Improving Data Freshness in Replicated Databases

Esther Pacitti

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

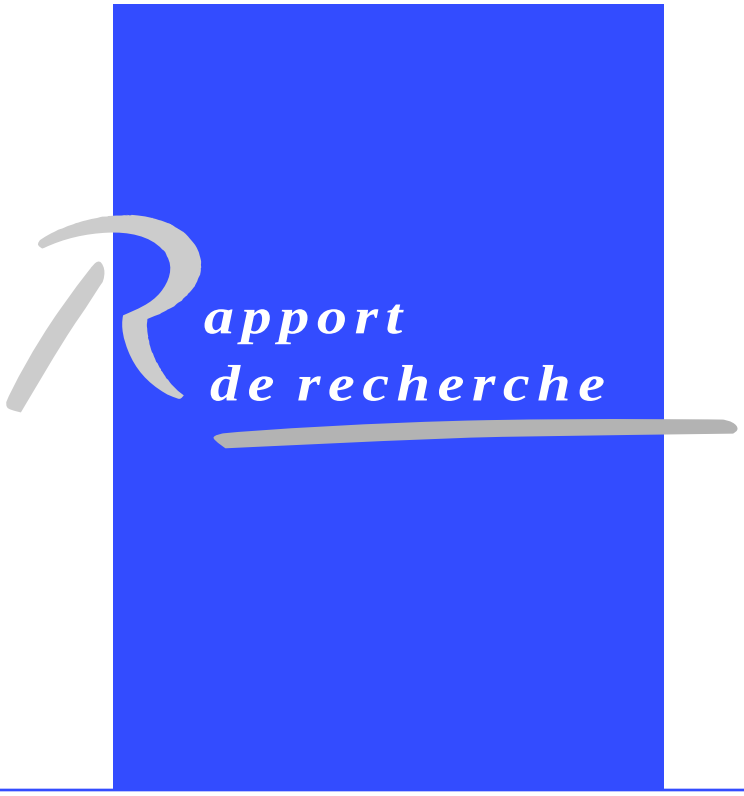
Improving Data Freshness in Replicated Databases

Esther PACITTI

No 3617

Feburary 1999

THÈME 1



ISSN 0249-6399

Improving Data Freshness in Replicated Databases

Esther PACITTI *

Thème 1 — Réseaux et systèmes
Projet Rodin

Rapport de recherche n° 3617 — Feburary 1999 — 114 pages

Abstract: Data replication is often used in distributed database applications to improve data availability and performance. Replicated data must be periodically refreshed using update propagation strategies. Most of the current strategies guarantee mutual consistency of replicated data but are inefficient. Lazy (or asynchronous) replication is an alternative, more efficient solution where mutual consistency is relaxed. It is needed in applications such as on-line financial transactions and telecommunication systems which require high freshness of replicated data. In this case, the concept of *freshness* is used to measure the deviation between replica copies. One focus of this paper is to propose strategies for improving data freshness. Another focus is to specify correctness criterias for different replication configurations, and to propose solutions for correctness enforcement. In this paper, we give a state of the art of replicated databases and we propose a new framework for lazy replication. We introduce two strategies that improve data freshness and an algorithm that enforces correctness for different configurations. Finally, we propose a simulation environment and a performance model to evaluate our strategies. The performance results show that our strategies outperform significantly existing strategies.

Key-words: Distributed Database Systems, Replication, Update Propagation

(Résumé : tsvp)

* Nucleo de Computacao Eletronica of UFRJ (NCE-UFRJ), BRAZIL, e-mail: Esther.Pacitti@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Amélioration de la Fraicheur des Données dans les Bases de Données Répliquées

Résumé : La réplication des données est souvent mise en oeuvre dans les bases de données distribuées pour améliorer la disponibilité des données et les performances. Les données répliquées doivent alors être périodiquement rafraichies en suivant des stratégies de propagation des mises à jour. La plupart des stratégies actuelles garantissent la cohérence mutuelle des données mais sont inefficaces. La réplication paresseuse (ou asynchrone) est une solution alternative, plus efficace qui relache la cohérence mutuelle. Elle est nécessaire dans les applications telles que les transactions financiéres ou les systèmes de télécommunication qui exigent une grande fraicheur des données. Dans ce cas, le concept de fraicheur permet de mesurer la déviation entre les copies répliquées (primaires et secondaires). Un premier objectif de cet article est de proposer des stratégies pour améliorer la fraicheur des données. Un autre objectif est de spécifier les critères d'exactitude pour différentes configurations de réplication, et de proposer des solutions pour la vérification de l'exactitude. Dans cet article, nous donnons un état de l'art des bases de données répliquées et nous proposons un nouveau cadre pour la réplication paresseuse. Nous introduisons deux stratégies pour améliorer la fraicheur des données et un algorithme de vérification de l'exactitude pour différentes configurations de réplication. Enfin, nous proposons un environnement de simulation et un modèle de performances pour évaluer nos stratégies. Les résultats de performances montrent la supériorité de nos stratégies par rapport aux stratégies existantes.

