Implementable Models for Replicated and Fault-Tolerant Geographically Distributed DataBases

Consistency Management for GlobData

Memory of the Ph.D degree

written by

Luis Irún Briz

Supervisors:

Prof. José M. Bernabéu Aubán

Dr. Francesc D. Muñoz i Escoí

Departamento de Sistemas Informáticos y Computación

Universidad Politécnica de Valencia

Valencia, Spain

July 15, 2003

Abstract

(English Version)

Lazy update protocols have proven to have a poor behavior due to the high abortion rate they produce in scenarios with a high degree of conflicts in the access to the information. This work studies lazy update protocols from a conservative point of view with respect to its applicability in the field of Distributed Databases.

To this end, a result of incompatibility between “laziness” and “serializability” is included, proving the necessity for the transactional guarantees provided by the consistency protocols to be relaxed, when this consistency management is not performed in an “eager” style.

On the other hand, this work also includes a study of the problem of the abortion rate from an statistical point of view, and being used as the basis for a new “not-so-lazy” approach, based on statistical properties of the abortion rate. In this sense, an exhaustive description of this new protocol is also included, in addition to an empirical evaluation of its implementation. These experiments are also used to experimentally validate the argumentation proposed.

The proposed protocol will be shown to produce lower abortion rates than the lazy update protocols, increasing -even at the time for some scenarios- the productivity of the system. An expression for these improvements will be also argued.

Finally, the same principles are used to provide a mechanism for providing a self-recovery ability to the fault tolerance aspect of the protocol. In addition, this recovering process avoids the use of log’s during the failures, and its behavior is also “not-eager”, avoiding the necessity of blocking transactions started in any node in the system (even in the recovering one).

(Versión Castellano)

Los protocolos de actualización perezosa han demostrado tener comportamientos inadecuados a causa de sus altas tasas de abortos en escenarios con un alto grado de conflictos en los accesos a la información. El presente trabajo aborda los protocolos de actualización perezosa desde un punto de vista crítico con respecto a su aplicabilidad en bases de datos distribuídas.

Para ello, se incluye un resultado de incompatibilidad entre "pereza" y "serializabilidad", que demuestra la necesidad de relajar las garantías transaccionales proporcionadas por protocolos de difusión no-ávida.

Por otro lado, también se incluye en este trabajo un estudio del problema de la tasa de abortos desde un punto de vista estadístico, que sirve como base para una nueva aproximación "no tan perezosa", basada en el comportamiento estadístico de la tasa de abortos.

En este sentido, una descripción exhaustiva de dicho protocolo es también incluída, junto con resultados experimentales de su implementación, que son también usados para validar experimentalmente los razonamientos usados.

El protocolo propuesto produce menores tasas de abortos que los protocolos de actualización perezosa, incrementando al mismo tiempo -en determinados escenarios- la productividad del sistema. Una expresión para esta mejora es también justificada convenientemente.

Por último, los mismos principios son usados para proporcionar un mecanismo de recuperación "no-ávida" de errores, que evita el bloqueo de cualquier nodo existente en el sistema (incluso el reincorporado), así como la gestión de "log's" en los nodos supervivientes.

(Versió Valencià)

Els protocols d'actualització peresosa han demostrat que tenen comportaments inadequats per causa de les altes taxes d'abortaments que presenten en entorns amb un alt grau de conflictes en els accessos a l'informació. Aquest treball aborda els protocols d'actualització peresosa des d'un punt de vista crític respecte de la seua aplicació en bases de dades distribuides.

Per tal cosa, s'inclou un resultat d'incompatibilitat entre "peresa" i "serialitzabilitat", que demostra la necessitat de relaxar les garanties transaccionals proporcionades per protocols de difusió no àvida.

D'altra banda, també s'inclou en aquest treball un estudi del problema de la taxa d'abortament des d'un punt de vista estadístic, que serveix com a base per a una nova aproximació "no del tot peresosa", basada en el comportament estadístic de la taxa d'abortaments.

En aquest sentit, una descripció exhaustiva de tal protocol es també inclosa, junt amb resultats experimentals de la seua implementació, que seràn també utilitzats per a validar experimentalment els raonaments plantejats.

El protocol proposat produeix menors taxes d'abortaments que els protocols d'actualització peresosa, incrementant al mateix temps -en determinats escenaris- la productivitat del sistema. Una expressió per aquesta millora es també raonada convenientment.

Per a concloure, els mateixos principis són utilitzats per a proporcionar un mecanisme de recuperació "no àvida" de errades, que evita el blocatge de qualsevol node existent al sistema (inclós el reincorporat), com ara la gestió de "log's" als nodes supervivents.

Agradecimientos

A mí se me da muy mal decir cosas brillantes. Sin esa pretensión, me gustaría agradecer el apoyo recibido y la ayuda prestada por muchas personas.

Para empezar, gracias a Alicia, mi esposa, mi amiga, y mi sufridora. Gracias a mis padres, que con muchos esfuerzos y sufrimientos me han dado todo lo que soy, y las herramientas para vivir honrada y justamente. Gracias también a Jesús, que me enseñó que la vida es mucho más que trabajo y responsabilidades.

También quiero agradecer especialmente a Paco su apoyo y consejos, sin los cuales este trabajo no hubiera salido adelante tan rápidamente, y a Pepe, cuya gran experiencia y claridad de pensamientos han sido principales para este trabajo.

Por último, me gustaría que estas líneas expresaran también la fuente última de mis esfuerzos, que encontré hace algún tiempo, y con la que he podido cultivar fortaleza de espíritu, capacidad de comprensión y de perdón. Hablo así del Aikido, una filosofía Japonesa, que el maestro Morihei Ueshiba recopiló y depuró hace ya un siglo, con el objetivo de transformar un arte marcial en un arte de paz, de armonía y amor.

List of Figures

List of Tables

List of Acronyms

API	Application Program Interface
COLU	Cautious Optimistic Lazy Update
COLUP	Cautious Optimistic Lazy Update Protocol
COPLA	Concurrent Object Platform
CORBA	Common Object Request Broker Architecture
DBMS	Database Manager System
FIFO	First In First Out
FOB	Full Object Broadcast
GODL	GlobData Object Definition Languaje
GOQL	GlobData Object Query Languaje
LOM	Lazy Object Multicast
LOMP	Lazy Object Multicast Protocol
ODL	Object Definition Languaje
ODMG	Object Data Management Group
OQL	Object Query Languaje
UDS	Uniform Data Store

Contents

Chapter 1
Introduction

1.1 Introduction

Fault tolerance and performance enhancement are two of the most important advantages derived from the use of techniques developed in distributed systems. To achieve these goals, replication of the information has proven to be a powerful and versatile approach, providing a wide range of algorithms fitting the needs of a number of problems and applications. In the area of distributed databases, consistency requirements introduce a new parameter in the problem, because the system must also manage the replicated information within a particular semantics. The proposed solutions in this field used to be centered in the consistency management as the main problem to solve.

In addition, consistency[Ell77, Sto79] of replicated data is the essence for query answering in distributed databases with multiple copies of data objects. Also a high availability of answers to queries is essential for many networked applications.

For tightly interconnected databases, consistency maintenance of replicated data is a fairly well-understood problem [BHG87a, WJ92]. Traditional approaches for replicating databases usually deploy fast local area networks (LANs) [KA98, KB94, JM90, Her90, Her87], using network-intensive protocols. However, for less tightly connected (e.g., internet-based and, more generally, wide-area-networked) applications, where the geographical spread of the data distribution involves larger distances, the network tends to be a limited resource. This means that, for wide-area networks (WANs), the issues of maintaining consistency and availability need to be addressed differently from solutions for LANs [LLSG92, FMZ94b, FMZ94a].

Nowadays, many distributed (i.e., networked) applications have to manage large amounts of data. Despite the increasing ubiquitousness of information, the access patterns to distributed data often feature a noticeable degree of geographical locality. Moreover, many applications require a high degree of availability, in order to satisfy the need of offering services at any time, to clients that are either internal or external to the networked application. A predominance of locality of access patterns usually suggests a partitioning of the database [Rah93, GHOS96]. In many scenarios, it may be convenient or even necessary to replicate the information in a set of servers, each one attending its local clients. The different replicas must then be interconnected, a WAN being usually the best-fit architectural option.

The typical kind of applications we have in mind are, for instance, widely distributed databases in wide area intranets of medium and large enterprises, for data warehousing or resource management, as well as extranet service provisioning of such enterprises. Examples where such intranets of databases are deployed advantageously are enterprises with several branch offices (e.g., banks, chains of retailers, super- and hypermarkets), telecommunication providers, travel businesses, logistics, etc. Examples of extranet services which benefit from replication protocols as discussed in this paper are customer relationship management, e-banking, virtually all kinds of e-business as well as most e-government applications. Common to all of these applications is that potentially huge amounts of data are maintained and replicated on distributed sites, while access patterns (or at least significant contingents thereof) are highly local. Efficiency and high availability of such services is key to their acceptance and success.

The GlobData project [IUF01, RMA$^+$02] strives to provide a solution for the kinds of applications just outlined. It does so by defining a specific architecture for replicated databases, together with an API and a choice of consistency modes for data access.

In distributed databases, different solutions have been proposed in the literature to conciliate performance, fault-tolerance and consistency management. In particular, a set of algorithms for replication control [WSP$^+$00] (known as *update everywhere* protocols) tries to maximize the use of every node in a replicated system, by means of the load balancing of the execution of each transaction in the system. This makes possible for every node in the system to execute a different transaction at the same time, but it becomes necessary, once the transaction is locally committed, to propagate its changes to the rest of the system. These techniques provide an improvement of the performance, but make it necessary to introduce some kind of consistency control in the system, and some kind of technique for the propagation of the updates performed in the executing node to the rest of the system.

In the GlobData project, a number of consistency protocols are proposed, that are capable of meeting the consistency requirements of the applications just outlined. Although the presented protocols differ in the priorities of their particular goals, each of them share a common characteristic: They are (more or less) optimistic (or, at least, cannot be classified as pessimistic), since transactions are allowed to proceed locally and are checked for consistency violation only at commit time. When a consistency violation is encountered, the transaction is aborted.

During the development of the project, we encountered that the protocols necessities of our target scenario (i.e. geographical distributed applications with a high degree of locality) could be well-fitted with a family of replication protocols classified as *lazy update protocols*. However, we encountered that this kind of protocols introduces a high number of abortions in the system, because they propagate the updates beyond the commit phase (in contrast to *eager update protocols*, where the whole system is updated inside the commit phase), making it possible for a transaction to read out of date data, and thus, there are consistency violations. But, an additional inconvenience was found for the *lazy update protocols*: the use of such techniques makes it impossible to guarantee one-copy serializability, unless the protocol is redesigned, in which case it becomes unusable due to the huge number of aborted transactions it introduces.

In this chapter, a number of basic concepts are presented, in order to establish a starting point for the discussion, and properly center the problem. To this end, section 1.2 presents the concept of Distributed Databases, and the model of applications that we are considering. In section 1.3 a number of techniques used in distributed databases, and described in the literature are summarized from two different points of view. Then, section 1.4 centers the *update propagation* problem. Finally, section 1.5 presents the contributions of the thesis, and section 1.6 concludes with the layout of the following chapters.

1.2 Distributed Databases

A **Distributed Database** is a distributed system where an information repository is managed among the nodes of the system. The way in which the nodes hold each local portion of repository, and the interoperability established between such nodes to manage aspects such as consistency of the maintained information, can differ from one approach to another.

Typically, a Distributed Database is a set of databases stored on multiple computers that appears to applications as a single database. Consequently, an application can simultaneously access and modify the data in several databases in a network. Each database in the system is controlled by its local server but cooperates to maintain the consistency of the global distributed database.

Distributed Databases comprise a particular field of Distributed Systems, where performance, and availability are not the only features studied by the researchers. In contrast, the goals of Distributed Databases are also centered in the maintenance of some critical properties of the algorithms as *consistency* of the managed information, or *serializability* of the executed operations.

1.2.1 Principles of Distributed Databases

Distributed Processing

Distributed Processing occurs when an application system distributes its tasks among different computers in a network. For example, a ”client-server” application typically distributes front-end presentation tasks to client nodes and allows a back-end database server to manage shared access to a database.

Distributed Databases make use of Distributed Processing to enhance features as performance, or to implement the cooperation between the different nodes in the system to achieve properties as consistency.

Database Replication

In order to provide availability to the database system, it is common to use *Database Replication*. The main difference between a pure *Distributed Database* and a *Replicated Database* is that, in a pure Distributed Database, the system manages a single copy of all managed database objects. Thus, Distributed Database Applications typically implement distributed transactions in order to gain access to both local and remote data and modify the global Database. In contrast, Replicated Database Applications manage multiple copies of the data, and the initiated transactions require additional effort in order to synchronize the value of all the replicated instances for each accessed object.

Replicated Databases

Thus, *Replication* is the process of copying and maintaining database objects in multiple databases that conform a Distributed Database system. To implement replication, there have been discussed several distributed database technologies, and this makes possible for database replication to offer a number of benefits to applications that were not possible with the use of a pure Distributed Database system. In particular, replication is commonly useful to improve the **performance** of the system, and it also enhances the **availability** of the database system. These enhancements can be provided because replication enables the access to multiple instances for the same data, and it benefits the *Distributed processing* in the system. For example, when replication is used, applications might normally access to local replicas of the database rather than a remote server. This minimizes the network traffic and improves the performance when there is a reduced number of write accesses in the system. Furthermore, a client application can continue its execution even when some server in the system experiences a failure, if any other server -with replicated data- remains accessible.

Homogeneity of a Distributed Database

In an *Homogeneous Distributed Database*, all the nodes are running the same DBMS to manage the local database. In addition all the nodes use the same communications software. For instance, a system where all of the nodes use PostgresSQL over TCP/IP. In contrast, in a *Heterogeneous Distributed Databases*, there are differences among the nodes with respect to the running DBMS , or the communications software.

Access Patterns

User applications perform their accesses to the information managed by a Distributed Database following certain *Access Patterns*. The relevance of these access patterns lies in the fact that the underlying Distributed Database can take advantage of the knowledge of such information to improve some features as, for example, performance or availability.

For instance, an application managing productivity of a number of work positions in a factory, makes intensive use of a reduced subset of database objects: the different "work positions". The rest of objects contained in the database, although necessary for the application, may be accessed with a lower frequency. The distributed database should provide service to any involving department, that gain read access to any work position, but only updates the information of the work positions included in the particular department. For applications with this kind of access patterns, a feasible approach should consider an architecture where accesses to this reduced subset of objects (i.e. the work positions) are benefitted in terms of performance with independence of the location of the initiating node. In addition, the updates initiated from a particular department should be also benefitted, with the minimal interference to the rest of the system.

As seen, the access pattern may help the designer of particular Distributed Database Solutions to employ a particular approach in order to benefit the user applications the system will offer its services to.

Access Locality is a particular parameter of Access Pattern determined by the trend of a system to classify the set of managed objects within a number of subsets, each one corresponding to the different nodes that initiate requests in the system.

Failures, Faults, and Errors

A failure appears[Nel90] in the system when the particular functions for which the system was designed cannot be accomplished, due to the presence of some error in its own, or in its environment, and these errors have been caused by different faults.

Thus, we can define *fault* as an anomalous condition, and the *error* will be a consequent disfunction of a fault in the system.

A system trying to minimize the number of errors should be designed to tolerate faults. Thus, the system behavior should be described for a wide range of situations, even including faults in the system. In addition, any component in the system should be able to detect not predicted faults, and mask them to the rest of the system, thus avoiding the propagation of an error. This behavior, however, is not always possible, because some faults cannot be processed or predicted in a proper way, or the system cannot reach a consensus on the presence of such faults[CHT92, HR99, LFA00].

Availability and Fault Tolerance

A system is considered *Fault Tolerant*[Cri91a, Cri91b] when its behavior is well defined in presence of faults, or when the system masks the faults of its components to the user applications. In other words: *Fault Tolerance* is the ability for a system to continue providing to its client applications the agreed (i.e. expected by the client with the correct behavior) services even in presence of faults of its components.

As seen above, Fault Tolerance has a strict definition, because it requires the supervision and control of any fault produced in any component of the system. In addition, user applications can also introduce faults in the system (due to programming errors, or any other reason), and the system should also manage this kind of faults.

Moreover, due to the difficulty for a system to provide fault tolerance, and the impossibility to predict any fault the system may suffer (even produced by user applications), some faults will be observed in some cases by other components in the system, or by user applications.

Stabilization

When a fault occurs, causing an error of some component in the system, there is often initiated an algorithm to recover the global normality of the system. This process, known as Failure *Stabilization*[Dij74, AH93], performs a number of tasks designed to rebuild the distributed structures needed by the underlying algorithms executed in the distributed system. In many cases, this reconstruction requires the cooperation of each node in the system[BG95], causing an interference in the services provided to user applications. In some situations, it may make unable the system to attend requests during the recovery.

The lower the stabilization time is, the more available the system becomes for user applications, and the best quality of service the system offers.

Failure Recovery

When a faulty component is re-included in the system, a particular Stabilization process must be performed. This process will be considered with special interest. In such situations, known as *Failure Recovery*, despite no fault can be considered here, the system must be reconciled with the recovered component, reconstructing the distributed structures in a similar way that the Failure *Stabilization* required. In the case of Failure Recovery, the time required by the stabilization process can be longer, because it may require a higher number of transmissions, needed for the reconciliation.

This makes Failure Recovery a critical issue in terms of availability, that a distributed system must take into account, in order to provide a good quality of service.

System Partitions

Within a distributed system, with a number of interoperating nodes, the interconnection network may also suffer faults. These kind of failures, mentioned in the previous sections as “faults in the environment”, can produce the isolation of a single node, that can be considered as a fault in the isolated node. In other cases, network faults may produce the isolation of a set of nodes. In the latter situations, it is considered that the system has been *partitioned*.

Partitioning is a particular problem to solve in distributed systems, because the nodes included in both partitions may be correct, and it is possible for them to consider the set of nodes to which its is unable to gain access as “faulty nodes”. Thus, both subgroups of nodes would be able to proceed without synchronization of one set with the other. As a result, both subgroups (or partition) will proceed with further updates locally initiated, dealing with inconsistent state of the whole system. This situation will become an important inconvenience when both subgroups are joined again, and a reconciliation must take place.

If the system allows any partition to proceed, the reconciliation process must take it into account, making it harder to accomplish and, in some cases, it is impossible to solve without human intervention.

To avoid this, a commonly used solution consists of allowing only one of the partitions to proceed, forcing the other partition to “freeze” its activity. One of the most frequently used criteria to determine if the partition is trusted to continue is based on the number of included nodes in the partition: a node can only proceed if it can gain access to -at least- a half of the nodes included in the original group.

This approach presents two main disadvantages:

- It becomes necessary to predefine the original number of nodes included in the system.

- It is possible for the entire system to be suspended. For example, this can occur in a system with N nodes, and a failure of two nodes occur. Then, if the system suffers a partition including in each subgroup $\frac{N}{2} - 1$ nodes (the number of total correct nodes is $N - 2$), both partitions will be considered as "minority partitions", and no node will attend requests, causing the entire system to be stopped.

1.2.2 Features Provided by a Replicated Database

In a Database system a number of benefits can be obtained with the use of an adequate Replicated Database. Nevertheless, in order to properly exploit these benefits, it is also important to keep in mind a number of dangers that a distributed database (and in particular, a replicated database) introduces in the architecture.

The main benefits that a Replicated database provides can be summarized as:

- **Service Improvement**, with respect to the quality of the service (QoS). The use of a Replicated Database, as seen above, enhances the performance, availability, and other parameters relating to the QoS.

- **Reduced Computing Cost**, due to particular characteristics of the client applications, an adequate configuration of a Replicated Database may minimize the interference between the different nodes in the system, taking advantage of the locality of access of the client applications.

- **Improved Resource Use**, as, for example, network consumption. This can be achieved by the use of replication, avoiding unnecessary transmissions when the local database server has a suitable replica of the required information. In addition, some particular techniques can be also used to reduce even more the amount of transferred data. These techniques are commonly based on the locality of accesses of the client applications, and make use of the concept of laziness for the propagation of the changes performed by a local transaction. The concept of laziness will be treated in detail later.

  If the Database is not Replicated, the items maintained in the Distributed Database will be held in different nodes. In these systems, if a transaction is executed in a node where the required information is not contained in, there becomes necessary to perform some kind of communication in order to complete the transaction execution. On the other hand, when the information is replicated along the nodes in the system, the necessity of communication can be reduced.

- **Control of Resources**, due the distributed system can implement a better load balance[CA82], the resources in the system can be controlled in a more accurate way.

This also increases the chance for the system to manage efficiently the resources, and the performance can also be improved.

- **Scalable Architecture**, in contrast to centralized solutions, where the system performance cannot be improved indefinitely, an adequate design of a Distributed Database can benefit the system with a scalable behavior. The increase of nodes participating in the Distributed Database Server might provide a proportional improvement (under certain limits) in critical characteristics as performance and availability of the system.

- **Decreased Response Time**, as one of the consequences of increasing the performance. Depending the used replication technique, and the consistency control implemented in the system, there can be benefited certain access patterns of user applications. For instance, to improve the response time of user applications that make intensive reads, it can be benefited read operations in the database, by intensively replicating the information along the nodes in the system.

- **Increase flexibility**, with respect to Fault Tolerance, and availability of the service. A Replicated Database improves these features of a system, and makes it easier to manage changes both external and internal, with stability of the provided services.

- **Promise of reduce hardware cost**, because a Distributed Database can be implemented over the concept of ”many cheap nodes”, based on the feasible scalability of Distributed Databases, and their capability of heterogeneity, the nodes containing the Distributed Database can be added, removed, and replaced with a lower cost than a centralized solution.

- **Rapid Application Development**. The development of proprietary solutions to include in a database application the advantages provided by a distributed database (as performance, or availability), constitute an additional effort during the life-cycle of the application. In contrast, the use of Distributed Databases enables a database application to transparently access to the information, with independence of the implementation of the underlying information repository. This separation between database access and database implementation is performed through the use of interfaces, that may conform some standard, to enable the application to gain access in a transparent way to the maximum power of the used Distributed Database.

  For instance, a database application using an Application Server to gain access to several databases, may implement availability by the use of some services provided by the Application Server. This implementation of Replication, however, will introduce additional costs in the implementation of the application. In contrast, if a Replicated Database is used to implement availability, the application can be implemented with independence to the underlying infrastructure provided by the database system.

- **Reduce development cycles**. In addition, the use of an existing solution for the Distributed Database makes it more efficient the development cycles of a database application. Comparing it with a “client-server” application, where a number of additional components must be designed, implemented, integrated and tested, the use of a Distributed Database centers all the efforts in the development of the client application. This simplifies the problem, and increases the productivity of the development cycle.

On the other hand, the use of Distributed Databases may introduce a number of inconveniences in the system. These inconveniences must be considered as real risks during the design, utilization and implementation of a particular Distributed Database. In addition, the satisfaction of some of them compromises the accomplishment of others.

- **Architectures**, for the implementation of the Distributed Database, and the mechanisms for the client applications to interoperate with the Database. Some architectures make it simplest the interoperability between the information repository (i.e. the Distributed Database) and the client applications, worsening, in contrast, the achievable performance of the system, or the flexibility of the system as a Query Answering Engine.

- **User Requirements won’t stay**. When database applications are studied, their requirements play a principal role for the election of the particular solution for the information repository. These requirements may vary along time, even after the application is completely developed, and deployed in the final environment. The utilization of an inadequate Replicated Databases (or, in general, a Distributed Database) as the information repository of a database application can make it harder for the application to be adapted to changes in the requirements. This is because the election of a particular Distributed Database technique has a high dependence on these requests.

  To avoid this, an adequate Replicated Database must take into account the parameterizability of its behavior, and the flexibility of the kind of offered services.

- **Data and Application Security** of the managed information. Some database applications consider security as one of the most important aspects to manage. The security must be managed from two points of view: on the one hand, developed applications must encourage the security by means of some kind of validation mechanism, and privacy protection of the application operations; on the other hand, the underlying information repository (in our case, the Distributed Database) must also provide for such applications an adequate support for the applications to provide these guarantees. In addition, the database system must enable the necessary mechanisms to trust the client connections, also preserving the privacy of the information managed by the client applications.

  Unfortunately, security used to be a characteristic conflicting with performance, because additional processes must be performed for preserving privacy to tasks as attending application requests, as well as achieving the necessary actions included as part of the Distributed Database Protocols (such as communications between the different nodes in the database).

1.2.3 The Target System

As seen above, distributed applications may use replicated databases for taking advantage of a heightened availability enabled by a multitude of replicas of each data object. Some of such distributed applications are used by companies with multiple branches or offices that are distributed in a wide-spread area (national or international). Most of these companies distribute their information per branch, ensuring that the data are mainly updated locally, by the branch where they were created. Examples of such companies could be hypermarkets and banks (although very many of them still use a centralized solution).

Such applications mainly deal with data objects related to the local branch, but sometimes they require additional access to data objects created in other branches. If these accesses need to be done on a remote database, they could be inefficient. So, it seems advisable to replicate all data objects, with independence to the company branch (and, in a sense, also to the local node embodied by the branch's database) where they have been created. Moreover, the majority of accesses made on non-local-branch data objects will be read-only accesses.

If data are replicated, “remote” accesses can be accomplished locally, improving access times as well as availability, in case of node failures in the network. However, each time a data object is modified, updates have to be propagated to a given number of database replicas (not necessarily to all, depending on the replication technique being used), and this also needs some extra time. Hence, considering the system performance, database replication is only convenient if the larger share of accesses do not modify the data objects. But this limitation can be overcome if we also consider the availability benefits of having multiple replicas for each object. After all, in a WAN, node failures or network partitioning are not uncommon.

In summary, we consider applications with a high degree of locality in their access pattern. In addition, the requirements of such applications include availability of the information, due to the high frequency of the failures in the system. Moreover, it is desirable for the system to enable some kind of access to the information in a minority partition to enhance even more the availability.

With respect to the interconnecting network, applications as the depicted above often use wide area intranets to communicate the involved nodes. This makes the network a very restricted resource, and its utilization must be optimized.

Finally, availability of the managed data is important for the applications we have in mind. The nodes in our target system can suffer faults, producing their disconnection from the system. Moreover, once the node is disconnected, its activity can sometimes be restarted after a time, producing the re-inclusion of the node in the system. This re-inclusion, as seen above, must be treated by the system with a particular reconciliation process in order to achieve a consistency of the information distributed along the system. In addition, the target applications will often require a high degree of availability of the services provided by the system, and this makes it necessary for the stabilization processes (both the executed after the node failure, and during the failure recovery), to allow any incoming client request to be attended during their execution.

1.3 Techniques Used in Distributed Databases

It is important, for any Database Manager System to provide a number of guarantees to their requesting clients. These guarantees are mainly related to the traditional ACID properties of a transaction:

- *Atomicity*, making indivisible the effects of a transaction. The phrase ”all or nothing” precisely describes this property.
- *Consistency*, ensuring that any item update in a database is consistent with updates to other items in the same database.
- *Isolation*, [GR93] needed when there are concurrent transactions in the system; i.e. transactions that occur at the same time, working with shared objects. Guaranteeing isolation consists of preventing conflicts between concurrent transactions accesses.
- *Durability*, enforcing the maintenance of the updates of committed transactions. That is, avoiding any lost of these updates. To provide durability, a system must be able to recover updates performed by any committed transaction if either the system or the storage media fails.

Moreover, for a Distributed Database Manager System, some of these properties are usually harder to guarantee, because the management is performed in each one of the nodes conforming the system, and state of the database is not fully known, but it is only known a partial view of the global state. Particular principles and techniques have appeared in the literature to solve these issues, and many formalisms have been also proposed to validate the accomplishment of the problem.

By the other hand, availability can be one of the requirements satisfied by a Distributed Database. To achieve this problem, that does not exist in the same manner in a centralized Database System, additional considerations and techniques must be taking into account.

1.3.1 Concurrency Control

In particular, concurrency control is one of the principal tasks to be specially observed in a Distributed Database. Concurrency control algorithms are the responsible to warrant the properties of Isolation, and Consistency to the Database.

Many techniques for the the Concurrency management have been proposed in the literature [BSJ80, Sch81], and a number of formalisms have been also used to validate the accomplishment of the transactional properties of a system applying certain Concurrency Control Protocol[BSW79].

Moreover, the following is a brief classification of such techniques, attending to the attitude of the Consistency Management with respect to the executed transactions. These techniques are not exclusively used for Distributed Databases, but they appeared for Centralized Databases, and have suffered evolutions in order to be applied as Distributed Solutions.

Locking Techniques

The earlier appeared approaches designed to control the concurrency in Centralized Databases were based on the principle of traditional **locks**.

These techniques, also known as “optimistic techniques” were based on the idea that each database element (considering “element” as a granularity abstraction) is a resource, that must be managed with a conservative approach: when a transaction is about to access to an element in the database, a lock corresponding to this element must be obtained first.

If the lock is owned by a previous transaction, the latest one must be suspended, until the lock is released by the owning transaction. Locks must be released after the completion of the commit phase, or when the owning transaction aborts.

The lock acquirement must be performed with a specific protocol, in order to guarantee atomicity of the locking operations. As the locking protocols used in resource access control, database locks can be acquired for a particular purpose (i.e. for read or write access). As a result, the Concurrency Control can make use of some common properties of these access modes, in order to relax the conditions that make a lock request to cause the suspension of the requesting transaction. For instance, many transactions can obtain the lock for reading the same object, if no other transaction own the lock for writing the object. Moreover, two transactions cannot share the ownership of a lock for writing the same object.

In Distributed Databases, the lock acquirement must be performed with a specific adaptation of the locking algorithms. A number of approaches have been proposed, and its behavior used to be based on two different approaches:

- **Voting Techniques**. The requesting transaction sends a number of messages to the rest of nodes in the system asking about the grants for the resource (i.e. the lock). Then, the rest of the nodes reply to this request, including a grant or a denial, depending on the local information. When the requesting node receives enough responses, the request is completed with an additional round of messages, containing the final state of the request (acquired/denied) to the set of nodes to which the original request was sent.

  The number of nodes included in the consensus[SSW79, Tho79, Gif79] round can differ when different approaches are used. For instance, one of the commonly used approaches always sends the request to $\frac{N}{2} + 1$ nodes. This enables the request to be consensuated with the entire system.

  Other approaches differentiate between the requests sent for the acquisition of a "read-lock" or for a "write-lock". This first lock implies only the request to one node of the system (i.e. the requesting node has all the needed information), and the later lock implies the request to be sent to all the nodes in the system. As it can be seen, read operations are benefitted from this approach, because they do not need to perform any communication with the rest of the system.

- **Non-Voting Techniques**. Usually based on the concept of *owner*[Her90], in these techniques the requesting transaction sends any lock request to a single node. This single node depends on the lock requested, and is known as **the manager of the lock**. The manager centralizes all the requests, and reduces the costs in communication of the algorithm, but can worsen the scalability and availability of the system, because all the requests are centralized to a particular node.

  Other approaches to implement the locking algorithm without voting techniques are based on the utilization of communication primitives, with particular guarantees that must be implemented at additional computational cost.

The main disadvantages of the use of *locking techniques* are centered on:

- **Increase of the response time** of the transactions, due to the necessity of blocking when conflicts arise.

  In addition, the locking mechanism, as explained above, is based on particular protocols that often make use -in their distributed versions- of expensive communication primitives that increase the computational consumption of an executing transaction.

- **Appearing of deadlocks** in the system. *Deadlocks* are an ancient problem in locking techniques. A simple deadlock arises when two transactions T_1 and T_2 are suspended waiting for resources (i.e. locks) that are owned to the other transaction.

  For example, suppose two transactions T_1 and T_2 that request two locks l_a and l_b to the database. Suppose that T_1 has obtained grants for the lock l_a, and is waiting for the lock l_b, that is at this moment granted to T_2. Now, T_2 can request the lock l_a to the database, being locked by the Concurrency Manager. As a result, both T_1 and T_2 are locked, and the "key" to unlock them is owned one by the other transaction. This makes it impossible for them to be unlocked without external intervention.

More complicated deadlocks may also occur, but the essence of them can be found in *wait-cycles*. The absence of wait-cycles in the executing transactions ensures that there exists no deadlock in the system.

A number of techniques to avoid deadlocks have been proposed, performing additional checks whenever a transaction must be blocked by the system. If the blocking transaction is suspected to be involved in a deadlock, the blocking process is replaced by the abortion of such transaction, thus avoiding the deadlock. Other techniques to avoid deadlocks consist on the establishment of certain rules for the transactions to request the resources (i.e. locks) of the system, that provides -under certain restrictions- guarantees about the existence of deadlocks.

In addition, it has also proposed deadlock detection as an alternative solution, that makes use of the execution of external processes, and aborting the transactions involved in any deadlock.

As an example, one of the most popular protocols in distributed databases is the "Two Phase Locking" (2PL) algorithm, based on the principle of progress. The transaction execution is divided into two phases: during the first phase, the transaction can only perform locking requests; the second phase is dedicated to release the locks acquired during the first phase. In other words: the second phase the transactions cannot request any locking request anymore.

For applications, the implications of 2PL are that long-running transactions will hold locks for a long time. When designing applications, lock contention should be considered. In order to reduce the probability of deadlock and achieve the best level of concurrency, to follow certain guidelines can be helpful. For example, the lock requests must be done in a certain order (i.e. the objects in the database must be ordered with certain criteria, and the transactions must request first the locks corresponding to the objects appeared before in such order).

