems use the update everywhere approach. This approach sacrifices some performance for much higher availability and easier re-integration of failed nodes. Even the UM system, which supports partial replication, uses update everywhere: it updates all sites owning copies of changed database items.

The main difference among these systems depends on the way they terminate the transactions.

The Postgres-R system uses a voting strategy.[4] The choice was made to disseminate less information (only the write-set of each transaction) and use one final round of messages to confirm the transaction's outcome.

The UM system also uses a confirmation step. However, this system tries to minimize the impact of using two rounds of messages by starting the confirmation step earlier, using the results of the optimistic atomic broadcast as an input to the RAC confirmation protocol.

Both the SMA and CNDS systems use a non-voting approach, propagating enough information to allow each node to decide for itself whether a transaction should commit or abort. SMA also takes advantage of optimistic delivery to speed up the confirmation process when possible.

2.7 Summary

This chapter presented a survey of a few replication systems that make use of atomic broadcast primitives as an aid in ordering conflicting transactions. To aid in the understanding of these systems, a brief survey of the used primitives was also presented. The next chapter will describe the overall architecture of the GLOBDATA project's middleware tool, COPLA.

[4]The reader should refer to [18] for a complete description of the algorithm.

Chapter 3

The GlobData Architecture

The aim of the GLOBDATA project is to build a middleware tool, named COPLA, 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 area network spanning different geographic locations. To support a diversity of environments and workloads, COPLA provides a number of replica consistency protocols. This effort was undertaken by a team of academic and industrial partners, under the European Union IST program (IST-1999-20997). The academic partners are the Instituto Tecnológico de Informática de Valencia (ITI), Spain; the Faculdade de Ciências da Universidade de Lisboa (FCUL), Portugal and the Universidad Pública de Navarra (UPNA), Spain. The industrial partners are GFI Informatique (GFI), France; and Investigación y Desarrollo Informático (IDI EIKON), Spain.

3.1 Operating scenario

The architecture of a software system like COPLA is naturally influenced by the expected usage pattern and by the network's layout on which it will run.

It is assumed that different areas, formed by groups of nodes, exist throughout the network. Nodes belonging to the same area are physically close. One example in the real world could be an enterprise with delegations in different cities, the computers in each delegation forming an area. To adapt itself to this model, the system is organized as follows:

- There is a set of special nodes, one per area, called managers. Each manager is in charge of serving the COPLA applications on its own area.
- Applications access only its nearest manager, i.e., the manager for its area. Each manager uses one database as a tool that provides data persistence. This fact (that there are many physical databases) is completely transparent to the applications.
- Each physical database holds, in theory, an exact replica of all persistent objects. In practice, the consistency protocol used will determine the information stored at each database.
- There is a protocol run among the managers (consistency protocol) to coordinate them so as they can:
  - provide up-to-date consistent objects to client applications.
  - distribute the results of transactions made by a client application.
  - resolve conflicts by concurrent transactions running on the same or on different areas.
  - provide some degree of fault tolerance, taking advantage of the fact that data is replicated.
- Transactions issued by a client are executed by its corresponding manager using data exclusively on its database (Again, the consistency protocol used will determine the exact details).

- If a manager fails, the system should continue operations as much as possible, always ensuring data integrity and consistency:
  - user applications that were being served by a failed manager will not be able to continue until it recovers (unless they can connect to another manager).
  - if the failed manager was the only one storing the up-to-date version of an object, user applications requiring that object will be suspended until it recovers. (Each consistency protocol will make an effort to make available two or more up-to-date replicas of each object).

3.2 COPLA components

The main components of a COPLA manager node's architecture are depicted in Figure 3.1. 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 [6] 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 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 Figure 3.1). This allows COPLA to be configured according to the characteristics of the environment where it runs.

As it is shown in Figure 3.1, the system is composed of three main modules:

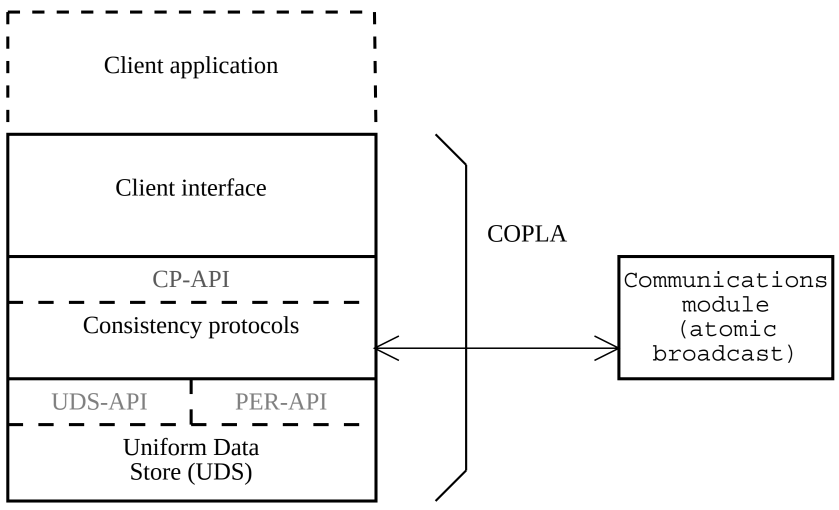


Figure 3.1: COPLA architecture

Client Interface: This module is what is exposed to the COPLA applications programmer. It is responsible for presenting an interface for developing client applications, and for communicating with the two other modules (consistency protocols and the UDS (uniform data store)) to perform its tasks. Clients access this module through CORBA calls.

Consistency Protocols: This module is in charge of maintaining the data stored in different replicas mutually consistent. The consistency protocols communicate with other nodes in the system using the communication module. The different protocols offer an uniform interface to the above layer (labeled CP-API in Figure 3.1). This allows COPLA to be configured according to the characteristics of the environment where it runs.

Uniform Data Store: 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 a dedicated interface, the PER-API.

The architecture of each module is sketched out in the next few sections, in a bottom-up order, to facilitate discussion.

3.3 The Uniform Data Store

As stated above, the UDS layer is responsible for the persistent storage of objects, and the interpretation of user queries.

In detail:

- Opening and closing transactions in the local object repository.
- Accessing objects or collections of objects in the local object repository.
- Committing and aborting transactions.
- Retrieving read and write sets in order to maintain data consistency by COPLA.
- Building transaction updates, containing the changes made by a transaction.
- Applying transaction updates.

To store objects in an efficient and transparent (to upper layers) manner, the UDS uses Proxy and Packet Objects. To accept user queries, it interprets GOQL statements. To accept repository specifications, it uses GODL schemata.

3.3.1 GODL and GOQL

GODL stands for GLOBDATA Object Definition Language. It is a declarative language used to define persistent objects for the COPLA tool. It is based on the Object Definition Language [6]. Using GODL, a programmer specifies the classes that will be present in the repository, and the relations between them. This definition will then be processed by a compiler, that will generate two outputs: a set of proxy objects (described below), and an SQL schema, to be placed in the underlying RDBMS.

The GOQL (GLOBDATA Object Query Language) language is used to present queries to the COPLA system. It is based upon Object Query Language [6], the companion standard to ODL. The UDS is in charge of translating, in run time, a GOQL query to a series of SQL statements that retrieve the desired object data from the RDBMS.

3.3.2 Proxy and Packet Objects

Here, the term proxy is used to refer to a local instance of a remote object. Since the objective is to maintain a high degree of transparency relative to the client application, all local transformations of object properties have to be reflected with a high degree of effectiveness in the uniform data store layer.

A proxy object is used to represent, on the programmer's side, an instance of a class (i.e., an object) that was defined through GODL. These instances are created as a result of a GOQL query to the system. It contains methods for manipulating the object's attributes and relations.

The proxy object is also responsible for persistently storing it's state. In order to do this, each proxy object is able to generate a representation of itself, called a packet object. This representation is designed to be interpreted by the UDS efficiently, for storage in the database, and for transmission over the network.

A packet object contains all the required information for an UDS at a COPLA site to instantiate a proxy object, to make it's data accessible to a client.

3.4 The consistency protocol layer

The consistency protocol (CP) layer is designed as a common interface for several different protocols. It is mostly a "placeholder" module, with little more than a specified API. Each consistency protocol then implements this API in order to perform its work.

Since the subject of this thesis is a consistency protocol designed to fit this interface, the CP layer and its interface will be described in greater detail in the following chapter.

3.5 The communications module

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 [35].

The protocol is an adaptation of the hybrid total order protocol presented in [29]. 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 their 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 [29] 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 [35] also supports the optimistic delivery of (tentative) total order indications [9, 26]. 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. However, ensuring perfect failure detection is very difficult without custom hardware, and as such the algorithms may provide inconsistent results. 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.

3.6 The client interface library

The client interface library (CLIB) is the visible part of COPLA to application programmers. It exposes an API, with which developers manipulate objects. The API is organized about the basic concept of a session. A session is

- A set of persistent objects.
- A set of threads that manipulate those objects.

It is the CLIB's responsibility to manage concurrent thread accesses to objects, instantiation of object proxies, and generally translating user requests into calls to the lower levels. These calls are either directly forwarded (for instance, a GOQL query is directly forwarded to the UDS), or are a result of user's requests (upon commit, the CP layer will have to be called to confirm the request).

In order to maintain the system efficient, the CLIB also provides a transparent caching mechanism. For example, if a user performs a query that returns one thousand objects, and then iterates over the resulting collection, the CLIB will retrieve objects in batches, and keep them in memory. So if the user backtracks on a collection, no database query will be necessary.

3.7 Interaction among components

As shown above, COPLA is a modular system, made of several interacting components. Here the interactions among them are detailed.

3.7.1 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.

3.7.2 Interaction with the consistency protocols

The CP-API 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 (some protocols may not always maintain the most recent version of an object 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 here.

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 requests 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.

3.8 Summary

This chapter described the overall architecture of the COPLA system, detailing each of its modular components, their roles within the system, and the general way they work and interact with each other. The next chapter will present the NonVoting consistency protocol algorithm for the COPLA middleware.

Chapter 4

The NonVoting Protocol For GLOBDATA

In this chapter, a replication algorithm for the COPLA tool is described. First, a generic strategy for exploiting group communication primitives for replication is presented. Then, an overview of the algorithm is presented, within the architectural description given in Chapter 3. A more thorough explanation of the interface between the different modules is given, and a complete exposition of the algorithm, matching that interface, is made. The code that implements the algorithm is reviewed. In the final section, an optimization to the algorithm is discussed, that takes advantage of a particular capability present in the communication protocol used in COPLA.

4.1 Architectural challenges

The GLOBDATA project is characterized by a unique combination of different requirements that make the design of a consistency protocol a challenging task. Namely, it 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 [26]).

- *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.

4.2 Replication using atomic broadcast

The use of group communication primitives (atomic/virtual synchronous broadcast) as a base for the construction of replicated database systems reduces complexity in respect to data consistency, fault tolerance and lock management. It also provides the system with a set of semantically strong primitives, which allow for simple implementations of strong consistency models [19]. Still, within group communication there are several possible approaches, each offering different semantics and levels of performance.

A transaction system must honor the ACID properties: Atomicity, Consistency, Isolation and Durability. The replication of the database poses particular problems to the implementation of each of the above properties. Below is described how the algorithm honors those properties within the COPLA system using an atomic broadcast primitive.