Mots-clé : Bases de données distribuées, Réplication Asynchrone, Propagation de mise á jour

Contents

Chapter 1

Introduction

Data replication is often used in distributed database applications to improve data availability and performance. Replicated data must be periodically refreshed using update propagation strategies. Most of the current strategies guarantee mutual consistency of replicated data but are not efficient. Lazy (or asynchronous) replication is an alternative solution where mutual consistency may be relaxed. It is needed in applications such as on-line financial transactions and telecommunication systems which require high freshness of replicated data. In this case, the concept of *freshness* is used to measure the deviation between replica copies (primary copies and secondary copies). The main focus of this thesis is to improve data freshness.

In lazy replication, replicated data may be placed at the nodes of a distributed system in different ways, thereby producing different configurations. A correctness criteria must be defined for each configuration to establish an order in which replicated data must be refreshed. An objective of this thesis to specify correctness criterias for different configurations, and to propose solutions for correctness enforcement. The background of this thesis is at the crossroads of two complementary, well-established areas of computer science, namely databases and distributed systems.

In this chapter, we give a general overview of this thesis. We first motivate the use of replicated databases and introduce several existing update propagation strategies. Second, we identify the problems with these solutions with respect to the applications we are interested in. Then, we present the objectives of the thesis and its contributions.

This chapter is organized as follows. Section 2 gives an overview of replicated databases. Section 3 presents the most known update propagation strategies and discuss their main problems. Section 4 presents the objectives of the thesis and, Section 5 presents its contributions. Section 6 introduces the thesis organization.

1.1 Replicated Databases

Organizations that use centralized systems are increasingly faced with the problem of sharing common data among large numbers of users. The main problem is to increase the computing power of the systems with the same level of data availability. There are two basic solutions for this problem. The first solution is to upgrade the computing power of the centralized system, by using, for instance, a more powerful machine, thereby making the central processing more efficient. However, a crash of the central system can threaten the entire organizations business. The typical highly-availability solution using redundant hardware and software remains very expensive. Furthermore, if users are placed at nodes remote from the central system, it may take a considerable amount of time to submit a query and see its results due to the round trips of messages necessary to communicate between the user node and the central server. Such delays may be unacceptable for applications where response time is crucial.

The second solution is to replicate data from the central system. In this case, a set of nodes store replicated data and users directly access these nodes instead of the central system. The advantage is that the communication delay to process a query may be greatly reduced because a query may not need to access the central system data. In addition, the crash of the central system may not block the nodes that store replicated data and users may still continue their normal data processing in case of failures.

The new problem now is how to update replicated data. There are several ways to do so. The most primitive, yet often used approach is to manually update replicated data. In this case, the database manager periodically updates replicated data by executing a program that applies the updates performed at the source data. In this solution, replicated data may be not correctly updated because of human errors. In addition, this approach does not scale up to many nodes and is slow at propagating updates. Therefore, automatic solutions (henceforth update propagation strategies) appear as a response to updating replicated data. In this case, data replication is made transparent to the users, that is, updates at the source data are propagated towards the nodes that carry replicated data and are then applied to refresh the replicated data.

There are several relevant points that must be considered when developing an update propagation strategy. The first one is the time at which updates are propagated. It may be immediately after each occurrence or periodically, for instance after every 15 minutes, or as soon as the users request it. Another relevant point is how each node deals with node failures. There are two cases. The first case is the recovery of a node that stores replicated data. The second case is the recovery of the central system node. The final point is concerned with network failures. In this case, a node may be isolated for a long period of time and it is important to find a way to give access to another node that stores replicated data, as an alternative solution.