To reduce the number of locks in the database, the concept of versions has been also used. These techniques allow transactions to read objects whose locks are owned for writing by other transactions. The first transaction can proceed without waiting for the release of the lock, but the value read by the transaction corresponds to a previous version for the object.

Reconciliation Techniques

As seen above, one of the disadvantages of the use of locking techniques is the increase of the response time of the transactions. This increase is mainly produced by the time the transactions are suspended, waiting for the release of locks owned by other transactions.

To eliminate this overhead, alternative techniques appeared[Sch81] in the literature as "Optimistic Techniques" or *Reconciliation Techniques*, which allow any transaction to proceed with no concurrency control until the transaction requests the commit to the Database Manager. Then, the Concurrency Control is performed, exploring the history of the transaction, and determining whether the transaction commit will produce a violation of some of the ACID properties, or the transaction can safely succeed its commit preserving these properties.

The mentioned process of checking the history of the committing transactions can determine that the transaction history offends the system. In this case, some measures have to be taken with the conflicting transaction. These measures are commonly known as *reconciliation process*, and the literature describes two alternatives:

- *Abort the conflicting transaction*, in order to avoid its changes to be permanent in the system.
- *Merge the changes made by the transaction* with the state of the system. These techniques, based on the principle of *reversible functions*, consider the history of the transaction as a chain of functions applied over the database. All these functions have a reverse function that annuls the effects of the application of such function. Thus, when a conflict is detected at commit time, then the history of the transaction is reversed. To this end, the history of the transaction is increased with the sequential reversion of each operation included in the history of the transaction, in reverse order. Thus, the effects of the transaction are annulled, and the original history is applied again, over a correct state of the database. Finally, the resulting history can be committed.

  These merging techniques are uncommon, and its interest remains nowadays in the field of theoretical research.

The way the checks are performed to determine the feasibility of the completion of a transaction commit must be also treated here. There are some different approaches to perform this check, but all of them consist of determining the serializability of the transaction (understanding the transaction as a history of operations applied in the system).

Moreover, the serializability of a transaction is always based on any mechanism used to associate each operation in the history with a certain order in the entire system history.

In other words, the serializability checks are based on the causality of the operations performed by a transaction. The differences between the different approaches reside in the technique used to determine the causal dependencies between these operations. Some of these alternatives are:

- *Timestamps*. In distributed systems[Gra78, MT85, Mul90, Lis91], these techniques have additional implications, due to the impossibility of consensuate a global time in an asynchronous networked system[RSB90, Lis91].
- *Clock Vectors*. Derivated of the idea of *Vectors of Lamport*[Lam78, Mul88], the causality is established with the association of these vectors to each object in the database.

  The solution, however, in practice is not useful, because it requires huge amounts of information to be stored[BJ87].
- *Version numbers* of the different objects in the database, plus a number of consensus rounds[SSW79, MT85] along the life of the transactions to consensuate[JM87] the latest versions in the global system for each objects.

Optimistic techniques have proven to provide better performance, comparing both Optimistic and Pessimistic approaches, in systems where lower level of conflicts exists. This is caused by the actions the reconciliation process must take in presence of such conflicts.

The reconciliation process, as described in previous paragraphs, typically causes the abortion of conflicting transactions. This is exactly the source of the main disadvantages of the use of *optimistic techniques*.

When a high degree of conflicts occur in a database, optimistic techniques introduce a high number of aborted transactions. This causes two dramatical effects:

- **Degradation of the quality of the offered service**, because the user applications perceive these abortions as a disruption of the system functionality, and are forced to retry the execution of the transaction.
- **Degradation of the system performance**, due to the computational time the aborted transaction have consumed, ending without benefits. This computational consumption must be considered [CL88] as part of the overhead introduced by the Concurrency Control.

Nevertheless, environments with a lower level of conflicts will be better suited to optimistic approaches, because there will appear a low number of aborted transactions, and the computational time will be lower than the needed by locking techniques used in pessimistic approaches.

Other Techniques

As occurs in other research fields, hybrid techniques also appeared trying to solve the Concurrency Control. A variety of proposals have been presented, with the aim of providing Concurrency Protocols getting together the advantages of Pessimistic and Optimistic techniques.

As an example, Speculative Concurrency Control[BB95], designed to be included in Real-Time Database Systems, relies on the use of redundant processes, named in the literature as “shadows”, which perform different computations “speculating” on alternative schedules. For each alternative, Consistency Control is used to detect possible conflicts in the speculation, and only one of the alternatives can be finally used. Speculative Concurrency Control algorithms make use of additional system resources to ensure that serializable executions are discovered and elected as soon as possible, thus the computational cost of the commit process can be reduced.

1.3.2 Replication Management

In a Distributed Database, an auxiliary -although critical- part of the Concurrency Control is the Replication Management. A Distributed Database Manager must ensure that the information accessed from any node of the system follows the property of Consistency. To this end, the Replication Management includes additional algorithms to guarantee the consistency of the accessed information. So, the Replication Management is responsible to provide these guarantees.

	Update Propagation	
U.Location	Eager Primary Copy	Lazy Primary Copy
	Eager Update-Everywhere	Lazy Update-Everywhere

Table 1.1: Replication Models in Distributed Databases

A categorization for database replication protocols[GHOS96] has been commonly adopted using two parameters:

- **When update propagation takes place (eager vs. lazy)**:

  In eager replication schemes, updates are entirely propagated within the boundaries of the transaction execution. This means that the commit completion can only arise when sufficient copies in the system have been updated.

In contrast, Lazy schemes, update a local copy, commit and only some time after the commit, the propagation of the changes takes place.

Eager approaches provide consistency in a straightforward way but it is expensive in terms of network consumption and performance. Lazy replication allows a wide variety of optimizations, but, since copies are allowed to diverge, inconsistencies might occur, and additional abortions can be caused by accesses to inconsistent information.

- **Who can perform updates (primary vs. update-everywhere)**:

  The primary copy approach requires all updates to be centralized first at one copy (the primary or master copy) and then they can be performed at the other copies. This simplifies replica control, but presents the disadvantage of the introduction of a single point of failure, and a potential bottleneck.

  The update-everywhere approach allows any copy to be updated, thereby speeding up access but at the price of making coordination more complex.

Comparing the Primary-copy and Update-everywhere approaches, in [WSP$^+$00], it is shown that update-everywhere protocols provide better behavior with respect to performance, and scalability of the resulting system. This is due to the potential bottleneck that a single node becomes in when primary copy approaches are used.

When comparing Eager versus Lazy propagation, the different approaches depend on the existent conditions in the system in a similar way that occurred with the optimistic and pessimistic approaches for Concurrency Control.

In particular, systems with lower conflict rates will be benefited when lazy protocols are used, because they make a lower use of network communications. In contrast, systems with a higher level of conflicts, a higher number of transactions will be aborted, and the quality of the service offered by the Database System will be degraded, in terms of performance and guarantees about the completion of the transactions.

1.4 Update Propagation in Update-Everywhere Systems

1.4.1 Eager Propagation

From a functional point of view, we will consider two types of protocols depending on whether they use distributed locking or atomic broadcast to order conflicting operations of the executing transactions:

- **Distributed Locking** to acquire the needed grants in the whole system before performing the access to the replica being modified. In these protocols, a 2 Phase Commit protocol[BHG87b] is needed during an agreement coordination phase in order to ensure that all nodes commit the transaction.

The 2PC process, does not need an Atomic Broadcast to be achieved as needed for the locking used in distributed systems. In contrast, the 2PC mechanism used in our ambit corresponds to the use of a Virtual Synchrony Cast mechanism in the distributed systems protocol.

- **Atomic Broadcast** techniques, use the total order guaranteed by ABCAST to provide a hint to the transaction manager on how to order conflicting operations. Thus, when any client submits a request to the system, although the request is initially received by a particular node, this node broadcasts the request to the rest of nodes included in the database system. Moreover, as any request is propagated with an atomic broadcast, there is established a consensuated total sorting of the requests at each node in the Distributed Database. Thus, instead of 2PL, the server coordination is done based on the total order guaranteed by ABCAST and using some techniques[KA98] to obtain the locks in a consistent manner at all sites. Then the operation can be executed and a response will be replied to the client.

The similarities between active replication and eager update everywhere using ABCAST are obvious. The main difference is the interaction between the client and the system: in distributed systems, the client broadcasts the request directly to all servers. In contrast, eager update everywhere techniques based on ABCAST attend a single request from the client applications, and is the system who broadcasts the request to all the replicating nodes.

1.4.2 Lazy Propagation

Lazy update propagation is designed to take advantage from repeated updates initiated in a node, and modifying the same set of objects. If such situation occurs, an eager approach must propagate several changes over the same set of objects. In contrast, lazy approaches can avoid the propagation of some of these changes, reducing the network consumption, and improving the overall performance of the system.

The basic principle utilized by lazy update propagation protocols consists of the possibility for the committing transaction to complete its commit phase before the update propagation is performed to all the rest of nodes in the system. Thus, some time after the transaction commits, the updates can be propagated to the other nodes. However, as in the case of eager update everywhere, there is needed a much more complicated coordination than with a primary copy approach. Since the other nodes might have run conflicting transactions during the time propagation is in progress. Moreover, the different sites might hold not only be stale, but it even be inconsistent.

So, the Consistency Control must decide which updates are the elected to be persistent, and which transactions must be aborted. The commit phase, however, must include same kind of atomic protocol to make it easier for the reconciliation process to determine the existence of conflicts in the execution of a transaction.

Note that laziness, while existing in distributed systems approaches[LLSG92], is not widely used. This is caused because those solutions are mainly developed for fault-tolerant purposes, making an eager approach more attractive. In contrast, lazy approaches are a straightforward solution when performance is the main issue. Response times have to be short when the protocol lack of communication requirements during the execution of transactions.

1.5 Contributions of the Thesis

In this thesis, the GlobData project is presented, and the architecture designed as a middleware for Data access is described. This architecture, named COPLA, provides a solution for database applications to gain access to a High Available Distributed Database, where different Consistency Protocols (including both Concurrency Control, and Replication Management), can be plugged in the system, in order to provide particular benefits to the different requirements of client applications.

In particular, this thesis centers its attention in the particularities of the type of applications described in this introduction. For which lazy update propagation techniques using an update-everywhere approach seem to be more adequate, due to the locality of the accesses, and the costs of the network communications.

Moreover, Lazy Update Propagation techniques are widely discussed here, providing as a starting point an incompatibility result that proves that the use of lazy update protocols defined up to now avoid a distributed database to guarantee serializability of the executed transactions if an optimistic consistency control is applied. In addition, the proof includes the theoretical extensions required for those algorithms to guarantee serializability of the transactions. Moreover, with the inclusion of those extensions, we proof the impracticability of the extended algorithm, due to the dramatical increase of the abortion rate.

The second contribution of this thesis consists of a statistical analysis of the abortion rate introduced by lazy update protocols. This abortion rate is, as seen along this introduction, the main disadvantage derived from the use of lazy update protocols. To understand the behavior of the abortion rate, we present an expression for the probability that an accessed object has to be stale, and thus cause the abortion of a transaction.

Following the discussion, the third contribution of the thesis consists of the application of the expression for the stale-abortion rate in order to propose a new approach of update protocol that provides the advantages of both eager and lazy update protocols, while avoids their disadvantages. The proposed protocol (called Cautious Optimistic Lazy Update Protocol) follows an optimistic approach for the concurrency control. The basic idea followed by the protocol consists of the interception of accesses performed by the transactions, then, the protocol predicts accesses to stale objects. If such prediction arises, then an update of the suspicious object is forced before the access is completed. Thus, the probability for a transaction to abort due to stale accesses is dramatically decreased, providing abortion rates similar to the ones obtained with eager update protocols. In addition, the protocol can be configured in order to vary its behavior from a pure eager approach, to a pure lazy update protocol.

In addition, the performance provided by the proposed replication protocol is very similar to the achieved with lazy update protocols, where communication and synchronization between the different nodes of the system is avoided, and the response time is consequently reduced with respect to eager update protocols and primary copy approaches. The protocol is completed with an auto-adaptative technique for tuning the behavior of the protocol to make it flexible to changes in the system characteristics.

The last contribution of the thesis consists of a new extension of the Cautious Optimistic Lazy Update Protocol, to enable lazy fault-recovery in the basic algorithm. The proposed modification makes it possible for the consistency protocol to re-incorporate recovered nodes in the system while maximizing the availability of the system. This implies that the client applications are allowed to continue performing requests to the system at any time, even during the stabilization times (i.e. during the failure stabilization, and the reconciliation process during the recovery). Thus, the quality of the service offered by the Database system will not be degraded when failures are detected, or during the node recovery.

1.6 Thesis Layout

The rest of this manuscript is structured as follows. Chapter 2 describes the GlobData project, and the COPLA architecture, implemented in the project to provide a middleware for client applications to access to a Distributed Database. The chapter starts with an outlined description of the architecture, and the goals of the project. Then, the different components of COPLA are defined, paying special attention to the Consistency Management.

In chapter 3 two basic protocols implemented in GlobData are described. These protocols, both applying optimistic concurrency control, make use of different approaches for the update propagation. On the one hand, a basic eager propagation protocol called FOB is discussed, and its implementation is detailed. On the other hand, the homologous lazy protocol (known as LOMP) is also described, and both protocols are then compared. The chapter also includes the implemented fault-tolerance protocol included for these two consistency protocols. Finally, the chapter concludes with a comparison of the different guarantees with respect to consistency and serializability that both protocols provide. The chapter also includes a summarization of the existing related work on similar researches.

The next chapter is dedicated to proof the impossibility, for lazy update protocols with optimistic consistency control, to provide strict serializability guarantees. The chapter starts with a number of formalisms that are used to express the traditional model, and the impossibility is then shown. An extension to the traditional model is provided, in order to enable a lazy update protocol to provide serializability guarantees, and the chapter concludes with a formal analysis of the theoretical behavior of any lazy algorithm providing these guarantees. As a result, there is shown that those implementation will be impracticable, due to the dramatical increase of the abortion rate it would introduce in the system.

In chapter 5, the abortion rate produced by lazy update protocols is studied from a statistical point of view. Thus, the chapter provides a statistical expression for the probability for an accessed object to be out of date (thus causing the abortion of the transaction). The expression is experimentally validated in the chapter, showing the accuracy of the prediction, other interesting properties. Finally, there is shown an application of the expression to update the predicted outdated objects, and theoretical boundaries of the improvement are detailed.

Chapter 6 details the application of the obtained expression, describing the implementation of the Cautious Optimistic Lazy Update Protocol, as a modification of the basic lazy update protocol described in chapter 3. Then, it is described the parameterization of the protocol, in order to make it to be an eager update protocol, or to have the behavior of a pure lazy update protocol. Further sections include a detailed presentation of the performance, and the abortion rate achievable with the protocol. Finally, the chapter describes a technique for the automatic tuning of the algorithm, enabling it to be configured with an auto-adaptative approach, in order to include flexibility with respect to changes in the system environment.

In chapter 7, there is presented the modification of the Cautious protocol to include self-recovery ability. This modification implements in a lazy manner the failure recovery of the algorithm, and is presented as sequential modification from the basic Cautious protocol.

Finally, chapter 8 includes the final conclusions of the thesis. To do this, the principal outlines are reviewed, paying special attention to the results obtained from the contributions. Then the chapter concludes with the future research lines derived from this thesis.

Chapter 2
The GlobData Project

When an organization starts an Internet project, a variety of objectives must be met. Many of the current Internet applications (i.e., e-bank applications) manage huge amounts of information. This information is mainly accessed with a strong geographical locality. In addition, these types of application usually strive for a high degree of availability, as often enough, these applications offer services not only to external, but also to internal clients, which must be capable of accessing the information at any time. The locality of the accesses suggests that in many cases the database can be partitioned [Rah93, GHOS96]. In many scenarios, it may be necessary to replicate the information in a set of servers, each one attending its local clients. The different replicas of the database must be then interconnected, a WAN being usually the best fit alternative.

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

When the databases containing the information must be replicated, it becomes necessary to introduce protocols and algorithms to provide a minimal set of guarantees about the consistency of the data [BHG87a, WJ92]. The traditional approaches for replicating databases are centered in the use of fast LAN's [KA98, KB94, JM90, Her90, Her87], where network-intensive protocols are used. In Internet applications, the network is a limited resource, and the problems introduced by the WAN must be appropriately dealt with [LLSG92, FMZ94b, FMZ94a].

Our researching group has experience with fault-tolerant systems and operating systems, including failure detectors[MGGB01], middlewares providing high availability to object-oriented architectures[GMB97a, MGB98, GMB97b, GMB99], and specific solutions for highly-available networking systems[IBBAME02, IBBAME01], as well as some operating systems developments [MB97, IB01].

The GlobData Project[IUF01, DMI$^+$03],marked into the V Framework Program of the Information Society Technologies, strives to provide a solution for these kinds of applications in which efficiency, availability and high volume data handling must be achieved. It does so by defining a specific architecture for a set of replicated databases, together with a programming API and a set of consistency modes for data access.

The aim of GlobData is to provide an architecture to enable distributed access to a database, granting fault-tolerance to the different nodes participating in the Database, with a reasonable overhead in term of performance.

In this sense, every client of the system architecture should be able to gain access to a distributed repository of information, in a similar way that it would be done in a centralized solution.

In addition, the system should provide consistency and isolation guarantees to the concurrent client applications executed in the system, benefitting in terms of performance -in the most of the possible- from particular characteristics of the client applications as locality in the accesses.

Finally, the global behavior of the system should become able to continue in presence of partial failures.

These functionalities can be provided by a number of alternatives. The simplest approach consists of the implementation (or use) of a specific distributed database, executed in each node in the system, and accessed directly by each client application. The main disadvantage of this approach is the poor portability of the solution, and the high cost of the mentioned implementation. In addition, the solution presents a poor scalability, providing a degraded behavior for systems with a high number of nodes.

Another option may be the implementation of a ”service replicator”, making use of a common database in each node in the system, and gaining access to the distinct instances of databases through a replicated service. The distributed database should access to that service to interact with the information repository. This alternative, although it can be useful for simple services, would not be useable for a database, because it makes no management of particularities as transaction conflicts, etc.

The adopted solution consists of the implementation of a *middleware* layer to complete the access to the logic Geographically Distributed Database over a Wide Area Network, implementing the information repository as a common database, executed in each node in the system.

The basic function of this *middleware* consists of providing to the different client applications (typically written in Java) isolation of the physical location of the different components of the distributed database, in addition to the way the replication has been actually implemented, the used concurrency control, or the followed failure recovery protocol.

The COPLA architecture [JAJ$^+$02, IUF01, MIG$^+$02b] provides a framework to integrate a wide variety of protocols capable of meeting the consistency requirements posed by GlobData's goals. Currently, a number of protocols [IMDBA03, RMA$^+$02, MEIBG$^+$01] , with slightly different goals have been implemented, but all of them share a characteristic: They cannot be classified as pessimistic, since transactions are allowed to proceed locally (except for "read" operations, as it will be discussed later), being checked for consistency violations at commit time. When a consistency violation is found, the transaction is rolled back.

Our implementation of the COPLA architecture has proven to be a flexible tool in order to experiment with different approaches for providing fault-tolerance, distributed access to databases, fault-recovery, etc.

We have implemented either eager, and lazy update protocols, and a number of different recovery algorithms. The results obtained by COPLA have helped us in the study of the implementability of those kind of protocols, and exhaustive studies have been performed about its behavior with respect to performance, load balancing, abortion rate, and other properties.

The different options implemented in COPLA have been designed to provide a particular solution to some kind of application. Thus, applications with a high degree of access locality, and performing intensive write operations have been proven to be best suited with optimistic, lazy update algorithms. In contrast, applications with a lower locality prefer eager update algorithms to obtain better performance.

The rest of the chapter is organized as follows: Section 2.1 describes the architecture of COPLA, and a number of common concepts used in COPLA; In section 2.2, 2.3, and 2.4 the different layers of the COPLA architecture are described in detail; Section 2.5 explains the functional component that manages the consistency in the system, and finally, section 2.6 concludes with a brief outline about some principles in consistency protocols.

2.1 Description of the Architecture

The COPLA architecture consists of three layers as depicted in figure 2.1. Further sections will explain with detail these different layers. From bottom to top, the following is a short description of them:

- *Uniform Data Store (UDS)*. This component manages the persistent data of a GlobData system. It interacts directly with a relational DBMS, storing there the persistent objects of the given application and the metadata of the consistency protocol.

  It isolates the upper layers from the actual storage system used. In practice, support for different RDBMSs will be provided in the final release of the UDS (currently, it

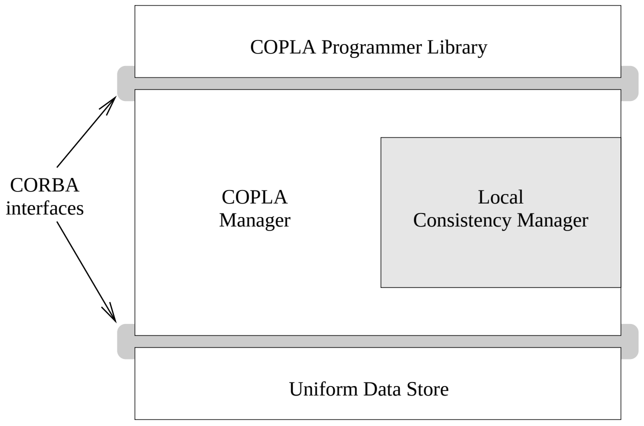


Figure 2.1: COPLA architecture.

only manages PostgresSQL repositories).

The definition of the application databases is made using GODL, a simplified version of the ODMG ODL language [CBB+00].

- *COPLA Manager*. The COPLA manager is the core component of the COPLA architecture. It manages database *sessions* (which may include multiple sequential transactions, working in different consistency modes) and controls the set of database replicas that compose the GlobData system. This manager also provides some caches to improve the efficiency of the database accesses.

  A *local consistency manager* is included in this layer. Multiple consistency protocol objects may be used in this component, but only one is allowed at a time. All consistency protocols share some characteristics. For instance, all of them receive event notification from the COPLA Manager when the user application accesses an object, and when a commit or rollback is requested. In addition, the instance of Consistency Manager in a node also receives events from other instances of Consistency Manager residing in other nodes, in order to locally apply changes made in other nodes, or requesting the local version of an object, or notifying the local Manager a remote session to start a commit (and hence, the Manager has a chance to opposite to this commit). The way all these events are managed depends on the consistency protocol being used. All of the communication among GlobData databases is managed by this component.

- *COPLA Programmer Library*. This library is the layer used in GlobData applications to access system services. It also provides some cache support and multi-threading optimizations that improve the overall system performance.

  The applications need not be installed in the same node where the COPLA manager or the UDS are placed. They only require this library layer on their nodes.

Between each pair of consecutive layers, there is a CORBA interface. So, each layer could be placed in a different node. To enable communication across layers, an object request broker (ORB) is needed. The current system release is implemented in Java, and the Java ORB of the Sun J2SDK is used.

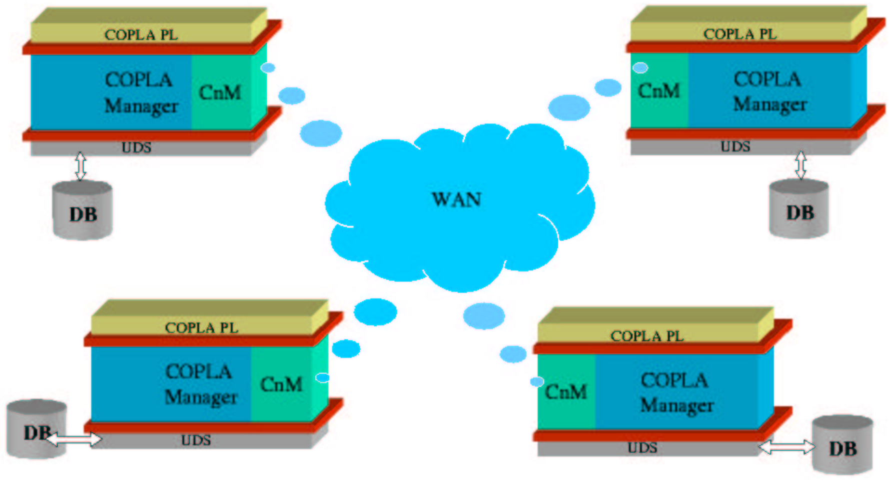


Figure 2.2: Typical COPLA system.

The communication between the COPLA nodes of the system is completely performed through the COPLA Manager of each node. In fact, is a particular component of the COPLA Manager, the Consistency Manager of each node, the responsible of such communications.

To simplify the communication tasks, COPLA also provides for the development of Consistency Managers a ”communication toolkit”, supplying a number of basic primitives that provide particular guarantees.

For example: it is possible for a particular Consistency Management to require atomic broadcast messaging[HT94]. To achieve this, COPLA provides the primitive

”total_broadcast(message, *set_of_nodes*)”.

Of course, it is not obligatory for a Consistency Manager to use these primitives, but they are provided to simplify some parts of such implementations.

Another important element supplied by COPLA is the Membership Monitor. Its main goal consists of the notification to every registered user component of any change in the composition of the COPLA system (i.e. any change in the list of actively participant nodes in the system).

Membership Monitors may not be needed by a particular Consistency Protocol, but some of them (as the protocols described in further chapters) can be simplified with the presence of a Membership Monitor. The description of the Membership Monitor implemented for COPLA can be found in [MGGB01].

Figure 2.3 depicts the internal architecture of the Consistency Manager of a COPLA node.

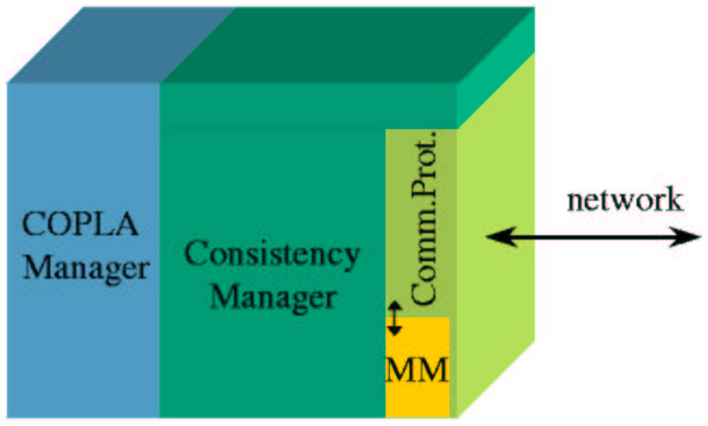


Figure 2.3: Consistency Manager architecture.

As shown in the figure, the main component of the Consistency Manager consists of a protocol managing the consistency control of the node. This protocol may make use of the membership protocol provided by COPLA (or any other monitor), and it can also use the builtin toolkit for the group communication.

As explained above, the consistency protocol may also include a recovery protocol, providing the system fault-tolerance and node recovering.

All the communication between the nodes in a COPLA system is exclusively performed through the consistency protocol. This means that two COPLA Managers (residing in two different nodes) are constrained to use an API within their local Consistency Manager in order to operate with local transactions, start the commit phase, recover their state after a failure, or any other operation.

2.1.1 Consistency Modes

As seen above, we consider as COPLA node a number of components, providing each one a different functionality to the system. In fact, the view provided to a user application is that a COPLA node is a *gate* to the Database, with independence of the concurrency control, replication protocols, or recovery mechanisms used by the system.

As other middleware to gain access to a transactional information repository (e.g. JDBC), COPLA provides particularities with respect to the transactional characteristics offered to the user applications. These characteristics, known in the COPLA architecture as *consistency modes*, provide a variety of guarantees for the sessions (i.e. transactional contexts) manipulated by the user applications.

These *consistency modes* are managed by COPLA allowing the user applications to manipulate some *session* attributes.

A session can be considered as a sequence of “transactions” made in the same database connection. Each of these “transactions” can be made in one of the following consistency modes:

- *Plain consistency*. This mode does not allow any write access on objects. It guarantees that all read accesses made in this mode follow a causal order; i.e., it is prevented that a read access obtains an object that causally succeeds the results of a later read access on the same or a related object.

  On the other hand, this mode imposes no restriction on the currentness of the objects being read. Thus, they may be outdated.

- *Checkout consistency*. This mode is similar to the traditional sequential consistency, although it does not guarantee isolation. Thus, if several sessions have read a given object, one of these sessions is allowed to promote its access mode to “writing”. However, if two of these sessions have promoted their access modes from reading to writing, one of them will be aborted.

- *Transaction consistency*. In this mode, the usual transaction guarantees: atomicity, sequential consistency, isolation and durability, are enforced.

A session always starts in plain mode. If the guarantees provided in this mode are not sufficient for the application, it can promote its consistency mode to checkout or transaction. In these two modes, all accesses are temporarily stored until an explicit call to the *commit()* or *rollback()* operations is made (with the usual meaning of such operations). Once one of these operations have been made, the session returns automatically to plain mode.

Thus, the programmer is able to choose the consistency mode of each session that composes her or his application, and this consistency mode can be varied as needed while a session is running.

As a result of a consistency mode change, the user application can receive an exception from the COPLA Manager. This exception is used to notify the user application the impossibility for this session to promote to the required consistency mode.

For example, this fact can appear when a partition in the system has been detected. In such situation, an isolated node cannot proceed with any modification in the database. In contrast, only sessions that are running in *Plain mode* can proceed during in an isolated node during a system partition.

2.2 The Programmer Library

This library provides to the user applications access to the resources supplied by COPLA .

The Library is composed by the following components:

- A set of library Java classes, common to any user application using COPLA as a database middleware. This can be considered as the *kernel* of the user interface of COPLA .
- A set of object classes, automatically generated by a specific compiler, that provide isolation to a specific application about the way they should access the COPLA infrastructure to obtain the values of the different objects needed by the application. The implementation of one of these classes consists of a proxy of an object managed by the COPLA Manager.

  The definition of these objects is done using an specific language GODL (GlobData Object Definition Language). This language is a subset of the ODMG standard for object definition (ODL).

  In addition, user applications also define interrogative operations (that is, queries) over the defined objects. These queries are also defined in a specific language known as GOQL. This is a subset of OQL, a standard language defined by ODMG.

The Programmer Library is executed as a part of each user application. Thus, it resides in the execution domain of the client.

2.3 Uniform Data Store: UDS

In the lower limits of the architecture of a COPLA node, the Uniform Data Store manages the information repository accesses. This component has a double responsibility:

First, it must manage the information of the persistent object that each user application defined with GODL. In this way, the UDS component makes use of a set of automatically generated classes for these persistent objects, in order to interact with the underlying database with the semantics defined by a particular application. In addition, it implements the interrogation functions, defined using OQL, and including this functionality in an additional set of classes. These *interrogation functions* conform the mechanism for an application to recover objects, in the same way an application using a relational paradigm uses SQL queries.

The second function of UDS consists of the management of the *metadata* required by the consistency manager to complete its tasks. These *metadata* are defined by each particular consistency protocol, and contain a number of information summarized for objects, sessions, or even nodes. Typically, these *metadata* should contain the version of each stored object, information about the executed transactions, etc.

The simplest implemented protocols only need a version number for each object, because they perform the concurrency control with an optimistic approach, based on versioning. Even more simple protocols can be based on pessimistic (locking) approaches, lacking of the necessity of manage metadata.

Although the user application performs the description of its persistent objects with an object-oriented paradigm, COPLA allows (with the help of the UDS component) to establish a correspondence between those definitions and any relational database. This correspondence is established in a transparent way for the user application, because the user applications just have to use the classes generated by the ODL compiler to this end (the *object classes*).

The UDS layer is only accessed by the COPLA Manager, and this access is performed through a CORBA interface. This allows to put in different machines the COPLA Manager and the UDS.

2.4 COPLA Manager

The interaction between the Programmer Library and the Uniform Data Store is located in the kernel of a COPLA node: the COPLA Manager. The main task of this module is to serve the information requested by the user applications acting as COPLA clients. This activity produces a number of responsibilities:

- *Consistency Control* of the data stored in the local node (by UDS).
- Spontaneous *Update* of the local information whenever it is needed or convenient.
- Local *Transactional Control*, managing the commit process to provide the adequate guarantees.
- *Propagation* of the updates performed in the local database to the rest of nodes in the system.

The COPLA Manager uses the UDS as the local information repository, both for the data owned by the user applications, and for the metadata, generated and maintained by the installed consistency protocols.

There are two ways for the COPLA Manager to be accessed. The *user applications* have a CORBA interface to gain access to the COPLA Manager, employing an API to reach the persistent objects in a particular transactional context. This API is not directly used by the user application, but this is the way the proxies (generated by the GODL and GOQL compilers) perform the invocations to the COPLA Manager services.

In addition, *other COPLA Managers* can also communicate with the COPLA Manager of a node from another COPLA domain. This communication is done with *events* initiated by the *Consistency Manager* that notify the receiver instance of COPLA Manager about different situations, or actions to be done; e.g. object updates, session abortions, obtention of locally stored objects, etc.

The different accesses required by a COPLA Manager to the local information are redirected, using a CORBA interface, to the UDS instance of the COPLA node.

2.5 Consistency Manager

The different COPLA Managers are installed one in each COPLA domain (i.e. node). As seen above, they get coordination between the rest through event notifications. But the generation of those events is managed by the *Consistency Manager*.