To satisfy the atomicity property, one must extend to the different hosts the requirement that, in spite of failures, either all of the operations in a transaction are performed or none of them are. For each transaction COPLA executes the sequence of operations in the scope of a transaction local to the delegate server (the host mediating the transaction between the client and the group of database servers). When the transaction attempts to commit, the set of operations to be performed is broadcast in a single message that cannot be partially delivered. The responsibility of ensuring the atomicity of the transaction at each host is delegated to the local transaction server.

The consistency property states that the execution of interleaved transactions is equivalent to a serial execution of the transactions in some order [3]. When using a distributed database server the problem is extended to the need of ordering transactions executing concurrently on different servers. The proposed algorithm serializes transactions by using total order protocols that ensure the delivery of all messages in the same order to every participant. It will rely on a total order uniform delivery protocol that guarantees ordered delivery even in the presence of failures of some participants.

Isolation of operations to concurrent transactions is ensured locally by keeping the set of database updates in memory, private to each transaction. To the remaining hosts this property is straightforward: remote hosts are not aware of uncommitted transactions.

Durability is locally provided by the transaction database server used by the system. When one of the hosts recovers from a crash it will synchronize its state with the remaining to ensure that he will learn of every committed transaction.

4.3 Replication strategies

Using, as in the previous chapter, the classification of database replication strategies introduced in [37], the strong consistency protocol for COPLA that is to be presented can be classified as belonging to the "update everywhere constant interaction" class. It is "update everywhere" because it performs 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 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 protocol described below explores one option of the third degree of freedom: the way transactions terminate (voting or non-voting).

4.4 The Non-Voting Protocol

4.4.1 Description

This protocol, initially described in [31, 30], is a modification of the one described in [24], altered to use a version scheme for concurrency control [3], and adapted to the COPLA transactional model.

The protocol uses, for each object, a version number. This version number 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 version number 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 [3], 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 these 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 o 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' must be aborted, because it has read data that was later modified (by a transaction t that modified o and committed before t', thus increasing o's version number).

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 list of objects read by the transaction and its version numbers, the list of objects written by the transaction, and the transactions modifications to 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 previously committed 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 4.1 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. 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

```
• 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 conflicts with committed transactions
              i. Abort.
        (b) else (consistent transaction)
              i. Abort all transactions conflicting with t
             ii. Commit the transaction.
```

Figure 4.1: Non-voting protocol

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)i. 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.[1]

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.

[1]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.

4.4.2 Using versions for concurrency control

As stated above, the presented algorithm is a modification of a previous work [24]. The main modification consisted in replacing the original lock-based concurrency control by a version number scheme.

This modification is important, and was made due to a requirement of the original protocol (hereby referred to as DBSM). Suppose the following scenario: two transactions, t_a and t_b, which conflict with each other, are executing on the same node. Then, both t_a and t_b request commit. The atomic broadcast protocol will take both messages, and deliver t_a before t_b.

In DBSM, t_b would hold write locks over objects that would be overwritten by t_a. However, t_b could not simply be aborted, since its commit request had already been sent. We would then have two possible scenarios:

1. If t_b would latter commit, then its writes would overwrite t_a's writes, and therefore t_a would not need to request those locks, or process the corresponding updates. This reasoning is known as the Thomas Write Rule [34].

2. If t_b would latter abort, then the database would have to be restored to a state without t_b, for example by applying t_a's redo logs to the database.

While scenario 1 would pose no problems to the COPLA architecture, scenario 2 would imply the existence of a mechanism that would allow a committed transaction to be undone. This would pose a severe problem, because SQL does not have such a construct, and usually databases only provide this functionality through an administrative interface.[2] As such, the algorithm could not be directly transposed to COPLA.

The choice of using a versioning scheme solves this problem. Since a transaction's updates are not applied until their order is determined by atomic broadcast,

[2]Since DBSM was designed to be integrated in an existing database kernel, this would not be a problem, because regular RDBMS have this functionality internally, for crash-recovery purposes.

there is no need to undo any committed changes caused by a subsequent abort of a running transaction.

4.4.3 Objects and classes

As stated in the description of COPLA, the system deals with data organized in objects, which are instances of a given class. So far in the description of this protocol, we have considered objects as isolated entities. However, in order to correctly enforce transaction serializability, the notion of class must also be introduced in the concurrency control mechanism used by the consistency protocol.

To do this, each class must also have a version number associated with it. The rules that establish the inclusion of these version numbers in the transactions' read and write sets are the following:

- Whenever an instance of class X is read by a transaction t that searches the entire class, that is considered a read of class X (in addition to a read of that particular instance), and X's OID and version number is included in RS_t.
- Whenever an instance of class X is created, written or deleted by a transaction t, this is considered a write in class X, and X's version number will be increased when t commits.

Note that the enforcement of the read rule is made in cooperation with the UDS module – the UDS will interpret the query and will include in the resulting read set either specific objects or the whole class, as appropriate.

4.5 Implementation

As described in Chapter 3, an interface was defined for the COPLA consistency protocol (CP) module. This would enable different protocols to be used, thus

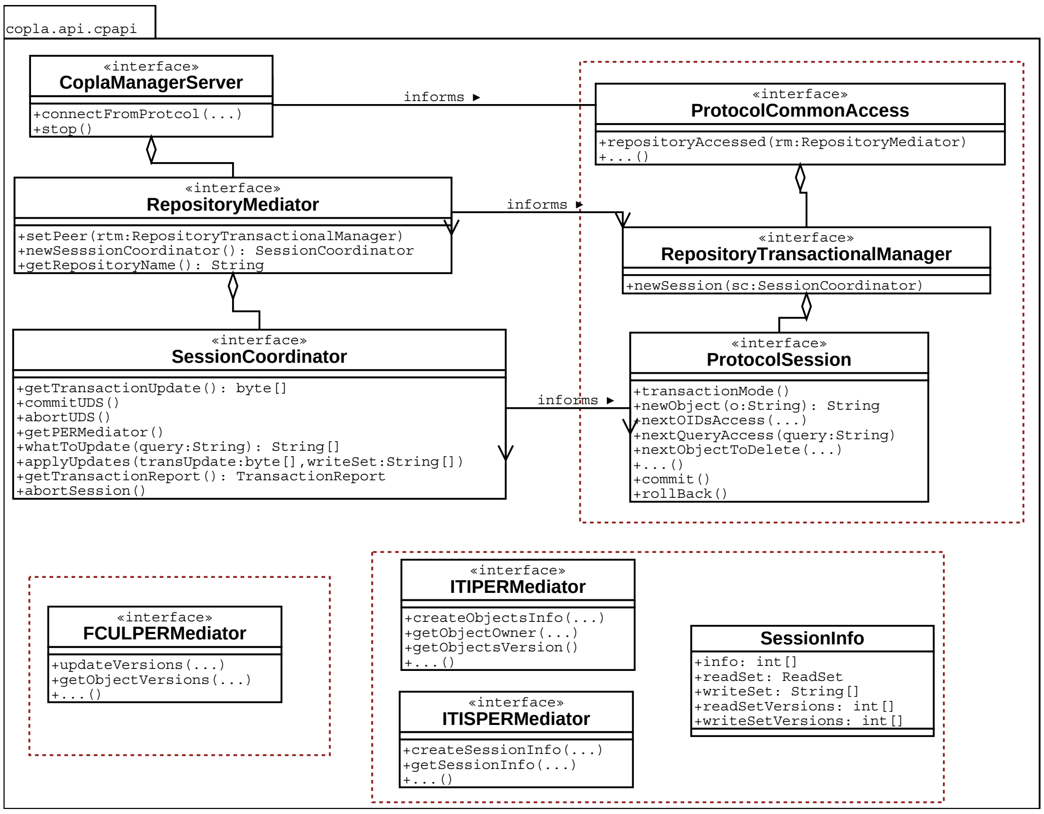


Figure 4.2: CP-API interface

giving greater flexibility.

Obviously, this interface will impact upon the algorithms detailed design, since it must conform to it. Below both the description of the interface, and the adapted design of the algorithm are described.

4.5.1 The CP-API interface

Figure 4.2 presents the complete CP-API interface. This interface is composed of three major parts:

1. The COPLA server interface part, composed by the CoplaManagerServer, RepositoryMediator and SessionCoordinator interfaces. This part is the interface exposed to the consistency protocols for working with the remainder

of the COPLA server.

2. The protocol interface part, composed be the three interfaces surrounded by a border in the top left of the figure, ProtocolCommonAccess, RepositoryTransactionalManager and ProtocolSession. Each consistency protocol must implement these three interfaces.

3. The persistence interface (PER-API) part, composed of the remaining classes in the diagram. This interface is divided in two parts (each surrounded by a border), that correspond to interfaces specified by the two teams developing consistency protocols for COPLA, FCUL (bottom left)and ITI (bottom right). The protocol described in this thesis uses only the FCUL interface. This interface, implemented by the UDS module, provides functions for storing metadata in a persistent way, and coordinated with COPLA transactions.

The server interface is divided in three classes:

CoplaManagerServer This class contains two calls: one to enable a CP to open a repository connection (this is required when a node that has no clients has to apply remote updates), and a call to stop the node (emergency shutdown, for example).

RepositoryMediator This interface is usually passed to the consistency protocol at initialization time. It represents a repository. It contains three calls: setPeer(), to set its peer class in the consistency protocol (the class over which it will make repository-related operations to the CP), newSessionCoordinator(), to allow the CP to create an independent SessionCoordinator, and getRepositoryName(), that returns the repository's name.

SessionCoordinator This interface represents an opened session in a repository, at the COPLA manager level. It is usually associated with a client session (that interfaces with client applications), a ProtocolSession, that represents a session within a CP, and an UDS transaction, that contains methods for manipulating the persistent objects. The first two associations may not be present: the CP may create an *independent* Sessioncoordinator for applying updates from a remote node, for example. Also, it may also be associated with a PERMediator object, that provides operations over persistent metadata. Those operations will be performed in the context of the session represented by the SessionCoordinator.

The following classes constitute the interface that each CP must implement.

ProtocolCommonAccess The interface for starting communications between the common COPLA code and the integrated consistency protocol. repositoryAccessed() is called by the common code to inform the protocol that an user application is going to work with a given repository.

RepositoryTransactionalManager Is in charge of coordinating the work on a given repository. Its interface to the common code side offers the method newSession() so the protocol can be told of session openings on the repository represented.

ProtocolSession The interface to inform the protocol of actions the user application want to perform on the current session. This way the protocol can make a decision on either allow the action to proceed, delay it while some objects are suitably updated or deny the action. For instance:

- nextOIDsAccess() is used to inform the protocol of a set of objects that are to be accessed.

- nextQueryAccess() tells the protocol the next query that is going to be performed. The protocol should check if all objects involved to perform the query are up to date on the local UDS.

- The commit() method asks the protocol whether the job done by the session can be committed. Some relevant actions implied by the committing are driven by the protocol: calling the commitUDS() method of the SessionCoordinator class to tell the UDS to commit the transaction as well as collecting from it the "transaction report" so its effects can be replicated on other nodes.

The PER-API interface consists of a single class, FCULPERMediator, that contains two relevant functions:

- The updateVersions method increments the version numbers of the given objects.

- The getObjectVersions function returns the version numbers of the given objects.

As stated above, a PERMediator object (which FCULPERMediator subclasses) is always associated with an opened session, represented by a SessionCoordinator instance. The operations it provides will be executed within the context of that session, and will thus only be applied when the session commits.