Once the choice of replicating data is done, another important aspect is the way data are placed across the nodes. A replicated database design can be either *fully replicated* where the entire database is stored at each node, or *partially replicated* where a partition of the database is stored at one or more nodes. In addition, the replication design for a specific

organization may be done using different configurations. For instance, an organization may have only one central node and a set of nodes that replicate its data or alternatively, an organization may host more than one central node. In this case, data is replicated from two different sources.

1.2 Data Replication Problems

One central goal of data replication is to guarantee a high level of performance and data availability. However, performance and availability must be analyzed within a multitude of other interacting factors. These factors include workload characteristics, application requirements and the update propagation strategy employed.

Moreover, the overall performance and availability of a system are affected respectively by the performance and reliability characteristics of its constituent hardware and software subsystems. As such, system characteristics (e.g., individual node speed and reliability, multiprogramming level) and network characteristics (e.g., network topology, communication bandwidth and degree of reliability) must be taken into consideration when assessing the potential costs and benefits of increased replication with respect to performance and availability. These various factors make replicated database design hard.

In this thesis, we are interested in the performance impact of the update propagation strategies. In the last decade, strategies based on the *two-phase-commit* have received much attention because they guarantee nice properties. One is mutual consistency of replicated data. However, *two-phase-commit* has performance problems.

In practice, applications accessing replicated data seldom require mutual consistency. It is often the case that data may remain stale for certain time intervals, or during specific operations without compromising the application semantics. Therefore, other update propagation strategies may be best suited in these cases.

1.3 Existing Solutions

We classify update propagation strategies in five categories. We now introduce each one and identify their applicability.

- With *two-phase-commit* (2PC) [GR93], whenever a transaction updates a replica copy, all other copies of the same source object are updated inside the same transaction. Therefore, all replica copies are updated synchronously, thereby guaranteeing that all replica copies are *mutually consistent*. However, this strategy does not scale up because it requires a round trip message between the node where the update is initiated and every other node having a replica of the same object. Furthermore, 2PC is a blocking protocol in the case of network or node failures. Hence, this strategy is useful for applications with a reduced number of nodes and a high available network system.

- With *Read One Write All* (ROWA) [BG84], a read operation may be executed on an arbitrary copy. A write operation has to be executed on every copy. The underlying transaction manager at each node synchronizes access to copies. The invariant condition of this type of strategy is that all available replica copies are up-to-date and are mutually consistent. This strategy may speed-up read operations but updates transactions as with 2PC. This means that like 2PC, write transactions may not perform well as the number of nodes increases and are not allowed to complete when it is not possible to communicate with a node because of a network failure. Hence, ROWA strategies are well suited for applications that are predominantly read-intensive.

- With *Quorum Consensus* [Tho79], often termed *voting* methods, writes are in general allowed to be recorded only at a subset (a *write quorum*) of the up nodes, as long as reads are made to query a subset (a *read quorum*) that is guaranteed to overlap the write quorum. The invariant condition of this type of strategy is that a write quorum of a replica copy is up-to-date and mutually consistent. Quorum consensus strategies are useful for applications that need to mask failures. However, performance may be compromised when the number of nodes.

- With *Lazy Master* replication, updates performed on a primary copy are first committed at the master node. Afterwards, each secondary copy is updated *asynchronously*, in a separate transaction [GN95, Sta95]. The invariant condition is that the primary copy is always up-to-date. Here, the problem of *not-up-to-date* copies is introduced. However, performance is improved because updates on the primary copy and secondary copies are done in separate transactions. In addition, network and node failures do not block the entire system. Thus, lazy master replication is useful for applications that require higher levels of performance and where a deviation between replica copies is acceptable.

- With *Lazy Group* replication, updates are performed asynchronously on arbitrary identical replica copies. Consequently, copies may become inconsistent. Inconsistency can be automatically detected and sometimes corrected [GHOS96]. No replica copy is guaranteed to be up-to-date and consistent. This strategy presents the performance benefit of lazy master strategies but users may read inconsistent replica copy values.

1.4 Thesis Objectives

This thesis was done in the context of the Data Warehouse Quality (DWQ) European project that focuses on concepts and solutions for improving information quality in data warehouses projects. Data replication is very useful in this context. Several warehouses use replication to consolidate data, in particular when source data change frequently. In this thesis, we focus on lazy master replication schemes for applications that require a high-level of performance. In this section, we give more details on the thesis objectives.