The Consistency Manager is the responsible of the following tasks:

- *Replication Management*, of the information updated in the local node, propagating those changes to the rest of nodes in the COPLA system.
- *Concurrency Control* of the different transactions initiated in the system, avoiding conflicting transactions to be committed. This control is the responsible for the maintenance of the guarantees expected by the consistency modes used by the user applications.
- *Fault-Recovery Management*, in order to deal with the failures of nodes participating in the system, and the further reconciliation of the whole system if a failed node recovers its state and recovers its participation in the system after a failure.

The *Consistency Manager* is a software piece that, although it does not have a CORBA interface, has been designed to provide to the COPLA Manager a certain API , as generic as needed to allow the COPLA Manager to plug a particular Consistency Manager. In this way, it is possible to implement a variety of Consistency Managers, in order to satisfy the particular necessities of a wide range of user applications.

2.6 Consistency Protocols

As seen in section 2.5, the Consistency Manager is the responsible in each node to provide guarantees about the consistency of the information manipulated by the user applications.

To perform this work, it needs to make use of a particular consistency protocol. This consistency protocol will be notified of every access performed by the user applications to the information managed by COPLA. In addition, the consistency manager notify to the installed consistency protocol about any commit or abort operations requested by the user. In order -for the user applications- to gain access to the database commit operation, the consistency manager must receive positive response from the notifications sent to the consistency protocol.

In COPLA , there can be plugged a variety of consistency protocols, each one satisfying particular requirements corresponding to specific user application necessities.

A number of classifications of consistency protocols have been presented in distributed systems research. These classifications use to be based in a number of parameters. In [WSP$^+$00], a three parameter classification is presented:

- *server architecture*, refers to the way the different request are addressed to the system. This parameter classifies a protocol into: **primary copy**, where every request is received and processed by each node in the system, and the **update everywhere** approach, based on the execution in a single node of the request, and a further propagation of the result to the rest of the system. In this last case, a new parameter (*update propagation*) classifies the protocol into **eager update propagation** (where the propagation of the results obtained in the executing node are completely propagated to the rest of the system during the commit phase), and **lazy update propagation** (that allows the propagation process to be completed beyond the commit termination).

  Update everywhere protocols provide a good scalability due to the load balancing they allow. The use of lazy of eager update protocols produces different behaviors determined by the characteristics of the executed transactions, and other considerations as the locality of the accesses.

- *server interaction*, classifies protocols taking into account the way the different nodes in the system interact between them. In protocols with **constant interaction**, the interaction is done just once, either at the beginning or end of the transaction. In contrast, **linear interaction** consists of the communication, during the life of a transaction, of each operation it does.

  Constant interaction protocols have a better performance due to the lower communication requirements, because they complete all their communication in a unique phase.

- *transaction termination*, classifies a protocol with respect to the way the transactions end. **Voting termination** protocols perform, before a transaction is committed, a number of rounds for the nodes in the system to reach a consensus about the convenience for the current transaction to be committed, or aborted. **Non-Voting termination** protocols are able to provide guarantees to the user applications without the use of any voting phase.

  Although "non-voting termination" approaches require less message rounds, they either need atomic reliable broadcasts (with total order delivery) if the updates are made at commit time, or all nodes need to execute completely all transactions, even those that finally will be aborted (if the broadcasts are made when the transactions start). So, at first sight, a "voting termination" approach seems better.

In addition, the consistency protocol, with independence of the distribution of the information, manages an important property in transactional environments: the *isolation* property.

A transactional system must guarantee the isolation of every transaction initiated in the system. This makes it necessary for the consistency protocol to avoid concurrent incompatible accesses to the same set of objects.

The traditional classification of the consistency protocols with respect to this parameter of *isolation management* refers to the position the system takes in it: A **pessimistic isolation management** is based on some kind of *locking* structure, suspending a transaction when it tries to gain access to a "protected" object. In the other hand, *optimistic isolation management* uses to allow any transaction to proceed until they reaches the commit phase. Then, the isolation property is evaluated, introducing the possibility for the transaction to be aborted.

Optimistic approaches take advantage of read-intensive systems, and environments with a low conflict rate, because the number of aborted transactions will be kept low. In contrast, pessimistic approaches are based on timestamps of the access instants. These protocols have a well behavior in systems where the number of abortions would be high with an optimistic approach. In such scenarios, pessimistic approaches introduces long delays in the transaction service time, because the time the locks are suspending transactions is high.

Chapter 3
Implementing Eager and Lazy Update Protocols

The COPLA architecture has been used, as described above, to be an experimental platform to test and compare a number of consistency protocols serving a number of typical database applications.

In this sense, there have been implemented a number of basic protocols from the first versions of COPLA. These basic protocols, based on the optimistic approach, make use of version numbers of the objects stored in the database to guarantee isolation and consistency. Others protocols are described in [MEIBG$^+$01, MIG$^+$02a, MIG$^+$02b, IMDBA03].

The existing difference between them lies in the mechanism used to propagate the updates made by the transactions. This chapter is addressed to detail two of these protocols, using different approaches for the update propagation, from an eager approach, to a lazy propagation.

The rest of this chapter is organized as follows: Section 3.1 describes the implementation in COPLA of an eager update protocol. In section 3.2, the lazy version of this protocol is described, detailing the main differences and similarities between them, and section 3.3 compares the performance of both implementations. Section 3.4 is addressed to the common implemented mechanism to provide fault tolerance to these protocols. Finally, section 3.5 details the main differences of the guarantees provided by the protocols, and section 3.6 summarizes the existing work related to replication protocols.

3.1 Implementation of an Eager Update Protocol: FOB

The FOB consistency protocol uses "*Full Object Broadcast*" of the session updates, once the session is allowed to commit. This is a basic eager approach for the update propagation, because all the communication performed to achieve this *consistency propagation* is performed inside the commit phase. In addition, as an optimistic approach, it needs to perform additional management to guarantee isolation and consistency, aborting transactions where needed.

To present the protocol, subsection 3.1.1 details the roles assigned by the protocol to the different nodes in the system, and subsection 3.1.2 details the steps followed by the FOB protocol.

3.1.1 Node Roles

Considering a given session that tries to commit, the nodes involved in its execution may have two different roles:

- *Active node*. The node where the COPLA Manager that has directly served the session's execution is placed.
- *Synchronous nodes*. All other nodes that have a COPLA Manager. In these nodes, the session updates will be eventually received, if such updates exist. Note that read-only sessions do not generate any database updates. Hence, these sessions do not have any synchronous node.

Moreover, in a given session, multiple objects may have been accessed. Before committing a session, some checks have to be made to ensure that the accessed objects' states were up-to-date. One of the nodes receives a distinguished role in these checks, and the others will accept its decisions.

Consequently, for each object, there exists its *owner node*. That is the node where the object was created; it is the manager for the *access confirmation requests* sent by the active nodes at commit time. The management of these access confirmation requests is similar to lock management, but at commit time. To this end, the owner node compares two object versions, the one sent in the request (which is the object version accessed by the requesting session), and the latest object version that exists in the database. If they are not equal, the request is denied and the session will be aborted because it has accessed an outdated object version. On the other hand, if they are equal and there is no other granted request in a conflicting mode (a conflict exists if one of the requests comes from a session that has modified the object), a positive reply is sent to that active node. An active node can commit a session if all access confirmation requests that it has sent have been replied positively.

3.1.2 Protocol

As described above, the FOB consistency protocol broadcasts object updates to all synchronous nodes when a session is allowed to commit. Consistency conflicts among sessions are resolved using object versioning. To this end, the protocol uses some metadata tables in the database where the current object versions can be stored. In this metadata, object versions are stored in order to compare, at commit time, the versions of the objects accessed by the transactions, and the current version for these objects in the local database at commit time. An additional locking is needed at commit time in order to ensure the atomicity of these operations.

The protocol processes the following steps:

1. In active nodes, sessions are created and executed without any additional check. They are allowed to proceed until they request their commit operation.

2. When an application tries to commit one of its sessions, such operation arrives at its COPLA manager, and before applying the commit to the associated UDS, the local consistency manager is informed thereof. To this end, the COPLA manager builds two sets containing the identifiers of all objects read and written in such a session. These are the *session read-set* and *session write-set*.

3. Once these sets have been received by the local consistency manager, the latter sends an *access confirmation request* to the owner of each of the objects in the sets. Such messages include the object identifiers, their accessed versions, the access modes (read or write) and the consistency mode used by the session (checkout or transaction, since plain mode does not need a commit operation).

4. The owner node of each object checks if this access confirmation request would conflict in any way with previous access confirmation requests granted to other sessions but not yet released. A conflict arises if the requesting session has written the object and there is another session that has previously obtained a write grant on the same object version.

   Additional conflicts depend on the consistency mode of the sessions involved in the check. If all sessions have used checkout mode, then a conflict only arises if the requesting session has modified the object and other read-access grants have been obtained previously by other sessions. But in checkout mode, a session does not run into conflicts by having read outdated object versions.

   On the other hand, if at least one of the currently committing sessions has used transaction mode, then conflicts arise when either the requesting session has used an outdated object version, or when there are multiple sessions accessing the object and at least one of the sessions has written it.

If the owner finds that a conflict arises, then it answers the access confirmation request with a *deny* reply. Otherwise, it sends a *grant* reply and the session identifier is recorded as “granted” until it explicitly releases this grant in step 5 or 8.

5. When the active local consistency manager receives the replies, and if at least one reply denies the access confirmation requests, then the session is aborted. However, if all of them grant the request, then the session will commit.

   If the session has been aborted, then a release message is sent to the object owners that had replied using a “grant” message.

6. If the session has been allowed to commit in the previous step, then the consistency manager of the active node broadcasts the session updates to all GlobData system nodes; i.e., to all nodes that have a consistency manager. This is a FIFO reliable broadcast[BJ89].

7. Once the update message is received, the active node for that session commits it. The synchronous nodes will also commit the session updates. But before doing so, they have to check that no local session has accessed any of the objects received in that update message. If such local sessions exist, they are aborted.

8. Once the update has been completed, the consistency managers placed in the synchronous nodes check if they are the owners of some of the objects updated. If that is the case, the grants set in step 4 are released immediately. Since no explicit message is needed to do so, this accomplishes the protocol.

3.2 LOMP: Basic Implementation of a Lazy Update Protocol

The basic optimistic Lazy Protocol implemented in GlobData is named ”*Lazy Object Multicast Protocol*” (LOMP), and was presented in [MIG+02b].

The protocol could be summarized as follows: during the execution of a transaction, there are no special actions to be done until the transaction requests its commit phase. Then, the system performs a voting round (similar to the one described for the FOB protocol), resulting in the abortion or granting of the transaction. If the transaction can success, its updates are not fully propagated, but it is done only to a subset of the system.

To detail the LOMP protocol, subsection 3.2.1 presents the different consistency modes provided by LOMP, and its main differences with respect to the guarantees provided by the eager approach. Subsection 3.2.2 details the roles assigned by the protocol to the different nodes in the system, and subsection 3.2.3 details the steps followed by the LOMP protocol.

3.2.1 Provided Consistency Modes

In the COPLA architecture, a session can be considered as a sequence of "transactions" made in the same database connection. Each of these "transactions" can be made in one of the following consistency modes:

- ***Transaction*** * ***consistency***. This mode is a slight variation with respect to the guarantees provided by the ***Transaction consistency***. In the transaction* consistency, atomicity, sequential consistency, isolation and durability are enforced in the maximum level a lazy update protocol can guarantee. In section 3.5 these differences are discussed, and chapter 4 provides a justification of this assertion.

- ***Checkout*** * ***consistency***. This mode is similar to the traditional sequential consistency, although it does not guarantee isolation. Its behavior is similar to the Checkout consistency, in the maximum level a lazy update protocol can guarantee.

In contrast, ***Plain consistency*** cannot be always guaranteed, due to the temporal inconsistency of some parts of the local database. In fact, this consistency mode can only be guaranteed if the node where the session is initiated is considered a *synchronous replica* for every object accessed by the session.

3.2.2 Node Roles

In contrast to the nodes depicted for the FOB protocol, LOMP makes use of the concept of node roles with respect to each object in the system.

Despite this, LOMP considers -as FOB does- the *active node* as the node where the COPLA Manager that has served the session's execution is placed.

Considering a given object manipulated during the activity of a session, the nodes involved in the system may have three different roles:

- *Owner node*. For a particular object, the node where this object was created. During the consensus process performed at commit time of a session, the owner of an object will be asked to allow this session to complete the commit. Thus, it is the manager for the *access confirmation requests* sent by the active nodes at commit time. The management of these access confirmation requests is similar to lock management, but at commit time. These requests are detailed in section 3.2.3.

  We will denote that a node N_k owns an object o_i with the expression $N_k = own(o_i)$.

- *Synchronous nodes*. If one of our goals is fault tolerance, it becomes necessary a set of nodes that provides guarantees about the last version written for a certain object. So, for each session that is committing, the owner of each written object must multicast this update to a set of synchronous nodes, within the commit phase.

  We will denote that a node N_k is a synchronous replica of an object o_i with the expression $N_k \in S(o_i)$.

- *Asynchronous nodes*. For an object, all the other nodes that have a COPLA Manager replicating the database. In these nodes, the session updates will be eventually received.

  We will denote that a node N_k is an asynchronous replica of an object o_i with the expression $N_k \in A(o_i)$.

  Note that: $own(o_i) \in S(o_i), \text{ and } A(o_i) \cap S(o_i) = \emptyset$.

3.2.3 Protocol

As described above, the GlobData-LOMP consistency protocol multicasts object updates to all synchronous nodes when a session is allowed to commit. Consistency conflicts among sessions are resolved with an optimistic approach, using object versions. To this end, the protocol uses some meta-data tables in the database where the current object versions can be stored. The protocol processes the following steps:

1. In active nodes, sessions are created and executed without any additional check. They are allowed to proceed until they request their commit operation.

2. When an application tries to commit one of its sessions, such operation arrives at its COPLA manager, and before applying the commit to the underlying database, the local consistency manager is informed thereof. To this end, the COPLA manager builds two sets containing the identifiers of all objects read and written in such a session. These are the *session read-set* and *session write-set*.

3. Once these sets have been received by the local consistency manager, the latter sends an *access confirmation request* to the owner of each of the objects in the sets. Such messages include the object identifiers, their accessed versions, the access modes (read or write) and the consistency mode used by the session (checkout or transaction). In practice, the number of messages sent can be reduced, by grouping the requests by owner. Thus, each owner will in fact grant the access to a set of objects.

4. The owner node of each object checks if this access confirmation request would conflict in any way with previous access confirmation requests granted to other sessions but not yet released.

A conflict arises if the requesting session has written the object and there is another session that has previously obtained a write grant on the same object version.

Additional conflicts depend on the consistency mode of the sessions involved in the check. If all sessions have used checkout mode, then a conflict only arises if the requesting session has modified the object and other read-access grants have been obtained previously by other sessions. But in checkout mode, a session does not run into conflicts by having read outdated object versions.

On the other hand, if at least one of the currently committing sessions has used transaction mode, then conflicts arise either when the requesting session has used an outdated object version, or when there are multiple sessions accessing the object and at least one of the sessions has written it. We will name the conflicts described above as "with deferrable denial". The reason of this name will be detailed later.

Other conflicts do not depend on the state of the lock granting, but depend on the state of the database, and are produced by the access to outdated objects within the session. This new kind of conflicts arise if the owner receives a request, asking a grant (for read or write access) for an object whose accessed version is lower than the current version of such object. This can occur if the requesting session is running in an asynchronous node of the accessed object, and this object has been accessed in an outdated version. We name these conflicts as "outdated conflicts".

If the owner finds that any kind of conflict arises, then it answers the access confirmation request with a *deny* reply. Otherwise, it sends a *grant* reply and the session identifier is recorded as "granted" until it explicitly releases this grant in step 5 or 8.

5. When the active local consistency manager receives the replies, and if at least one reply denies the access confirmation requests, then the session is aborted. However, if all of the asked nodes grant the request, then the session will commit. If the session has been aborted, then a release message is sent to the object owners that had replied using a "grant" message.

   Step 4 can be revised now. If the owner replies immediately about a conflict "with deferrable denial", double abortions can occur, because its is possible that the conflict does not effectively exists (i.e. the session granted for the conflicting object can be aborted by other conflicts). For example, it is possible that the conflict arise because session S_2 requested a grant for the object o_1 to its owner node, and this owner has previously granted a write access for o_1 to a session S_1. If S_1 accessed to other objects, then a set of requests was sent by S_1 to the respective owners of its accessed objects. Some of these other nodes can deny the request, and S_1 will finally be aborted. In order to avoid double abortions, the denial reply of S_2 can be delayed until the grant of the conflicting objects is released. Then, if the conflict persists, the denial is sent; otherwise, a grant can be replied to the requesting session S_2.

   A negative reply to an access confirmation request consists not only of a denial of the request, but if the denial is consequence of an outdated conflict, the denial can also contain an update of outdated objects. This occurs when the request has been sent from an asynchronous node of an object. In this case, the owner node can also take advantage of the denial reply to update the requesting node.

6. If the session has been allowed to commit in step 5, then the consistency manager of the active node multicasts the session updates to all the synchronous replicas of each updated object. This is a reliable multicast. Note that the protocol does not need to use atomic broadcast here, because it uses a voting algorithm to acquire the “locks”.

7. Once the update message is received, the active node for that session commits it. The synchronous nodes will also commit the session updates. But before doing so, they have to check that no local session has accessed any of the objects received in that update message. If such local sessions exist, they are aborted.

8. Once the update has been completed, the consistency managers placed in the synchronous nodes check if they are the owners of some of the objects updated. If that is the case, the grants set in step 4 are released immediately. Since no explicit message is needed to do so, this accomplishes the protocol. For the rest of owner nodes, placed in asynchronous replicas, an explicit message is sent to release the granted “locks”.

9. An asynchronous process must be run in every owner node, in order to ensure that every asynchronous node of each object eventually receives an update of the object. This process can be executed in background, as a low-priority process, in order to minimize the interference in the system.

In 3.6 it will be detailed the classification presented in [WSP$^+$00] for the consistency protocols. Following this classification, based on three parameters, the protocol presented in this section can be classified as follows:

Update everywhere with lazy propagation, because each session is actually executed in a single node (the active node for this session), and the updates are propagated -in a lazy manner- to every replica in the system.

Optimistic constant interaction between the nodes, because all the communication is concentrated at the end of each session execution.

Voting termination, because the commit process needs the consensus of every node owning any object accessed by the session.

3.3 Comparing the Protocols: (FOB vs. LOMP)

The algorithms described above for both eager and lazy update propagation approaches implemented in GlobData present in their implementation a number of peculiarities that make it hard to make a comparison without benefit one of them or the other one.

These peculiarities mainly consist on the different accesses primitives offered by the underlying Uniform Data Store (UDS) component. There can be used two mechanisms to change the value of a set of objects:

- `UpdateObjects(PackedObject objs[])`. The method is used when the update is performed using the full state of each object. A `PackedObject` is a structure used by UDS to store or transmit the state of an object existing in the database.
- `ApplyUpdate(TransactionReport rpt)`. In contrast, when the UDS is requested for apply the same changes that another transaction did, then the adequate method is `ApplyUpdate`. The method uses a parameter of the type `TransactionReport` to represent these actions to be taken by the UDS component.

In the implementation of COPLA , these two routines are quite different, its application has different implications for the database, and the use of one of them or the another produce different performance behavior for the final applications.

The routine `ApplyUpdate` will produce a faster update of the whole actions taken by a transaction, and thus, is the more adequate for an eager update propagation protocol. The cost of the routine has a weak dependency to the amount of object changes performed in the transaction.

In contrast, `UpdateObjects` perform the update of particular objects in a separate flavor, fitting with the philosophy proposed by lazy update propagation protocols. However, the performance of the routine depends on the amount of updated objects, and its cost is increased when the updated objects have either aggregated objects or relations with other objects.

The heterogeneity of the operations is the reason for the implementations of our FOB and LOM protocols to be difficult to compare without benefiting any approach.

The comparison shown here includes the implementation of FOB using the `ApplyUpdate` routine, while the implementation of LOM uses for the propagation the `UpdateObjects` routine. Our experiments shown that ,when a set of six objects is modified, the cost of the application of the update using the later operation is up to three times the time spent by the update performed using the `ApplyUpdate` routine.

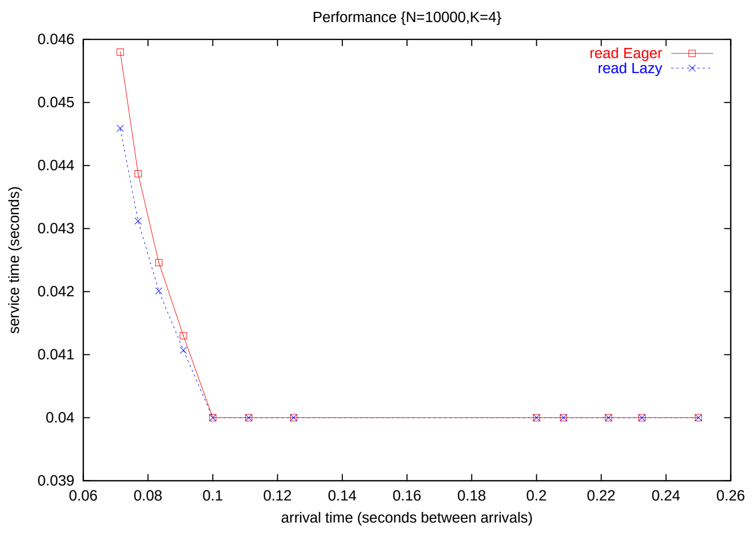


Figure 3.1: Evolution of the performance for read-only transactions with different load rates

Thus, the presented results will benefit the FOB approach, due the low cost of the update propagation.

To perform the experiments, we have executed in a system composed by four nodes a test application that initiates transactions. There are two different kinds of transactions that the test application can start:

- *read-only transactions*, that accesses in read mode to tree different objects in the database.
- *read-write transactions*, that accesses in read mode to six different objects in the database, and then changes the value of three of these objects.

Each node in the system executes the test application, parameterizing the desired load rate. This *load rate* consists of the establishment of the elapsed time between two initiations of transactions. Thus, it can be said that the system will be saturated when the service time exceed this *inter-arrivals time*.

Finally, the access pattern of our test application is mainly local (i.e. the 75% of the accessed objects are own by the node where the transaction is initiated).

For read-transactions (see figure 3.1), there are observed similarities in the service time offered by the system when any of the compared protocols is used.

In contrast, figure 3.2 shows the better service time offered for write-transactions by the lazy protocol in scenarios where a few number of conflicts appear. This is caused because the lazy approach, even when a faster update technique is used (i.e. `ApplyUpdate`),

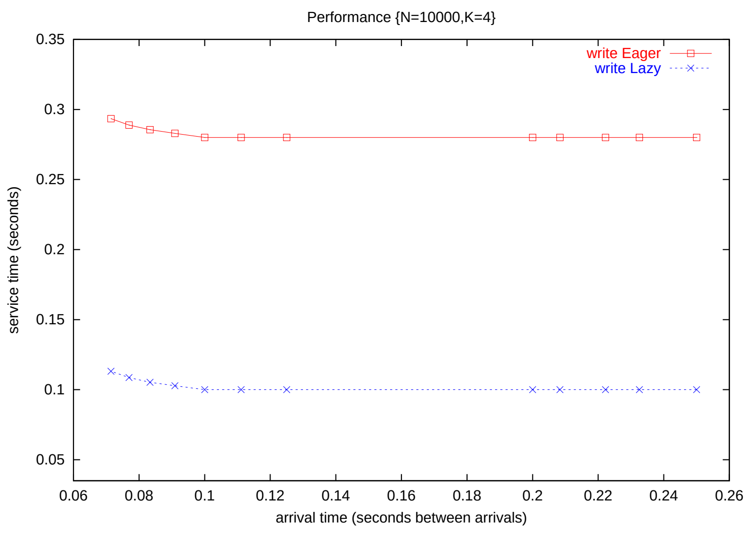


Figure 3.2: Evolution of the performance write transactions with different load rates

will not perform these updates before an abortion occurs (or the asynchronous process is activated). Thus, the service time of write-transactions will not include the time spent in the propagation of the updates to the rest of the system for each executed transaction.

However, the cost of the performance obtained by lazy update protocols is the abortion rate. Figure 3.3 evidences the important abortion rate introduced by the use of LOMP, in contrast to the low amount of transactions aborted in the system when the FOB protocol is used.

3.4 Fault Tolerance

The fault model supported by the FOB and LOMP protocols is ”pause crash” [FHHR85, SS83, Cri91b]. In this kind of failures, when a node is considered to be failed, all correct nodes are eventually informed of such failure, and if the node is recovered, it has access to any information stored before the failure.

Since data objects are replicated in several RDBMSs, the protocol (both FOB or LOMP) is able to tolerate failures of part of the system nodes. To this end, the following protocol details must be considered.

- *Session completion.* If the active node of a session fails before it completes the reliable broadcast of the session updates, the occurrence of such a session is unknown on the rest of nodes. Hence, the session has to be aborted when the active node recovers.

  However, if the update reliable broadcast has been completed, the session has been committed on all system nodes. In this case, the only node that probably has not

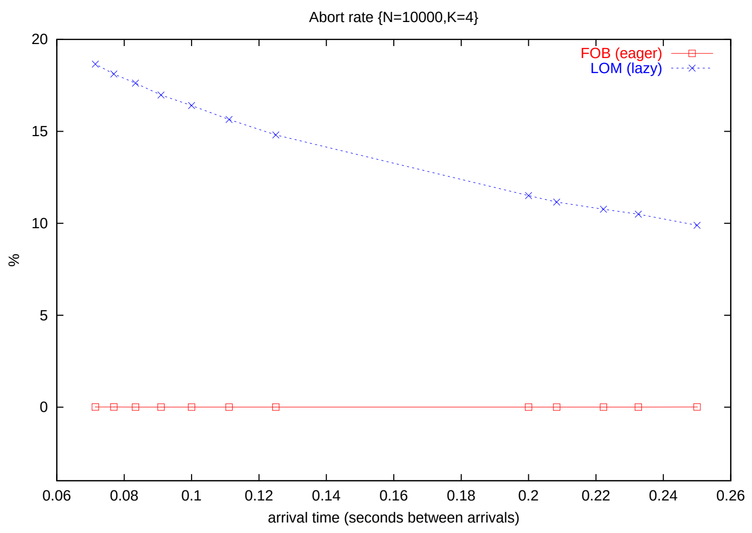


Figure 3.3: Evolution of the abortion rate for different load rates

committed that session is the active one. But this does not matter, since the session updates will be transferred to that node when it recovers, if needed.

Another undesirable effect of this kind of failure is related to the access grants that the session may have obtained. Since the session has not been committed, these grants would remain assigned to the faulty node, preventing other sessions from obtaining access to such objects. The solution to this problem is easy. When the access confirmation requests have been received by an owner node, the identifier of the node that has made such requests is memorized and associated to the requests in some data structure of the owner. When the membership service notifies that a node has failed, this data structure is scanned and all grants assigned to it are automatically released.

- *Ownership role migration*. When a node fails, all objects that were created in there have lost their owner. To replace it, the node with the next identifier in increasing order is chosen as a *temporary owner* for such objects. This temporary owner retains its role until it also fails and is replaced by another one, or until the original owner recovers.

  Moreover, some steps are needed to obtain the lists of access grants that the faulty owner had when it failed. COPLA uses a membership service that notifies all live nodes about any membership change (either join or failure). When such a notification is received, each consistency manager scans its list of received access grants and builds a message containing all the information of all grants given by the faulty owner. This message is then sent to the temporary owner that will replace the faulty one. If a node does not hold any grant of this kind, it must send an empty message.

  The temporary owner collects all such messages and builds a list of its granted confirmation requests. New access confirmation requests are not replied until such a list has been rebuilt.

- *Node recovery*. The recovery steps needed when a node rejoins the GlobData system are thoroughly described in [IF02]. An outline of these steps is provided subsequently.

    1. Once a given node has failed, all remaining live nodes memorize the OIDs of all objects updated in all sessions committed since then, and associate these notes to the faulty node identifier, i.e., they add the OIDs to a hash table or some data structure which is indexed by node identifiers. Hence, all live nodes have registered the same state.
    2. Once a faulty node recovers and tries to join again the GlobData system, all live nodes freeze their respective databases. To this end, if some session has started the commit protocol or has modified at least one of the objects owned by its local node, it is allowed to terminate. Other sessions are blocked until the joining node is integrated in the system.
    3. Once the allowed sessions have terminated, the live nodes send a message to the joining one, communicating that their local databases are prepared for the joining procedure. The joining node waits for all such messages. When it has received all of them, it broadcasts a message to these nodes, indicating that it expects the database updates.
    4. The aforementioned nodes of the GlobData system reply to this message by another one, including the contents of all objects whose OIDs can be found in the hash table described in the first step and that are owned by the replying node. Hence, the database updates are collected from different nodes.

        Note that this recovery protocol can be used in both consistency protocols outlined here. Although the distributed collection of database updates as described in the previous paragraph does not provide any advantage or inconvenience for the FOB consistency protocol, it permits a fast collection when the updates are propagated lazily (LOMP). Instead of using another approach for the FOB case, we prefer to use the same recovery steps as in the lazy consistency protocol.
    5. Later, the joining node applies all such updates using a newly created transaction, which will be committed when the whole set of replies has been received.
    6. Finally, the joining node broadcasts another message, indicating that the joining process has concluded and allowing the blocked sessions to go on.

3.5 Provided Guarantees

During the description of the consistency modes provided by LOMP, it was announced a subtle difference between the guarantees supplied by LOMP and the ones supplied by an eager protocol.

The inability for LOMP to provide stronger guarantees is caused by the lazy essence of the protocol. As any lazy protocol, in commit time, LOMP does not completely update, to every node in the system, the modifications locally performed by a transaction. In contrast, some of these updates are performed before the commit phase is completed.

As explained above, this asynchronism of the update propagation makes it possible for a session to access outdated objects. The consistency manager must detect such situations, and abort these untrusty transactions.

The mechanism to detect outdates can be completed by the use of versioning (or timestamping) techniques by comparing, for each accessed object, the version read by the transactions with the latest version held in a synchronous replica for such object.

These techniques, as it can be easily seen, are based on the assumption that a transaction can only recover the value of an object through straightforward, atomic ”read” operations in base of object identifiers.

But the mentioned difference dwells in the capability for a real transaction to recover objects through queries, based on the values of attributes. This makes it possible for a transaction to alter the recovered set of object identifiers, resulted from the execution of a query in other transaction, without interfere with the basic conception of the ”readset” of the querying transaction.

Let’s see an example:

- Suppose an application where transaction T_1 obtains a list with the employers of an organization having a salary higher than a certain threshold (s_0). The transaction

  – first executes a query, obtaining the adequate object identifiers,

  – and then recovers each object contained in the obtained collection.

  Consider the obtained collection of oid’s as $\{o_1, o_2, o_3\}$. Let’s also suppose that, in a certain moment, the transaction T_1 is executing the recovery of each object.

- During this recovery, and before T_1 tries to commit, another transaction T_2 is started, and increases the salary of a certain employer o_4 above the threshold salary (for instance $s_0 + 100$). Then, T_2 starts its commit, and succeeds. Note that this can occur because the isolation and consistency management are performed using the object as the minimal database item.

- Now, the first transaction continues recovering objects, and concludes its execution with a commit.

- The ”readset” of T_1, as described above, only contains the object identifiers recovered (i.e. $\{o_1, o_2, o_3\}$), but the second transaction, completed before the termination of T_1, should introduce a new object in the set of identifiers recovered by T_1, and thus, T_1 should be aborted.

This should be the same effect that a similar scenario:

- Suppose the same application, where the same transaction T_1 obtains the list of employers having a salary above s_0. The transaction
  - first executes a query, obtaining the adequate object identifiers,
  - and then recovers each object contained in the obtained collection.

  As above, consider the obtained collection of oid’s as $\{o_1, o_2, o_3\}$. Let’s also suppose that, in a certain moment, the transaction T_1 is executing the recovery of each object.
- During this recovery, and before T_1 tries to commit, another transaction T_3 is started, and decreases the salary of a certain employer o_2 below the threshold salary (for instance $s_0 - 100$). Then, T_3 starts its commit, and succeeds.
- Now, the first transaction continues recovering objects, and concludes its execution with a commit.
- The ”readset” of T_1, contains the object identifiers recovered (i.e. $\{o_1, o_2, o_3\}$), and the writeset of T_3 is now $\{o_2\}$. Thus, this transaction, completed before the termination of T_1, now conflicts with the set of identifiers recovered by T_1, and thus, T_1 is aborted.

These examples illustrate the nuance between a strict transaction isolation and the relaxed transaction modes provided by LOMP, or any other Lazy Update Protocol.

In chapter 4, it will be justified the incompatibility of pure lazy update protocols and isolation guarantees.

3.6 Related Work

Many work is currently performed around consistency management, replication and fault-tolerant systems.

The use of a middleware to allow user application to gain access to a replicated database has been already presented in [JPPMKA01] for clustering systems. The approach, uses eager replication to provide a Fault-Tolerant, distributed database cluster providing good results with respect to scalability and performance.

To achieve these three characteristics, it has been proposed the use of communication primitives with particular properties, in order to take advantage of the guarantees these primitives provide, and perform straightforward implementations of the adequate distributed algorithms.

In particular, some contributions[AAES97] proposed the use of broadcast primitives for the implementation of replication and consistency management for distributed systems. In [KA98], several replication protocols are implemented, mainly using atomic broadcast as the basic communication primitive. The idea has been also improved using different levels of isolation to improve performance[KA00].