4.5.2 Class and function structure

The implementation code is divided in two main parts: one part implements the functions defined in the CP-API. The other consists in an independent thread that listens to incoming messages. The class structure is shown in Figure 4.3

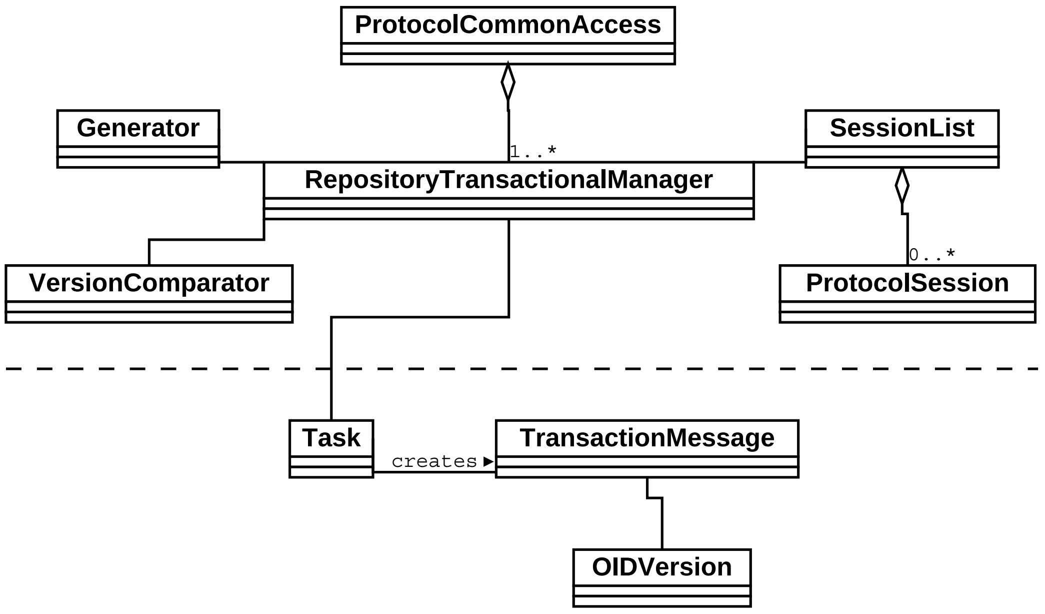


Figure 4.3: Class structure of the protocol implementation

The first part (above the dashed line) is composed of six classes. Three of them (RepositoryTransactionalManager, ProtocolCommonAccess and ProtocolSession) provide the implementation of the protocol part of the CP-API (see Figure 4.2). The other three are auxiliary classes, they will be described below.

The second part is composed by three classes. The Task class contains the code to process incoming messages, and the other two classes constitute the messages themselves.

Each class and its functions will now be described in greater detail.

ProtocolCommonAccess This class is the entry point for the protocol code. It contains functions to initialize and stop the protocol, and a function that is called by the COPLA common code to inform the protocol that a given repository has been accessed.

RepositoryTransactionalManager This is the main class of the protocol. Its methods are:

- **newSession()** This function is called by the common code to create a new session.
- **commitSession()** Attempts to commit a given session.
- **abortSession()** Aborts a given session.
- **checkTransactionConsistency()** This function checks whether a given transaction's read set is consistent, i.e., if that transaction has not read data that was modified.
- **abortConflictingTransactions()** This function aborts running transactions whose read sets intersect the given write set. The actual work is performed by the corresponding function in SessionList.
- **newView()** Called by Task to inform the protocol that there is a membership change of the server group. A view change may happen on a node failure, node join or network partition.

SessionList This class holds all the running sessions.

- **addSession()** Adds a session to the list.
- **removeSession()** Removes a session from the list.
- **getSession()** Returns a specific session.
- **abortConflictingTransactions()** This function aborts all transactions whose read sets intersect the given write set.

Generator This class is responsible for creating unique session identifiers and unique object identifiers.

- **nextSID()** Generates a unique SID (Session IDentifier).
- **nextOID()** Generates a unique OID (Object IDentifier).

ProtocolSession This class represents a transaction at the consistency protocol level.

transactionMode(), checkoutMode() These methods switch the operating mode for this transaction. Since the FCUL protocols always operate in transaction mode, these methods do nothing.

nextOIDAccess() / nextQueryAccess() / nextMOAAccess() / nextMRAccess() / nextMLAAccess() Verifies which OIDs are going to be accessed and add them to the read set of the transaction.

nextURChange(), nextUOAChange() Verifies which OIDs are going to be changed and add them to the read set of the transaction.

newObject() Adds the created object to the read and write set of the transaction.

nextObjectToDelete() Adds the object to be deleted to the read and write set of this transaction.

nextRemoveAllOIDs() Adds the given OIDs to the list of deleted objects. This list must be maintained because of cascading deletes.

getState() Returns the state of this transaction (aborted, committed or running).

setState() Sets the state of this transaction. Changing the transaction from running to committed or aborted commits or aborts the associated UDS transaction.

intersects() Checks if the given write set intersects this transaction's read set.

commit() Attempts to commit this transaction.

abort() Aborts this transaction.

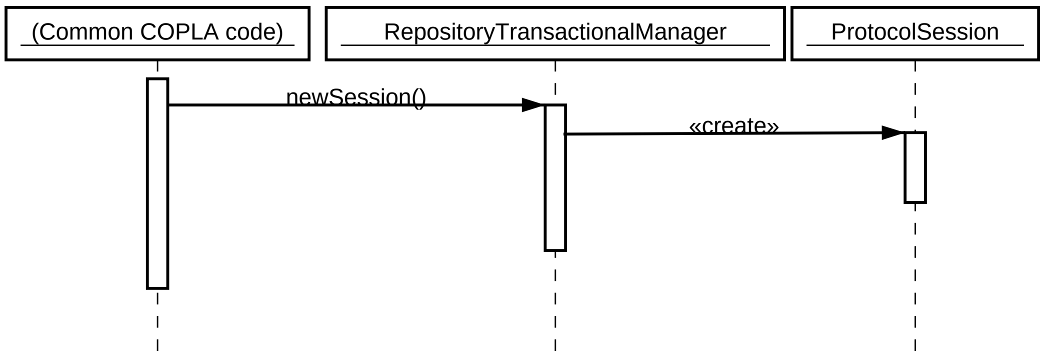


Figure 4.4: Execution of a newSession request

close() Frees any resources associated with this transaction.

VersionComparator This class contains a single function, lessThan() that checks whether a version is older than another, taking in to account version number wrap-around.

Task This class contains the code that runs in a separate thread and processes incoming messages.

run() Main message processing loop.

send() Sends a message through atomic broadcast.

In order to clarify the descriptions above, we show some common executions with interaction diagrams.

In Figure 4.4, the execution of a newSession request is shown.

The execution of a commit request is shown in Figure 4.5. The COPLA manager first makes a commit request to ProtocolSession, which in its turn forwards it to RepositoryTransactionalManager. This object is responsible for constructing a transaction message to be sent to the network. To do so, it first obtains the versions of the objects involved in the transaction, through the getObjectVersions() method. Then it constructs the message and forwards it to the Task object,

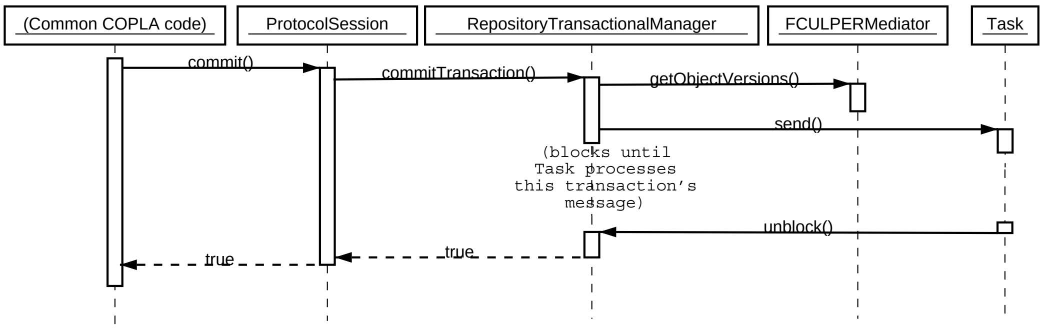


Figure 4.5: Execution of a commit request

and blocks waiting for it to be ordered by the atomic broadcast protocol. The Task object is responsible for sending the message to the total order protocol, and to unblock the correct session when a message is delivered by the communication module. When RepositoryTransactionalManager is unblocked, it will verify if the transaction can be committed or not, through the checkTransactionConsistency() method, and if so, commit the transaction in the UDS and return a true response to the common COPLA code.

4.5.3 Detailed algorithm

In this section, the detailed pseudo-code of key algorithm functions will be examined, to illustrate the choices presented above. First, let us examine a more refined description of the protocol, in figure 4.6. Here we can already see that the algorithm must be separated in two: one thread runs independently, receiving incoming transaction messages, and the other functions run within the main application thread. This "unfolded" version fits better within the class structure presented in the section above. The main thread consist of procedures that are called by the CLIB at various stages of a session: the initialize method is called when a session is created, the UDSAccess method whenever the session reads objects (listed in the parameter *l*), and the commit method when a session requests a commit. All these methods receive a parameter, *t*, that indicates the session to act upon. The Task thread is simply in charge of processing incoming messages, and validating them. The pseudo-code presented in the figure uses two auxiliary methods: udsCommit and udsAbort instruct the UDS layer to commit or abort a transaction.

Reading and checking versions Here a small, but nevertheless important implementation detail must be considered. Like it was described above, accesses to the database, whether objects or persistent metadata, must be done through a SessionCoordinator. Normally, a SessionCoordinator is associated with a ProtocolSession. However, the protocol sometimes requires access to the database *outside* the context of any user transactions. To that effect, it uses an independent SessionCoordinator (ISC) to perform these accesses.

With the above in mind, let us examine the source for the three functions that cooperate to perform the commit function, corresponding to the isConsistent, abortConflicting and commit functions in figure 4.6. The function headers make it clear which one is which.

```
/**
 * Checks if the transaction is consistent , i.e., it has not read stale
 * data.
 *
 * @param rs the read set of the transaction .
 * @return true if the transaction is consistent , false otherwise.
 */
boolean checkTransactionConsistency(OID_Version[] rs) {
    boolean ret = true;

    try {
        String [] oids = new String[rs.length];
        for (int i = 0; i < rs.length; i++)
            oids[i] = rs[i].oid;

        OID_Version[] lv = null;
```

Main thread

```
procedure initialize($t$)
    $pread_t \leftarrow \emptyset$
    $lv_t \leftarrow \emptyset$

procedure UDSAccess($t$,$l$)
    $pread_t \leftarrow pread_t \cup l$

function isConsistent($t, rs_t$)
    $lv_t \leftarrow$ getVersions($rs_t$)
    for each $< o,v >$ in $rs_t$ and $< o,v' >$ in $lv_t$
        if $v < v'$
            return false
    return true

procedure abortConflicting($t, ws$)
    for each running transaction $rt$
        if $pread_{rt} \cap ws \neq \emptyset$
            udsAbort($rt$)

procedure commit($t$)
    $rs_t \leftarrow$ getReadSet($t$)
    $ws_t \leftarrow$ getWriteSet($t$)
    if $ws_t = \emptyset$
        if isConsistent($t, rs_t$)
            udsCommit($t$)
        else
            udsAbort($t$)
    else
        ABSend($< t, rs_t, ws_t >$)
```

Task thread

```
Upon ABReceive($< t, rs_t, ws_t >$):
    if isConsistent($t, rs_t$)
        abortConflicting($t, ws_t$)
        udsCommit($t$)
    else
        udsAbort($t$)
```

Figure 4.6: Non-voting protocol (detail)

```
17            synchronized (iscLock) {
18                lv = indepFpm.getObjectVersions(oids);
19
20                for (int i = 0; i < rs.length; i++) {
21                    int j;
22                    for (j = 0; ! lv[j].oid.equals(rs[i].oid); j++);
23
24                    if (VersionCompare.lessThan(rs[i].version, lv[j].version)) {
25                        ret = false;
26                        break;
27                    }
28                }
29
30                indepSessionCoord.commitUDS();
31            }
32
33            return ret;
34        } catch (NoSuchOIDException ex) {
35            error("checkTransactionConsistency:␣bad␣oid");
36            ex.printStackTrace();
37            Util.fatalCondition();
38        } catch (PersistentLayerException ex) {
39            Util.fatalCondition();
40        }
41
42        return false;
43    }
```

The above function is very straightforward: it first obtains from the database the versions it currently owns for the requested objects. This involves creating a list of OIDs to pass to the database (lines 12–14), and then performing the call (line 18). The obtained versions are then compared to the versions present in the given read set (lines 20–28), using the VersionComparator class. The only difficulty here is that the two lists might not be ordered the same way, OID wise, hence the need for two nested “for” loops. Finally, the ISC transaction is ended.