Lazy replication has been widely used by organizations for different types of applications. Its performance improvements and flexibility are recognized. There is a variety of terms that may be used to define a replication design. However, several terms are ambiguous and some others are still missing to make a clear characterization of a replication design. Our first objective is to define a framework that may be used to specify any lazy master design and define its correctness criteria.

The second point we address is the problem of reducing the deviation between a primary copy and its secondary copy due to the time needed to propagate changes performed on primary copies towards secondary copies. Intuitively, we use the term *freshness* to indicate the proportion of updates that are reflected by a given secondary copy but have nevertheless been performed the primary copy. We propose new update propagation strategies that improve freshness and compare them to existing ones. In addition, we provide a functional architecture for lazy master update propagation strategies.

The third point we address is correctness enforcement. Correctness enforcement in lazy master replication is related to the order in which updates on primary copies are applied on secondary copies at each node for different types of configurations. For instance, consider a replication configuration with two master nodes and two slave nodes. Each master stores a primary copy, that are different among themselves, and each slave node stores a secondary copy of each primary copy, one from each master node. Now suppose that each primary copy is updated in a specific chronological order. We focus on the order in which these updates must be applied at each slave node in such a way that users that read the secondary copy values have a consistent view of the world. Consistent here means that, at each node, users should see the same sequence of events.

The fourth point we address is the validation of our solutions. The evaluation of update propagation strategies is a difficult task since several performance factors may be involved (e.g. node speed, network bandwitch and others). Analytical evaluation is typically very complex, hard to understand and hard to thrust. Therefore, we are interested in finding implementation solutions to our solutions and defining a performance evaluation environment to study the performance improvements brought by the update propagation strategies we propose.

1.5 Thesis Contributions

In this thesis, we concentrate on replication strategies that improve freshness in lazy master schemes and correctness enforcement for different configurations. We propose original solutions to these problems with the following contributions:

- a detailed overview of replicated databases showing its most relevant concepts and classifying the most known update propagation strategies [PV98];
- the current applicability of data replication in different domains such as data warehousing [PV98];

- a formal framework for lazy master replication;
- a correctness criteria for different types of configurations;
- a functional architecture for each component node of a lazy master scheme [PSdM98, PS98];
- two algorithms that improve data freshness [PSdM98, PS98];
- an algorithm that enforces correctness for different configurations;
- implementation solutions for our algorithms;
- a simulation environment and a performance model to evaluate our update propagation strategies [PSdM98, PS98];
- results that show the improvements brought by our strategies [PSdM98, PS98].

1.6 Thesis Organization

This thesis contains four main chapters, followed by a general conclusion. Chapter 2, present the state of the art of replicated databases. We recall the concepts of distributed databases with special attention to transactions and failures. Then, we define replicated databases. We also define the concept of a data warehouse since it uses replication as a solution to refresh warehouse data. Next, we cover the fundamental update propagation strategies that are the replication protocols used to manage replicated data. Update propagation is one major issue of replication because it must achieve a good trade-off between freshness of replicated data and cost of replication. We identify and define the principles of six update propagation classes. To illustrate, we provide some examples for each class. We pay special attention to lazy replication which is the basis for this thesis. We also discuss the principles of *broadcast protocols* used to provide fault-tolerance to the update propagation strategies. We then present and compare the replication solutions used by the existing database systems. Furthermore, we present the current trends in data replication such as large scale replication, mobile computing and data warehousing.

In Chapter 3 present a formal framework for lazy master replication. Our framework is defined using five basic parameters: *ownership, configuration, transaction model, propagation* and *refreshment*. The three first ones, ownership, configuration and transaction model, are informally presented in Chapter 2. The last two ones, propagation and refreshment, define when updates must be propagated from a primary copy and applied to a secondary copy, respectively. The configuration parameter is related to the placement of replicated data among the nodes of the distributed system. We consider three basic configurations that correspond to data dissemination, data consolidation and logical partition [Dav94a] since they are most used in practice. In addition, we consider a hybrid configuration that can be built using basic configurations. Finally, for each configuration we define its correctness criteria.

In Chapter 4, we address the problems of freshness improvement and correctness enforcement in lazy master replication schemes. We first present a lazy master system architecture. With respect to freshness improvement, we propose a propagation strategy called *immediate-propagation*. It works as follows: updates to a primary copy at some master node are immediately propagated towards the other secondary copies held by slave nodes without waiting for the commitment of the original update transaction. We propose two update propagation strategies to support immediate-propagation: *immediate-immediate* and *immediate-wait*. With *immediate-immediate*, the transaction that updates a secondary copy (i.e. refresh transaction) is started at a slave node as soon as the first update operation is received from the master node. With *immediate-wait*, a refresh transaction is started at a slave node after the complete reception of all updates (of the same transaction) from the master node. In addition, we also present how the basic algorithm used in several available systems, *deferred-immediate*, fits in our architecture.