The use of group communication primitives, however, has been also shown as a solution for the reconfiguration problem[KBB01]. Such proposals discuss reconfiguration issues arising in replicated databases built using group communication primitives, and in particular those that implement Virtual Synchrony.

Moreover, other proposals to achieve concurrency control and recovery have been also shown, using less expensive communication primitives. In [SAS99, JPPMA00], reliable FIFO and totally ordered multicast are used to implement replication and recovery. This kind of multicast allows all the replicas to have the same “view” of the external events, and they are observed in the same order at every system node.

These kinds of techniques often have the particularity of needing specific scheduling to execute the requests in the system nodes. The requirement of such schedulers often consist on determinism guarantees or constraints.

Another technique used to develop replicated data services is to use Consensus[Sch90, Lam96] to complete some of the tasks included in such algorithms.

Distributed consensus protocols are used when a set of processes need to agree upon the outcome of an operation. Notion of majority, quorums, etc, are commonly used by these consensus protocols, and in order to establish these notions, many of them need to know some information about which processes are involved in the execution of the protocol.

Optimistic consistency protocols have been also widely discussed[Alo98, JP03]. They have proven to be a good approach with respect to performance, but at the high cost of increasing the abortion rate of the system, dealing with a degradation of the achieved performance improvement (and even the saturation of the system) for environments where conflicts can often arise for the executing transactions. Some alternatives have been pointed to relax the inconveniences of such protocols, often based on reordering the incoming requests of any system. Other proposals consist on the use of the concept of “pre-commits” for the transactions locally committed, but for which the update propagation has not been already completed.

In summary, the election of the replication model[JPAK01] is a problem by itself, that has been also treated widely in the literature.

Chapter 4
About the Incompatibility Between Laziness and One-Copy Serializability

In section 3.5, an intuitive scenario was presented to justify the impossibility of LOMP to provide strict transaction guarantees.

The current chapter includes a wide justification about the relaxation in the consistency modes provided by an optimistic protocol, when it wants to propagate the local updates in a lazy style.

To do this, the chapter is organized as follows: section 4.1 presents the basic formalisms used in further argumentations; in section 4.2, the basic semantics traditionally used for the specification of an optimistic lazy protocol is described, and examples are also included; then, section 4.3 presents the impossibility for such models to provide strict serializability when queries are not included in the, and an extension of this semantics is provided in order to include the conflicting scenarios. There are also provided a number of clarifying examples; finally, section 4.4 provides a reasoned result about the derived consequences of using lazy update protocols for the extended semantics, and section 4.6 summarizes with some conclusions.

4.1 Used Formalisms

To model a transactional environment, it is needed to formalize a number of concepts:

- ***Database Item***

  In our model, the minimal unit of information stored in the database is an object. This means that each object in the database can be referenced with a one-to-one identifier.

- ***Transaction:*** we will consider a transaction as a number of operations, that have to be executed inside a *transactional context*; i.e.: following a number of restrictions, described by the ACID properties:

  – *Atomicity,* making indivisible the effects of a transaction. The phrase ”all or nothing” precisely describes this property.

  – *Consistency,* ensuring that any item update in a database is consistent with updates to other items in the same database.

  – *Isolation,* [GR93] needed when there are concurrent transactions in the system; i.e. transactions that occur at the same time, working with shared objects. Guaranteeing isolation consists of preventing conflicts between concurrent transactions accesses.

  – *Durability,* enforcing the maintenance of the updates of committed transactions. That is, avoiding any lost of these updates. To provide durability, a system must be able to recover updates performed by any committed transaction if either the system or the storage media fails.

  In this section, we will center the attention on the Isolation property, because this is the property that depends in a strongest way on the synchronism of the update propagation algorithm.

- ***Operation***

  To illustrate the concept of isolation, we need to resort to the concept of *operation*. A transaction can be defined as a sorted sequence of operations over a set of information items (e.g. the database).

  So, we need to provide a comprehensive list of the operations that can be included in a transaction. The following is a simplified list of such operations:

  – read(object-id)→value

  – write(object-id,newvalue)

  So, it can be seen that the read operation obtains the value of an object stored in the database, and the write operation updates the value of an object in the local database.

- ***Isolation*** In order to categorize the provided guarantees of a system, let’s suppose two transactions (T_1 and T_2) being executed concurrently. It is possible to distinguish four isolation levels:

  – ***degree 0***. transaction T_1 does not overwrite data updated by T_2 (”dirty data”).

  – ***degree 1***. Provides the guarantees of *degree 0*, plus T_1 does not commit any write operation until it completes all its write operations (until the end of T_1 arises).

- ***degree 2***. Provides the guarantees of *degree 1*, plus T_1 does not read dirty data from T_2.
- ***degree 3***. Provides the guarantees of *degree 2*, plus T_2 do not dirty (i.e. write) data read by T_1 before the T_1 commits.

These ”isolation levels” were originally described in [GR93] as ”degrees of consistency”.

The isolation concept can be enunciated as the following property:

”Two transactions T_1 and T_2 being executed concurrently (and working with shared objects) can complete their commit phases if and only if there can be found a serialized order for the execution of both transactions.”

Let’s see an example of the maintenance of the isolation property. Suppose three transactions T_1, T_2 and T_3 being executed concurrently. Let’s consider these transactions as the following sequences of operations:

- $T_1 = read(o_1), read(o_2), read(o_3)$
- $T_2 = read(o_2), write(o_2)$
- $T_3 = read(o_3), write(o_3)$

Now, consider a particular timing for the execution of these transactions:

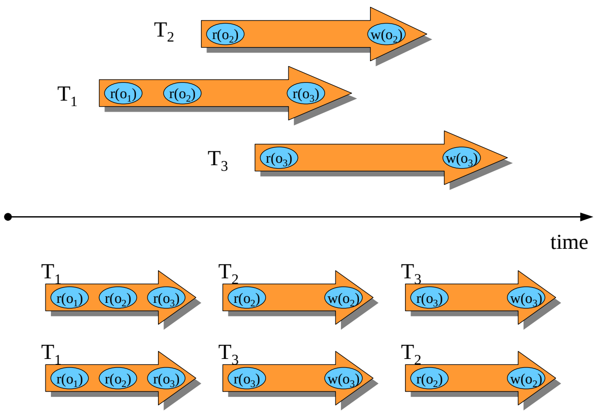


Figure 4.1: Isolation Example.

Figure 4.1 shows how the particular occurrence of the execution of T_1, T_2 and T_3 can be *serialized* in two different ways with the same results for every transaction. Note that transactions T_2 and T_3 accesses different objects. So, both transactions, although they are executed at the same time, can be serialized in any order between them.

In contrast, figure 4.2 shows an example where there is a unique serialization following the isolation property. In this figure, three transactions are started:

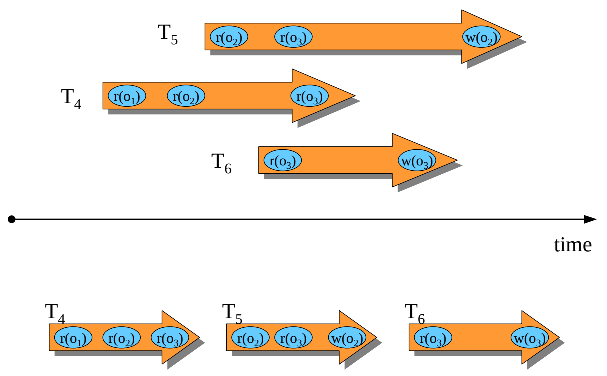


Figure 4.2: Isolation Example 2.

- $T_4 = read(o_1), read(o_2), read(o_3)$
- $T_5 = read(o_2), read(o_3), write(o_2)$
- $T_6 = read(o_3), write(o_3)$

Finally, figure 4.3 shows an example where there cannot be found a serialization of the transactions following the isolation property. The transactions involved in the figure are:

- $T_7 = read(o_2), read(o_3), write(o_2)$
- $T_8 = read(o_2), read(o_3), write(o_3)$

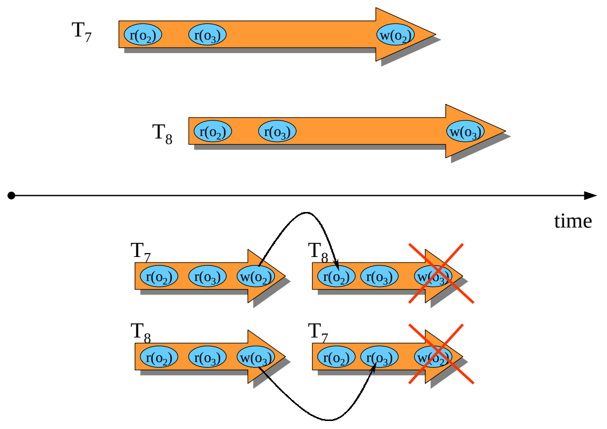


Figure 4.3: Isolation Example 3.

In these examples, we have shown that, when concurrent transactions are executed, it is possible for the system to be unable to guarantee the isolation property. In those situations, the system may abort a number of transactions to preserve the serializability of the committed transactions, and thus, the isolation property in the system.

To calculate the serializability of two transactions, two common approaches have been adopted: pessimistic approaches are based on the use of blocking mechanism (usually locks) to avoid the co-existence of transactions that conflicts with respect to their accessed objects; on the other hand, optimistic approaches allow transactions to proceed without blocking, and needs a process of “reconciliation” usually during the commit phase that only allow to succeed the commit to a transaction that did not access to any conflicting object. To do this check, optimistic approaches use to be based on version numbers (or timestamps) associated to each accessed object, and increasing this version each time the object changes.

Pessimistic approaches avoid a transaction to proceed when it is trying to access to an object accessed in a conflicting way by another transaction. This solves the problem of isolation by blocking temporally the execution of the transaction, but it introduces the possibility of ”deadlocks” i.e. two transactions blocked one on an object locked by another transaction. Deadlocks can also involve more than two transactions. When a deadlock is encountered, the system must abort one or more of the involved transactions.

In contrast, the optimistic approach allows any transaction to proceed, until the transaction enters the commit phase. Then, the timestamps of every object accessed by the transaction are compared to the latest timestamps stored in the database. If the database holds a newer version of any accessed object, the transaction must be aborted. The advantage of the optimistic approach consists of the reduction of the transaction time (they are never blocked). The disadvantage is that it is more possible for a transaction to be aborted at commit time.

The rest of the chapter will be centered in the optimistic approach, making use of any kind of version numbers (or timestamps).

4.2 Basic Semantics of Lazy Update Transactional Environments

This section includes a basic formalization of a database, and its involved elements. These will be used to formalize the property of isolation (serializability) from the classic point of view.

4.2.1 General concepts

Let’s see a number of formalizations for the basic concepts in database semantics.

Database

A database B can be defined as:

$B = \{O(B), V_B, w_B\}$ where:

$O(B) = \{o_i\}, o_i$ objects contained in the database B
$V_B : O(B) \rightarrow N$,function providing ”Current object version in B”
$w_B : O(B) \rightarrow V$,function providing ”Current object value in B”

We are considering V as the set of all the possible values that can get any object in the database.

Transaction

A database transaction T can be defined as:

$T = \{R(T), W(T), V_T, w_T\}$ where:

$R(T) = \{r_i\}, r_i$ readset of (set of objects read by) the transaction T
$W(T) = \{w_i\}, w_i$ writeset of (set of objects written by) the transaction T
$V_T : R(T) \rightarrow N$, function providing ”Object version read by T”
$w_T : W(T) \rightarrow V$, function providing ”Object value written by T”

Every transaction satisfies that: $W(T) \subseteq R(T)$

Note that this formalization assumes that the amount of existing objects in the database is unchanged. This does not mean that the model excludes deletions or insertions of new objects. In contrast, the model aims to consider a database object as unrepeatable. Thus, a deleted object can be considered as ”present, but not usable” for further transactions. The insertion can be considered as the first use of a ”previously existing object”. This will be discussed in section 4.2.4.

Initial Database

A database B_0 is considered initial when:

$B_0 = \{O(B_0), V_{B_0}, w_{B_0}\}$ satisfying:

$\forall o \in O(B) : V_B(o) = 0,$

Inconsistent Database

The notation for the Inconsistent Database will be Θ. Further subsections will describe the situations that transforms a correct database into Θ.

Applying a Transaction over a Database

When a transaction T is applied in a database B, the result can be considered as a new instance of database (B'):

$T[B] = B' = \{O(B'), V_{B'}, w_{B'}\}$ being:

$$O(B') = O(B)$$

$$\wedge$$

$$\begin{aligned} &\forall o \in O(B): \\ &\quad o \in W(T) \rightarrow V_{B'}(o) = V_B(o) + 1 = V_T(o) + 1 \wedge w_{B'}(o) = w_T(o) \\ &\quad \wedge \\ &\quad o \notin W(T) \rightarrow V_{B'}(o) = V_B(o) \wedge w_{B'}(o) = w_B(o) \end{aligned}$$

in other words:

$$O(B') = O(B)$$

$$\wedge$$

$$\begin{aligned} &\forall o \in O(B): \\ &\quad V_{B'}(o) = \begin{cases} V_B(o) & , o \notin W(T) \\ V_B(o) + 1 = V_T(o) + 1 & , o \in W(T) \end{cases} \\ &\quad \wedge \\ &\quad w_{B'}(o) = \begin{cases} w_B(o) & , o \notin W(T) \\ w_T(o) & , o \in W(T) \end{cases} \end{aligned}$$

Applicability of a Transaction to a Database

When a transaction T is applicable to a database B, the result (i.e. $B' = T(B)$) must be a correct database ($B' \neq \Theta$). This principle is formalized as the basis for the correct progress of a database:

$$T(B) = \begin{cases} T[B] & , B \neq \Theta \wedge \forall o \in R(T) : o \in O(B) \wedge V_T(o) = V_B(o) \\ \Theta & , \text{in other case} \end{cases}$$

4.2.2 Commutability of Transactions

To formalize the isolation property, we introduce now the concept of commutability.

Two transactions T_1 and T_2 are commutable when it is satisfied:

$$T_1(T_2(B)) = T_2(T_1(B))$$

Intuitively it can be seen that:

$$T_1(T_2(B)) = T_2(T_1(B)) \longleftrightarrow (R(T_2) \cap W(T_1) = \emptyset) \wedge (R(T_1) \cap W(T_2) = \emptyset)$$

Proof $T_2(T_1(B)) = \{O(B''), V_{B''}, w_{B''}\}$ where:

$$O(B) = O(B') = O(B'')$$

$$\wedge$$

$$\forall o \in O(B) \tag{1.1}$$

$$V_{B''}(o) = V_B(o) + \begin{cases} 0 & , o \notin W(T_1) \\ 1 & , o \in W(T_1) \end{cases} + \begin{cases} 0 & , o \notin W(T_2) \\ 1 & , o \in W(T_2) \end{cases} \tag{1.1.1}$$

$$\wedge$$

$$w_{B''}(o) = \begin{cases} w_B(o) & , o \notin W(T_2) \wedge o \notin W(T_1) \\ w_{T_1}(o) & , o \notin W(T_2) \wedge o \in W(T_1) \\ w_{T_2}(o) & , o \in W(T_2)) \end{cases} \tag{1.1.2}$$

$$\wedge$$

$$\forall o \in R(T_1) \tag{1.2}$$

$$o \in O(B)$$

$$\wedge$$

$$V_{T_1}(o) = V_B(o)$$

$$\wedge$$

(from the definition of T(B)...) $\forall o \in R(T_2)$ (1.3)

$$o \in O(B)$$

$$\wedge$$

$$V_{T_2}(o) = V_{B'}(o) = \begin{cases} V_B(o) & , o \notin W(T_1) \\ V_{T_1}(o) & , o \in W(T_1) \end{cases}$$

in the same way,

$T_1(T_2(B)) = \{O(C''), V_{C''}, w_{C''}\}$ where:

$$O(B) = O(C') = O(C'')$$

$$\wedge$$

$$\forall o \in O(B) \tag{2.1}$$

$$V_{C''}(o) = V_B(o) + \begin{cases} 0 & , o \notin W(T_2) \\ 1 & , o \in W(T_2) \end{cases} + \begin{cases} 0 & , o \notin W(T_1) \\ 1 & , o \in W(T_1) \end{cases} \tag{2.1.1}$$

$$\wedge$$

$$w_{C''}(o) = \begin{cases} w_B(o) & , o \notin W(T_1) \wedge o \notin W(T_2) \\ w_{T_2}(o) & , o \notin W(T_1) \wedge o \in W(T_2) \\ w_{T_1}(o) & , o \in W(T_1)) \end{cases} \quad (2.1.2)$$

$$\wedge$$

$$\begin{aligned}
&\forall o \in R(T_1) \\
&\qquad o \in O(B) \\
&\qquad \wedge \\
&\qquad V_{T_1}(o) = V_{C'}(o) = \begin{cases} V_B(o) & ,o \notin W(T_2) \\ V_{T_2}(o) & ,o \in W(T_2) \end{cases}
\end{aligned} \tag{2.2}$$

$$\wedge$$

$$\begin{aligned}
&\forall o \in R(T_2) \\
&\qquad o \in O(B) \\
&\qquad \wedge \\
&\qquad V_{T_2}(o) = V_B(o)
\end{aligned} \tag{2.3}$$

We can now express the commutability as:

$$T_1(T_2(B)) = T_2(T_1(B)) \longleftrightarrow ((1.1) = (2.1)) \wedge ((1.2) = (2.2)) \wedge ((1.3) = (2.3))$$

The first subexpression,

$$(1.1) = (2.1) \longleftrightarrow \forall o \in O(B) : V_{C''}(o) = V_{B''}(o) \wedge w_{C''}(o) = w_{B''}(o)$$

satisfying $\forall o \in O(B)$ (3.1)

$$\begin{aligned}
&V_{B''}(o) = V_B(o) + \begin{cases} 0 & ,o \notin W(T_1) \\ 1 & ,o \in W(T_1) \end{cases} + \begin{cases} 0 & ,o \notin W(T_2) \\ 1 & ,o \in W(T_2) \end{cases} \text{(from 1.1.1)} \\
&= \\
&V_{C''}(o) = V_B(o) + \begin{cases} 0 & ,o \notin W(T_2) \\ 1 & ,o \in W(T_2) \end{cases} + \begin{cases} 0 & ,o \notin W(T_1) \\ 1 & ,o \in W(T_1) \end{cases} \text{(from 2.1.1)}
\end{aligned}$$

from (1.1.2) and (2.1.2), we can see that: $\forall o \in O(B)$ (3.2)

$$\begin{aligned}
&w_{B''}(o) = \begin{cases} w_B(o) & ,o \notin W(T_2) \wedge o \notin W(T_1) \\ w_{T_1}(o) & ,o \notin W(T_2) \wedge o \in W(T_1) \\ w_{T_2}(o) & ,o \in W(T_2)) \end{cases} \text{(from 1.1.2)} \\
&= \\
&w_{C''}(o) = \begin{cases} w_B(o) & ,o \notin W(T_1) \wedge o \notin W(T_2) \\ w_{T_2}(o) & ,o \notin W(T_1) \wedge o \in W(T_2) \\ w_{T_1}(o) & ,o \in W(T_1)) \end{cases} \text{(from 2.1.2)} \\
&\longleftrightarrow \\
&\forall o \in O(B) : \neg(o \in W(T_1) \wedge o \in W(T_2))
\end{aligned}$$

For the second expression: $(1.2) = (2.2) \longleftrightarrow$

$$\forall o \in R(T_1) \tag{3.3}$$

$$\begin{array}{lcl}
o \in O(B) & & o \in O(B) \\
\wedge & = & \wedge \\
V_{T_1}(o) = V_B(o) & & V_{T_1}(o) = \begin{cases} V_B(o) & ,o \notin W(T_2) \\ V_{T_2}(o) & ,o \in W(T_2) \end{cases}
\end{array}$$

$$\longleftrightarrow$$

$$\forall o \in R(T_1) : o \notin W(T_2)$$

And the third expression: $(1.3) = (2.3) \longleftrightarrow$

$$\forall o \in R(T_2) \tag{3.4}$$

$$\begin{array}{l} o \in O(B) \\ \wedge \\ V_{T_2}(o) = V_B(o) \end{array} \quad = \quad \begin{array}{l} o \in O(B) \\ \wedge \\ V_{T_2}(o) = \begin{cases} V_B(o) & , o \notin W(T_1) \\ V_{T_1}(o) & , o \in W(T_1) \end{cases} \end{array}$$

$$\longleftrightarrow$$

$$\forall o \in R(T_2) : o \notin W(T_1)$$

Thus, from (3.1), (3.2) and (3.3) it follows that:

$$T_1(T_2(B)) = T_2(T_1(B)) \longleftrightarrow \begin{cases} \forall o \in O(B) : \neg(o \in W(T_1) \wedge o \in W(T_2)) \\ \wedge \\ \forall o \in R(T_1) : o \notin W(T_2) \\ \wedge \\ \forall o \in R(T_2) : o \notin W(T_1) \end{cases}$$

and thus,

$$T_1(T_2(B)) = T_2(T_1(B)) \longleftrightarrow \begin{cases} \forall o \in R(T_1) : o \notin W(T_2) \\ \wedge \\ \forall o \in R(T_2) : o \notin W(T_1) \end{cases}$$

being it equivalent to the expression: (4.1)

$$T_1(T_2(B)) = T_2(T_1(B)) \longleftrightarrow \begin{cases} R(T_1) \cap W(T_2) = \emptyset \\ \wedge \\ R(T_2) \cap W(T_1) = \emptyset \end{cases}$$

$\square$

We have proven that the condition for a couple of transactions to be commutable over a database can be summarized as a comparison between its readsets and writesets. The following subsections will introduce the concept of consistency.

4.2.3 Causal Dependency

Once commutability is defined, we can formalize concepts as causal dependency in terms of readsets and writesets.

First Order Causal Dependency

We express the causal dependency of two transactions T_1 and T_2 using its manipulated objects:

"Considering two transactions T_1 and T_2, acting over a database B, the transaction T_2 causally depends at first order on T_1 (and we express this as $T_1 \xrightarrow{1}_B T_2$) when the writeset of T_1 is included in the readset of T_2".

$$(T_1 \xrightarrow{1}_B T_2) \longleftrightarrow W(T_1) \subseteq R(T_2)$$

An Interesting Property of Causal Dependency

From (4.1) it can be seen that:

Considering a correct database ($B \neq \Theta$), and two transactions T_1 and T_2 applicable to B, these two transactions are not commutable over B if and only of there exists a causal dependency between them.

$$T_1(T_2(B)) = T_2(T_1(B)) \longleftrightarrow ((T_1 \overset{1}{\not\to}_B T_2) \wedge (T_2 \overset{1}{\not\to}_B T_1))$$

That is:

$$T_1(T_2(B)) \neq T_2(T_1(B)) \longleftrightarrow ((T_1 \xrightarrow{1}_B T_2) \vee (T_2 \xrightarrow{1}_B T_1))$$

General Causal Dependency

Consider a given database B, and two transactions T_i and T_j. In general, we will say that T_j causally depends on T_i (and we will express it as $T_i \xrightarrow{*}_B T_j$) if there exists a sequence of causal dependencies (at first order) between T_i and T_j.

$$(T_i \xrightarrow{*}_B T_j) \longleftrightarrow \exists\{T_1, \ldots, T_k\} : T_i \xrightarrow{1}_B T_1 \xrightarrow{1}_{B_1} \ldots \xrightarrow{1}_{B_k} T_k \xrightarrow{1}_{B_k} T_j$$

4.2.4 Isolation

This subsection provides a formalism for the isolation property, using the causal dependency to build a number of expressions.

Causal Possibility

A set of transactions $S = \{T_i\}$ is causally possible in a database B (and we express this as $\{T_i\}(B) \neq \Theta$) if and only if there cannot be found in S two transactions with two (mutual) causal dependencies, and if for any transaction contained in S, it is possible to find in S a sequence of transactions applicable over B (resulting B'), and being it possible to apply T to the resulting database B'.

$$\{T_i\}(B) \neq \Theta \longleftrightarrow \begin{cases} \nexists T_j, T_k \in S : T_j \overset{*}{\rightarrow}_{B_j} T_k \wedge T_k \overset{*}{\rightarrow}_{B_k} T_j \\ \wedge \\ \forall T \in S : \exists T_1, \ldots T_r \in S : T(T_1(\ldots(T_r(B))\ldots) \neq \Theta \end{cases}$$

Subset of Causally Dependent Transactions

Consider a set of transactions $S = \{T_i\}$, and another transaction $T \notin S$. We will name as *subset (of S) of transactions for which T is causally dependent* (and this will denote as $\overset{*}{\rightarrow}(S, T)$), to the subset of S containing all the transactions T_j for which T causally depends on.

$$\overset{*}{\rightarrow}(S = \{T_i\}, T) = S' = \{T_j\} : S' \subseteq S \wedge (\forall T_i \in S' : T_i \overset{*}{\rightarrow} T)$$

Last Transaction for an Object ”o”

Consider a set of transactions $S = \{T_i\}$. We will name as *Last Transaction for a certain object ”o”* to the last transaction in S that modified such object.

$$T_o^{>}(S = \{T_i\}) = T \in S : o \in W(T) \wedge \nexists T_j \in S, T_j \neq T : o \in W(T_j) \wedge T \overset{*}{\rightarrow}_B T_j$$

Sub-Database of Interest

Consider a database B, and S a set of transactions $S = \{T_i\}$ applicable in B. We will name as *Sub-Database (of B) of Interest for S* to the subset of B formed only with objects contained in the readset of some transaction in S.

$$\mathcal{I}(B, S = \{T_i\}) = B' = \{O, V, w\} : \begin{cases} O = \bigcup_{T_i \in S} R(T_i) \\ V(o)_{,o \in O} = V_B(o) \\ w(o)_{,o \in O} = w_B(o) \end{cases}$$

Isolated Application

Consider a database B, and S a set of transactions $S = \{T_i\}$ causally applicable over B. We will name *Isolated application of S over B* to a new database $\mathcal{A}(B,S)$ resulting from the application of S over B with causal correction.

$$\mathcal{A}(B, S = \{T_i\}) = B' = \{O, V, w\} : \begin{cases} O = O(B) \\ V(o)_{,o\in O} = \begin{cases} V_B(o) & \nexists T_i : T_i = T_o^{>}(S) \\ V_{T_i}(o) & \exists T_i : T_i = T_o^{>}(S) \end{cases} \\ w(o)_{,o\in O} = \begin{cases} w_B(o) & \nexists T_i : T_i = T_o^{>}(S) \\ w_{T_i}(o) & \exists T_i : T_i = T_o^{>}(S) \end{cases} \end{cases}$$

Isolation

Consider the database B_0, and $S = \{T_i\}$ a set of transactions applicable in an isolated way over B_0 (and resulting B). Consider a new transaction T applicable over B.

The application of $S \cup \{T\}$ over B_0 is equivalent (with respect to the sub-database of interest for $\overset{*}{\rightarrow} (S,T) \cup \{T\}$) to the application of $\overset{*}{\rightarrow} (S,T) \cup \{T\}$ over B_0.

In other words: if it is applied, over a database B_0, a set of transactions S, and it is applied to the result a new transaction T, it is obtained as a result a new database that, with respect to the causability of T, is equivalent to the application of the subset of $S \cup \{T\}$ causally interdependent.

$$\mathcal{I}(\mathcal{A}(B_0, S \cup \{T\}), \overset{*}{\rightarrow} (S,T) \cup \{T\}) = \mathcal{I}(\mathcal{A}(B_0, \overset{*}{\rightarrow} (S,T) \cup \{T\}), \overset{*}{\rightarrow} (S,T) \cup \{T\})$$

Proof

Consider B a database, and I_1 the subset of the database accessed by $S \cup \{T\}$.

$$I_1 = \{O_1, V_1, w_1\} : \begin{cases} O_1 = \bigcup_{T_i \in S\cup\{T\}} R(T_i) \\ V_1(o) = \begin{cases} V_{B_0} & , \nexists T_i : T_i = T_o^{>}(S \cup \{T\}) \\ V_{T_i} & , \exists T_i : T_i = T_o^{>}(S \cup \{T\}) \end{cases} & , \forall o \in O_1 \\ w_1(o) = \begin{cases} w_{B_0} & , \nexists T_i : T_i = T_o^{>}(S \cup \{T\}) \\ w_{T_i} & , \exists T_i : T_i = T_o^{>}(S \cup \{T\}) \end{cases} & , \forall o \in O_1 \end{cases}$$

In the same way, consider I_2 the subset of the database accessed by $\xrightarrow{*}(S,T) \cup \{T\}$.

$$I_2 = \{O_2, V_2, w_2\} : \left\{ \begin{array}{l} O_2 = \bigcup_{T_i \in \xrightarrow{*}(S,T) \cup \{T\}} R(T_i) \\ V_2(o) = \left\{ \begin{array}{ll} V_{B_0} & , \nexists T_i : T_i = T_o^{>}(\xrightarrow{*}(S,T) \cup \{T\}) \\ V_{T_i} & , \exists T_i : T_i = T_o^{>}(\xrightarrow{*}(S,T) \cup \{T\}) \end{array} \right. , \forall o \in O_2 \\ w_2(o) = \left\{ \begin{array}{ll} w_{B_0} & , \nexists T_i : T_i = T_o^{>}(\xrightarrow{*}(S,T) \cup \{T\}) \\ w_{T_i} & , \exists T_i : T_i = T_o^{>}(\xrightarrow{*}(S,T) \cup \{T\}) \end{array} \right. , \forall o \in O_2 \end{array} \right.$$

Obtaining the expression:

$$I_1 = I_2 \longleftrightarrow \forall o \in R(T_i), T_i \in \xrightarrow{*}(S,T) \cup \{T\} : T_o^{>}(S \cup \{T\}) = T_o^{>}(\xrightarrow{*}(S,T) \cup \{T\})$$

Let's see in which conditions the expression is satisfied $\forall o \in R(T_i), T_i \in \xrightarrow{*}(S,T) \cup \{T\}$:

$$\begin{array}{lll} T_o^{>}(S \cup \{T\}) = & T_q \in S \cup \{T\} : o \in W(T_q) \wedge & (a.1) \\ & \nexists T_s \neq T_q, T_s \in S \cup \{T\} : o \in W(T_s) \wedge T_q \xrightarrow{*}_B T_s & \\ T_o^{>}(\xrightarrow{*}(S,T) \cup \{T\}) = & T_k \in \xrightarrow{*}(S,T) \cup \{T\} : o \in W(T_k) \wedge & (a.2) \\ & \nexists T_s \neq T_k, T_s \in \xrightarrow{*}(S,T) \cup \{T\} : o \in W(T_s) \wedge T_k \xrightarrow{*}_B T_s & \end{array}$$

In addition:

$$(\xrightarrow{*}(S,T) \cup \{T\}) \subseteq (S \cup \{T\})$$

From (a.1) and (a.2) we can obtain:

$$\left. \begin{array}{l} T_q = T_o^{>}(S \cup \{T\}) \longrightarrow o \in W(T_q) \longrightarrow o \in R(T_q) \\ T_k = T_o^{>}(\xrightarrow{*}(S,T) \cup \{T\}) \longrightarrow o \in W(T_k) \end{array} \right\} \longrightarrow$$

$$\{T_k \rightarrow T_q\} \longrightarrow$$

$$\left\{ \begin{array}{l} T_q \in \xrightarrow{*}(S,T) \\ \vee \\ T_q = T \end{array} \right\} \longrightarrow T_q \in \xrightarrow{*}(S,T) \cup \{T\}$$

Let's suppose that $T_k \neq T_q$: then, it will be satisfied (considering that T_k is the last applicable transaction of $\xrightarrow{*}(S,T) \cup \{T\}$, and T_q is the last applicable transaction of $S \cup \{T\}$) that:

$$T_q \notin \xrightarrow{*}(S,T) \cup \{T\}$$

and thus:

$$T_q = T_k$$

that conflicts with the hypothesis. Finally,

$$\mathcal{I}(\mathcal{A}(B_0, S \cup \{T\}), \xrightarrow{*}(S,T) \cup \{T\}) = \mathcal{I}(\mathcal{A}(B_0, \xrightarrow{*}(S,T) \cup \{T\}), \xrightarrow{*}(S,T) \cup \{T\})$$

□

This proves the condition for a database to progress from a consistent instance into another consistent instance.

Completing the Model with Insertions and Deletions

Up to this point, we have worked with transactions, but there are not specified the operations contained in a transaction.

We will now consider all the operations that are possible in this model, and their effects over the transaction:

- **Read operation**, increases the readset of a transaction.
- **Write operation**, increases the writeset of a transaction. It must be preceded of a read operation on the same written objects.
- **Deletion operation**, is a particular case of write operation. This is the **last** operation over the involved object.
- **Insertion operation**, is the second particularity of write operation. This is the **first** operation over the involved object. This assumes that a initial database contains every object accessed in the existence of the system.

The included definition for *Insertion* and *Deletion* is a simplification of the complete model, but does not introduce differences from a model containing specific behaviors for these operations.

We can consider the insertion not only as the increase of the readset and writeset of the transaction, but also as the increase of the object set of the database. The results exposed in this section keep their correctness, because no previous transaction was able to access the new object.

In the same way, a deletion can be considered to decrease the object set of the database (in addition to the increase of the readset and writeset of the transaction). The results exposed in this section also keep their correctness, because no further transaction will access to the deleted object.

4.2.5 Laziness

As it can be seen, the classic model includes the ”read” operation as the unique way to access to the value of an object. In this sense, we can now express the concept of *laziness* in the next way:

When a transaction terminates all its operations, it initiates the commit phase. During this phase, the system must determine if the transaction can succeed the commit, or if it must be aborted to preserve the isolation property.