This procedure is done inside a synchronized block, because it makes use of the ISC. Since there is only one independent coordinator, not synchronizing the access to it would mean that operations from concurrently executing transactions would be intermixed in the same ISC transaction.

```
1  /**
2   * Aborts transactions that conflict with the given write set.
3   *
4   * @param ws the write set.
5   * @param sid the session id of the incoming transaction.
6   */
7  synchronized void abortConflictingTransactions(String[] ws, SID sid) {
8      for (Enumeration i = htable.elements(); i.hasMoreElements();) {
9          ProtocolSession s = (ProtocolSession) i.nextElement();
10         if (s.getState() == ProtocolSession.EXECUTING) {
11             if (!s.sid.equals(sid) && s.intersects(ws)) {
12                 s.setState(ProtocolSession.ABORTED);
13             }
14         }
15     }
16 }
```

This function is a direct translation from pseudo-code: it runs through the session list, checking if each executing session's read set intersects the given write set. If so, the offending session is aborted. The intersects function called on line 11 is a simple check against the session's read set.

```
/**
 * Commits a session.
 *
 * @param session the session to commit.
 * @param tr the session's transaction report.
 *
 * @return true if the session commited ok, false otherwise.
 */
boolean commitTransaction(ProtocolSession session, TransactionReport tr) {
    int state = -1;

    synchronized (session) {
        // check if this session was aborted while it was executing
        state = session.getState()
    }

    if (state == ProtocolSession.ABORTED) {
        SID oldSid = resetSession(session);
        return false;
    }

    synchronized (viewLock) {
        if (myState == RECOVER) {
            session.setState(ProtocolSession.ABORTED);
            session.rollBack();
            resetSession(session);
            return false;
        }
    }

    synchronized (session) {
        if ((state = session.getState()) == ProtocolSession.ABORTED) {
            // abort processing
        } else {
            try {
                OID_Version[] rs = buildRsVersions(tr.readSet, session.reads);

                byte[] tu = null;
                tu = session.getTransactionUpdate();

                if (tr.writeSet.length > 0) { // read-write transaction
                    String src = null;
                    synchronized (viewLock) {
                        src = viewState.view[localState.my_rank].toString();
                    }

                    /* Processing for finding cascaded deletes: if an object
                     * is in WS but not in RS or Created then it is a
                     * cascaded delete.
                     */
                    boolean found = false;
                    for (int i = 0; i < tr.writeSet.length; i++) {
                        found = false;
                        for (int j = 0; j < rs.length; j++) {
                            if (rs[j].oid.equals(tr.writeSet[i])) {
                                found = true;
                                break;
                            }
                        }
                        if (!found) {
                            found = session.createdObjs.contains(tr.writeSet[i]);
                    }
                    if (! found) {
                        session.deletedObjs.add(tr.writeSet[i]);
                    }
                }

                String [] sco = (String []) session.createdObjs.toArray(new String[0]);
                String [] sdo = (String []) session.deletedObjs.toArray(new String[0]);
                TransactionMessage tm =
                    new TransactionMessage(session.sid, rs, tr.writeSet, tu, sco, sdo, src);
                TxMessage msg = new TxMessage(tm.munchTrans());
                task.send(msg);
            } else { /* read-only transaction */
                if (checkTransactionConsistency(rs)) {
                    session.setState(ProtocolSession.COMMITED);
                } else {
                    session.setState(ProtocolSession.ABORTED);
                }
            }

            // wait for the state to be updated
            while ((state = session.getState()) == ProtocolSession.EXECUTING) {
                session.wait();
            }
        } catch (PersistentLayerException ex) {
            Util.fatalCondition();
        } catch (TransactionUpdateException ex) {
            Util.fatalCondition();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
            error("commitTransaction:␣wait␣interrupted");
        }
    }
}

SID oldSid = resetSession(session);

if (state == ProtocolSession.COMMITED) {
    return true;
```

```
101     } else {
102         return false;
103     }
104 }
```

This is one of the most complex functions, that implements the commit procedure of a session. A thing to note is that access to the actual session data must be made under a lock on the session itself. This must be done because an arriving message may abort the very session being processed. The same reasoning applies to accessing the group view information; a view change can occur at any time the function is executing.

The function begins by checking if the session was already aborted (lines 12–20), or if the node is in a recover state (lines 22–29). Then the main commit procedure begins, on line 31, again with an abort check (because the lock on the session object had been released).

First, the versions for each object read by the transaction are collected, in line 36. This procedure simply performs the union of both sets passed to it. The first is the set passed to the commit function, the other is the reads collected during transaction execution.[3] Then, the transactions updates are retrieved (line 39). The procedure is then split in two: like was stated above, read-only transactions can be processed more efficiently. The handling of read-write transactions is made in lines 42–73.

First, the identity of the node is determined (42–45). Then, some processing must be done for finding cascaded deletes (lines 47–66). Cascaded deletes occur when an object that references other objects is deleted. If the referenced objects do not have any other references pointing to them they are deleted as well (i.e., the first delete provoked a *cascade* of deletions). These cascaded deletes are added to the session's write set (tr.writeSet in the code), but might not be included in

[3] In practice, in the current COPLA implementation these sets are exactly the same. In theory, however, the first set might contain additional reads, because of class relationships, for example.

the session's read set, because they might not have been read by the session at all. This part then finishes by constructing the transaction message and sending it (lines 68–73).

Lines 75–79 contain the handling for read-only transactions, which is much simpler: all that is required is to verify if the transaction has not read stale data (line 75), and commit or abort accordingly.

The function will then suspend the current thread until the system reaches a decision about the transaction (this will be done within the Task code, explained below). Note that this test continues immediately in case the transaction was read-only.

In the end, the function resets the session data (this is done to mirror the behavior of user sessions, that can be used continuously as well), and returns the appropriate value.

```
public void run() {
    while (true) {
        CommAPIMessage msg = null;

        msg = commApi.receive();
        if (msg instanceof SendToAll) {
            if (msg instanceof TxMessage) {
                TransactionMessage tm =
                    TransactionMessage.deMunchTrans(((TxMessage)msg).getData());
                if (recover) {
                    recMsgQueue.addLast(tm);
                } else {
                    handleTxMessage(tm);
                }
            }
        } else if (msg instanceof SendTo) {
            if (msg instanceof LogRequestMessage) {
                handleLogRequest((LogRequestMessage) msg);
            } else if (msg instanceof LogReplyMessage) {
                handleLogReply((LogReplyMessage) msg);
            }
        } else if (msg instanceof CommAPIView) {
```

```
23            // new view
24            blocked = false;
25            while (! msgQueue.isEmpty())
26               commApi.send((CommAPIMessage) msgQueue.removeFirst());
27
28            rtm.newView((CommAPIView) msg);
29         } else if (msg instanceof CommAPIBlockOk) {
30            synchronized (blockLock) {
31               blocked = true;
32               commApi.send(msg);
33            }
34         }
35      }
36   }
```

As described above in the implementation's architecture, the Task class is a thread, whose main loop continuously reads and processes incoming messages. Above the source code for this loop is presented.

This loop can receive four types of messages. Transaction messages are handled by another function, shown below. If the system is in recover mode, these messages are queued for later processing. Point-to-point messages are used by the recovery procedure. The other two message types are views and block requests. View messages inform the system that group membership changed.

Block messages are "fake" messages (i.e., no actual transmission over the network is performed), and a requirement of the view synchrony system. The are generated when a view change is about to occur, and they establish a basic contract between the communication system and its users.[4] When the user receives this message, it will send any pending messages it has. Then, it will send the received block message, and will refrain from sending any more messages until it receives a view change message (during this period the user may receive other messages). This allows the communication system to stabilize, and fulfill the view synchrony requirements. This is the reason for the code in lines 25–26: messages

[4]Here, the communication system user is the CP layer.

to be sent while the system is blocked are placed in a queue, that is flushed when the system unblocks.

```
/**
 * Handles transaction messages.
 * @param msg the received message.
 */
private void handleTxMessage(TransactionMessage tm) {
    // check for created objects on remote tx.
    OID_Version[] rsToCheck = null;
    if (tm.createdObjs != null && tm.createdObjs.length != 0
            && (!rtm.isLocal(tm.sid))) {
        // a remote tx has created objects, must add entries
        // in version table
        rsToCheck = splitSet(tm.readSet, tm.createdObjs);
    } else {
        rsToCheck = tm.readSet;
    }

    synchronized (currWrites) {
        for (int i = 0; i < tm.writeSet.length; i++) {
            currWrites.add(tm.writeSet[i]);
        }
    }

    if (rtm.checkTransactionConsistency(rsToCheck)) {
        rtm.abortConflictingTransactions(tm.writeSet, tm.sid);
        // notify transactions, so that aborted transactions wake up
        // and free the UDS tx.
        synchronized (currWrites) {
            currWrites.notifyAll();
        }
        if (rtm.isLocal(tm.sid)) {
            // it's a local transaction, let's commit it.

            // write the metadata on persistent storage
            ProtocolSession s = rtm.sessionList.getSession(tm.sid);
            if (s != null) {
                // update versions
                String[] delob = null;
                String[] wsToUpdate = null;
                if (s.deletedObjs.size() != 0) {
                    delob = (String[]) s.deletedObjs.
41                          toArray(new String [0]) ;
42                  }
43                  if ( s . createdObjs . size () != 0) {
44                      wsToUpdate = splitWriteSet (tm. writeSet ,
45                              tm. createdObjs ) ;
46                  } else {
47                      wsToUpdate = tm.writeSet;
48                  }
49                  Util . updateVersions (s . sc , wsToUpdate, delob, "");
50                  addLogEntry(s.sc , tm. sid , curr_seq_num,
51                          tm.pObjs, tm. writeSet ) ;
52                  s. setState ( ProtocolSession .COMMITED);
53                  curr_seq_num++;
54              }
55          } else {
56              // it 's a transaction from another node
57              writeInUDS(tm.sid , curr_seq_num, tm. writeSet ,
58                      tm. createdObjs , tm. deletedObjs ,
59                      tm.pObjs);
60              curr_seq_num++;
61          }
62      } else {
63          // inconsistent transaction
64          if (rtm. isLocal (tm. sid )) {
65              ProtocolSession s = rtm. sessionList . getSession (tm. sid ) ;
66              if ( s != null) {
67                  s. setState ( ProtocolSession .ABORTED);
68              } // if s is null then a transaction that arrived
69                 // before has already aborted s
70          } // nothing to be done for a remote aborting transaction
71      }
72      synchronized ( currWrites ) {
73          currWrites . clear () ;
74          currWrites . notifyAll () ;
75      }
76  }
```

This function handles incoming transaction messages, and implements the second half of the commit procedure. It begins by separating from the read set any objects the incoming transaction might have created (lines 7–15). They are included in the read set, but cannot be passed to the consistency check (on line 23), because versions for them do not exist in the database yet (since they were created by this transaction, which has not committed yet).