In the second part of this chapter, we focus on a solution for correctness enforcement. In our solution, we propose to fix a multicast protocol and to adapt and extend a scheduling algorithm [GM97] to our replication architecture which has the following principle: updates are scheduled for execution using their original update transaction timestamp values. In other words, the first update committed at a master node must be the first committed update at each other node.

In Chapter 5, we propose solutions to implement our algorithms with respect to our architecture. In addition, we propose a simulation environment and a performance model to validate and measure the performance improvements brought by our *immediate-immediate* and *immediate-wait* update propagation strategies compared to the *deferred-immediate* strategy, as well as the scheduling procedures necessary to enforce correctness. Our model is simple, yet taking into account all relevant factors. Our performance evaluation takes into account a data consolidation configuration. We choose this configuration because it is widely used in data warehouse applications. Furthermore, it allows the evaluation of our strategies when the number of master nodes increases (we consider up to 8 master nodes). We show exhaustively the impact of different types of parameters such as the transaction workload at each master node, the transaction sizes, the network delay to propagate a message and abort ratios.

In Chapter 6, we conclude the thesis by summarizing the main results and discuss avenues of future research.

Chapter 2

Replicated Databases

2.1 Introduction

Data replication has become the major solution to the management of data distributed across multiple sites of a computer network. By increasing data availability and query performance, data replication satisfies the stringent requirements of many demanding applications such as banking and on-line financial transactions. Therefore, all the major DBMS vendors now provide a sophisticated replication capability which frees the users from the manual management of replicated data. There are many different ways to perform data replication and there is a broad and deep literature exploring these alternatives. In this chapter, we present an overview of replicated databases focusing on concepts, architectures and techniques.

We first recall the concepts of distributed databases with special attention to transactions and failures. Then, we define replicated databases in the context of distributed databases. We also define the concept of a *data warehouse* since it uses replication as a solution to *refresh* warehouse data. Next, we cover the fundamental *update propagation* strategies that are the replication protocols used to update replicated data. Update propagation is one major issue of replication because it must achieve a good trade-off between *freshness* of replicated data and cost of replication. We identify and define the principles of six *update propagation* classes. To illustrate, we provide some examples for each class. We pay special attention to *Lazy* (or asynchronous) replication which is very efficient at improving freshness. We also discuss the principles of *broadcast protocols* used to provide fault-tolerance to the update propagation strategies. We then present and compare the replication solutions used by the existing database systems. Furthermore, we present the current trends in data replication such as large scale replication, mobile computing and data warehousing.

This chapter is organized as follows. Section 2 presents the main concepts of distributed databases, transaction processing and identifies failures in distributed systems. Section 3 defines replicated databases. Section 4 gives an overview of replication in data warehousing. Section 5 defines and illustrates the update propagation strategies which has been proposed

in the litterature. Section 6 discusses the current research issues in data replication. Finally, Section 7 concludes.

2.2 Distributed Database Systems

Distributed database technology is the basis for data replication. Thus, we first introduce distributed databases and present its major components. We also present the concepts of transaction processing as well as distributed transaction management. These concepts are key for understanding update propagation strategies. We give special attention to failures since their support must be part of each update propagation strategy.

2.2.1 Definitions

A distributed database is a collection of multiple, logically interrelated databases distributed over the nodes of a computer network [OV98]. A distributed database management system (distributed DBMS) is the software system that manages the distributed database system (DDBS) and makes distribution *transparent* to the users. There are several reasons why distributed databases are developped. First, they fit naturally in the structure of a decentralized organization. Distributed databases are the natural solution when several databases already exist in an organization and the necessity of performing global applications arises. Second, if an organization grows by adding new autonomous organizational units, distributed databases support *incremental growth* with the minimum degree of impact on the existing units. Third, when enabling replication, distributed DBMS can yield much higher performance and availability than centralized systems.

Several commercially available distributed database systems have been developed by the vendors of centralized DBMS. They contain additional components which extend the capabilities of centralized DBMS by supporting communication and cooperation between several instances of DBMS which are installed at different nodes of a computer network. The software components which are typically necessary for building a distributed database in this case are: the DBMS component, the data communication component, the data dictionary, and the distributed database component.