If the system determines that the transaction can succeed the commit, for each modified object o_i, a **lazy** protocol propagates the update of such object in the following way:

- *during the commit phase*, the change made by the transaction over o_i is propagated to the synchronous set of nodes for such object. In a strictly lazy protocol, this set only includes the active node.

- *beyond the commit phase*, the changes are eventually propagated to the rest of nodes (i.e. the asynchronous set of replicas for o_i).

In order to determine the necessity for a committing transaction to be aborted, this mechanism makes it necessary for the protocol to check, for each object accessed by the transaction, the version of such objects contained in the readset. Thus, the causality must be kept, and the properties of consistency and isolation can be also guaranteed. In such situations, a common lazy protocol should update the conflicting objects in the local database, in order to reduce the number of further abortions.

In addition, the checks performed by the protocol only consider, for each object o_i contained in the readset, one of the synchronous set of nodes of o_i, due to the guarantees provided by the ”synchronism” of the update phase.

Note that the checks performed by the protocol make use of the results presented above for the definition of isolation and consistency, and must be done at commit time for each transaction.

4.2.6 Formalization of Laziness

Due to the possibility for each node to hold different versions for the same object in a certain moment, we need to extend the basic definition of Database, in order to characterize this.

Database Extension

So, keeping the properties shown above, a database can be reformulated as:

$$B = \{O^{\{1..K\}}, V, w\}$$

where $O^{\{i\}}$ contains the object instances maintained in the node N_k. With this approach, the object o_i in an eager model is separated now in $o_i^1 \ldots o_i^K$, being K the number of nodes in the system.

Meta-Transaction

Now, we must extend the concept of transaction to fit our proposal:

$$\Gamma[B] = B' = \{O^{\{1..K\}}, V', w'\}$$

being Γ a meta-transaction. As defined in terms of transaction, Γ will keep the properties defined above for the basic transactions.

The possible meta-transactions in our model are described in further subsection.

Node Projection

To formalize the meta-transactions, we will use the concept of *Node Projection*:

$$\begin{aligned} R(T)|_k = &\quad \{o_i^{\{k\}} : o_i \in R(T)\} \\ W(T)|_k = &\quad \{o_i^{\{k\}} : o_i \in W(T)\} \\ V(T)|_k = &\quad V(o_i^{\{k\}}) \\ w(T)|_k = &\quad w(o_i^{\{k\}}) \end{aligned}$$

In summary, the projection of an element (a readset, writeset, version application, or value application) consists of a mapping from objects (without node characterizations) into instances of those objects, characterized in a particular node.

Γ as "Transaction Execution"

A meta-transaction Γ can symbolize the execution of a transaction T, initiated in the node N_k (consider $T = \{R(T), W(T), V(T), w(T)\}$).

In this case, the semantics of the meta-transaction is described as:

$$\Gamma = T^{\{k\}} = \{R(T)|_k, W(T)|_k, V(T)|_k, w(T)|_k\}$$

and

$$T^{\{k\}}[B] = B' = \{O^{\{1..K\}}, V', w'\}$$

where, applying the definition of transaction:

$$\forall o_i^{\{k\}} \in W(T)|_k \Rightarrow \quad \begin{cases} V'(o_i^{\{k\}}) = V_T(o_i) + 1 \\ w'(o_i^{\{k\}}) = w_T(o_i) \end{cases}$$
$$\forall o_j \notin W(T)|_k \Rightarrow \quad \begin{cases} V'(o_j) = V(o_j) \\ w'(o_j) = w(o_j) \end{cases}$$

Γ as "Update Propagation"

A meta-transaction Γ can also symbolize the propagation of one of the updates performed by a transaction $T^{\{k\}}$ (initiated in the node N_k), to another node N_q. We consider that, for each object o in the writeset of the transaction $T^{\{k\}}$, it will be performed a propagation to each node in the rest of the system ($N_{j\neq k}$). We denote the propagation to a node N_q of the object o_i updated by transaction $T^{\{k\}}$ (initiated in node N_k), with the expression $P^{\{q\}}(T^{\{k\}}, o_i)$.

So, the semantics of the meta-transaction is described as:

$$\Gamma = P^{\{q\}}(T^{\{k\}}, o) = \{\{o^{\{k\}}\}, \{o^{\{q\}}\}, V_{T^{\{k\}}}, w_{T^{\{k\}}}\}$$

and

$$P^{\{q\}}(T^{\{k\}}, o)[B] = B' = \{O^{\{1..K\}}, V', w'\}$$

where, applying the definition of transaction:

$$o_j = o^{\{q\}} \Rightarrow \quad \begin{cases} V'(o_j) = V_{T^{\{k\}}}(o_j) + 1 \\ w'(o_j) = w_{T^{\{k\}}}(o_j) \end{cases}$$
$$\forall o_j \neq o^{\{q\}} \Rightarrow \quad \begin{cases} V'(o_j) = V(o_j) \\ w'(o_j) = w(o_j) \end{cases}$$

Note that the readset of $P^{\{q\}}(T^{\{k\}}, o_i)$ is established as $\{o^{\{k\}}\}$ in order to express the causal dependency $T^{\{k\}} \xrightarrow{*} P^{\{q\}}(T^{\{k\}}, o_i)$. This will be used in further sections.

Example

To illustrate the behavior of an eager system, using this model, suppose a system with K nodes, where a transaction T is initiated in N_k, and updates two objects (o_1 and o_2).

Now -as the propagation is performed in an eager way, and transaction completion is atomic- no other transaction can be initiated in the system during the propagation.

The following is a stack representation of the different operations applied one over the previous:

$$\left[\begin{array}{c} T^{\{k\}}[B] \\ P^{\{1\}}(T^{\{k\}}, o_2) \\ P^{\{1\}}(T^{\{k\}}, o_1) \\ P^{\{2\}}(T^{\{k\}}, o_2) \\ P^{\{2\}}(T^{\{k\}}, o_1) \\ \ldots \\ P^{\{k-1\}}(T^{\{k\}}, o_2) \\ P^{\{k-1\}}(T^{\{k\}}, o_1) \\ T_B^{\{q\}} \end{array}\right] \begin{array}{l} \text{initiation in } \mathrm{N_k} \text{ of T over a Database B} \\ \text{propagation to } \mathrm{N_1} \text{ of } \mathrm{o_2} \\ \text{propagation to } \mathrm{N_1} \text{ of } \mathrm{o_1} \\ \text{propagation to } \mathrm{N_2} \text{ of } \mathrm{o_2} \\ \text{propagation to } \mathrm{N_2} \text{ of } \mathrm{o_1} \\ \ldots \\ \text{propagation to } \mathrm{N_{k-1}} \text{ of } \mathrm{o_2} \\ \text{propagation to } \mathrm{N_{k-1}} \text{ of } \mathrm{o_1} \\ \text{initiation in } \mathrm{N_q} \text{ of another transaction } \mathrm{T_B} \end{array}$$

Age of a Database (Time of an Operation)

To formalize laziness, we will use the concept of age of a database (or time of an operation). This concept can be expressed as the number of operations, applied subsequently (one over the result of another) from an original database B_0, to the current database B.

Formalized with a recursion:

$$t(B) = \begin{cases} 0 & , B = B_0 \\ 1 + t(B') & , B = \Gamma[B'] \end{cases}$$

Propagation Time

Suppose a transaction $T^{\{q\}}$, initiated in N_q, and containing in its writeset o_i.

With the principle of time of an operation, we formalize the *propagation time to the node* N_k *of the object* o_i *that was updated by* $T^{\{q\}}$ *at* N_q as the time of the operation Γ that symbolizes the propagation $P^{\{k\}}(T^{\{q\}}, o_i)$:

$$pt^{\{k\}}(\Gamma[B], T^{\{q\}}, o_i) = \begin{cases} t(T^{\{q\}}) & \text{if } \mathrm{q = k} \\ t(B) + 1 & \text{if } \Gamma = \mathrm{P^{\{k\}}(T^{\{q\}}, o_i)} \\ pt^{\{k\}}(B, T^{\{q\}}, o_i) & \text{other case} \end{cases}$$
$$pt^{\{k\}}(B_0) = \infty$$

Intuitively, when a transaction is initiated in N_k, the propagation time of the transaction for this node is 0. For the rest of the nodes, if there exists a propagation operation P to the node N_q, the propagation time to the node N_q of the transaction corresponds to the time of this operation P. Finally, if there not exists a propagation for the transaction to a node, the propagation time for this node is ∞.

This expression can be extended for every object contained in the writeset of a transaction. Thus, we obtain an expression to determine the time for a transaction $T^{\{q\}}$ to be entirely propagated to the node N_k:

$$pt^{\{k\}}(B, T^{\{q\}}) = MAX_{o_i \in W(T^{\{q\}})} \left(pt^{\{k\}}(B, T^{\{q\}}, o_i) \right)$$

We extend again this expression, providing a formalization for the *time of the complete propagation* for a transaction $T^{\{q\}}$ to every node in the system:

$$pt(B, T^{\{q\}}) = MAX_{n \in \{1..K\}} \left(pt^{\{n\}}(B, T^{\{q\}}) \right)$$

Laziness

In these terms, we can now formalize the concept of *laziness* of a system as the elapsed *time* between the application of a transaction $T^{\{q\}}$ in its initiating node (N_q), and the *complete propagation* of such transaction:

$$\Psi(B, T) = pt(B, T) - t(B, T)$$

A system must be considered as lazily updated when this difference exceeds the minimum value $\Psi_E(T)$:

$$\Psi_E(T) = (K - 1) \times |W(T)|$$

Note that Ψ_E is the minimum elapsed time in an eager system, because in such systems, when a transaction T_A is applied, it cannot be applied any transaction T_B before all the propagations are completed for each object in the writeset of T_A (i.e. $|W(T)|$ objects), and for each node different to the initiating one (i.e. $K - 1$ nodes).

Example of Laziness

To illustrate the behavior of a lazy system, using this model, suppose a system with $K = 3$ nodes, where a transaction T is initiated in N_k, and updates two objects (o_1 and o_2).

Before the updates made by T are completely propagated, transaction T_B can be applied (with some conditions). This can be represented as a stack diagram:

$$\begin{array}{r} t(T) \\ t(T)+1 \\ pt^{\{1\}}(T) = t(T)+2 \\ t(T)+3 \\ t(T)+4 \\ pt(T) = pt^{\{2\}}(T) = t(T)+5 \end{array} \begin{bmatrix} T^{\{3\}}[B] \\ P^{\{1\}}(T^{\{k\}}, o_2) \\ P^{\{1\}}(T^{\{k\}}, o_1) \\ P^{\{2\}}(T^{\{k\}}, o_2) \\ T_B^{\{2\}} \\ P^{\{2\}}(T^{\{k\}}, o_1) \end{bmatrix} \begin{array}{l} \text{initiation in } \mathrm{N_3} \text{ of T over B} \\ \text{propagation to } \mathrm{N_1} \text{ of } \mathrm{o_2} \\ \text{propagation to } \mathrm{N_1} \text{ of } \mathrm{o_1} \\ \text{propagation to } \mathrm{N_2} \text{ of } \mathrm{o_2} \\ \text{initiation in } \mathrm{N_2} \text{ of } \mathrm{T_B} \\ \text{propagation to } \mathrm{N_2} \text{ of } \mathrm{o_1} \end{array}$$

In the example, it can be seen that $pt(T) > t(T) + 4$ (note that $4 = (K - 1) \times |W(T)|$). This is caused by the intrusion of the transaction T_B during the propagation process. This situation models a lazy update propagation system.

Checking Serializability in Lazy Systems

In section 4.2.4, a condition to progress from a consistent database into another was proved. Now, we can apply these result to our lazy model.

Thus, consider a system with two nodes (i.e. $K = 2$), and two transactions T_A and T_B with $T_A \xrightarrow{1}_B T_B$ (for example, consider that $R(T_B) = W(T_A) = \{o_0, o_1\}$).

Now, suppose that the propagation of T_A has not completed when T_B is applied. The sequence of operations can be represented with a stack diagram:

$$\left[\begin{array}{c} T_A^{\{0\}}[B] \\ P^{\{1\}}(T_A^{\{0\}}, o_2) \\ T_B^{\{1\}} \\ P^{\{1\}}(T_A^{\{0\}}, o_1) \end{array}\right] \begin{array}{l} \text{initiation in } \mathrm{N}_0 \text{ of } \mathrm{T}_\mathrm{A} \text{ over a Database B} \\ \text{propagation to } \mathrm{N}_1 \text{ of } \mathrm{o}_1 \\ \text{initiation in } \mathrm{N}_1 \text{ of another transaction } \mathrm{T}_\mathrm{B} \\ \text{propagation to } \mathrm{N}_1 \text{ of } \mathrm{o}_0 \end{array}$$

Now, reconsider the previous stack, expressing the readsets and writesets of each operation. This will help us to depict the causal dependencies for the operations:

$$\left[\begin{array}{c} T_A^{\{0\}}[B] \\ \swarrow \\ P^{\{1\}}(T_A^{\{0\}}, o_2) \\ \swarrow \\ T_B^{\{1\}} \\ \Nwarrow \\ P^{\{1\}}(T_A^{\{0\}}, o_1) \end{array}\right] \begin{array}{l} R = \{o_0^{\{0\}}, o_1^{\{1\}}\} \\ \\ R = \{o_0^{\{0\}}\}, W = \{o_0^{\{1\}}\} \\ \\ R = \{o_0^{\{1\}}, o_1^{\{1\}}\} \\ \\ R = \{o_1^{\{0\}}\}, W = \{o_1^{\{1\}}\} \end{array}$$

Where we find a causal dependency $T_B^{\{1\}} \xrightarrow{1} P^{\{1\}}(T_A^{\{0\}}, o_1)$, that makes the resulting sequence inconsistent, as seen in 4.2.4.

Despite the conflicting operation is $P^{\{1\}}(T_A^{\{0\}}, o_1)$, this operation must be applied in the system, because it is a propagation of a previously applied transaction (T_A). So, the system should avoid this kind of situations to exists. To do this, the transaction T_B should be aborted, in order to avoid it to be considered in the sequence before every object accessed by T_B is updated in N_1.

Thus, the system should check, for each transaction T_B, if its application will make it impossible to apply some update propagation (in our example $P^{\{1\}}(T_A^{\{0\}}, o_1)$), due to a causal dependency.

In other words: it is not possible to apply a transaction T_B over the system if there exists any object from its readset that is not updated in the node where T_B is initiated:

$$\forall o_i \in W(T_B^{\{q\}}) : \forall T_L : o_i \in W(T_L) \rightarrow pt(T_L) < t(T_B)$$

Proof

Let $T_B^{\{q\}}$ be a transaction for what the system is questioning its abortion.

$$T_B^{\{q\}} = (R(T_B)|_q, W(T_B)|_q, V(T_B)|_q, w(T_B)|_q)$$

If there exists a transaction T_L^k (initiated in node N_k) that modified one of the objects accessed by T_B (let o_i be this object), and it has not already propagated the update to the node where T_A was initiated (i.e. N_q), then it will satisfied that:

$$o_i \in R(T_b^{\{q\}}) \wedge o_i \in W(T_L^{\{k\}}) \wedge pt^{\{q\}}(B, T_L^{\{k\}}, o_i) \geq t(T_B^{\{q\}})$$

thus, from the definition of $pt(B, T, o)$, this implies that:

$$\begin{cases} k \neq q \\ \wedge \\ \nexists \Gamma_i : P^{\{q\}}(B, T_L^{\{k\}}, o_i) = \Gamma \wedge t(\Gamma) < t(T_B^{\{q\}}) \text{ with } \mathrm{T_B^{\{q\}}} \xrightarrow{*} \Gamma \end{cases}$$

The justification of the first subexpression is that if $k = q$, then both transactions have been initiated in the same node, and the definition of pt says that it will be satisfied that $pt^{\{q\}}(B, T_L^{\{q\}}, o_i) = t(T_L^{\{k\}} < t(T_B^{\{q\}}$.

The second subexpression indicates is justified with the readset and writeset of $T_B^{\{q\}}$ and Γ respectively: the object o_i is contained in both sets $R(T_B^{\{q\}})$ and $W(\Gamma)$, and so it is easy to see that $R(T_B^{\{q\}}) \cap W(\Gamma) \neq \emptyset$, thus, this satisfies the causal dependency $T_B^{\{q\}} \xrightarrow{*} \Gamma$.

From these results we can obtain the conclusion that eventually, there will be applied a transaction Γ corresponding with the propagation of o_i updated by the transaction $T_A^{\{k\}}$ to the node N_q. This transaction, will conflict with the questioned transaction $T_B^{\{q\}}$, and this will not preserve isolation and consistency. This makes it not applicable the hypothesis of the proof, and demonstrates that:

$$T_B^{\{q\}} \text{ can be applied if}$$

$$\forall o_i \in W(T_B^{\{q\}}) : \forall T_L : o_i \in W(T_L) \rightarrow pt(T_L) < t(T_B)$$

□

4.3 Extending the Basic Semantic with Queries

Up to this point, we have presented the classic model describing concurrent transactions over a database, providing a formalization of consistency and isolation, and the application of such results to lazy update consistency protocols.

Nevertheless, the set of operations included in this classic model is not a realistic enumeration. We can illustrate this with a simple assertion:

*”Any real database application recovers its accessed objects not only through direct operations, but also using **queries**.”*

This is the main imperfection of the classic model when it is applied to databases. Queries cannot be modeled in the classic model as an increase in the readset of the transaction, if the readset is understood as a set of **objects** contained in the database. Section 3.5 presented an intuitive example of the expressive lack remarked here.

In order to include queries in the model, we must consider now the behavior of such operations, and the interaction with the database, and the rest of included operations.

4.3.1 Formalization of Queries

The execution of a query over a database consists of the recovery of a number of objects stored in the database, that satisfy a certain condition included in the query.

Such condition can be expressed as evaluations of expressions involving attributes of the considered objects (that belong to a certain class), including in the recovered list of the objects that satisfy the condition.

Thus, a query operation can be considered as a ”read” operation over a number of attributes of the involved classes, retrieving a number of object identifiers.

Once the query is completed, further ”object-read” operations may be executed to recover the value of every object contained in the report of the query.

This approach includes in the model the ”attributes” of the classes maintained in a database as part of the readset of the transactions. Thus, these attributes can be considered as ”meta-objects”, and be treated as common objects. To maintain this meaning, the ”object-read” and ”object-write” operations must be also reformulated.

4.3.2 Extending Operations with Attributes

Considering attributes as ”meta-objects” of the database, the concept of transaction keeps its meaning and the properties justified along this chapter.

The question consists of reformulating the possible operations to include the attributes in its definitions.

- Let a_{oid} be the attribute ”object identifier”, common to every object in the database.
- A **query**, evaluating a number of attributes $\{a_i, \ldots, a_j\}$ to build the reported list of object identifiers can be defined as an increase of the readset of the transaction:

  $R'(T) = R(T) \cup \{a_i, \ldots, a_k\} \cup \{a_{oid}\}$.
- An **object-read**, recovers an object value (o_i) using its object identifier. Thus, the operation can be considered as an increase of the readset of the transaction:

  $R'(T) = R(T) \cup \{o_i\}$.
- An **object-write**, changes the value of an object o_i. Considering the object as the minimal data unit, the write-object operation modifies every attribute of the object. Thus, the operation can be considered as an increase of the writeset of the transaction:

  $W'(T) = W(T) \cup \{o_i\} \cup \{a_k\}, \forall a_k \in \{$ attributes of $o_i\}$.

  Note that we consider that every *object-write* is preceded by an *object-read* operation.

This reformulation keeps the validity of the results obtained in the chapter, because it maintains the readset/writeset paradigm.

4.4 Impracticability Result

In previous sections, we proved that the traditional basic model[BS80], without the inclusion of queries in the causality analysis, are unable to guarantee one-copy serializability. To solve this, an extension of the basic model was proposed, in order to include the queries in the model, and make it possible for a lazy update protocol to provide serializability guarantees.

In this section, an intuitive justification is presented for the discussion for the validity of lazy update protocols including queries in the consistency checks. Further sections will formalize the discussion.

4.4.1 Intuitive Discussion

We can now reconsider the definition of laziness expressed in section 4.2.5, taking into account the inclusion of attributes in the readsets and writesets of the transactions:

When a transaction terminates all its operations, it initiates the commit phase. During this phase, the system must determine -in the same way that the classic depicted mechanism- if the transaction can succeed the commit, or if it must be aborted to preserve the isolation property.

If the system determines that the transaction can succeed the commit, for each modified object o_i, a **lazy** protocol propagates the update of such object in the following way:

- *during the commit phase*, the change made by the transaction over o_i is propagated to the synchronous set of nodes for such object. For the "meta-objects", such synchronous set will be specified later.
- *beyond the commit phase*, the changes are eventually propagated to the rest of nodes (i.e. the asynchronous set of replicas for o_i).

The convenience for a committing transaction to be aborted is determined in the same way that the basic approach, checking for each object accessed by the transaction the version of such objects contained in the readset; i.e. the system checks for the committing transaction T, if $\exists T_j : T_j \overset{*}{\rightarrow}_B T$.

As depicted in section 3.5, queries makes it necessary to extend the expressions presented in section 4.3 for the readset and writeset of a transaction, in order to include the causality of the accesses performed by the queries to attributes of classes in the database. Thus, in order to keep the causality, and guarantee the properties of consistency and isolation, the protocol must include the "meta-objects" in the checks to determine the convenience for the transaction to be aborted. To this end, the system must determine if

$$\exists T_j \ldots T_k : W(T_j) \subseteq R(T_{j+1}) \wedge \ldots \wedge W(T_{k-1}) \subseteq R(T_k) \wedge W(T_k) \subseteq R(T)$$

and $T_j \ldots T_k$ have not been yet applied in the local database.

Finally, for the aborted transactions, the extended common lazy protocol should update the conflicting objects *and meta-objects* in the local database, in order to reduce the number of further abortions.

It is easy to see -observing the formalizations of queries and updates- that the local update of the *meta-objects* (as attributes) can only be completed by the application in the local database of every update performed in the system involving the attributes accessed by any query executed by the aborted transaction.

In addition, the checks performed by the protocol must now consider every node in the system, due to the capability of each node to perform updates, and thus, changes in these *meta-objects*. This is a worst condition compared to the basic voting algorithm presented in 4.2.5, because there are more involved nodes, and the number of needed updates will be higher.

In summary, during the commit phase it becomes now necessary to update every ”meta-object” involved in the committing transaction. Considering that the queries executed in a common database application use a wide range of attributes, the conclusion is that every update made in a node will be propagated to the rest of the system as soon as there will be executed any query in such nodes.

This situation will cause that the number of aborted transactions will be seriously increased, making the system unusable. To avoid this, the alternative solution consists of the propagation, as soon as possible, of every update made by the committing transaction to every node in the system.

To make the system as usable as possible, this update must be done during the commit phase, in order to minimize the elapsed time of the outdate. This alternative, as it can be seen, is the exact approach used by any **eager** update protocol.

4.5 Formalization of the Result

As seen in this chapter, a Database is considered in our modeled system as the result of a number of operations applied sequentially, from an original Database.

Laziness Degree

To formalize the previous discussion, we need to introduce the concept of *Laziness Degree* of a database (we denote this as $\mathcal{D}(B)$).

The *Laziness Degree* of a Database B is defined considering the transactions applied in B completely (i.e. considering every T_i satisfying that $pt(T_i) \neq \infty$).

For each completed transaction T contained in the Database B, we define the *Laziness Degree* as the ratio:

$$\mathcal{D}(B,T) = \frac{\Psi(B,T)}{\Psi_E(B,T)}$$

If we consider every completed transaction, we extend the previous expression to the complete Database:

$$\mathcal{D}(B) = \frac{\sum_{T_i:pt(T_i)\neq\infty} \mathcal{D}(B,T_i)}{|\{T_i : pt(T_i) \neq \infty\}|}$$

Formalization

Let's see how $\mathcal{D}(B)$ is reduced in the model when the attributes are included as meta-objects in the database to support queries properly.

Suppose a system containing C different classes of objects (let $C_1 \ldots C_C$ be these classes). In a particular execution B, we have executed queries in the transactions applied in B.

It can be seen that in the extended model, where queries are considered in order to guarantee one-copy serializability, the probability for a transaction to succeed its commit phase satisfies:

$$PC(T) = \prod_{q_i \in T} PC_q(q_i) \times \prod_{o_j \in R(T)} PC_o(o_j)$$

Where $1 - PC_q(q_i)$ is the probability for the query contained in T_i to cause the abortion of T_i. In addition, $1 - PC_o(o_j)$ is the probability for o_j (i.e. an object accessed by T_i), to cause the abortion of T_i. Note that the basic model should only consider the second factor for $PC(T)$.

The expression indicates that two conditions have to be satisfied to succeed the commit of a committing transaction T_i:

- The query contained in T_i must not introduce conflicts (i.e. the classes accessed by the query are up-to-date).
- The objects accessed by T_i must be also up-to-date.

Now, we can observe $PC_q(q_i)$, in order to determine an upper bound:

$$PC_q(q_i) = \prod_{c_i \in \{1..cpq\}} PC_c(c_i) = PC_c[c_i]^{cpq}$$

being *cpq* the number, in mean of classes accessed by a query. Now, we can reformulate $PC_c[c_i]$ (i.e. the probability for a class c_i to be up-to-date in a particular node), as:

$$PC_c[c_i] = (PC_{T,c_i})^{K \times wtps \times \delta(c_i)} \tag{4.1}$$

where $wtps$ is the number of write transactions per second committed in each node, and $\delta(c_i)$ is the elapsed time from the last update of the class c_i in the node. Note that $K \times wtps$ is the total number of write-transactions per second committed in the system.

The expression for PC_{T,c_i} symbolizes the probability, for a concurrent write transaction T, to commit without update objects of the class c_i. This expression satisfies that:

$$PC_{T,c_i} = \left(1 - \frac{1}{C}\right)^{cwt}$$

where cwt is the number, in mean, of classes modified per write-transaction. The expression for PC_{T,c_i}, as seen, increases when the number of classes in the database is high. Replacing this in the expression 4.1:

$$PC_c[c_i] = \left(1 - \frac{1}{C}\right)^{K \times wtps \times \delta(c_i) \times cwt}$$

Now, we can see that $\delta(c_i) \geq \frac{C}{wtps}$, because -in the node where the query is executed-, there are also applied transactions updating objects of a certain class. Thus, it will be satisfied that:

$$PC_q[q_i] \leq \left(1 - \frac{1}{C}\right)^{K \times cwt \times cpq \times C}$$

Finally, as $cpq \geq 1$, we can simplify the expression as:

$$PC_q[q_i] \leq \left(1 - \frac{1}{C}\right)^{K \times cwt \times C} \tag{4.2}$$

This can be used as an upper bound for the probability for a query included in a transaction not to cause the abortion of such transaction.

As $PC_q[q_i] \leq 1$, it will decrease the probability for a transaction to succeed its commit phase in absence of queries in the model (i.e. the simple model), this model allows objects to be recovered directly, and there is no necessity for the use of queries. In contrast, the extended model, where queries are also considered, introduces $PC_q[q_i]$ as a factor in the probability for a transaction to succeed the commit phase.

In addition, the expression for $PC_q[q_i]$ decreases exponentially with higher values for K. This implies that the model will not be scalable with respect of the number of nodes in the system.

In chapter 5 we will present the expression for the abortion rate in lazy systems where strict one-copy serializability is not required (i.e. queries are not included in the abortion checks). Figure 4.4 shows a comparison, for a particular system configuration, of the successful commit probability in both systems.

The figure shows the effects of the inclusion of queries in the check for different system configurations, varying the number of objects per node (N in the figure).

In the figure, the notation PC_s is used for the probability of a transaction to succeed its commit phase when queries are considered in a lazy system. We use PC_r to denote the

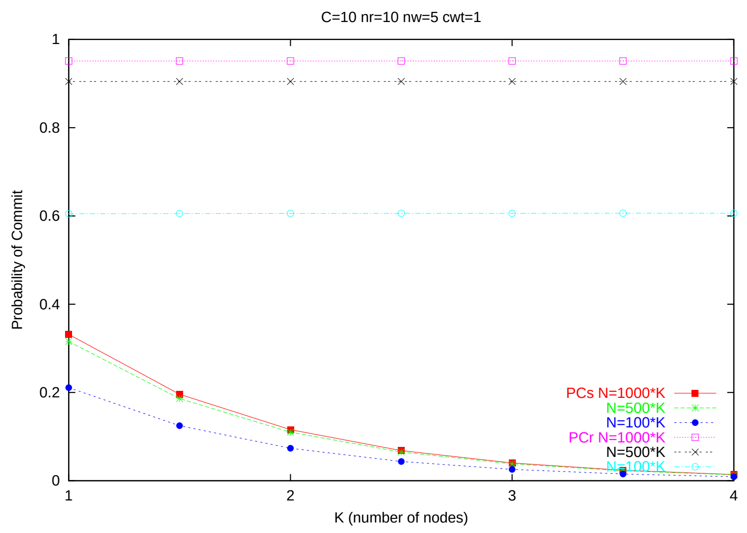


Figure 4.4: Comparison of the Probability of Commit, in a Lazy environment, when including one-copy serializability

probability of a transaction to succeed its commit phase when queries are not considered in the same lazy system. Note that PC_s is an upper bound, while PC_r is calculated with the statistical expression. In addition, the depicted system constitutes a good case for a lazy update protocol, due to the low conflict rate (in terms of objects accessed).

It can be seen that in a system with 4 nodes, the check will cause the abortion of almost the 100% of the initiated transactions when queries are considered. In contrast, if they are not included in the check, the consistency control of the same system should only abort the 5% of the initiated transactions.

4.6 Conclusions

In this chapter, we have presented the problem of serializability from the traditional point of view, formalizing a database as the history formed with the application of transactions, understood as a number of operations over the managed objects. In these definitions, some formalizations have been also included about the serializability of a transaction over a database.

The model has been discussed with examples illustrating the semantic gap produced by the use of queries in real databases, and the necessity for the model to include queries in order to provide a complete condition for a transaction to be serializable, and thus, applicable over a database. Following this discussion, the model has been extended to include queries, and the mentioned condition has been provided as a comparison between readsets and writesets of the involved transactions, including as a part of such sets the attributes read by the queries of the transactions.

Once the model has been extended, a discussion has been presented about the practicability of the concurrency control needed to provide serializability guarantees when the complete model is considered. The discussion, making use of statistical considerations, showed that the strict consistency control, based on the extended model, will produce a dramatical increase in the abortion rate of the initiated transactions, due to the high number of conflicts produced by the analysis of the queries.

As an alternative to the high abortion rate, the prevention of such aborts can be only performed by the propagation, as soon as possible, of the changes made by any transaction in the system. This propagation "as soon as possible" can be related to other families of propagation protocols such as *eager update propagation protocols*.

As a conclusion, we have determined that the inclusion of queries in the consistency control of lazy update systems, although it is needed in order to guarantee one-copy serializability, makes the system unusable, due to the high degree of abortions the strict consistency control will cause.

Chapter 5
Analysis of the Abortion Rate on Lazy Update Protocols

In chapter 3, we presented two implementations of consistency protocols, using different approaches for the update propagation. Then, section 3.5 enunciated with an example the inability of the LOMP protocol (as any lazy protocol) to preserve one-copy serializability. This was proven in 4, providing an incompatibility result of laziness and strict isolation.

Nevertheless, the consistency and isolation provided by lazy update protocols can still be useful to a wide range of applications. This affirmation gives sense to the research in the area of laziness and its applications, its inconveniences, problems, and solutions.

With respect to the latest, lazy update protocols have proven to have a critical inconvenience in contrast to eager approaches: the dramatical increase of the abortion rate in scenarios with a high degree of access conflicts. This inconvenience makes unusable the traditional lazy update protocols in certain scenarios, because an unacceptable number of started transactions will terminate with an undesirable abort.

To understand the problem, this chapter presents a set of expressions describing the abortion rate. In the presentation, we model a complete system including nodes, the sessions executed, and the objects accessed by the sessions.

Section 5.1 includes the description of such modeled system, in order to formalize an analysis of the abortion rate in section 5.2. In section 5.3, an empirical validation of the model is presented, and section 5.4 will provide a theoretical analysis of an improvement of the lazy approach. Finally, section 5.5 includes some conclusions about the applicability of the expressions.

5.1 The Modeled System

The targeted system of our analysis follows a number of considerations, designed to configure a scenario as close as possible to a general environment that, although simplified, is able to fit the requirements of the kind of environment we are centered in. This environment was described in the introduction, and has considerations about client applications, system load, pattern of accesses, interconnection network, etc.

In summary, these adopted assumptions are the following:

- There are K COPLA managers running in the system. Each one can be considered as a "node" $N_{k=1..K}$.
- Each node in the system manages a complete replica of the database. This database contains N objects.
- A session S can be written as a tuple $S = [R(S), W(S), m(S)]$ where:
  - $m(S)$ is the consistency mode in which the session has been initiated (checkout/transaction). For this study, we consider that every session in the system only uses the *transaction* consistency mode.
  - $R(S)$ is the set of objects read by the session S. It is also named "readset of S". $R = \{r_i\}_{i=1..|R|}$
  - $W(S)$ is the set of objects written by the session S (or "writeset of S"). $W = \{w_i\}_{i=1..|W|}$
- We assume that $W(S) \subseteq R(S)$. And the objects contained in $R(S)$ and $W(S)$ can be expressed as tuples: $o_i = [id(o_i), ver(o_i), val(o_i), t(o_i), ut(o_i)]$ where:
  - $id(o_i)$ is a unique identifier for the object. The identifier includes the owner node (the node where the object was originated), a sequential number established within the context of each node, and other information used to calculate conflicts.
  - $ver(o_i)$ is the version number of the accessed object.
  - $val(o_i)$ is the value read (or written) by the session for the object.
  - $t(o_i)$ is the local time the object was accessed at.
  - $ut(o_i)$ is the local time the object was more recently updated at.