The purpose of the currWrites array (lines 17–21, 27–29 and 72–75) will be described furthed below, together with the addReads code.

It then proceeds by checking if the transaction has read any stale data. If it did, it will abort the transaction (exactly how will be examined further below). If not, it will abort all conflicting transactions. Then the procedure splits in two, for local and remote transactions.

For local transactions, it will write the version data to persistent storage (line 49), add an entry to the recovery log (50), and then commit the transaction (52). Line 53 increments the recovery log sequence number. Note that this whole sequence may be skipped, in case there is no entry in the active session list for the incoming transaction (lines 34–35). This means that a previous transaction has aborted the current one, and as such the message is simply discarded.

For remote transactions, it will perform the same steps as the local case, except that it will apply the transaction's updates before committing. The process is done outside this function simply to ease error handling.

Inconsistent transactions are handled on lines 62–71. For local transactions, it will locate the corresponding ProtocolSession object, and abort the session. For remote transactions, the message is simply discarded.

```
1  /**
2   * Adds the specified objects to this session's read set.
3   *
4   * @param objs the {@link OID pt.fcul.copla.manager.protocol.common.OID}s
5   * to add to the read set.
6   */
7  void addReads(String[] objs) throws AbortException {
8      LinkedList versToGet = new LinkedList();
9
10     // check if incoming tx is writing on our objects
11     synchronized (rtm.task.currWrites) {
    boolean hold = true;
    boolean found = false ;
    while (hold) {
        if ( this . state == ABORTED) {
            hold = false ;
        } else {
            found = false ;
            for ( int i = 0; i < objs . length ; i++) {
                for ( Iterator j = rtm. task . currWrites . iterator () ; j .hasNext() ; )
                    {
                    if ( objs [ i ]. equals ((( String ) j . next () ))) {
                        found = true;
                        break;
                    }
                }
            }
            if (found) {
                try {
                    rtm. task . currWrites . wait () ;
                } catch ( InterruptedException e) {
                    e. printStackTrace () ;
                }
            } else {
                hold = false ;
            }
        }
    }

    if ( this . state == ABORTED) {
        throw new AbortException();
    }

    synchronized (reads) {
        for ( int i = 0; i < objs . length ; i++) {
            if (OID.quickIsClassId (objs [ i ]) ) {
                if ( reads . classIds .add(objs [ i ]) ) {
                    versToGet.add(objs [ i ]) ;
                }
            } else {
                if ( reads .setOfOIDs.add(objs [ i ]) ) {
                    versToGet.add(objs [ i ]) ;
                }
            }
            }
            if (versToGet.size() != 0) {
                try {
                    FCULPERMediator fpm =
                        (FCULPERMediator) this.sc.getPERMediator(ProtocolType.FCUL
                            );
                    OID_Version[] ov =
                        fpm.getObjectVersions((String[]) versToGet.toArray(new String
                            [0]));
                    for (int i = 0; i < ov.length; i++) {
                        reads.versions.put(ov[i].oid, new Integer(ov[i].version));
                    }
                } catch (PersistentLayerException e) {
                    e.printStackTrace();
                    Util.fatalCondition();
                } catch (NoSuchOIDException e) {
                    error("addReads:␣no␣such␣oid");
                    e.printStackTrace();
                    Util.fatalCondition();
                }
            }
        }
    }
}
```

Above we can see the addReads code. This function is called whenever the protocol is informed that a session will read objects. In principle, this function would be very simple: obtain the current version of the objects to be read, and add the OIDs and corresponding versions to the read set of the session.

However, there is a problem. The whole design of the COPLA tool requires that the underlying RDBMS supports the SQL isolation level READ COMMITTED. In this mode, running transactions are allowed to read only committed data. This means that if a transaction t_1 writes on an object x, and a transaction t_2 also wants to write x, t_2 will be placed on hold until t_1 commits, because t_1 already has an exclusive lock on x.[5]

[5]Theoretically, if the RDBMS uses an optimistic approach, it would abort t_2. However, the most widely used RDBMS, including PostgreSQL, over which the COPLA prototype runs, use lock-based concurrency management.

Now, imagine that the transaction t_1 above was being executed on behalf of a COPLA session writing on an object. And t_2 was the Task thread applying the transaction changes of a transaction. You now have a deadlock, because when the session requests a commit, it will send its message through. However, that message will never arrive, because the Task thread is busy writing on the database. And the Task thread will never unblock, because the session will never commit!

This case can occur when a running session writes to an object after Task has aborted conflicting transactions (line 24 of handleTxMessge), but before it actually writes to the database (lines 49–52 of the same function). This problem is further compounded by the COPLA architecture: like it was described in Section 3.7.2, the consistency protocols are not informed of object writes, they are made directly by the CLIB layer. The solution is to handle session reads as potential writes (because to write on an object a session must read it first).

The following mechanism was implemented in the code. An array named currWrites contains the writes that the Task thread is performing at the time. Before handleTxMessage writes to the database, or aborts any transactions, it fills this array, on lines 17–21. If a session is going to read objects, it must first check if they are currently being written by Task, on lines 11–37 of addReads. If they are, then the session waits for the Task thread to finish (note that addReads is called *before* any read is performed, so this prevents the offending session from blocking Task's transaction). When handleTxMessage aborts conflicting transactions, it wakes up all transactions waiting on currWrites, on lines 27–29. This will cause sessions that were "caught" by abortConflictingTransactions to be aborted (note, in addWrites, that the waiting loop of line 11 is broken in case the session is aborted, and an exception is thrown on lines 39–41 to abort the session). Finally, after the writes are performed on the database, handleTxmessage clears the array and wakes up all transactions that were waiting on currWrites (these were the transactions that would cause the deadlock), on lines 72–75. The sessions waiting on addReads will wake up, and their check on currWrites (lines 19–26) will fail, enabling them to proceed.

```
class VersionCompare {

   private static final int MSB_MASK = 0x80000000;


      /* if v_one is smaller than v_two */
      public static boolean lessThan(int v_one, int v_two) {
         if ((v_one & MSB_MASK) != (v_two & MSB_MASK))
                  return ((v_one << 1) > (v_two << 1));
         else
                  return (Math.abs(v_one) < Math.abs(v_two));
      }
}
```

Above we can see the source code for the version compare function. The version number mechanism must deal with wrap-around. A version number is a 31-bit number, with the leftmost bit (the sign bit) reserved for wrap-around signaling, and starts with 0. When it reaches $2^{31} - 1$, it will wrap around, and continue with negative numbers. As such, -2 is actually greater than $2^{31} - 3$, for example.

4.6 Optimistic delivery

As described in Chapter 3, 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 the consistency protocol. The tentative order allows the protocol 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 writing data behind the cost of uniform delivery, effectively doing both things in parallel.

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.

In terms of the implementation, the Task code is the only one that will change. Below we can see the modified code.

```
public void run() {
    while (true) {
        CommAPIMessage msg = null;

        msg = commApi.receive();
        if (msg instanceof SendToAll) {
            if (((SendToAll) msg).getOptimistic()) {
                TransactionMessage tm =
                    TransactionMessage.deMunchTrans(((SendToAll)msg).getData());
                if (recover) {
                    recMsgQueue.addLast(tm);
                } else {
                    handleOptTx(tm);
                }
            } else {
                TransactionMessage tm =
17                      TransactionMessage.deMunchTrans(((SendToAll)msg).getData());
18                  if ( recover ) {
19                      recMsgQueue.addLast(tm);
20                  } else {
21                      handleTxMessage(tm);
22                  }
23              }
24          } else if ( msg instanceof SendTo) {
25              if ( msg instanceof LogRequestMessage) {
26                  handleLogRequest((LogRequestMessage) msg);
27              } else if ( msg instanceof LogReplyMessage) {
28                  handleLogReply((LogReplyMessage) msg);
29              }
30          } else if ( msg instanceof CommAPIView) {
31              // new view
32              blocked = false ;
33              while (! msgQueue.isEmpty())
34                  commApi.send((CommAPIMessage) msgQueue.removeFirst());
35
36              rtm.newView((CommAPIView) msg);
37          } else if ( msg instanceof CommAPIBlockOk) {
38              synchronized (blockLock) {
39                  blocked = true;
40                  commApi.send(msg);
41              }
42          }
43      }
44  }
```

Here we see the new main loop for the Task thread. An incoming transaction message can now be delivered optimistically (line 7), or in final order.

```
/**
 * Handles transactions delivered with tentative order.
 * @param msg the received message.
 */
private void handleOptTx(TransactionMessage tm) {
    OptMsg om = new OptMsg(tm);

    if ( checkAgainstQueue(om)) {
        optMsgQueue.addLast(om);
        om. consistent = checkPhase(om);
    } else {
```

```
12          discardMsgQueue.add(om);
13      }
14  }
```

Above we see the (very simple) code for initially handling optimistically delivered messages. First, the message is first checked against all opt-received but not yet confirmed transaction messages (line 8). If any of these writes on an object that the current transaction has read, then the transaction will quite likely be aborted. Has such, it is placed on the discarded message queue (line 12), which contains transactions that were aborted during the opt-deliver phase. If the message passes the test, it will be placed on the opt-received message queue and proceed to the check phase (lines 9–10).

The code for the check phase is identical to the handleTxMessage procedure of the original implementation, save two details: it does not commit the transaction after performing the writes on the database, and it returns the outcome of the consistency check in the end.

```
1   /**
2    * Handles transaction messages received in final order.
3    * @param msg the received message.
4    */
5   private void handleTxMessage(TransactionMessage tm) {
6       OptMsg om;
7
8       // check if this message has not been discarded
9       chkOm.tm = tm;
10      chkOm.sid = tm.sid;
11      if (discardMsgQueue.contains(chkOm)) {
12          discardMsgQueue.remove(chkOm);
13          if (checkPhase(chkOm)) {
14              commitPhase(chkOm);
15          }
16          return;
17      }
18
19      while (!(om = (OptMsg) optMsgQueue.removeFirst()).tm.sid.equals(tm.sid) &&
20                  optMsgQueue.size() != 0) {
```

```
21        // all these messages are out of order
22        if (rtm.isLocal(om.tm.sid)) {
23          ProtocolSession s = rtm.sessionList.getSession(om.tm.sid);
24          // don't notify client, tx will try to commit when final order arrives
25          s.abortNotNotify();
26        } else {
27          om.sc.abortSession();
28          try {
29            om.sc.closeSession();
30            om.sc = null;
31          } catch (PersistentLayerException e) {
32            error("PersistentLayerException␣closing␣indep.␣SC");
33            e.printStackTrace();
34            Util.fatalCondition();
35          } catch (CloseSessionException e) {
36            error("CloseSessionException␣closing␣indep.␣SC");
37            e.printStackTrace();
38            Util.fatalCondition();
39          }
40        }
41        discardMsgQueue.add(om);
42      }
43
44      commitPhase(om);
45    }
```

When a transaction message's ordering is confirmed by the communications module, it is delivered in final order, and processed by the above procedure.