5.2 Probability of Abortion

We can define the probability for a session S to be aborted as: $PA(S) = 1 - (PC_{conc}(S) \cdot PC_{outd}(S))$ where:

- $PC_{conc}(S)$ is the probability that the session concludes without concurrency conflicts.
- $PC_{outd}(S)$ is the probability that the session concludes without accessing to outdated objects.

The goal of this section is to determine the value of $PC_{outd}(S)$, in order to predict the influence our LOMP has into the abortion rate in the system. To this end, we can calculate this probability in terms of the probability of a session to conclude with conflicts produced by the access to outdated objects ($PA_{outd}(S)$):

$$PC_{outd}(S) = PC_{outd}(r_i)^{nr}$$

taking nr as the number (in mean) of objects read by a session,

$$PC_{outd}(S) = PC_{outd}(r_i)^{\frac{\sum_k nr_k}{K}} \tag{5.1}$$

moreover, $PC_{outd}(r_i)$ is the probability for an object r_i to have an updated version in the instant the session accesses it. This probability can be expressed in terms of the probability for an object to be accessed in an outdated version ($PA_{outd}(r_i)$) as:

$$PC_{outd}(r_i) = 1 - PA_{outd}(r_i)$$

now, let's see the causes of these conflicts: we took r_i as an asynchronous object in the active node that has not been updated since $ut(r_i)$; the outdated time for r_i satisfies $\delta(r_i) = t(r_i) - ut(r_i)$; it can be seen that $PA_{outd}(r_i)$ depends on the number of sessions that write r_i having the chance to commit during $\delta(r_i)$. Let PC_{T,r_i} be the probability for another concurrent session T (that has success in its commit) to finalize with $r_i \notin W(T)$. Then,

$$PA_{outd}(r_i) = 1 - (PC_{T,r_i})^C$$

where C depends on the number of write-sessions that can be committed in the system during $\delta(r_i) \dots$

$$PA_{outd}(r_i) = 1 - (PC_{T,r_i})^{\sum_k wtps_k \times \delta(r_i)} \tag{5.2}$$

now, we can reformulate PC_{T,r_i} as $PC_{T,r_i} = P[r_i \notin W(T)]$ and, considering $W(T) = \{w_1, w_2, \dots w_{nw(T)}\}$, then in mean, it will be satisfied that:

$$PC_{T,r_i} = \Big(P[r_i \neq w_{j \in \{1..nw\}}]\Big)^{nw}$$

taking nw as the mean of $|W(T)|$ for every write-session in the system.

$$PC_{T,r_i} = \Big(P[r_i \neq w_{j \in \{1..nw\}}]\Big)^{\frac{\sum_k nw_k}{K}} \tag{5.3}$$

The next step consists of the calculation of $P[r_i \neq w_{j \in \{1..nw\}}]$. To do this, we must observe the number of objects in the database (N). The probability that an accessed object is a given one is $\frac{1}{N}$, thus: $P[r_i \neq w_{j \in \{1..nw\}}] = 1 - \frac{1}{N}$

Finally, the complete expression can be rewritten as follows:

$$PA_{outd}(r_i) = 1 - \left(1 - \frac{1}{N}\right)^{\frac{\sum_k nw_k}{K} \times \sum_k wtps_k \times \delta(r_i)} \tag{5.4}$$

This expression provides a basic calculation of the probability for an object access to cause the abortion of the session by an out-of-date access.

The expression can be calculated with a few parameters. Only nw_k and $wtps_k$ must be collected in the nodes of the system in order to obtain the expression. Thus, it becomes possible for a node to estimate the convenience for an object to be locally updated before being accessed by a session. This estimation will be performed with a certain degree of accuracy, depending on the “freshness” of the values of nw_k and $wtps_k$ the node has. The way the expression can be used, and an adequate mechanism for the propagation of these parameters will be presented in chapter 6.

5.3 Experimental Validation of the Model

We have validated the algorithm presented above by implementing a simulation of the system. In this simulation, we have implemented nodes that concurrently serve sessions, accessing to different objects of a distributed database. We have also modeled the concurrency control, and the lazy update propagation used by LOMP.

5.3.1 Assumptions

The assumptions for the implementation of the simulation [CM79, BCM87, Bag90] are compatible with the ones taken for the model calculation, and the values have been established to increase the number of conflicts produced by the transactions executed in the system (i.e., this configuration shows a “worst-case” scenario for our protocol):

- There are 4 nodes in the system, each holding a full replica of the database, that contains 20 objects. Each node executes transactions, accessing the database.
- For every object, a local replica holds the value of the object, and the version corresponding to the last modification of the object. That is, the only synchronous replica for each object is its owner node.
- There are three kinds of transactions, with a probability to appear of 0.2, 0.4, and 0.4

respectively:

- ”Type 0”, or read-only transactions: reads three objects.
- ”Type 1”, or read-write transactions: reads three objects, then writes these three objects.
- ”Type 2”, or read&read-write transactions: reads six objects, then writes three of the objects read.

- The model supports the locality of the access by means of the probability for an accessed object to be owned by that node (i.e. the node where the transaction is started).
  - For read-only transactions, this is 1/4, (as the system contains 4 nodes, this models no locality for read-only transactions).
  - For read-write transactions, and read&read-write transactions, the probability is 3/4 (i.e. the number of local accessed objects should be 3 times higher than the number of accessed objects owned by other nodes).
- The cost of each operation are shown in time units (t.u.):
  - Read operation in a local database: $LR = 0.01\, t.u.$
  - Write operation in a local database: $LW = 0.02\, t.u.$
  - Cost of a local-update request: $LUR = K_{LUR} \times LR, K_{LUR} = 5$
  - Cost of a confirmation request: $CR = 0.6 \times LUR$
- In order to provide a complete study of the algorithm, subsection 6.4.3 will include different executions of the simulation, with values for the constant K_{LUR} in the range $[1..10]$.
- The simulation time has been set at 1,422 t.u., discarding the first 2 t.u. as stabilization time for the simulation. This allows to start up to 60,000 transactions.

Due to the characteristics of the expression shown in section 5 the algorithm becomes very sensible for low values of the established threshold. To relax this, we have applied an escalation to the basic expression of $PA_{outd}(o_i)$. The goal of this escalation is to distribute the values provided by $PA_{outd}(o_i)$ in a more homogeneous way. The scaled function $PA_{tra}(o_i)$ just expands the main range of values taken by $PA_{outd}(o_i)$ into a wider scale, and compacts the residual range.

The rest of the chapter studies each parameter of the protocol using the normalized expression (PA_{tra}).

5.3.2 Accuracy of the Prediction

When the expression exceeds the established threshold for an object, and an update request is sent, it is possible for the response of this request to contain the same version for the requested object (e.g. when the object was, in fact, up to date). We name this situation "Inaccurate prediction".

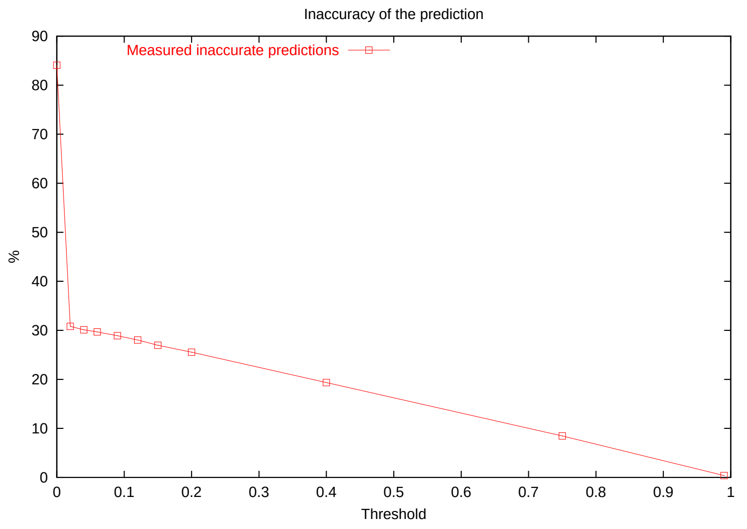


Figure 5.1: Evolution of the inaccuracy of the prediction for different thresholds

The more accurate the predictions are, the less overhead the algorithm introduces in the system. This accuracy of the predictions will be given by the set threshold: higher values for the threshold should provide more accurate predictions.

The figure 5.1 shows the evolution of the inaccuracy of the prediction, for different values of the threshold. For lower values of the threshold, the number of update requests is very high, and many of them are unnecessary. In contrast, a higher threshold produces a lower number of update requests, and only the most likely stale objects will be asked for update.

In general, it can be observed that higher values for the threshold increase the accuracy of the prediction.

The evolution of the inaccuracy with respect to the amount of LocalUpdate messages is shown in the figure 5.2. The optimum line is also shown, and corresponds with the diagonal. The more accurate the prediction is, the closest the curves are. The studied implementation differs from the ideal line with a lower bound pattern, and it is shown that is quite proximal to the ideal.

5.4 Theoretical Boundary of an Improvement

The first sections of this chapter have been dedicated to the study of an statistical expression determining the probability for a particular object access, to obtain an out to date value, thus causing the abortion of the requesting transaction.

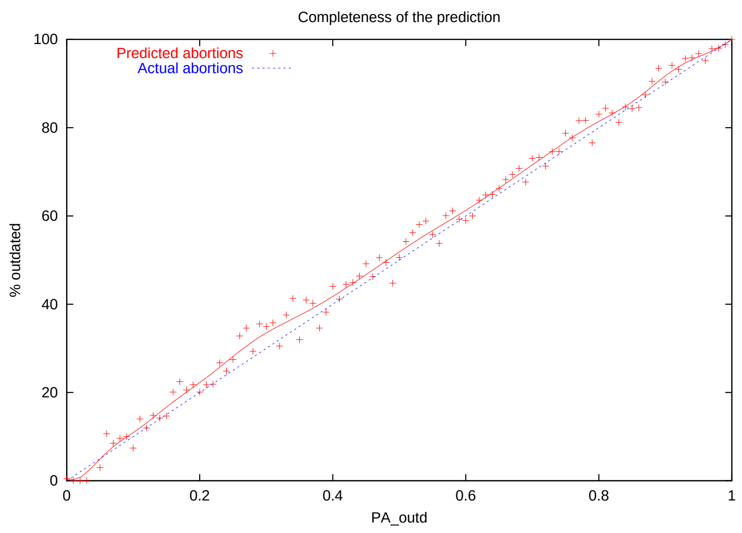


Figure 5.2: Evolution of the inaccuracy for different P[update]

We can make use of such expression, in order to determine the achievable improvement for a transaction, in terms of abortion rate, when the average outdate time of the objects is decreased.

Unfortunately, this decrement in the outdate time will cause a degradation in the service time of the executed transactions, and this must also be taken into account.

In this section, we present a theoretical boundary for the reduction of the abortion rate that an adequate exploit of the expression can provide.

5.4.1 Preliminaries

The used expression for the probability of an access to be a stale-access was presented in (5.4) as:

$$PA_{outd}(r_i) = 1 - \left(1 - \frac{1}{N}\right)^{\frac{\sum_k nw_k}{K} \times \sum_k wtps_k \times \delta(r_i)} \tag{5.5}$$

In the expression, the elapsed time between two consecutive updates of the object r_i is expressed as $\delta(r_i)$.

We can perform a serial analysis in order to determine the mean value for $PA_{outd}(r_i)$. The analysis can be easily performed applying differentiate calculus to the expression. The obtained expression is showed to be:

$$PA_{outd} = 1 - \left(1 - \frac{1}{N}\right)^{\frac{\sum_k nw_k}{K} \times \sum_k wtps_k \times \delta} \tag{5.6}$$

where δ is the mean value for the $\delta(r_i)$ in a system execution, and PA_{outd} is the mean value for the $PA_{outd}(r_i)$ in the same system execution.

Now, we can obtain $PC_{outd} = 1 - PA_{outd}$, being:

$$PC_{outd} = PC = \left(1 - \frac{1}{N}\right)^{nwwt \times wtps \times \delta} \tag{5.7}$$

where $nwwt$ is the number, in mean, of objects written by write-transaction in the system, and $wtps$ is the total number of transaction executed in the system per second.

5.4.2 Average Outdate Time

To simplify, let's suppose that transactions are distributed homogeneously along the system history.

Imagine the system execution history as a line, where a number of transactions are sequentially executed. For a certain object o_i, the probability for an executed transaction to read o_i will be $\frac{nr}{N}$, where nr is the number, in mean, of objects read by any executed transaction (either read or write transactions are included here), and N is the number of objects in the database.

The probability of the object o_i to be updated by a lazy replication protocol depends on the probability for a transaction that read o_i to be aborted by a stale-access (i.e. $PA_T = PA^{nr}$). Thus the probability for an object to be updated by a generic transaction will be:

$$PA^{nr} \times \frac{nr}{N}$$

Now, let tps be the number of transactions executed in the system per second. Thus, there will be $up(o_i) = tps \times PA^{nr} \times \frac{nr}{N}$ updates of o_i per second. Finally, we can express $\delta(o_i)$ as $\frac{1}{up(o_i)}$, and, in mean:

$$\delta = \frac{1}{tps \times PA^{nr} \times \frac{nr}{N}} \tag{5.8}$$

If the accessed objects are updated along the transaction, the value for δ will be decreased proportionally to the amount of updates performed during the transaction execution.

To model this, a simple approach can be expressed with the following expression:

$$\delta' = PC \times d'_T + (1 - PC) \times \delta \tag{5.9}$$

where d'_T is the duration of a transaction when the updates are performed along its execution. For the aborted transactions, (i.e. $(1 - PC)$), the mean outdate time is unchanged (δ). In contrast, for committed transactions, the new outdate time is decreased to d'_T (i.e. the duration of the transaction).

Now, the duration of a transaction when the updates are performed will depend on the number of requested accesses that are actually updated along the transaction execution ($nr \times P_{UPD}$), and the cost of each of these updates (K_{UPD}). Note that P_{UPD} is the probability for a requested object to be updated. Thus, if $P_{UPD} = \frac{1}{2}$, there will be forced to be updated the half of the objects requested by a transaction.

In summary, the expression for d'_T can be composed by:

$$d'_T = d_T + nr \times P_{UPD} \times K_{UPD} \tag{5.10}$$

Replacing 5.10 in 5.9, the new outdate time will follow the expression:

$$\delta' = \delta \times (1 - PC) + PC \times (d_T + nr \times P_{UPD} \times K_{UPD}) \tag{5.11}$$

This result will be useful in section 5.4.3, where the achievable abortion rate is specified in terms of δ'.

5.4.3 Abortion Rate

In mean, we can say that the achievable commit rate will be, observing equation (5.7):

$$PC' = \left(1 - \frac{1}{N}\right)^{nwwt \times wtps \times \delta'} \tag{5.12}$$

Replacing δ' in the expression, we obtain:

$$PC' = \begin{cases} \left(1 - \frac{1}{N}\right)^{nwwt \times wtps \times \delta \times (1-PC)} \\ \times \\ \left(1 - \frac{1}{N}\right)^{nwwt \times wtps \times PC \times (d_T + nr \times P_{UPD} \times K_{UPD})} \end{cases} \tag{5.13}$$

That can be rewritten as:

$$PC' = \begin{cases} PC^{(1-PC)} \\ \times \\ PC^{PC \times \frac{d_T + nr \times P_{UPD} \times K_{UPD}}{\delta}} \end{cases} \tag{5.14}$$

and simplified as:

$$PC' = PC^{1 + PC\left(\frac{d_T + nr \times P_{UPD} \times K_{UPD}}{\delta} - 1\right)} \tag{5.15}$$

Now, we can replace δ with the expression obtained in 5.8, the resulting expression is:

$$\frac{PC'}{PC} = PC^{PC\left((d_T + nr \times P_{UPD} \times K_{UPD}) \times (tps \times (1-PC)^{nr} \times \frac{nr}{N}) - 1\right)} \tag{5.16}$$

From the equation 5.16 we obtain that the improvement of the probability for an object to be accessed in an adequate way (i.e. not a stale access), is determined by $\frac{PC'}{PC}$, and it will be benefitted from the decrease of the established value or any of the following expressions:

- d_T, the duration of the transactions.
- $nr \times P_{UPD} \times K_{UPD}$, the number of updated objects, and the computational cost of each of these updates.
- tps, the amount of committed transactions per second (including both read-only and read-write transactions).
- $\frac{nr}{N}$, the relation between the amount of objects accessed per transaction (read-only or read-write transactions) and the total number of objects contained in the database.

To simplify the expression 5.15, we can denote as Δ to the existing relation between d'_T and δ (i.e. $\Delta = \frac{d_T + nr \times P_{UPD} \times K_{UPD}}{\delta}$). The resulting expression is:

$$PC' = PC^{1+PC\times(\Delta-1)} \Longrightarrow \frac{PC'}{PC} = PC^{PC\times(\Delta-1)} \tag{5.17}$$

Let's see an example for the improvement achievable in a extremely simple system, where $nr = 1$. In such system, we can establish as a parameter the probability for a requested object to be previously updated (i.e. P_{UPD}), and then study the achieved improvement for different values of Δ (and, consequently, different computational overheads).

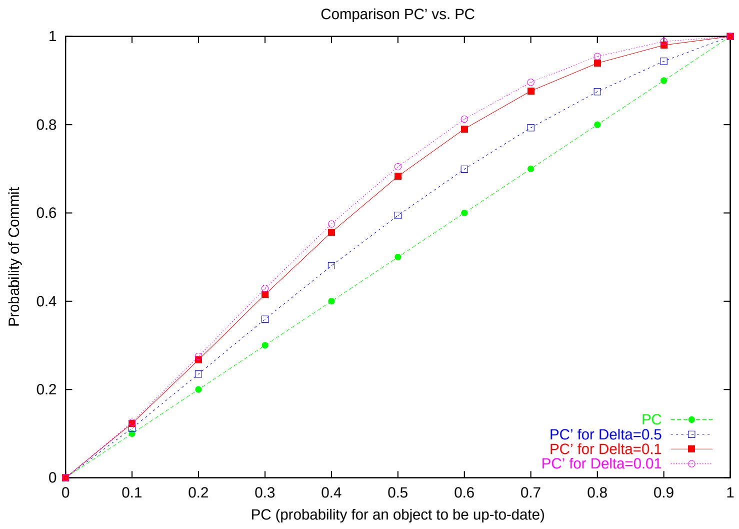


Figure 5.3: Evolution of the improvement for different Δ

Figure 5.3 shows how the commit probability can be improved, when the update-time is decreased to the half, up to the 120% of the original commit time. When the update-time is decreased ten times, the improvement reaches the 140%. Lower values of Δ provides marginal improvements, at a higher computational costs. When transactions accessing to more than one objects are considered, the results show a higher differentiation for the improved abortion rate.

These results points to the convenience, in the scenarios fitting the parameters described above, to apply the techniques postulated by the presented discussion.

5.5 Conclusions

A statistical analysis has been performed in this chapter, in order to provide an expression for the probability for a requested access to obtain a stale value of the required object.

The application of the expression has been also discussed, in order to determine the convenience, using a general algorithm, to update along the execution of a transaction, the objects predicted to be stale. This discussion has provided a set of conditions, in base to a number of parameters, where these generic algorithms can improve the abortion rate of a lazy update protocol.

Consequently, the improvement has been also studied, in base to the established decrement of the update-time of the accessed objects, giving as conclusion that such reductions may considerably improve the probability for an object to be updated.

Thus, the next suitable step should consist on the concrete specification of the mentioned algorithm, and the further validation of the conclusions.

Chapter 6
COLUP: The Cautious Optimistic Lazy Update Protocol

As seen in chapter 3, the main disadvantage of LOMP , as a lazy protocol, consists of its high abortion rate. This is caused by the increase of the number of transactions aborted due to outdated objects.

When a session enters in the commit phase, a lazy protocol needs to ensure that for every accessed object it has been obtained an updated version. If it cannot be guaranteed, then the consistency manager must abort the transaction. This basic mechanism, formalized in chapter 4, makes it necessary for the consistency protocol to abort a number of transactions due to accesses to outdated objects.

This situation becomes very frequent in scenarios where the number of concurrent access increases. Moreover, such behavior can make unusable the system.

The behavior of COLUP is very similar to the basic Lazy protocol (LOMP) mentioned above, but it makes use of the results obtained in chapter 5 to predict -with a certain accuracy- the probability of an accessing object to be outdated in the local database. This prediction is then used to locally update the object, and reduce the probability of abortion of such transaction.

The rest of the chapter is organized as follows: section 6.1 describes the COLU protocol in terms of a modification introduced in the basic lazy protocol (LOMP) depicted in 3.2. Section 6.2 details the particularities of the implementation of these modifications, and section 6.3 presents the flexibility of the protocol. Section 6.4 includes an experimental measurements of the COLU protocol. In section 6.5, an auto-adaptative algorithm to minimize the abortion rate with an acceptable performance is also provided. Finally, section 6.6 includes some conclusions.

6.1 Protocol Description

The aim of COLUP is centered to this undesirable behavior, trying to avoid this abortion increase. The main principle used by COLUP [IBMEBA03b, IBMEBA03a] consists in the calculation of a prediction for an accessed object to be outdated.

To reduce the number of abortions avoiding the use of locks, it becomes necessary to update the local version of an object before it is accessed by a local transaction. To do this, the first approach should use the following principles:

- Any consistency protocol should always guarantee that, for each object, a particular subset of the nodes in the system will always hold an updated value. This set of ”synchronous” nodes for each object can vary, depending on the particular protocol, from one single node (pure lazy protocols) to the complete system (eager protocols).
- To reduce the probability for a transaction to read an outdated value of an object, the simplest way consists, for each object accessed by a transaction, of preceding such access with a request, asking about the more recent value of such object to one of the ”synchronous” nodes for this object.
- In commit time, the behavior should be maintained unchanged, because this approach cannot guarantee that the objects accessed by the transaction have not been modified since the access. In contrast, it is only reduced the probability for an accessed object to be considered outdated at commit time.

But this technique is not adequate, because the time spent in performing a request every time the transaction makes an access will be very inefficient. This inefficiency is produced because in most of the cases the objects accessed by a transaction will be updated, being it unnecessary the ”update request” performed by the protocol.

Moreover, this inefficiency can be reduced using the results obtained in chapter 5, to perform the ”update request” only when the probability for an accessed object to be outdated exceed a pre-established threshold.

The rest of this chapter explains how these concepts can be introduced in the LOM protocol, and the properties of the resulting protocol.

6.1.1 Modification of the LOM Protocol

The way the COPLA manager decides to send or not an update request for an object that is about to be accessed is quite simple:

1. During the life of a session, an access to an object o_i is requested by the application. This access is intercepted by the COPLA manager of the active node, and then the probability of being outdated is calculated for the object o_i. If the active node is a synchronous (or owner) node for o_i, this probability is 0. This is because the protocol ensures that incoming updates of the object will abort every transaction conflicting with the updated object.

   If the active node is an asynchronous node for o_i, then the COPLA manager will use the expression of $PA(o_i)$. To perform this calculation, it is needed $\delta(o_i)$: the time elapsed from the last local update of o_i.

2. If this probability exceeds a certain threshold T_c, then the COPLA manager sends the update request to the owner of o_i. If the threshold is not reached, the protocol continues as described in section 3.2. Section 6.4.2 will present an empirical approximation to determine an optimum value for the threshold T_c.

3. In other case, after the COPLA manager allows the session to continue, it waits for the update response. This response indicates whether the local version of o_i is updated, or outdated (then, the response contains the newest version for the object). If the local version must be updated, then the update is applied in the local database, and the update time is also written down.

4. Once the COPLA manager has ensured that o_i is updated, the required access to the object can continue.

By forcing to update an object before the session accesses it, the COPLA manager decreases the value of δ_i, to the length of the session. Thus, the chance for a session to success the commit phase is higher.

This technique implies that every update performed in the asynchronous nodes of an object must include a timestamp of the instant the update is performed at. Note that it is not necessary to use global time, but it is only necessary the local time for each node.

An expression for the probability for an object to cause the abortion of a session has been presented in section 5. This expression can be used to predict the convenience for a session to ensure that an object that is asynchronously updated has a recent version in the local database.

In order to apply these results, it becomes necessary to establish a threshold of $PA(o_i)$ to consider the object “convenient to be updated”. In section 5.3 a study of the accuracy of this expression is presented.

An adequate value for this threshold should minimize the number of abortions caused by accesses to outdated objects, and keeping low the number of updates for the system.

The minimization of the number of updates, will increase the number of sessions executed in the system per second, because it will decrease the resources used by the update propagation. On the other hand, this minimization will cause an increase in the number of aborted sessions, because the number of outdated objects will also be increased.

The higher the threshold is, the less number of abortions will occur in the system, but the higher updates will be done, and a higher overhead will be introduced in the system.

The implementation of this principle introduces a new request in the LOM protocol. Now, the active node for a session will send "Update requests" to the owners of the accesses objects, in order to get the updated versions for such objects. This update request message can be sent along the session execution, in order to maintain updated (in a certain degree) the objects being about to be accessed by the session.

This technique will reduce the probability of abortion caused by the accesses to outdated objects within the sessions. This improvement is based on the outdated time, shown as $\delta(o_i)$ in the expression of $PA_{outd}(o_i)$

Section 5.3 describes the behavior of the protocol for different thresholds, and section 6.5 provides an adaptative algorithm to converge to a threshold near to the optimum.

6.2 Inclusion of the Expression for the Abortion Rate

In the previous section, it has been presented a description of the COLUP modification in front to the basic lazy protocol (LOMP).

This modification mainly consists of an evaluation of the probability for an object to be outdated when it is about to be accessed by a transaction.

This evaluation makes use of the basic expression presented in 5:

$$PA_{outd}(r_i) = 1 - \left(1 - \frac{1}{N}\right)^{nwwt \times wtps \times \delta(r_i)} \tag{6.1}$$

This expression makes use of a number of parameters, that should be measured by each node in the system, in order to be propagated to each other node.

With the latest received parameters from each node, a single node can summarize this information, obtaining an approximation for each parameter needed to evaluate the expression.

Thus, it can be calculated the number of write-transactions committed per second in the system (wtps) with the expression:

$$wtps = \sum wtps_k, \forall N_k \ in\ the\ system$$

And the number of objects written in mean for each write-transaction (nwwt) as:

$$nwwt = \frac{1}{K} \sum nwwt_k, \forall N_k \ in\ the\ system$$

being K the number of nodes in the system (i.e. $k \in [1..K]$).

Once a node has summarized all the needed information, the expression can be evaluated for each required object, just obtaining its *outdate time* (represented in the expression as $\delta(o_i)$). This *outdate time* can be calculated with the current **local time** of the node, and the annotated time, for the object o_i, the last time it was written.

Note that it is only necessary to use the local time of the node, and there is no need to maintain a global time in the system. This is because the $\delta(o_i)$ is a gradient of time, and this is calculated as a difference between the last annotated time $t_o^{N_i}$ and the current time $t_{now}^{N_i}$. For a particular node N_i, the difference between its local time and a supposed global time ($d_i = t^{N_i} - t^{global}$) should be unchanged along the execution of the system, because the progress of every local clock can be homogeneous. Then, it is simple to see that the difference d_i will not affect to the evaluation of the $\delta(o_i)$:

$$\delta(o_i) = t_{now}^{global} - t_o^{global} = (t_{now}^{N_i} - d_i) - (t_o^{N_i} - d_i) = t_{now}^{N_i} - t_o^{N_i}$$

The last consideration to evaluate the probability for an accessing object to be outdated is the role, with respect to the accessing object, of the active node of the transaction (the node where the transaction is executed, and hence the accessing node).

If the active node of the transaction is a synchronous replica of the accessing object, then the probability for the object to be outdated will be always 0. This is exactly the guaranty provided by the ”synchronism”.

In the other hand, asynchronous replicas of an accessed object can hold an outdated version of such object, and the probability for it to occur must be evaluated with the expression of PA_{outd}.

Thus, the complete expression to be used can be summarized as:

$$
\begin{aligned}
& P(o) = && \begin{cases} 0 & , N \notin A(o) \\ PA_{outd}(o) & , N \in A(o) \end{cases} \\
with \\
& PA_{outd}(o) = && 1 - (1 - \tfrac{nwwt}{N})^{\delta(o) \times wtps} \\
& \delta(o) = && t_{now}^{N_i} - t_o^{N_i} \\
& ps = && \sum wtps_k, \forall N_k \; in\ the\ system \\
& nwwt = && \tfrac{1}{K} \sum nwwt_k, \forall N_k \; in\ the\ system
\end{aligned}
\tag{6.2}
$$

Where $A(o)$ represents the set of nodes preconfigured as ”synchronous” replicas for the accessed object o.

6.3 From Eager to Lazy Update

Another particularity of the COLUP approach is that the protocol can be parameterized to have the behavior of both eager and lazy update protocols.

Thus, it is possible to use COLUP to implement an eager consistency protocol, and it is also possible for COLUP to have the behavior of a pure lazy protocol.

In the description of the protocol, there was mentioned that, for each object, the system maintains a set of synchronous replicas, all of them maintaining a synchronized copy of the object (note that one of these synchronous replicas is the owner node of the object).

To provide synchronization guarantees, the COLUP protocol propagates *within the commit phase* the updates performed over an object to all its synchronous replicas. Thus, this propagation is performed as an eager update propagation protocol, because the transaction termination only arises when all of the synchronous replicas are updated.

In addition, the COLUP protocol, as defined above, doesn't perform any analysis about the probability of an object to be stale, when the requesting transaction is in a synchronous replica for such object. This means that synchronous replicas will not need to perform any spontaneous updates for their synchronous objects.

Now, extending the set of synchronous replicas, for each object, to the entire system, we will obtain an update propagation protocol which ensures that, for each node in the system, all the updates are propagated within the commit phase, and no other updates are needed anymore. This is precisely the definition of an eager update propagation protocol.

On the other hand, the description of COLUP includes the specification of a threshold to be compared with the probability for an accessing object to be outdated. If this probability exceeds the established threshold, the update is forced, requesting to the object owner the latest version of the object. In other case, the access is performed without performing the update at access time.

Thus, if no update is forced along the life of a transaction (i.e. it has been established threshold to 1.0), and the synchronous set for each object is restricted to the owner node, no update will be performed in the commit phase. In addition, these updates will only be performed when a transaction is aborted due to any access to outdated objects. This is the definition of a pure lazy update propagation protocol.

In summary, to make COLUP to have the behavior of an eager update protocol, the configuration must be:

- **Synchronous set for each object: the entire system**

On the other hand, to implement a pure lazy behavior with COLUP , the configuration must be:

- **Synchronous set for each object: only the owner node**
- **Threshold = 1.0**

Thus, we have shown that the COLUP protocol is versatile enough to cover a wide range of solutions.

6.4 Measured Results

In order to evaluate the improvement of the approach postulated by COLUP , we have implemented a simulation of a basic COPLA system. In this simulation, a number of nodes initiate a number of transactions of several types.

The assumptions and the simulation are identical to the ones described in 5.3.

The concurrency control is performed with the COLU protocol, using as a threshold a preconfigured value.

In this section, we present an empirical evaluation of the behavior of our approach, when different thresholds are used.

The adequate value of this threshold should reduce the number of abortions in the system, maintaining below reasonable values the time spent in communication of the updated values.

6.4.1 Abortion Rate

The main goal of the COLUP algorithm consists of the reduction of the abortion rate. This reduction is achieved by the adjustment of the value of a threshold. When the probability for an object to be outdated ($PA(o_i)$) is greater than this threshold, an update request is sent to the owner of the object. This reduces the probability of abortion of the transaction, because it is reduced the time the objects are out of date in each node.

Figure 6.1 shows the evolution of the abortion rate when different thresholds are used. In

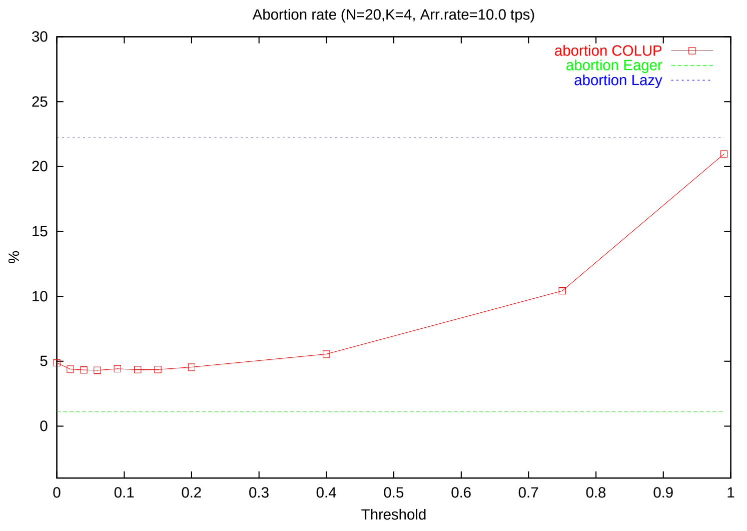


Figure 6.1: Evolution of the abortion rate for different thresholds

one extreme, if the threshold is set to 0, the protocol tries to update every accessed object, introducing the lower abortion rate.

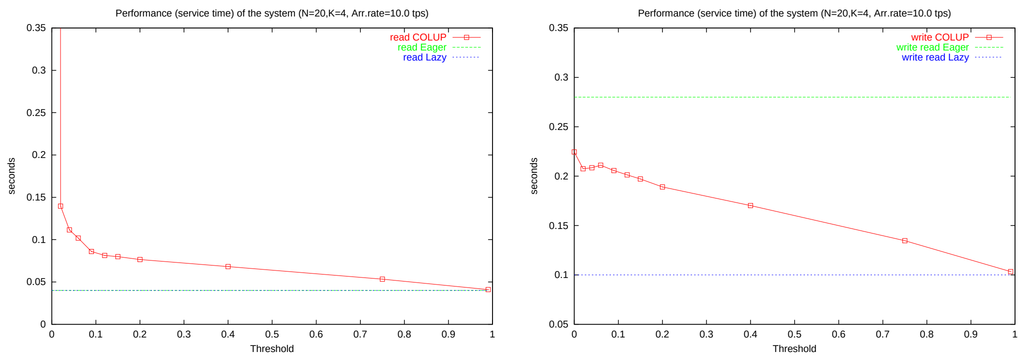
In this case, the behavior of the protocol is similar to an eager protocol. In such algorithms, every object update is always propagated to each node in the system. The disadvantage of the eager approach appears when the same object is updated several times before another node performs an access to it. In contrast, in the case of COLUP with $Threshold = 0$, the disadvantage consists of the unnecessity of performing update requests when the accessed object is already up-to-date.

On the other hand, when a $Threshold = 1$ is set, the protocol behavior is the same than a pure lazy protocol: no update is performed, unless a transaction is aborted.

Increasing the value of the threshold, the abortion rate will also be increased.

6.4.2 Performance

On the middle point, it can be found a balance between accuracy and abortion rate. In this point, the time spent in aborted transactions will be low, and the overhead introduced by the updates will be always useful. Then, the number of transactions committed per second should be maximized, keeping low the abortion rate. The service time of the server transac-



(a) Read-only transactions (b) Read-write transactions

Figure 6.2: Evolution of the service time for different thresholds

tions (see figure 6.2) is improved when the established threshold is increased. This is caused by the lower number of update requests introduced by a higher threshold during the transaction execution. For lower thresholds, the amount of update requests is greater, and thus, the overhead introduced by the protocol is also increased, providing worst performances to the user application.

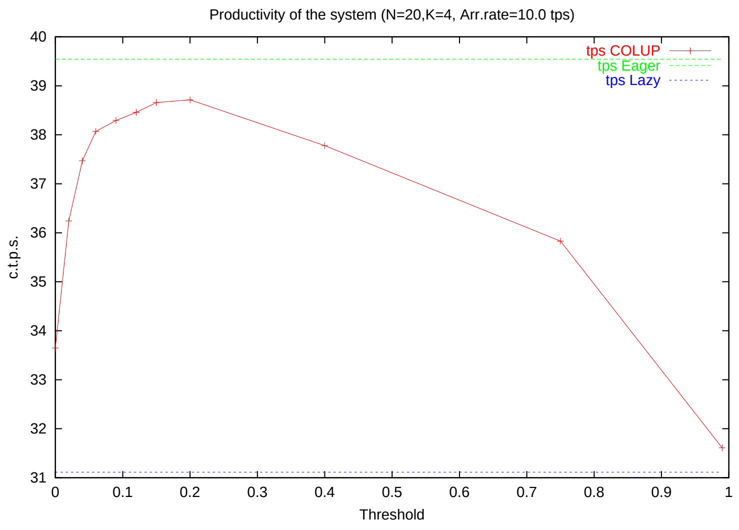


Figure 6.3: Evolution of the number of transactions committed per t.u. (c.t.p.s.) for different thresholds

To summarize abortion rate and service time, in figure 6.3, the productivity of the system is analyzed through the number of transactions served per time unit.

It is shown how the number of transactions committed when the COLUP gives a pure lazy behavior (i.e. $Threshold = 1.0$) is higher than the productivity obtained for a "paranoid COLUP" (i.e. $Threshold = 0.0$). On the middle point, it can be found a threshold maximizing the number of commits per time unit.

We have seen that the lower the threshold is, the lower the abortion rate becomes. These experiments show that the evolution of the system for different thresholds has an optimum value that maximizes the performance and productivity of the system. In addition, the abortion rate when this threshold is used, can be kept below a reasonable limit, very near to the optimum provided by an eager approach.

6.4.3 Optimum Threshold

We have seen that COLUP has a behavior similar to the eager protocol with respect to the abortion rate. In addition, we encountered that in scenarios with non-saturated systems (i.e. the client requests arrive with a frequency lower than the service time offered by the database system), our protocol offers a performance similar to the one achieved with a lazy approach. In a saturated system, however, we encountered that the features offered by COLUP also depend on other parameters of the system. So, we consider an important study to determine, in the worst case for our COLU protocol, the empirical limits of the performance and abortion rate achievable when the protocol is used.

This "worst case" has been considered as follows: In each participating node, a test client application is executed. A number of transactions are initiated by each client application during their execution. The elapsed time, for each client application, between two consecutive initiations will be the minimum. To do this, the client application will initiate the next transaction immediately (i.e. without waiting) after the reception of the system response for the previous transaction. The main difference with the load pattern used in previous sections is that in this test, the notion of "arrival frequency" has been disappeared as a parameter of the experiments.

In this "worst case" in terms of system load, we pretend evaluate the behavior of the algorithm for different environment parameters. To do this, we have considered that the most relevant parameter is the cost introduced by the localUpdates. If this cost is high, it will not be convenient to perform any local update of ”potentially outdated objects”, because the cost of this local update will be higher than the time spent in the transaction if the commit phase cannot be completed (and thus, the transaction is aborted).

To analyze this, we have performed several simulations, giving different values to this cost

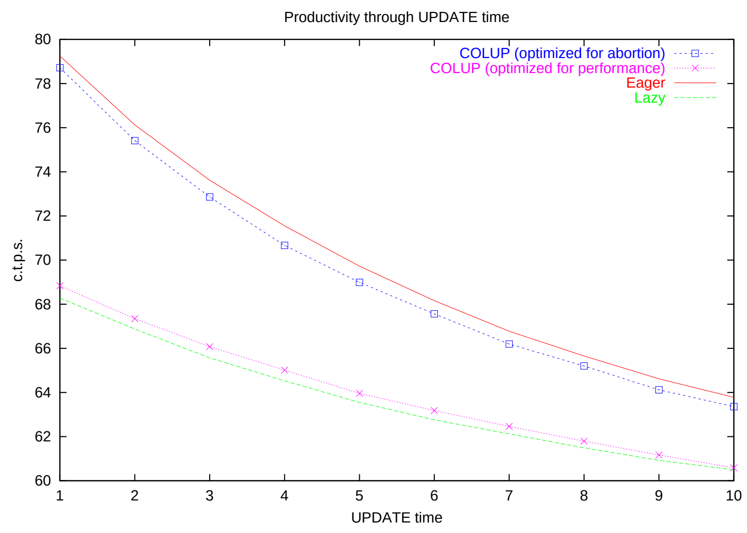


Figure 6.4: Evolution of c.t.p.s. in the optimum for different K_{LUR}

(i.e. making K_{LUR} take values from $[1..10]$). For each K_{LUR}, the simulation includes an environment with a high degree of conflicts (4 nodes, replicating a database with a total number of 20 objects). The rest of the parameters have been unchanged from the detailed simulation in section 6.4.3. For each K_{LUR}, we have taken the optimum value of the thresholds, in terms of productivity (committed transactions per second), and also the optimum threshold has been taken, looking for the lower values for the abortion rate. Finally, for each configuration, the experiment has been repeated, two more times: first, there has been used an eager algorithm (FOB), and then, a pure lazy update protocol (LOMP).

These simulations will be useful to determine the ranges of performance and abortion rate the COLU protocol will be able to provide, when an optimum threshold is established. On one hand, when the threshold is established to minimize the abortion rate, the performance and productivity of the system will be consequently degraded, because the time spent by the protocol in keeping up-to-date the accesses objects will be excessive. On the other hand, when the threshold is optimized to provide a high performance to the system, there will be consequently increased the abortion rate of the initiated transactions, because there will be reduced the number of `LocalUpdate` requests to improve the performance, and a higher number of stale objects will be accessed by the initiated transactions, and thus, they will be aborted.

Figures 6.4 and 6.5 show the evolution of the algorithm for different values of K_{LUR}, either for a threshold optimized for improving the performance, and for a minimized abortion rate. We can observe in figure 6.4 how the reduction of the cost of an update request (lower values of K_{LUR}) leads the system to provide a better productivity with independence of the chosen algorithm, or the applied optimization policy. In addition, it is shown how the COLUP approach is capable to provide productivities higher to those provided by the lazy

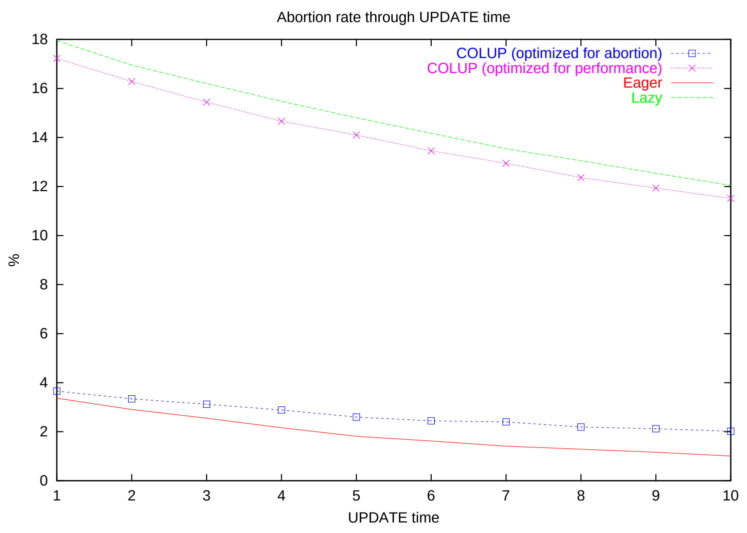
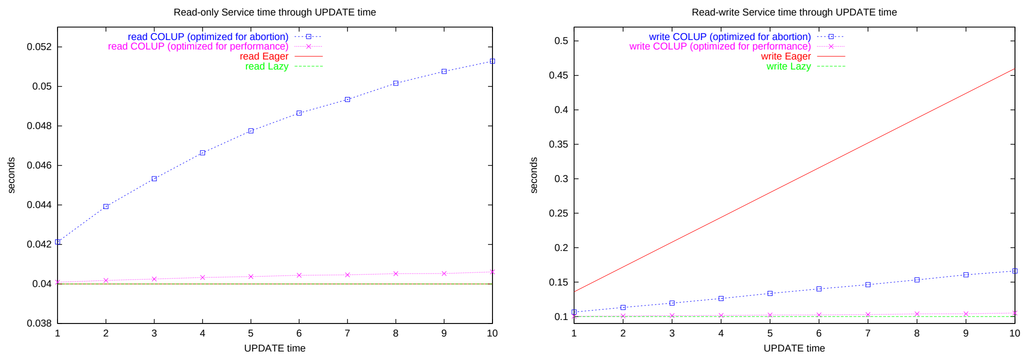


Figure 6.5: Evolution of the abortion rate in the optimum for different K_{LUR}

approach, when the abortion rate is benefitted. Moreover, when the performance is benefited to establish the threshold, the productivity achievable with the COLU protocol is very near to the one provided with the eager approach.

A similar discussion can be taken for the abortion rate. In figure 6.5, it is shown how the abortion rate benefits from increased costs of an update request. In this case, the abortion rate achievable with the COLUP approach is always lower to the abortion rate produced by a lazy approach, and is is very close to the one provided by the eager algorithm if the threshold is optimized to this end. Finally, the service time offered by the COLUP approach



(a) Read-only transactions (b) Read-write transactions

Figure 6.6: Evolution of the service time in the optimum for different K_{LUR}

is shown in figure 6.6, where it can be observed the ranges of service time achievable with the COLU protocol. For read-write accesses (see figure 6.6(b)), the service time of COLUP is always lower than the one obtained with the eager approach (that is higher to the service time provided by a lazy approach), reaching the minimum value -near to the minimal established by the lazy approach- when the threshold is optimized for performance. On the other hand, with respect to the read-only transactions, both eager and lazy approaches provide the same service time. In contrast, the service time provided by the COLU protocol is always higher to them. Moreover, the service time of COLUP is very close to the one provided by the lazy protocol when the threshold is established to improve the performance, and it is increased with a logarithmic pattern in the worst case when the threshold is optimized for the minimization of the abortion rate.

Note the figures 6.6(a) and 6.6(b) have different scales, and the service time of a read-only transaction is always lower than the one obtained for a read-write transaction.

As a conclusion, the abortion rate obtained with COLUP will be always lower than the one achievable with the lazy approach. In addition, it will be very close to the minimum abortion rate established by the eager approach.

On the other hand, the performance and productivity achievable with the COLUP approach will vary from the ones provided by the lazy approach to the eager one, depending on the established threshold for the COLU protocol.

The establishment of this threshold will depend on the necessities and pattern access of the user application of the system.

6.5 Heuristic Run-Time Search of the Optimum Threshold

In section 5.3 we have shown how the protocol behavior can vary from a pure lazy protocol to a behavior similar to the one obtained with an eager approach. This tuning can be performed by the adequate adjustment of the used threshold to be compared with the value of PA_{outd}.

We have also shown how, for a concrete scenario, it can be found the value for this threshold making the protocol enhance the performance of the system, while it is kept low the abortion rate.

But this tuning, as dependent of the system, should not be static. In order to provide flexibility to the protocol, a modification should be introduced, to make it able to find itself a value for the threshold that approximates the behavior of the protocol to the optimum performance.

6.5.1 Modification of the Protocol

In section 5.3 a relationship has been found between the accuracy of the update requests and the performance of the system. Moreover, we have also proven that the performance is also affected by the abortion rate. Our adaptative algorithm exploits this, trying to adjust the threshold in order to minimize the inaccuracy, but keeping low the abortion rate.

- When the system starts, the initial threshold is set to a mean value ($T_0 = 0.5$).
- When an update request is performed (the value of PA for an object exceeds the current threshold T_i), the protocol obtains as a response the updated object, and the latest version. If this updated version corresponds to the version already held in the local database, the update request must be considered as "vain". This imprecision is annotated in the transaction context, and summarized at commit time.
- At commit time, the protocol performs the voting phase to achieve the consensus about the transaction. As a result, the transaction can be aborted due to a number of conflicting objects. This fact is also annotated in the context of the transaction.
- When the commit phase is completed (either with an abortion or with the confirmation of the transaction), the collected annotations are summarized, and the threshold is modified:

  $$T_{i+1} = \frac{K_{hist} + Q(N_{vain}, N_{fail})}{(K_{hist} + 1)} \times T_i$$

  being

  $$Q(N_{vain}, N_{fail}) = \left\{ \begin{array}{ll} K_{inc} & \text{, if } N_{vain} > N_{fail} \\ 1.00 & \text{, if } N_{vain} = N_{fail} \\ (1 - K_{inc}) & \text{, if } N_{vain} < N_{fail} \end{array} \right\}, \; with\ K_{inc} > 1$$

  The expression $Q(N_{vain}, N_{fail})$ gives values below, equal, or behind 1, in order to decrease, keep, or increase the value of T_{i+1}.

  The constant K_{hist} depends on the variability of the system (mainly K_{LUR}). To provide faster adaptability, K_{hist} should take lower values.

  The constant K_{inc} depends on the variability of the transactions, taking lower values for systems with homogeneous transactions.

6.5.2 Validation

We have included the adaptative algorithm in our simulation, and the experiments performed for section 6.4.3 have been repeated.

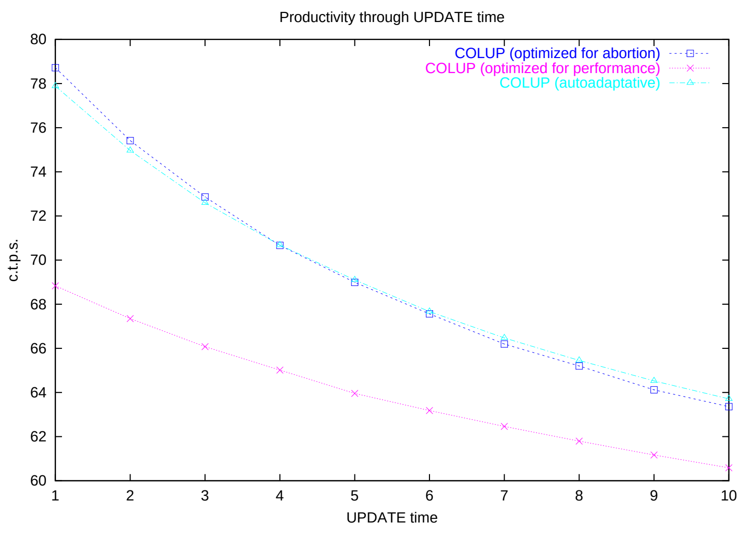


Figure 6.7: Comparison for different K_{LUR} of the optimum (c.t.p.s.) with adaptative threshold

The results of these experiments (figure 6.7 and 6.8) show the proximity of the behavior obtained with the adaptative approach, versus the behavior obtained with an optimum value set with experimental techniques. As it will be shown, it can be obtained a near to the optimal value for the productivity, while the abortion rate is kept below reasonable limits.

For the performance, as shown in figure 6.7, the distance between the two alternatives is always lower than 5%, and it is less than 1% in the 96% of the cases. Thus, the adaptative search of the optimum threshold provides an acceptable productivity, and offers a simple way to establish automatically an adequate threshold.

With respect to the abortion rate, figure 6.8 shows how the auto-adaptative method also provides acceptable results with respect to the reduced abortion rate provided by the election of an optimum threshold. In our experiments the minimal abortion rate was established around the 3.7% in the worst case, and the abortion rate achieved with the auto-adaptative algorithm was, for the same environment, below the 4.7%. This ratio is the same for different environments, providing an acceptable value for the abortion rate, while the obtained productivity of the system keeps near to the maximum achievable.

6.6 Conclusions

In this chapter, the COLU protocol has been presented as an implementable alternative to the existing lazy approaches. This protocol, based on the prediction of the probability for an accessed object to be outdated, produces abortion rates near to the ones provided by eager protocols, maintaining the performance of the system near to the achieved by lazy update

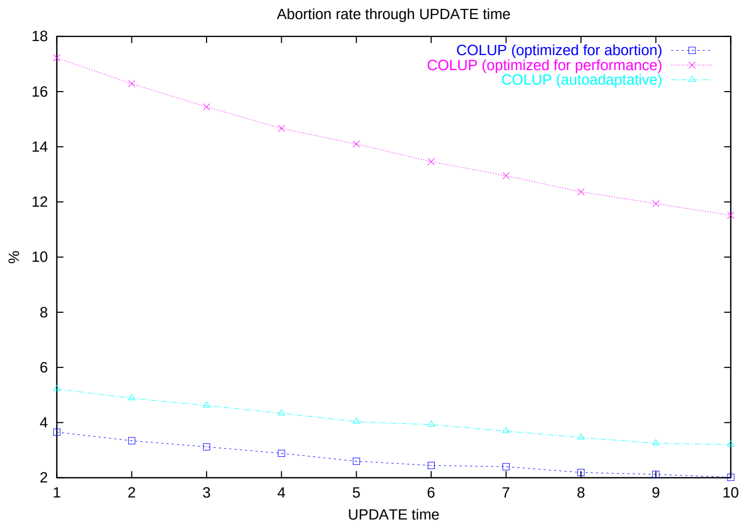


Figure 6.8: Abortion rate with adaptative threshold

protocols.

The presented basic approach can be configured, depending on the adequate establishment of a static threshold, both to improve the performance of the system, and to reduce the abortion rate produced in the system. Intermediate values will produce in the system values for the abortion rate lower than the ones provided by a lazy update protocol, at the cost of a degradation in the performance provided by the system. Thus, the configuration of such threshold should be determined by some parameters defined in the system, such as the network capacity, and should fit the necessities of the client applications of the distributed database system.

To make it more flexible, the basic algorithm has been improved with an auto-adaptative technique that makes it possible for the protocol to find itself an adequate, near to the optimum, value of the threshold. This algorithm is again parameterized with a number of constants, determining the necessities of the user applications with respect to abortion rate. As a result, the final algorithm provides independence of fluctuations of the system load, and environment characteristics, providing a well-fitted solution with the necessities of the client applications, with independence to environmental changes.

As an example, a general configuration of the auto-adaptative algorithm has been also included. The behavior of the COLU protocol when this auto-adaptative technique is used is shown to provide the 98% of the optimum performance. In addition, for the same configuration of the algorithm, the abortion rate is maintained below reasonable limits, producing an improvement of such abortion rate of about the 92% of the maximum possible improvement.

As a conclusion, the proposed COLU protocol, with the inclusion of the auto-adaptative technique to flexibly establish an adequate value for the threshold, has proven to be an implementable protocol to solve the update propagation in a replicated database, with a behavior near to the optimum, either in systems with a low load degree, and systems near to saturation. The characteristics offered by the protocol joins the goodnesses of eager and lazy update protocols minimizing their inconveniences.

Chapter 7
Providing Lazy Self-Recovery Ability to COLUP

In chapter 6 we presented a lazy update protocol, with a good behavior with respect to the abortion rate and an empirical mechanism to obtain the optimum value for the threshold of $P(o_i)$ has been also shown. However, the calculus of $PA(o_i)$ can be used for other techniques than the evaluation of the convenience for an update. This is the case of the *Fault Tolerance*. In this section, we will show a modification of the COLU Protocol, in order to provide fault tolerance to the protocol. This modification is based on two principles: role migration, and the calculation of $PA(o_i)$.

To this end, section 7.1 describes the concept of *lazy recovery*, and section 7.2 presents the basic proposed recovery protocol. This proposal is revised in section 7.3 to include a number of particularities. Finally, section 7.4 includes conclusions about the capabilities, behavior and guarantees provided by the protocol.

7.1 Notion of Lazy Recovery

Fault-Recovery was presented in the introduction as the processes needed by the system to stabilize the normal system behavior when some node has been suffered any fault in its functionality, or when a faulty node recovers from its abnormal state, and is considered to be re-inclosed in the system.

Earlier jobs about fault-tolerant systems[Avi76] presented fault-tolerant systems as systems where redundancy is needed in order to allow partial failures in some components. The basic techniques described in these jobs, although they were initially presented for hardware systems, have not lost validity along the time.

The principles presented were based on the replication of such components of the system that may be reliable in order to ensure the functionality of the entire system even when some of those components suffer a failure.

When these principles are ported to software systems, we encounter that redundancy can again help us in the task of providing guarantees about system availability.

Many techniques have been discussed from the earlier approaches in order to provide these guarantees[EASC85]. In the particular field of replicated databases, as a system where the information is replicated, and accessed within transactional contexts[KB85, BJ86, Jos85, Bir86], additional considerations must be taken.

In summary, they can be classified in two main groups:

- *Self-stabilization Systems*[Dij74], where the fault-tolerance is achieved with a minimal amount of additional effort, and the entire system can proceed even when failures or recoveries arise.
- *Recovering Systems*[SS83, Nel90, Cri91b, HT94], where the recovery is performed with specific (and often weighted) algorithms designed to reestablish the normal functionality of the system when a failure is detected, or when a faulty node recover its normality, and is re-inclosed in the system.

The second kind of recovery, makes it easier to design distributed algorithms, because they don't include in its normal functionality the failure detection, and the adaptation of the system to such changes is also excluded from the algorithms. In addition, these considerations used to be an overhead in the self-stabilization algorithms with respect to the *Recovering Systems*, because the latter have no need to perform any additional action during their normal execution (i.e. in absence of failures).

On the other hand, *Recovering Systems* include important overheads when the system suffers failures, or re-inclusions of recovered nodes. This is because specific algorithms must be run in the system when such situations occur, in order to reestablish the normal functionality of the system. This inconvenience is not present in *Self-stabilization Systems*, because these systems lack of such specific processes.

Moreover, the specific algorithms mentioned above must be run at least [BK88] when a *system reconfiguration* arises. This is due to two facts: first, a fault-tolerant system uses to be based on replication, and the different replicas must be adjusted to have an adequate behavior; second, a distributed system maintains a distributed state, that must also be treated in presence of system reconfigurations.

In a database system the most weighted management used to be the maintenance of the replicated information in the databases before a system reconfiguration. Moreover, a node recovery conforms a worst case for such reconfigurations, because the information managed by the recovered node may be incomplete, outdated and even inconsistent with the information maintained in the rest of replicas in the database. Thus, the information managed by the recovering node must be synchronized using the data held by the rest of nodes in the database system.

To do this, it can be seen that a huge amount of information may be necessary to be transmitted, and the consistency of the system can be compromised if the process is not achieved carefully. To provide guarantees about the consistency, the recovering node must be avoided to initiate any transaction until the information held in its database is updated. In addition, the rest of the system nodes should take into account that the recovering node is not completely updated, and perform any additional action to guarantee the consistency of the initiated transactions in such nodes. Some approaches even avoid the entire system to initiate transactions during the recovery of a node.

On the other hand, the recovery of a node can be accomplished in a more “graceful” way. Making use of laziness, a recovering node may consider that the information held in its database is asynchronously maintained, and the entire system can continue its normal functionality without the necessity of executing any additional actions. This allow the entire system to proceed without introducing overheads in the used resources.

In addition, the recovering node can reestablish asynchronously the original state of the adequate set of objects when a locally initiated transaction performs a request about such objects.

This approach can be named *Lazy Recovery*, because the re-inclosed node performs such recovery in a lazy manner, updating its state from the rest of the system with a sequence of asynchronous operations.

The approach will reduce the overhead of the recovery process, and allow any node in the system to proceed with normality, even before the recovering node has completely updated its state.

7.2 Modification of the COLU Protocol

Each node in the system runs a copy of a *membership monitor*. This monitor is a piece of software that observes a preconfigured set of nodes, and notifies its local node about any change in this set (either additions or eliminations). Each node is labeled with a number, identifying it, and providing an order between every node in the system. The membership monitor used for the LOM Protocol is described in [MGGB01], and can be also used to provide the membership service to our COLU Protocol. The rest of this section shows the differences between the LOM and the COLU protocols.

1. When the membership monitor notices a node failure (let N_f be the failed node), a notification is provided to every surviving node in the system. This notification causes for each receiving node to update a list of *alive nodes*. The effect of these notifications will be a logical migration of the ownerships of the failed node. Further steps will explain the term *logical*.

2. During the execution of a session, a number of messages can be sent to the different owners of the objects accessed by this session. If a message must be sent to a failed owner N_f, then it will be redirected to the new owner for the involved object. This new owner can be assigned in a deterministic way from the set of synchronous replicas of the object (e.g. electing as new owner the node with an identifier immediately higher to the failed one). Let N_n be the new owner for the accessed object.

   The determinism of the election is important to guarantee that every surviving node redirects its messages to the same node (N_n).

   Note that the messages sent to a node can involve more than one object. This will generate a unique message to the new owner, because every object in the original message had the same owner, and so, will have the same substitute.

3. When the synchronous replica N_n receives a message considering the node as an owner, then the message is processed as if N_n was the original owner. To this end, if the received message was an *access confirmation request*, then the lock management must be performed by N_n, replying the request as shown in section 3.2. Moreover, if the received message was an *update request*, then the new owner should reply to the message sending the local version of the object. The update message will be detailed in further steps.

   This behavior maintains the consistency because the new owner of an object will be always elected from the set of synchronous replicas of the object. This guarantees that the value for the object maintained in the new owner is exactly the same value the failed owner had.

4. Whenever the original owner node N_f is recovered from the failure, every alive node will be notified by its local membership monitor. Then, further messages sent from the nodes to the owner N_f must not be redirected to N_n, because the node N_f has been recovered now. In addition, the recovering node sends a specific message (*"I am back"*) to the node that managed its owned objects (i.e. the temporally owned N_n. This message will serve to synchronize the activity of both nodes.

   All the alive nodes recognize the recently recovered node by sending a *greeting message*. A *greeting message* sent from a node N_a to the recently recovered node N_f contains a list of locks granted to the node N_a by the temporally owner N_n. Using the contents of these lists, the node N_f can generate the structures needed to manage the locks again. Thus, there becomes unnecessary for the N_n to continue managing these locks.

Step 8 includes a more detailed description of the contents of the *greeting messages*.

5. Nevertheless, a recently recovered node N_f will receive request messages concerning owned objects that may have been updated during the failure period. In order to manage this situation, a recovered node must consider every object held in its local database as an "asynchronous replica". This consideration will be done for an object o_i until either an *update reply* or *access confirmation reply* is received from a synchronous replica of the object. These replies will be received in the situations described in steps 6 and 7.

6. If an *access confirmation request* is received by a recently recovered node N_f, and the involved object has not been already *synchronized* in the node (i.e. the concerning object has not been already updated from a synchronous replica), then N_f must force the synchronization. This synchronization is performed with an *update message* sent to a synchronous replica of the object. The reply to this *update message* will ensure the local database to hold an updated version of the requested object.

   Once the object is updated in the local database, the *access confirmation request* can be processed as described for a standard owner node.

7. The recovered node can also process sessions during the synchronization period. These sessions will access to a set of objects. As we see in step 5, every accessed object must be temporally considered as an *asynchronously maintained* node until the object is synchronized. Note that an object maintained asynchronously in the node does not need to be synchronized.

   The treatment for the objects accessed by a local session will depend on the next classification:

   - Objects with a synchronous maintenance (i.e. either objects owned by the active node, or objects for which the active node is a synchronous replica).
   - Objects for which the active node is an asynchronous replica.

   The treatment for the objects with a synchronous maintenance in the active node will be similar to the recovery of the synchrony described in step 6. When the session asks its local COPLA manager about an object *originally* owned by the node, then an *update request* is sent to a synchronous replica of the object. The reply to this *update message* will ensure the database of the active node to hold an updated version of the requested object, and the response to the session can be completed.

   For the objects maintained asynchronously, the standard treatment can be used, taking into account that the period of outdatedness should include the time the active node was down.

8. Another way for a node N_f to recover the ownership of an object can be found in the *greeting messages* received by N_f from each alive node. These messages were introduced in step 4. When a node N_a sends a *greeting message* to N_f, the message not only contains a list of locks obtained by N_a, but it also contains the last value (and version) for each locked object o_i. This information is enough for N_f to consider synchronized each object o_i.

   In addition, the temporally owner N_n can receive additional locks during the greeting phase (i.e. before N_n receives the *"I am back"* message). In this situation, N_n must process every incoming request until the *"I am back"* message is received, abstaining from sending the *greeting message* to N_f before it occurs. Then, N_n composes its *greeting message*, including the new set of locks, and sends it to the recovered node.

   The recovered node N_f cannot answer requests from any node, until every *greeting message* is received, and the complete list of locked objects can be rebuilt.

9. In order to ensure that a recently-recovered node N_f achieves a correct state for its originally synchronized objects (i.e. the node receives an update message for each object o_j that satisfies $N_f \in S(o_i)$), an asynchronous process becomes necessary to be run.

   This process, will be executed as a low-priority process, and will send an *update request* for each object not already synchronized in N_f.

   Note that the interference of such process in the performance of N_f should be low, because it will only be scheduled during idle periods.

10. The asynchronous process should also include the update, in the local database of the recovered node N_f, of any new object created during the time the node was failed. To perform this update, a simple algorithm may be followed by N_f just at the beginning of its recovery:

    - When N_f recovers from a failure, a query is performed to the local database in order to retrieve the identifier for the more recently inserted object owned by every node in the system. This can be done due to the construction of the object identifiers. As a consequence of these requests, N_f knows, for each node, the last inserted object.
    - Until N_f has received the information from its local database, it will be locked any update in its local database. This ensures that the response of the requests does not include any update performed after the recovery of N_f.
    - In addition, and concurrently with these requests, every node in the system sends the *greeting message* to the recovered node. In this message, explained in 8, additional information can be included. Moreover, a *greeting message* sent from a node N_i to N_f will include the identifier of the more recently object created by the node N_i.

- The comparison of the information contained in the *greeting messages,* with the values collected from the local database, makes N_f know, for each node N_i, the *lost insertions* for each node (i.e. the range of objects inserted during the failure).
- In addition to these object identifiers, the asynchronous process performs further requests to its local database in order to retrieve a complete list of object identifiers owned by each node in the system, and managed in a synchronous way in N_f.
- Objects contained in this list of *synchronous identifiers* will be considered as asynchronously maintained objects, until an update message will be sent to its owner node and, as a response of this request, an up-to-date value is obtained, and it is possible to guarantee in the local database of N_f the "synchronism" of such objects. Then, the identifier can be removed from the list of *synchronous identifier*.

In order to update every object in the local database, the asynchronous process will use the collected information about *lost insertions* to perform *update requests* to each owner node about these objects.

The behavior described in this section can be summarized by the rewriting of the expression for PA presented in equation (6.2), and considering $d(N_f)$ as the set of objects with synchronous management in N_f, but not already synchronized (note that it is satisfied that $\forall o_i \in d(N_f) : N_f \in S(o_i)$):

$$P(o_i) = \begin{cases} 1 & , o_i \in d(N_f) \\ 0 & , o_i \notin d(N_f) \wedge N_f \in S(o_i) \\ PA_{outd}(o_i) & , N_f \in A(o_i) \end{cases} \tag{7.1}$$

The expression includes either the access to already synchronized objects, and the access to synchronous objects that have not already been synchronized.