First, we must check if the transaction being processed was not placed on the discard queue earlier (line 11). If so, then we must repeat the check phase, to find out if the transactions that would potentially abort the current one actually committed (and thus aborted this transaction). If the check phase returns true, then we commit the transaction immediately (line 14).

In case the message was not in the discard queue, then we must verify the ordering of all the other already opt-received transactions. Any message that in between the current top of the queue and the opt-delivered version of the current transaction is out of order (line 19).[6] These transactions will then be aborted. For local transactions, the process is very similar to aborting an inconsistent transaction in the regular version of the protocol: the session is found on the session list, and aborted (lines 23–35).[7] For remote transactions we need to abort the session opened by the protocol for applying their modifications (lines 28–39). Finally, these aborted transactions are placed on the discarded queue (line 41), since they were aborted during their optimistic phase. After this process is complete, the transaction proceeds to the commit phase (line 44).

```
/**
 * Commits the given transaction .
 * @param om the transaction to commit.
 */
private void commitPhase(OptMsg om) {
    if (om. consistent ) {
        if (rtm. isLocal (om.tm.sid)) {
            ProtocolSession s = rtm. sessionList . getSession (om.tm.sid);
            if (s != null) {
                s. setState ( ProtocolSession .COMMITED);
            }
        } else {
            try {
                om.sc.commitUDS();
                om.sc. closeSession () ;
            } catch ( PersistentLayerException e) {
                e. printStackTrace () ;
                 Util . fatalCondition () ;
            } catch (CloseSessionException e) {
                e. printStackTrace () ;
            }
        }
    } // if its not consistent , all was done on checkPhase
}
```

[6]Note that when the system is stable and the optimistic and final orders are the same, the opt-delivered version of the message is at the top of the queue.

[7]The reason for not notifying the client is for code simplicity. IF the client was notified, it would cause a reset of the session data, which means that when the final message for the aborted transaction was received, then the code would have to take in to account the possibility of an opt-delivered version not being present in any of the queues, complicating the code.

The commit phase simply checks if the transaction is consistent (line 6), and then commits it.

It should be noted that this optimization was not tested in the evaluation shown in the next chapter, because its implementation at the time of the evaluation was not completed.

4.7 Summary

In this chapter, I presented the NonVoting consistency algorithm. A general description of the algorithm was given, followed by a description of the COPLA module where the algorithm is implemented, and the implementation code itself. A discussion of an optimization to the algorithm concluded this chapter. In the next chapter I will present an evaluation of the algorithm against other consistency algorithms.

Chapter 5

Evaluation

This chapter will present an evaluation of the protocol, against the other two protocols implemented for COPLA. This evaluation is based upon comparison tests performed by the academic partners, to evaluate the different consistency protocol implementations, and reports made by the industrial partners, which had the responsibility of designing and implementing independent testing applications, both for correctness and performance tests.

First, a brief description of the other two protocols is provided. Then, the testing conditions and test results are given. Each of the consistency protocols will be evaluated according to the following criteria:

Number of messages The number of messages used by each protocol to commit a transaction.

Transaction distribution For the total number of transactions performed at a node, the percentage of commits and aborts.

Transaction average time The average time a transaction takes to complete.

These criteria will be evaluated under three test scenarios, that attempt to emulate the effects of different usage patterns. Each scenario is run in two network configurations, LAN and WAN.

As it was mentioned in the previous chapter, because the implementation of the optimized version of the protocol, taking advantage of optimistic message delivery, was completed only at a very late stage of the project, it could not be included at the time these tests were performed.[1]

An analysis of these results is then performed, according to different expected usage patterns of COPLA, to determine what is the most appropriate protocol for each situation.

5.1 The other COPLA protocols

5.1.1 The full object broadcast protocol

The FOB protocol was created by the ITI Valência team. It uses the concept of object ownership. The delegate node of a transaction creating an object gets its ownership.

To decide the outcome of some transaction, the delegate node calls the owners of all objects used by that transaction passing the version number v read by that transaction for that object and the operation to be performed (either read or write). The owners deny permission to access the object if v is lower than the current object version hold at the object owner or if some other transaction has been granted permission to write that object.

If the delegate node collects permissions for all objects, it will reliably broadcast the write set of that transaction to every node in the system, with the updated version numbers. The reliable broadcast messages must be delivered in First-In-First-Out (FIFO) order with respect to the sender. When the permission is refused

[1]Obviously, the optimized implementation was tested for correctness, but the testing conditions were not the same, and as such the test results would not be comparable.

by at least one node, the delegate node will call those that have already grant him permission and cancel the operation.

In this protocol, all active nodes are expected to have the latest version of all objects, retrieved from the message reliably broadcasted by the delegate nodes. In the case of failure of some node, a deterministic algorithm temporarily transfers the ownership of its objects for some other node.

5.1.2 The voting protocol

The Voting was the second protocol created by the FCUL Lisboa team. This protocol uses two different messages, broadcast by the delegate node of some transaction t, one using a total order protocol, and the other a reliable broadcast protocol. The first is issued when t is requested to commit and includes the *write set*. The second is issued when the delegate node can inform the remaining participants of the final outcome for t which can be either COMMIT or ABORT (in the literature this second message is called the "vote").

Each node keeps track of changes to the objects retrieved by its transactions and crosses this information with the *write set* of every transaction for which it receives a COMMIT vote. It will abort all local transactions whose read set intersects the write set of committed transactions.

The above condition holds until the delegate node gets its first message totally ordered and decisions of all previous transactions have been issued. After this point no concurrent transaction can change the objects in the *read set* of t and it is safe to vote for COMMIT that transaction. The *Abort* message on the other hand can be issued much earlier, as soon as the *write set* of some other transaction has intersected t's read set.

Nodes receiving COMMIT decisions should apply the changes in the *write set* to their local replicas of the database.

```
1  * * *
2  * * *
3  10.101.84.1 (10.101.84.1)  0.779 ms  0.688 ms  0.618 ms
4  FCUL-7500.net.ul.pt (194.117.0.249)  1.307 ms  5.481 ms  4.785 ms
5  ROUTER25.Hssi3-0.Lisboa.fccn.pt (193.136.1.229)  1.678 ms  1.861 ms  1.781 ms
6  ROUTER7.ATM2-0.139.Lisboa.fccn.pt (193.136.1.33)  3.850 ms  8.673 ms  10.444 ms
7  ROUTER1.GE0-2-0.5.Lisboa.fccn.pt (193.137.0.11)  6.963 ms  5.488 ms  12.796 ms
8  rccn.es1.es.geant.net (62.40.103.49)  27.657 ms  12.653 ms  15.762 ms
9  * * *
10  GE1-1-0.EB-IRIS2.red.rediris.es (130.206.220.2)  12.984 ms  16.352 ms  14.229 ms
11  AT0-0-0-2.EB-Valencia0.red.rediris.es (130.206.224.6)  25.256 ms  27.022 ms  19.950 ms
12  upv-router.red.rediris.es (130.206.211.186)  210.957 ms  212.411 ms  207.150 ms
13  kabrakan-dmz.net.upv.es (158.42.255.1)  198.307 ms  197.711 ms  207.377 ms
14  atlas-rou.net.upv.es (158.42.255.33)  207.809 ms  196.410 ms  180.919 ms
15  sidi3.iti.upv.es (158.42.51.73)  195.779 ms  196.814 ms  248.047 ms
```

Figure 5.1: The output of a typical trace route between hosts in FCUL (Lisbon) and ITI (Valencia)

5.2 The testing environment

This section describes the testing conditions under which the protocols will be evaluated. First, the testbed itself is described, followed by the test applications. We also analyze the expected abort rate of the test application, to serve as a guideline for verifying test results. Finally, we shall look at each of the testing scenarios.

5.2.1 Testbed

Tests have been made in local and wide-area networks. The local area network is a 10Mb/s Ethernet connecting three computers of the research lab of the FCUL computer science department. The wide-area environment connected two hosts in Lisbon with one node at ITI facilities in Valência. Measures were taken by night. The average round trip delay is of 30*ms* with approximately 1% packet loss. A typical trace of the route followed by packets between the hosts in Lisbon and Valência is presented in figure 5.1. First two lines show each one firewall on the entry of the Computer Science Department which refuse to reply with ICMP packets. None of the firewalls disturbs nodes connectivity.

All tests are performed using three hosts: three hosts in Lisbon, for the LAN tests, and two hosts in Lisbon and one in Valência for the WAN tests. Hosts in Lisbon, here named $L1, L2$ and $L3$ have 800MHz Pentium III processors with 512MB RAM running Linux Mandrake 8.2 (kernel version 2.4.19) and Java Virtual Machines v. 1.4.1 (build 1.4.1-b21). The host in Valência (named V1) is a 450MHz Pentium III processor with 256MB RAM running RedHat 7.2 (kernel version 2.4.18) and jdk 1.4.0 (build 1.4.0-b92).

5.2.2 The test application

The test application is composed of two independent programs, one performing updates to some objects, called the *app_writer* and another querying the database, called the *app_reader*.

When it is started, the *app_writer* adds four objects of the same class to the database. Each of these objects is simply composed of an integer, initialized so that the sum of all objects in the system is 999. At random intervals ranging from 0 to 2000 milliseconds, the writer attempts to perform a transaction. Each transaction will decrease by some amount the value of one of the objects and increase, by the same amount, the value of another object, so that the sum of all objects remains constant in the system. The amount is a random integer such that the value stored in the object that is decreased remains greater or equal to zero and the final value of the increased object remains lower than 1000.

The *app_reader* queries the database also at random intervals between 0 and 2000 milliseconds and presents the values of the objects read. It also checks the integrity of the database, stopping the application if the sum of all objects becomes different from the expected constant. It is interesting to notice that *app_reader* applications perform *read-only transactions*. All protocols optimize this kind of transactions by keeping them local to the delegate node.

The set of objects read and written is configurable in both programs. One can instruct the *app_writer* to operate exclusively over the objects it has created or to move amounts between any two objects in the system. The *app_reader* can also be tailored to peek only the objects inserted by some writer or to retrieve the entire database state.

The programs proceed echoing to the screen their progress. In the case of the *app_reader*s, they notify the user about the aborted transactions or show the values retrieved from the database for the committed transactions. For each transaction, *app_writer*s describe the amount being moved, the objects operated and the transaction outcome. After 5 minutes of execution, both *app_reader* and *app_writer* output to the screen the number of committed and aborted transactions and the average time for each transaction. Programs are manually started in a fast succession. In order to prevent the latest applications to be started from benefiting from a lower degree of concurrency, the programs continue to execute after this period of time. The programs are manually stopped after every instance has printed out its final results.

5.2.3 Expected abort rate

To provide a clearer overview of the results presented, it is therefore important to estimate in theory the expected abort rate. The following paragraph provides a raw estimate of this rate using the testbed application.

The test application operates over a quite limited set of objects. Each update transaction picks at random two (distinct) objects out of twelve. The tuple space is therefore, $12 \times 11 = 132$. Two concurrent transactions will conflict if they pick at least one object in common. In the above tuple space, there are 42 tuples sharing at least one object, so the probability of this event is $\frac{42}{132} = \frac{7}{22}$. Each *app_writer* will sleep a random interval between each transaction. One also needs to consider the probability of two transactions to occur simultaneously. Figure 5.3 shows that it is reasonable to assume that update transactions will take 600*ms* to execute. The probability of some other transaction to start in this interval is $\frac{600}{2000} = \frac{3}{10}$. So, the probability of one transaction to start concurrently with the execution of another and that both share at least one object is given by the product of both values above or $\frac{3}{10} \times \frac{7}{22} = \frac{21}{220}$. All scenarios described bellow execute three writers, one at each host. Therefore, each transaction may be aborted by two others and the expected abort rate for update transactions becomes $2 \times \frac{21}{220} \approx 19\%$.

The reader component of the application always queries the entire database. Any read-only transaction will be aborted if some update transaction commits concurrently with its execution. Based on the values presented in figure 5.3, one can estimate that read-only transactions have a duration of approximately $350ms$. The abort rate of read-only transactions will be given by the probability of success of the update transaction times the probability of the update transaction to finish while the read-only is executing. The existence of three writers require that the above value be multiplied by three so the abort rate for read-only transactions becomes $3 \times \frac{350}{2000} \times \frac{89}{110} \approx 36.4\%$.

These values are substantially higher than what might be expected for typical commercial databases. One should also notice that the above values take into the calculation only the most significant factors. While some of them result from the nature of the system (like the transactions execution time), others like the number of objects or the characteristics of the read-only transaction are introduced by the test application, which was not particularly tailored for benchmarking.

5.2.4 Evaluated scenarios

The tests described here try to compare the performance of the protocols under different configurations. We now proceed to describe each of the performance tests that have been made.

Scenario #1: one active replica This scenario is used as a control for the tests realized by the industrial partners. In this scenario, only one reader and one writer are used and they query the same database replica. The remaining two nodes simply mirror the state of the active replica. Because only one program is updating the database, it is expected that the writer always commits its transactions.

Scenario #2: one owner, several clients This scenario extends the previous one by placing one reader and one writer over each replica. However, all objects have been created only by one node.

Scenario #3: shared data As in the previous scenario, one reader and writer is placed over each replica. However, both are free to randomly access any objects in the system, which could have been inserted by them or by other nodes. This is the scenario that inserts a higher degree of concurrency in the system.

5.3 Evaluation

This section compares the results achieved by each protocol in the scenarios outlined above. First, we will analyze the number of messages exchanged by each protocol, since this is independent of the usage pattern, and will help us to better understand the results obtained by each protocol. Then we shall examine the results for the other criteria under each scenario.

5.3.1 Number of messages

The number of messages required by each protocol to commit a message is presented in Table 5.1.

The number of messages used by the FOB protocol depends on the read set and

Protocol	# messages Type	Content
FOB	$2 * [0..n-1]$ PtP	use request
	1 RB	WS
Voting	1 UTO	WS
	1 URB	Decision
NonVoting	1 UTO	RS versions + WS

n: the number of replicas in the system
PtP: Point-to-Point
RB: Reliable Broadcast
URB: Uniform Reliable Broadcast
RS: Read Set
UTO: Uniform Total Order
WS: Write Set

Table 5.1: Messages issued by each protocol to commit one transaction

number of replicas in use. Two point-to-point messages are exchanged between the delegate node and each node owning at least one of the objects in the read set. The delegate node omits this step for the objects owned by himself. The low-level number of messages used by the remaining protocols is implementation dependent. Uniform Total Order, for example, requires at least three rounds of messages. Each round may be implemented in one single message if IP Multicast is used (what is well suited for LANs) or over several messages if standard point-to-point UDP or TCP protocols are used. The following sections will analyze this subject with greater depth when it shows to be advantageous for the reader.

5.3.2 Scenario #1

The rate of committed and aborted transactions for scenario #1 is presented in figure 5.2. As expected, none of the protocols aborted any update transactions because there are no concurrent updates to the database.

Figure 5.3 shows that the FOB protocol achieves a lower transaction average time, particularly on LANs. Further below we will see that the difference be-

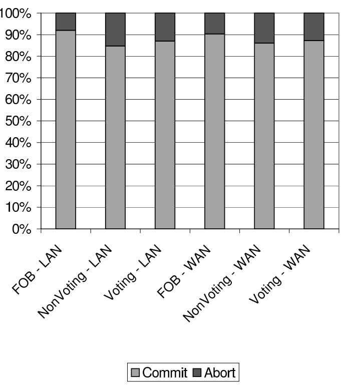


Figure 5.2: Read-only transactions distribution in scenario #1

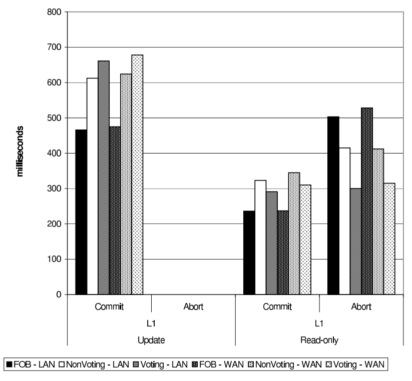
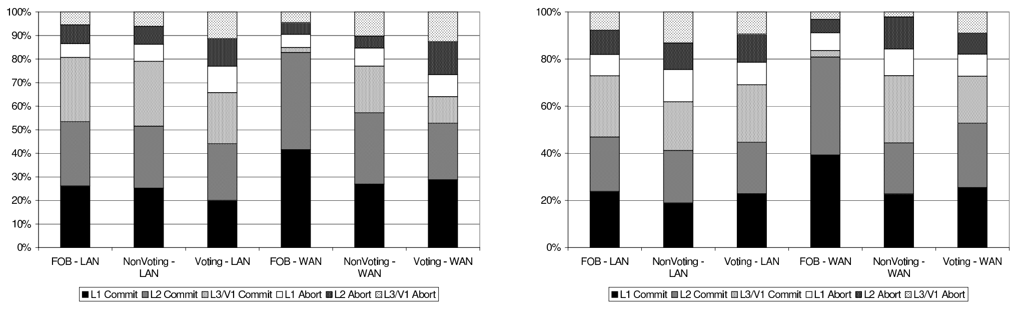


Figure 5.3: Transactions average time (ms) in scenario #1

tween FOB and the remaining two protocols is greater in the industrial partner's performance tests, because they introduced a much higher load in COPLA, using simultaneously several writers. The number of messages required by each protocol is reflected in the average transaction times. Since node $L1$ owns all objects and is the delegate server of every transaction, it does not need to send any message requesting permission to change the objects. In this case, FOB issues a single Reliable Broadcast message with the state update after committing the transaction. Update transactions require one uniform total ordered message for the NonVoting protocol and two for the Voting protocol. Uniform total order protocols require



update read-only

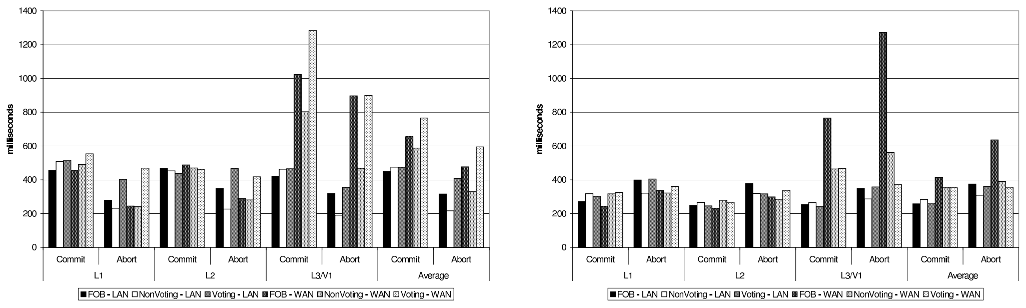
Figure 5.4: Transactions distribution in scenario #2

at least three communication steps for each message and this cost is reflected on the average update transaction time. All protocols query the local database state for read-only transactions, so no messages are exchanged. The values presented for read-only transactions are related with the implementation complexity to ensure transaction consistency. One can notice that, in general, aborting a read-only transaction is a more time consuming operation than committing it. Notice that the theoretical estimate of the abort rate is consistent with these results. The value presented before should be divided by 3 because only one writer is updating the database, which suggests values around 12%.

The results for each protocol are quite similar when one of the nodes suffers from high latency w.r.t. the remaining. In this scenario, the remote replica is passive in the sense that it does not provide any contribution to the protocol. Its work is limited to receive and apply database updates.

5.3.3 Scenario #2

The results for scenario #2 were taken creating all objects in node $L1$. Figures 5.4 and 5.5 present respectively the outcome rate and the average time for this scenario.



update read-only

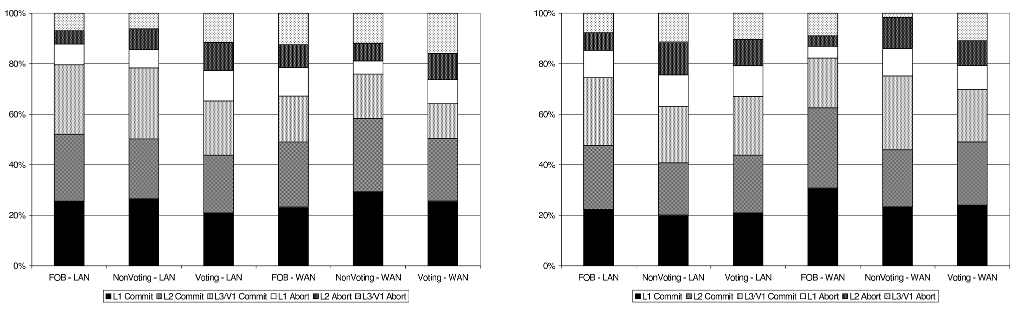
Figure 5.5: Transactions average time (ms) in scenario #2

The ownership used by FOB does not hurt significantly nodes $L2$ and $L3$ but becomes visible for the remote node $V1$. The contribution of this node for the total number of executed transactions is minimal. The NonVoting protocol is the one that shows to be less prejudiced by having a remote node: the execution time of committed and aborted transactions is almost stable across all nodes and for either update or read-only transactions and it is the one who presents a lower time for committed and aborted update transactions in $V1$.

The advantages of using LANs are also visible in the higher success rate of read-only transactions for the WAN tests: because the number of committed update transactions decreases, the probability of aborting read-only transactions also becomes lower. The use of two total order messages for committing a transaction significantly hurts the number of committed transactions for the Voting protocol: on average, transactions take longer to get their outcome in the Voting protocol.