It can also be used by any surviving nodes N_s, because they are synchronized replicas for all its synchronous objects (i.e. $d(N_s) = \emptyset$).

7.3 The Extended Modification

In the previous section, a basic technique to provide fault tolerance has been described in terms of a modification to the COLU protocol. In this approach, when a node fails, the remaining nodes don't need to perform any specific action.

The implementation of a *replication degree* of T is accomplished with the synchronous replication of the information of each object at least along T nodes.

Let's suppose a system where it has been established a *replication degree* of two. This means that the system will only guarantee the full functionality of the system in presence of less than two failures over the original configuration; i.e. it can occur that subsequent failure of two nodes deal with a system stop.

This undesirable effect can be avoided (or at least attenuated) if the number of synchronous replicas of each object is always maintained over the established *replication degree*. To achieve this, it becomes necessary, whenever this level is decreased (i.e. when a failure of a synchronous replica is detected), to promote the role of a node previously considered asynchronous, making it synchronous for the objects maintained by the failed node in a synchronous way.

The modification of the recovery flavor of COLUP performs the following steps:

- When the system detects a failure of the node N_f, the Membership Monitor of each remaining node in the system notifies the corresponding local Consistency Manager of such failure.
- For the rest of the nodes, one of the nodes N_p acting as an asynchronous replica for the set of objects owned by the node N_f, will be promoted, for this set of objects, to synchronous replica. To this end, a number of actions have to be done in this node:
- The election of the promoting node N_p must be done with a deterministic algorithm. This algorithm is quite simple: It must be always guaranteed the following property.

  "For each object o_i, considering $N_w(o_i)$ as the owner node for o_i, and the replication degree as T, then the set of synchronous replicas for the object is always $S(o_i) = \{N_k | k = w, w+1, \ldots w+T-1\}$, where the operation $+$ only considers the alive nodes".
- To satisfy this property, when a node N_f fails, every node N_i proceeds to recalculate the set $S(o_i)$ for each object owned by the failed node. Three situations can occur at this point:
  - If N_i was synchronous, and it is not the new owner of the objects, then the normal behavior described in section 7.2 is applied.
  - If N_i was synchronous, and becomes the new owner of the object, then additional actions will take place.
  - If N_i was asynchronous, and now it is synchronous, but not the owner, then it should be considered synchronous, but *not synchronized* for o_i. Thus, the node

N_i will follow the same actions as if it was recovering from a failure for the objects o_i.

The depicted technique will promote in a lazy way an asynchronous node N_p to synchronous replica for the set of objects owned by the failed node. Moreover, this new synchronous replica, at the beginning of its promotion, will not be synchronized for each object in this set. Hence, it is possible that a sequence of failures of the new owner nodes $N_{f_2}, \ldots, N_{f_T}$ deal with a situation of lack of synchronous, synchronized replicas of a particular object.

As a consequence, it becomes necessary for the owner node of o_i to maintain the count of synchronous replicas that are currently synchronized with itself. If a node is the unique synchronized replica of an object, and the node fails, no other node will be able to recover the adequate version for this object. Therefore, no other node will be able to promote to owner of this object, and there will only exist a set of nodes synchronous, but *desynchronized* for this object.

When a unique synchronized node for the object is recovered, then it must be again considered, as described above, synchronous for this object. But, in contrast to the common case, it can be considered *synchronized*, because no other node can have been considered owner of this object during the failure, and hence, it has been impossible for any transaction to change the value of the object.

7.4 Conclusions

In this chapter we have presented *Lazy Recovery* as a good approach to achieve fault-recovery in distributed database systems, where there exists an important necessity of performing an update of the replicated information maintained by a re-inclosing node after a failure and further recovery.

In addition, we have presented a lazy recovering protocol that makes use of the principles applied by the COLUP algorithms to recover the full state of a re-inclosed node without the necessity of suspending the activity of the node during such process. Moreover, there is no node in the system suspending its activity during the system-recovery, because this recovery is performed exclusively by the re-inclosed node, and following a lazy paradigm.

The abortion rate, however, is also managed with the statistical conservative techniques on which the COLU protocol is based. This decreases the abortion rate of the transactions initiated in the recovering node, and the recovery algorithm takes profit of the infrastructure used by the replication protocol to propagate the changes.

As a result, the proposed recovery algorithm will not interfere in the functionality of the system, even allowing the recovering node to proceed immediately after its re-inclusion.

Moreover, the performance of the rest of the system will be also unchanged, because the only node making additional work will be the recovering one. On the other hand, this recovering node will suffer this overhead with a lazy policy, making it possible for the local scheduler of the recovering node to proceed with the update of the local replica with a conservative policy, using the idle time of the node to advance part of such updates.

Chapter 8
Conclusions

The current chapter will present contents of this thesis, summarizing the provided contributions, and the possible implications of these contents in future works.

8.1 Contributions of the Thesis

In this thesis we have formally shown the impossibility to conjugate both laziness and serializability in a useable update propagation protocol, and we have formalized a relaxed consistency mode, that makes lazy update protocols usable -although with certain lacks- with an acceptable loss of serializability, keeping the consistency of a distributed database. Then a statistical study of the abortion rate introduced by the use of lazy update protocols has been performed, in order to understand the behavior of the disadvantage that this abortion rate conforms. With this study, a statistical derivation of a lazy update protocol has been proposed, providing a consistency protocol that achieves the performance of a lazy update protocol, and keeps the abortion rate near to the one offered by an eager update protocol. Finally, the algorithm has been completed to provide fault tolerance to the system, by applying a *lazy self-recovery* technique in the protocol.

The following sections describe in more detail the mentioned contributions.

8.1.1 Impracticability of One-Copy Serializability on Lazy Protocols

We have started with a presentation of the problem of serializability from the traditional point of view. To this end, a formalization has been established for concepts such as **transaction**, defined as a sequence of operations over objects contained in the database, and **database**, defined as the history formed with the application of transactions.

Making use of these definitions, some principal formalizations have been also included. The concept of **causal dependency** has been expressed in algebraic terms, making use of sets of objects accessed in different ways by a transaction (readsets and writesets). Then, the **serializability** of a transaction over a database was defined in base to a condition relating the causal dependencies existent between a transaction and the previously applied transactions.

The model has been discussed with examples illustrating the semantic gap produced by the use of queries in real databases. Thus, we evidenced the necessity for the model to be extended with queries in order to provide a complete condition for a transaction to be serializable, and thus, applicable over a database.

Following this discussion, the model was extended to include queries, and the mentioned condition was provided as a comparison between readsets and writesets of the involved transactions. The contribution of such comparison is located in the inclusion, as a part of such sets, of the attributes read by the queries included in the transactions. Thus, to extend the basic model, we reformulated the semantics of the increased range of operations that a transaction can include.

Once the model was extended, a discussion was also included about the practicability of the concurrency control needed to provide serializability guarantees when the complete model is considered. This discussion made use of statistical considerations, and showed that the strict consistency control required to provide one-copy serializability, and based on the extended model, will produce a dramatical increase in the abortion rate of the initiated transactions in the system. Moreover, we showed that this increase is caused by the high number of conflicts produced by the inclusion of the queries in the analysis of serializability.

As an alternative to this high abortion rate, the prevention of such aborts was proposed, and it was also shown that this prevention could be only performed by the propagation, as soon as possible, of the changes made by any transaction in the system.

Finally, we encountered that this “as soon as possible” propagation can be related to other families of propagation protocols such as *eager update propagation protocols*.

As a conclusion, we determined that the inclusion of queries in the consistency control of lazy update systems, are essential in order to guarantee one-copy serializability, but it makes the system unusable, due to the high degree of abortions the strict consistency control will cause.

8.1.2 Analysis of the Stale-Abortion Rate

Despite the obtained impracticability result for a strict one-copy serializability control for lazy update protocols, we formalized the relaxed consistency control, obtained when the queries are not included in the model. The relaxed mode appears when the executed queries are excluded from the checks performed by the consistency control manager.

Moreover, we encountered that this relaxation of serializability may be also useful for a wide range of applications, for which the derived loss of serializability is not dramatical, or can be controlled by the client application itself.

However, existing studies of lazy update protocols showed that they are not recommended for environments with a high degree of consistency conflicts, due to the high level of abortions it introduces. These abortions are mainly produced by the access to stale (i.e. out of date) objects by the transactions. The reasons of such stale accesses point to the asynchrony of the update propagation, that is completed beyond the commit phase in any lazy update protocol.

Thus, a statistical analysis of the abortion rate introduced by lazy update protocols has been performed. To understand the behavior of the abortion rate, we presented an expression for the probability that an accessed object had to be stale, and thus cause the abortion of a transaction.

This expression was adequately validated, with simulated models, and the results showed that the predictions established with the use of the proposed expression had a high degree of accuracy when compared to the real amount of stale accesses.

A mechanism was presented to make use of the expression to reduce the abortion rate. In this model, an executed transaction may perform updates of some of its accessed objects in order to reduce the probability for such accesses to be stale. This update is determined in base of a prediction performed with the discussed expression.

The model included parameters such as cost of network communications, and the system load, in order to bound the improvement achievable with the proposed technique.

The obtained results show a range of values for the environmental parameters where the proposed technique can be considered a practical solution. Thus, for such scenarios, we encountered the technique suitable to be implemented as an alternative to the traditional lazy update protocols, with an improvement in the produced abortion rate.

8.1.3 Improved Update Propagation Protocol: COLUP

Following the discussion, a particular implementation of such technique was presented, using the application of the expression for the stale-abortion rate in order to propose a new approach of update protocol. This new approach, should be able to provide the advantages of both eager and lazy update protocols, while avoiding their disadvantages.

A complete description of the proposed protocol (called Cautious Optimistic Lazy Update Protocol) was presented, and its policies were also discussed. In particular, we showed that the algorithm follows an optimistic approach for the concurrency control.

The basic idea followed by the protocol consists of the interception of each access performed by the transactions. Then, the protocol takes the control, and predicts the probability for such accesses to end as a stale-access. When such prediction indicates that a stale-access is about to occur, then the protocol performs an update request of the suspicious objects before the access is completed. For each object, this request is addressed to a particular system node, acting as the “owner” of such object.

Thus, the probability for a transaction to be aborted due to stale accesses can be dramatically decreased, making it possible for the system to provide abortion rates similar to the ones obtained with eager update protocols.

Moreover, the performance of the system can be kept near to the one provided by a lazy approach, also improving the quality of service of the system.

In addition, we show that the protocol can be configured in order to vary its behavior from a pure eager approach, to a pure lazy update protocol. This versatile feature makes the protocol an adequate alternative to any replication technique, because the use of COLUP will allow a system to apply either eager replication control, and lazy replication control, in addition to the improved COLUP approach.

From Eager to Lazy

As described above, one of the main advantages offered by the COLUP approach can be found in its versatility. This versatility allows a system using the protocol to implement a wide range of replication controls.

On one hand, COLUP can be configured to provide an eager update propagation, distinctive of the synchronous replication techniques. On the other hand, our proposal can be configured to provide the behavior of a pure lazy update protocol, making use of an optimistic, asynchronous replication management.

This is achieved by the protocol with the tuning of two parameters:

- *Size of the set of synchronous replicas* ***(S)*** for each object in the database. When the size is established to 1, for a certain object o_i, the only node that provides guarantees about the adequate value of o_i is its owner node. In contrast, when the size is established to K (i.e. the number of nodes existing in the system), every node will provide these guarantees.
- *Established Threshold* ***(T)*** compared to the evaluated expression for the probability of stale-accesses. Then the evaluation of the expression exceeds the threshold for a

certain object, an update request is performed, and the latest version for the object is requested to be applied in the local node. Thus, higher threshold produces a lower number of update requests.

In order to use COLUP as a pure lazy update propagation protocol, $S = 1$ must be established, and the value established for threshold lacks of relevance.

In contrast, when the COLUP algorithm is used as an eager protocol, it must be configured with $S = K$, and $T = 1$. Thus, there will be no system node (apart from the owner node) updated during the commit phase, and the threshold will be never exceeded, and consequently, there will be no update performed during the life of the transactions (there will occur only after an abort).

In the middle point, the COLU protocol can be configured to provide a certain degree of availability (adjusting the S parameter), while it can be also tuned the degree of *cautiousness* with the T parameter.

Abortion Rate from Eager, Performance from Lazy

We also presented an empirical validation of the improvement achievable by our proposal. To do this, a number of simulated environments were tested, and a number of characteristics of the system were measured.

In particular, a study was shown including two main characteristics: performance and abortion rate. The election of these two characteristics was motivated by the fact that they have proven to be opposed. Thus, it was foreseeable for the system to enforce the one at the cost of worsening the other.

The experiments were repeated for different system loads, only varying the established threshold, and establishing the S parameter to the minimum (i.e. $S = 1$). Our results show that the performance achievable by the COLU protocol with an adequate configuration is very similar to the one obtained with a pure lazy approach. This result validates the prediction done during the description of the versatility of the algorithm.

In addition, we also encountered that the adjustment of an adequate value for T enables the COLU protocol to provide abortion rates near to the ones obtained with eager approaches. This result did suggest us that an adequate configuration of the threshold may produce a low abortion rate while high values for the performance are provided.

Auto-adaptative Tuning

The protocol was then completed with an auto-adaptative technique for tuning the behavior of the protocol to make it flexible to changes in the system characteristics. This technique is based on the automatic adjustment of the threshold, during the execution of the system, in order to maximize the performance, while the abortion rate is kept under reasonable boundaries.

The performance provided by the proposed technique was shown to be very similar to the performance achieved with lazy update protocols, where communication and synchronization between the different nodes of the system is avoided, and the response time is consequently reduced with respect to eager update protocols and primary copy approaches.

In addition, the abortion rate was reduced in a 92% respecting to that produced by the use of a pure lazy update protocol. Thus, our Cautious approach makes it possible to dramatically increase the performance of the system, at the cost of a marginal increase of the abortion rate with respect to eager approaches.

8.1.4 Lazy Fault-Recovery

Performance, Productivity, and Abortion Rate are not the only objectives of our proposal. In addition, another important issue discussed along this thesis has been the necessity of availability and fault-tolerance of the considered client applications.

Thus, fault-tolerance conforms an additional point to be treated here. This has been included as a new extension of the Cautious Optimistic Lazy Update Protocol, to enable lazy fault-recovery in the basic algorithm.

We understand as *lazy fault-recovery* a stabilization process, initiated when a recovered node is re-included in the system, that is completed along the time, by updating the information required by the re-included node with a lazy philosophy.

The technique, in contrast to the self-stabilization techniques, makes it easier to implement a fault-recovery mechanism, because the process is not included as part of the normal behavior. In addition, self-stabilization techniques often introduce overheads during the normal execution of the system.

Moreover, *lazy fault-recovery* allows any system node to proceed without blocking any transaction during the recovery process. This is satisfied even in the re-included node, because the state of such node is updated with lazy policies. This uninterrupted progress of the system provides a clear advantage in contrast to the common specific processes designed to be executed when a re-inclusion arises. Such techniques often need parts of the system to be frozen during the re-inclusion process.

The proposed modification makes it possible for the consistency protocol to re-incorporate recovered nodes in the system while maximizing the availability of the system. This implies that the client applications are allowed to continue performing requests to the system at any time, even during the stabilization times (i.e. during the failure stabilization, and the reconciliation process during the recovery). Thus, the quality of the service offered by the Database system will not be degraded when failures are detected, or during the node recovery.

Finally, as the technique is included as part of the COLU protocol, the abortion rate of the transactions initiated on the re-included node will be also maintained under the suitable boundaries.

8.2 Implications and Future Work

To conclude this thesis, we will present a number of implications of the presented work, and the planned future work related to the studied field will be also mentioned.

One of the main contributions of the thesis has been the justification of the impossibility for a lazy update protocol to provide one-copy serializability guarantees in systems where queries must be executed in a transactional context.

The repercussion of such result in the field of distributed databases should not determine the closure of the researching line, in the field of databases, of lazy update protocols. Nevertheless, lazy protocols defined from now should take into account the impossibility shown in this work for lazy update protocols to be applied to environments where a strict consistency control is not mandatory.

For the COLUP approach, it can be considered as the starting point of a new line in the research of replication control, and even as a consistency control technique. The statistical approach has been exploited in many other fields[All78], and consistency control can also be benefitted from the experience of such contributions. So, COLUP should be considered as one application of these techniques in the field of distributed consistency control, and further uses of such techniques should be also studied.

Finally, *lazy fault-recovery* has been also proposed as an alternative to the fault-tolerance maintenance, in contrast to self-stabilization algorithms and conventional (i.e. synchronous) fault-recovery algorithms.

Asynchrony in fault-recovery is not a new technique for distributed systems in general, but the use of such techniques in transactional systems, or distributed databases has not been exploited.

8.2.1 Future Work

The thesis, as exposed above, can be considered as a starting point in the developing of statistical techniques for replication control. In this line, another approaches can be taken in order to improve the advantages provided by our approach, minimizing network traffic, and improving the abortion rate.

Other, more accurate expressions for the abortion rate can been also studied, in order to increment the capabilities of the system.

On the other hand, the abortions studied in this work only refer to stale-abortions. However, other kinds of abortions can also occur and they are also suitable to be studied in order to propose alternative consistency control techniques, walking on the middle point between optimistic and pessimistic approaches.

Bibliography

[AAES97] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. *Lecture Notes in Computer Science*, 1300:496–503, 1997.

[AH93] Efthymios Anagnostou and Vassos Hadzilacos. Tolerating transient and permanent failures. In *Proceedings of the 7th International Workshop on Distributed Algorithms (WDAG93)*, pages 174–188, 1993.

[All78] A. O. Allen. *Probability, Statistics and Queueing Theory*. Computer Science and Applied Mathematics Series. Academic Press, New York, 1978.

[Alo98] L. Alonso. Optimistic data object replication for mobile computing. In *9th IFIP/IEEE Workshop on Distributed Systems: Operations and Management*, October 1998.

[Avi76] Algirdas Avižienis. Fault-tolerant systems. *IEEE Transactions on Computers*, 25(12):1304–1312, December 1976.

[Bag90] Rajive L. Bagrodia. An integrated approach to the design and performance evaluation of distributed systems. In *Proceedings of the First International Conference on Systems Integration*. IEEE Computer Society, April 1990.

[BB95] Azer Bestavros and Spyridon Braoudakis. Value-cognizant speculative concurrency control. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 122–133. Morgan Kaufmann, 1995.

[BCM87] Rajive L. Bagrodia, K. Many Chandy, and Jayadev Misra. A message-based approach to discrete-event simulation. *IEEE Transactions on Software Engineering*, SE-13(6), June 1987.

[BG95] Kenneth P. Birman and Bradford B. Glade. Reliability through consistency. *IEEE Software*, pages 29–41, May 1995.

[BHG87a] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

[BHG87b] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA, EE.UU., 1987.

[Bir86] K. Birman. ISIS: A system for fault-tolerance in distributed systems. Technical Report TR 86-744, Department of Computer Science, Cornell University, Ithaca, NY, April 1986.

[BJ86] Ken Birman and Thomas Joseph. Low cost management of replicated data in fault-tolerant distributed systems. *ACM Transactions on Computer Systems*, 4(1):54–70, February 1986.

[BJ87] Ken Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *11th ACM Symp. on Operating System Principles*, pages 123–138. ACM SIGOPS, 1987.

[BJ89] Ken Birman and Thomas Joseph. Reliable broadcast protocols. In *Arctic '88*. Addison-Wesley, 1989.

[BK88] Ken Birman and Ken Kane. Causally consistent recovery of partially replicated logs. Technical Report TR 88-949, Department of Computer Science, Indiana University, November 1988.

[BS80] P. A. Bernstein and D. W. Shipman. The correctness of concurrency control mechanisms in a systems for distributed databases — (SDD-1). *ACM Transactions on Database Systems*, 5(2), March 1980.

[BSJ80] Philip A. Bernstein, David W. Shipman, and James B. Rothnie Jr. Concurrency control in a system for distributed databases (sdd-1). *TODS*, 5(1):18–51, 1980.

[BSW79] P.A. Bernstein, D.W. Shipman, and W.S. Wong. Formal aspects in serializability in database concurrency control. *IEEE Transactions on Software Engineering*, 5(5):203–216, May 1979.

[CA82] T. C. K. Chou and J. A. Abraham. Load Balancing in Distributed Systems. *IEEE Transactions on Software Engineering*, SE-8(4), July 1982.

[CBB$^+$00] R.G.G. Cattell, D.K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, editors. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann Publishers, January 2000. 300 pgs., ISBN 1-55860647-5.

[CHT92] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. In Maurice Herlihy, editor, *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing (PODC'92)*, pages 147–158, Vancouver, BC, Canada, 1992. ACM Press.

[CL88] M. J. Carey and M. Livny. Distributed concurrency control performance: A study of algorithms, distribution and replication. Technical report, 758, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, USA, March 1988.

[CM79] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[Cri91a] Flaviu Cristian. Reaching agreement on processor-group membership in synchronous distri buted systems. *Distributed Computing*, 6(4):175–187, 1991.

[Cri91b] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[Dij74] Edsger W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[DMI$^+$03] Hendrik Decker, Francesc Muñoz, Luis Irún, Antonio Calero, Francisco Castro, Javier Esparza, Jordi Bataller, Pablo Galdámez, and Josep Bernabéu. Enhancing the availability of networked database. In *14th International Conference on Database and Expert Systems Applications - DEXA 2003*, Lecture Notes in Computer Science, Prague, Czech Republic, 1-5 September 2003. Springer-Verlag.

[EASC85] A. El-Abbadi, D. Skeen, and F. Cristian. An efficient, fault-tolerant protocol for replicated data management. In *Proceedings 4^{th} SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 215–228, Portland, 1985. ACM.

[Ell77] C. A. Ellis. Consistency and correctness of duplicate database systems. *Operating Systems Review*, 11, 1977.

[FHHR85] F.Cristian, H.Aghili, H.Strong, and R.Dolly. Atomic broadcast: From simple diffusion to byzantine agreement. In IEEE, editor, *15th International Annual Symposium on Fault-Tolerant Computing Systems.*, pages 200–206, 1985.

[FMZ94a] F. Ferrandina, T. Meyer, and R. Zicari. Correctness of lazy database updates for object database systems. In *POS*, pages 284–301, 1994.

[FMZ94b] F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *Proceedings of the Twentieth International Conference on Very Large Databases*, Santiago, Chile, 1994.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, Canada, 1996.

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *Proceedings* 7^{th} *Symposium on Operating System Principles*, pages 150–161, Pacific Grove, 1979. ACM.

[GMB97a] Pablo Galdámez, Francesc D. Muñoz Escoí, and José M. Bernabéu Aubán. HIDRA: Architecture and high availability support. Technical report, DSIC-II/14/97, Depto. de Sistemas Informáticos y Computación, Un iv. Politécnica de Valencia, May 1997.

[GMB97b] Pablo Galdámez, Francesc D. Muñoz Escoí, and José M. Bernabéu Aubán. High availability support in CORBA environments. In F. Plášil and K. G. Jeffery, editors, *24th Seminar on Current Trends in Theory and Practice of Inform atics, Milovy, República Checa*, volume 1338 of *LNCS*, pages 407–414. Springer Verlag, November 1997.

[GMB99] Pablo Galdámez, Francesc D. Muñoz Escoí, and José M. Bernabéu Aubán. Garbage collection for mobile and replicated objects. In J. Pavelka, G. Tel, and M. Bartosek, editors, *26th Seminar on Current Trends in Theory and Practice of Inform atics, Milovy, República Checa*, volume 1725 of *LNCS*, pages 379–386. Springer Verlag, November 1999.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Gra78] J. N. Gray. Notes on database operating systems. In *Operating Systems: An advanced course*, volume 60 of *Lecture Notes in Comp. Sci.*, pages 393–481. Springer-Verlag, 1978.

[Her87] M. Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Transactions on Database Systems*, 12(2):170–194, June 1987.

[Her90] Maurice Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. on Database Sys.*, 15(1):96–124, March 1990.

[HR99] Michel Hurfin and Michel Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4):209–223, 1999.

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, May 1994.

[IB01] Luis Irún-Briz. MIC: An interdomain call mechanism for linux. In *Actas de las IX Jornadas de Concurrencia*, Sitges, Españo, June 2001.

[IBBAME01] L. Irún-Briz, J.M. Bernabéu-Aubán, and F.D. Muñoz-Escoí. HARL: A high available router for linux. In *Proc. of the IEEE-YUFORIC'2001*, Valencia, Spain, November 2001.

[IBBAME02] Luis Irún-Briz, José M. Bernabéu-Aubán, and Francesc D. Muñoz-Escoí. Design and implementation of high availability routing for linux: HARL. In *Anexo de las actas de las X Jornadas de Concurrencia*, Jaca, Españo, June 2002.

[IBMEBA03a] Luis Irún-Briz, Francesc D. Muñoz-Escoí, and Josep M. Bernabéu-Aubán. COLUP: The cautious optimistic lazy update protocol. In *Actas de las XI Jornadas de Concurrencia*, pages 149–162, Benicassim, Españo, June 2003.

[IBMEBA03b] Luis Irún-Briz, Francesc D. Muñoz-Escoí, and Josep M. Bernabéu-Aubán. An improved optimistic and fault-tolerant replication protocol. In *Proceedings of 3rd. Workshop on Databases in Networked Information Systems (DNIS)*, Lecture Notes in Comp. Sci. Springer-Verlag, 2003.

[IF02] ITI and FCUL. Implementation of the scattered data manager. Technical report, D07, GlobData Working Group, IST Programme, project number: IST-1999-20997, May 2002.

[IMDBA03] Luis Irún, Francesc Muñoz, Hendrik Decker, and Josep M. Bernabéu-Aubán. Copla: A platform for eager and lazy replication in networked databases. In *5th Int. Conf. Enterprise Information Systems (ICEIS'03)*, volume 1, pages 273–278, April 2003.

[IUF01] ITI, UPNA, and FFCUL. COPLA programming interface, deliverable 04 (workpackage 01). Technical report, Globdata, IST Programme, project number: IST-1999-20997, June 2001.

[JAJ$^+$02] J.Esparza, A.Calero, J.Bataller, F.Muñoz, H.Decker, and J.Bernabéu. Copla - a middleware for distributed databases. In *3rd Asian Workshop on Programming Languages and Systems (APLAS '02)*, pages 102–113, 2002.

[JM87] S. Jajodia and D. Mutchler. Enhancements to the voting algorithm. In *Proceedings* 13^{th} *VLDB Conference*, pages 399–405, 1987.

[JM90] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Trans. on Database Sys.*, 15(2):230–280, June 1990.

[Jos85] Thomas Joseph. *Low Cost Management of Replicated Data*. PhD thesis, Department of Computer Science, Cornell Unviersity, Ithaca, NY, 1985.

[JP03] R. Jimenez-Peris and M. Patino-Martinez. Towards Robust Optimistic Approaches. In *Future Directions in Distributed Computing*, volume LNCS-2584 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[JPAK01] R. Jimenez-Peris, M. Patino-Martinez, G. Alonso, and B. Kemme. How to Select a Replication Protocol According to Scalability, Availability, and Communication Overhead. In *IEEE Int. Conf. on Reliable Distributed Systems (SRDS'01)*, New Orleans, Louisiana, October 2001. IEEE CS Press.

[JPPMA00] Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Sergio Arévalo. Concurrent transactional replicated servers. In *Proceedings of the 2000 ACM symposium on Applied computing*, pages 655–660. ACM Press, 2000.

[JPPMKA01] R. Jimenez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database. In *Proc. of Int. Conf. on Dependable Systems and Networks, DSN'01 (Fast Abstract)*, 2001.

[KA98] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *International Conference on Distributed Computing Systems*, pages 156–163, 1998.

[KA00] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, 2000.

[KB85] et.al. Ken Birman. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, SE-11(6):502–508, June 1985.

[KB94] N. Krishnakumar and A. J. Bernstein. Bounded ignorance: A technique for increasing concurrency in a replicated system. *ACM Trans. on Database Sys.*, 19(4):586–625, December 1994.

[KBB01] B. Kemme, A. Bartoli, and Ö. Babaoğlu. Online reconfiguration in replicated databases based on group communication. In *Proceedings of the Internationnal Conference on Dependable Systems and Networks (DSN2001)*, Göteborg, Sweden, June 2001.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, pages 558–565, July 1978.

[Lam96] B. W. Lampson. How to build a highly available system using consensus. In Babaoglu and Marzullo, editors, *10th International Workshop on Distributed Algorithms (WDAG 96)*, volume 1151, pages 1–17. Springer-Verlag, Berlin Germany, 1996.

[LFA00] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, Nürnberg, Germany, 2000. IEEE Computer Society Press.

[Lis91] Barbara Liskov. Practical uses of synchronized clocks in distributed systems. In Luigi Logrippo, editor, *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing (PODC'90)*, pages 1–10, Montéal, Québec, Canada, August 1991. ACM Press.

[LLSG92] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. on Comp. Sys.*, 10(4):360–391, November 1992.

[MB97] Francesc D. Muñoz Escoí and José M. Bernabéu Aubán . The NanOS microkernel: A basis for a multicomputer cluster opera ting system. In H. R. Arabnia, editor, *Proc. of the 3rd International Conference on Parallel and Distri buted Processing Techniques and Applications, Las Vegas, Nevada, EE.UU.*, pages 127–135. CSREA, July 1997.

[MEIBG$^+$01] F.D. Muñoz-Escoí, L. Irún-Briz, P. Galdámez, J.M. Bernabéu-Aubán, J. Bataller, and M.C. Bañuls. GlobData: Consistency protocols for replicated databases. In *Proc. of the IEEE-YUFORIC'2001*, pages 97–104, Valencia, Spain, November 2001.

[MGB98] Francesc D. Muñoz Escoí, Pablo Galdámez, and José M. Bernabéu Aubán. ROI: An invocation mechanism for replicated objects. In *Proc. of the 17th IEEE Symposium on Reliable Distributed System s, Purdue Univ., West Lafayette, IN, EE.UU.*, pages 29–35, October 1998.

[MGGB01] Francesc D. Muñoz Escoí, Óscar Gomis Hilario, Pablo Galdámez, and José M. Bernabéu-Aubán. Hmm: A cluster membership service. In Rizos Sakellariou, John Keane, John R. Gurd, and Len Freeman, editors, *Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, Proceedings*, volume 2150 of *Lecture Notes in Computer Science*, pages 773–782. Springer, 2001.

[MIG$^+$02a] Francesc Muñoz, Luis Irún, Pablo Galdámez, José Bernabéu, Jordi Bataller, and Mari-Carmen Bañuls. Flexible management of consistency and availability of networked data replications. *Flexible Query Answering Systems (FQAS '02)*, 2522:289–300, October 2002.

[MIG$^+$02b] Francesc Muñoz, Luis Irún, Pablo Galdámez, Josep Bernabéu, Jordi Bataller, and Mari-Carmen Bañul. Globdata: A platform for supporting multiple consistency modes. *Information Systems and Databases (ISDB'02)*, pages 244–249, 2002.

[MT85] Sape J. Mullender and Andrew S. Tanenbaum. A Distributed File Service Based on Optimistic Concurrency Control. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, Shark Is., WA, 1985.

[Mul88] Sape J. Mullender. Distributed Operating Systems: State-of-the-Art and Future Directions. In R. Speth, editor, *Proceedings of the EUTECO 88 Conference*, pages 57–66, North-Holland, Vienna, Austria, November 1988.

[Mul90] Sape J. Mullender. *Distributed Systems*. ACM Press, 1990.

[Nel90] Victor P. Nelson. Fault-tolerant computing: Fundamental concepts. *IEEE Computer*, 23(7):19–25, July 1990.

[Rah93] E. Rahm. Empirical performance evaluation of concurrency and coherency control protocols for database sharing systems. *ACM Trans. on Database Sys.*, 18(2):333–377, June 1993.

[RMA$^+$02] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicent. The globdata fault-tolerant replicated distributed object database. In *Proceedings of the First Eurasian Conference on Advances in Information and Communication Technology, Teheran, Iran*, October 2002.

[RSB90] Parameswaran Ramanathan, Kang G. Shin, and Ricky W. Butler. Fault-tolerant clock synchronization in distributed systems. *Computer*, 23(10):33–42, October 1990.

[SAS99] Heiko Schuldt, Gustavo Alonso, and Hans-Jörg Schek. Concurrency control and recovery in transactional process management. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*, pages 316–326. ACM Press, 1999.

[Sch81] Gunter Schlageter. Optimistic methods for concurrency control in distributed database systems. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 125–130. IEEE Computer Society, 1981.

[Sch90] F. Schneider. Implementing fault-tolerant services using the state machine appr oach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[SS83] R.D. Schlichting and F.B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. In *ACM Transactions on Computer Systems*, pages 222–238, 1983.

[SSW79] J. Seguin, G. Sergeant, and P. Wilms. A majority consensus algorithm for the consistency of duplicated and distributed information. In *Proceedings* 1^{st} *International Conference Distributed Computing Systems*, pages 617–624. IEEE, 1979.

[Sto79] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering*, SE-5:188–194, May 1979.

[Tho79] R. H. Thomas. A majority consensus approach to concurrency control. *ACM Transactions on Database Systems*, 4:180–209, 1979.

[WJ92] O. Wolfson and S. Jajodia. Distributed algorithms for dynamic replication of data. In *ACM PODS'92, Symposium on Principles of Database Systems*, pages 149–163, 1992.

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