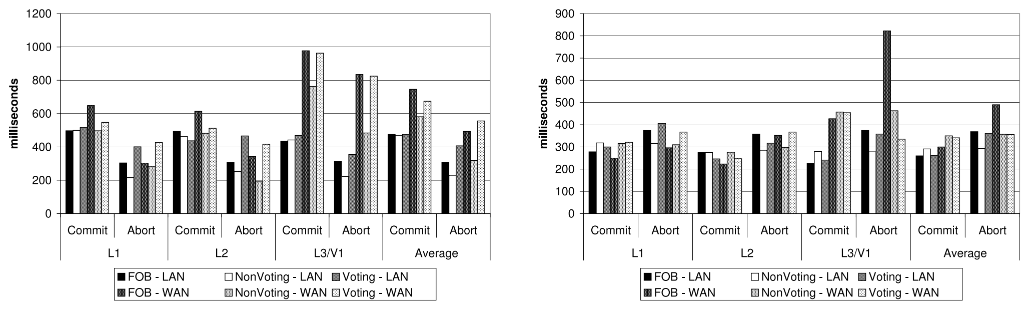
5.3.4 Scenario #3

The results for scenario #3 are presented in figures 5.6 and 5.7. In all tests. Voting and NonVoting protocols used a sequencer based total order protocol. In this protocol, the definition of the message order is centralized in one of the participants,



update read-only

Figure 5.6: Transactions distribution in scenario #3



update read-only

Figure 5.7: Transactions average time (ms) in scenario #3

named the sequencer. In these runs, the sequencer node was $L2$. An implementation feature explains the good results achieved by both protocols in this node: the messages issued by the sequencer are immediately ordered before going into the network. The Voting protocol is the one who most benefits because it uses two totally ordered messages for each update transaction.

The impression left by scenario #2 that the FOB protocol is not particularly sensitive to data distribution in LANs is confirmed: it presents the best average results for both update and read-only commits and the lower abort rates. However, when used over a WAN, FOB performance begins to degrade. It presents the worst performance for committing update transactions on all nodes when one of them is remote. While the commit time increases for every protocol, FOB is the one for whose values grow more substantially. For the Voting and NonVoting protocols, the presence of a remote node is almost negligible for those nodes near the coordinator. As expected $V1$ performs worst in these protocols but the two totally ordered messages penalize more substantially the Voting protocol.

Although the substantial increase of the transaction duration when comparing the LAN and WAN environment, the commit rate of the NonVoting protocol does not decrease substantially, remaining above the Voting protocol rate in LANs.

The locality of the read-only transactions makes its performance almost unaffected by the distance of the nodes. As usual, the commit rate grows in inverse proportion of the performance for the update transactions.

5.4 Analysis of the results

Previous scenarios have shown that several different factors can influence the consistency protocols performance. This section summarizes previous results by making some suggestions on the adequacy of each consistency protocol for different usage patterns. These consider the following factors:

- *number of replicas*;
- *system load* that accounts for the number of requests to the system per unit of time;
- *system coverage* either LAN or WAN;
- *primary data access location* which is concerned with the number of accesses to objects created in a different node.
- *concurrency* measures the degree of concurrent access to the same set of objects, what could result in a significant number of transaction aborts.

Name	# replicas	load	coverage	access	concurrency
Small	small	light	LAN	local	small
Med-local	medium	medium	LAN	local	small
Med-wid	medium	medium	WAN	distributed	high
Large	large	high	WAN	local	high

Table 5.2: Combinations of some factors that influence the decision of the consistency protocol to use in COPLA

A set of possible scenarios combining these factors have been devised and are presented in table 5.2. Each of these combinations will be explained bellow.

Small: When all nodes are located in one LAN the FOB protocol shows to be the most adequate. The scenarios above do not show a significant distance between FOB and the remaining protocols, however, the distance between both protocols is emphasized when all protocols are placed under a higher load. The results from industrial partners also show that both the Voting and NonVoting protocols require more memory and processing time, what makes them inadequate for situations where load peaks occur, for example in web applications.

Med-local: As the number of hosts grows, the FOB advantage will depend on the distribution of the creation of objects. In this protocol, the number of messages per transaction varies with the number of object owners: those that created the objects involved in a transaction. If the number of objects is stable and they were mostly created in one single node[2] then the number of messages used by FOB will always be small and it will keep the advantage against the remaining protocols. However, if object instantiation is scattered against a large number of replicas and the typical transaction requires access to objects created in several nodes, the

[2]for example, this will be the case when the data has been ported from an existing relational database by a specialized application

NonVoting protocol may be preferable. To prevent the number of messages for this protocol from growing with the number of replicas, it is advisable to use the IP Multicast module. This module brings no benefit to the protocols for a small number of replicas. The Voting protocol performance will have a greater increase with the utilization of IP Multicast than the NonVoting protocol.

The NonVoting protocol performance will degrade for transactions having a huge number of objects in their read sets. In this case, it will probably be necessary to fragment the low level datagram, increasing message transmission time. This case will also be unfavorable for the FOB protocol when data is scattered across a large number of replicas. The Voting protocol is unaffected by the dimension of the read set. It is probable that the Voting protocol achieve a better performance in this situation.

Med-wide: This case extends the previous one by distributing the replicas across a Wide Area Network. As scenario #3 shows, the NonVoting protocol provides a better performance than the remaining for both update and read-only transactions. The lack of support for IP Multicast in the Internet prevents this optimization from being applied in this case. However, the indulgent total order protocol will reduce the asymmetries between the nodes in the LAN where the sequencer resides the remaining by virtually placing one sequencer in each LAN. While this will not hide the latency from the system, it will improve performance over the conventional sequencer algorithm used in the tests on this report.

As in the previous case, the size of the read set will affect both the FOB and the NonVoting protocol, possibly turning the Voting protocol to be the best choice.

Large: In this case, nodes are grouped in clusters connected by LANs, and each cluster is connected to each other by a WAN. The case where each node cluster will access mostly to its own data, although it is replicated over all other units was studied in the WAN tests for scenario #2. FOB has shown to be the most favorable protocol in this situation. However, it should be noticed that FOB provides a highly unfair environment for hosts outside the LAN that will see a huge number of their transactions attempting to access foreign data aborted and suffer a high delay. Collecting system-wide data (statistical analysis, for example) shows to be difficult when the load is high in some node clusters.

5.5 Protocol comparison conclusions

As shown in the previous sections, the protocol presented in this work operates well in the operating scenarios devised for COPLA. It is the best choice for one of the three cases considered above, and can be a good contender for other scenarios, depending on the particularities of the transaction load.

5.6 Summary

In this chapter we evaluated the performance of the NonVoting replication algorithm against other consistency protocol implementations for COPLA. We showed that the algorithm performs well for some expected usage patterns, thus being a useful addition to the COPLA system. The next chapter finalizes this thesis, and establishes some guidelines for future work.

Chapter 6

Conclusion

Replicated databases are becoming increasingly common. They are no longer restricted to small clusters of machines, sitting next to each other in one room, connected together by a very high speed network. The increasing internationalization of businesses and organizations demands that database systems expand with them. Although large-scale data replication solutions existed before, they often traded relaxed consistency semantics for acceptable performance.

Recently, developments in group communications technology have shown us that we can offer interesting ordering guarantees on broadcast messages, while maintaining good performance levels, even on wide scale area networks (WANs). Taking advantage of these developments, several authors proposed to take advantage of these efficient communication primitives to build replication algorithms that offer strong semantic guarantees, and also good performance on WANs.

This thesis approached the problem of replica management in a distributed object-oriented database system. It presented a protocol to ensure data consistency across the different nodes of the system. This protocol relies on recent advances in group communication techniques, and the use of atomic broadcast as a building block to help serialize conflicting transactions.

This work reached a series of important goals.

- It contributed to the design of the COPLA tool, enabling it to make use of several different consistency protocols, for maximum flexibility.
- It successfully adapted an existing replica consistency protocol, and implemented it within the established framework.
- The resulting implementation was evaluated against competing protocol implementations, and found useful on some of the operating scenarios for COPLA, thus proving the validity of this work.

6.1 Future work

Despite the fact that COPLA is complete, and the GLOBDATA project is over, there are still important issues to pursue with this replication technology, maybe even independently of the COPLA tool.

Although the presented solution was complete, and implemented, it did not address the problem of failed node recovery. Evidently, the proposed replication management solution would not be complete without a recovery algorithm. We have defined and implemented a simple approach to recovery. This simple recovery algorithm is described in the extended version of [30], but would require further work to offer optimal performance.

Also, there is an optimization, which although it was considered in the published papers [31, 30], was not implemented, mostly due to lack of time. This optimization, named deferred updates, aimed to exploit the *locality* of data, that is, the fact that on some replicated systems, distinct subsets of data are only changed at one node, while the other nodes simply read it (one example of this is sales records of a company with delegations in several countries). To exploit this, when a transaction that altered some data committed, the message it sent would contain only the identifiers of the changed objects, and not the transaction updates themselves. The other nodes would keep a record of the node that had the latest version of the data, and would request it in case it was needed.

The other direction of research would be to adapt the current implementation to work on a regular SQL database. I believe that the current implementation is abstract enough that no significant changes would be required, and the fact that other systems [2] have proposed similar architectures is encouraging support.

Bibliography

[1] Y. Amir and J. Stanton. The Spread wide area group communication system. Technical Report CNDS 98-4, Center for Networking and Distributed Systems, Johns Hopkins University, Baltimore, U.S.A., 1998.

[2] Y Amir and C. Tutu. From total order to database replication. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, July 2002.

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

[4] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th Symposium on Operating System Principles*, pages 123–187, November 1987.

[5] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

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

[7] Johns Hopkins University Center for Networking and Distributed Systems. http://www.cnds.jhu.edu.

[8] T. Chandra and S. Toueg. Unreliable failure detecturs for reliable distributed systems. *Journal of the ACM*, 43(1), March 1996.

[9] 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.

[10] M.J. Fisher, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

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

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

[13] 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.

[14] R. Guerraoui. Revisiting the relationship between non blocking atomic commitment and consensus problems. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG-9), LNCS 972, Springer Verlag*, pages 87–100, Le Mont Saint Michel, France, September 1995.

[15] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, Computer Science Department, 1998.

[16] 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.

[17] R. Jiménez-Peris, M. Patiño Martínez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems 2002 (ICDCS'02)*, Vienna, Austria, July 2002.

[18] 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.

[19] B. Kemme and G. Alonso. Transactions, messages and events: Merging group communication and database systems. In *Proceedings of the 3rd ERSADS European Research Seminar on Advances in Distributed Systems*, Madeira Island, Portugal, April 1999.

[20] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the 26th International Conference on Very Large Databases*, Cairo, Egypt, September 2000.

[21] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[22] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, June 1994.

[23] M. Patiño Martínez, R. Jiménez Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proceedings of DISC'00, LNCS 1914*, pages 315–329, Toledo, Spain, October 2000.

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

[25] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. Technical Report SSC/1999/008, École Polytechnique Federale de Lausanne, Switzerland, March 1999.

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

[27] PostgreSQL. http://www.postgresql.org.

[28] A. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 341–352, Montreal, Canada, August 1991.

[29] 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.

[30] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. The GlobData fault tolerant replicated distributed object database. In *Proceedings of*

the First Eurasian Conference on Advances in Information and Communication Technology, pages 426–433, Teheran, Iran, October 2002. An extended version is available at the author's home page, at http://www.di.fc.ul.pt/˜ler.

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

[32] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS-13)*, pages 561–568, Pittsburgh, Pennsylvania, USA, May 1993. IEEE Computer Society Press.

[33] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine. In *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA 2001)*, Cambridge, MA, USA, October 2001.

[34] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[35] P. Vicente and L. Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, pages 92–101, Osaka, Japan, October 2002.

[36] R. Vitenberg, I. Keidar, G. Chockler, and D. Dolev. Group communication specifications: A comprehensive study. Technical Report CS99-31, Comp. Sci. Inst., The Hebrew University of Jerusalem note =.

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