The DBMS component implements all database procedures necessary to define and manage local data such as concurrency control and transaction processing. The distributed database component implements all the procedures used to manage remote data and gives access to the local data. Notice that the access to the local data may be done via the DBMS component. The data dictionary stores information about the distributed data in the network using the underlying DBMS. The data dictionary is itself a distributed database containing metadata. As such, it can also be fragmented or replicated across multiple sites. Finally, the data communication component implements the communication primitives such as broadcast and multicast.

An important property of distributed DBMS is whether they are *homogeneous* or *heterogeneous*. The term *homogeneous* distributed DBMS refers to a DBMS with the identical

DBMS at each node, even if computers or operating systems are dissimilar. An heterogeneous distributed DBMS uses instead a least two different DBMSs. Heterogeneous DBMS add the problem of translating between the different data models of the different local DBMS.

2.2.2 Transactions

A transaction is a unit of consistent and reliable computation. Intuitively, a transaction takes a database, performs an action on it, and generates a new version of the database, causing a state transition. This is similar to what a query does, except that if the database was consistent before the execution of the transaction, we can now guarantee that it will be consistent at the end of its execution regardless of the fact that (1) the transaction may have been executed concurrently with others, and (2) failures may have occurred during its execution.

In general, a transaction is considered to be made up of a sequence of *read* and *write* operations on the database, together with computation steps. In that sense, a transaction may be thought of as a program with embedded database access queries [Pap86].

A transaction always terminates, even when there are failures. If the transaction can complete its task successfully, we say that the transaction *commits*. If, on the other hand, a transaction stops without completing its tasks, we say that it *aborts*. Transactions may abort for a number of reasons, which are not discussed here (see [GR93] for a complete presentation). We assume that a transaction aborts itself because of a condition that would prevent it from completing its tasks successfully. Additionally, the DBMS may abort a transaction due to, for example, deadlocks or other conditions. When a transaction is aborted, its execution is stopped and all of its already executed actions are *undone* by returning to the state before their execution. This is also known as *rollback*.

The importance of commit is twofold. The commit command signals to the DBMS that the effects of that transaction should now be reflected in the database, thereby making it visible to other transactions which may access the same data items. Second, the point at which a transaction committed is a *point of no return*. The results of the committed transaction are *permanently* stored in the database and cannot be undone.

Transactions read and write some data. This has been used as the basis for characterizing a transaction. The data items that a transaction reads are said to constitute its *read set (RS)*. Similarly, the data items that a transaction writes are said to constitute its *write set (WS)*. Note that the read set and write sets of a transaction need not be mutually exclusive. Finally, the union of the read set and write sets of a transaction constitutes its *base set* ($BS = RS \cup WS$).

The consistency and reliability aspects of transactions are due to four properties: (1) atomicity, (2) consistency, (3) isolation, and (4) durability. Together, these are commonly referred as the "ACIDity" of transactions. In the following, we discuss each of these properties.

Atomicity refers to the fact that a transaction is treated as a unit of operation. Therefore, either all the transaction's actions are completed, or none of them are. This is also

known as the *all-or-nothing* property. Notice that we have just extended the concept of atomicity from individual operations to the entire transaction. Atomicity requires that if the execution of a transaction is interrupted by any sort of failure, the DBMS will be responsible for determining what to do with the transaction upon recovery from the failure. There are two possible courses of action: it can either be terminated by completing the remaining actions, or it can be terminated by undoing all actions that have already been executed.

A DBMS ensures transaction atomicity by *undoing* the actions of incomplete transactions. In order to do so, the DBMS maintains a *history log* of all writes to the database. A crucial property of the log is that each write action must be recorded in the log (on the disk) *before* the corresponding change is reflected in the database itself. Otherwise, if the system crashes just after making the change in the database but before the change is recorded in the log, the DBMS would be unable to detect and undo this change. This property is called *Write-Ahead Log* [Gra79].

Consistency refers to transaction correctness. In other words, a transaction is a correct program that maps one consistent database state to another. Verifying that transactions are consistent is the concern of semantic data control. Ensuring transaction consistency, on the other hand, is the objective of concurrency control mechanisms, which is briefly discussed below.

A transaction consistency is ensured when its *serial* execution can be considered correct. It establishes the way for defining the correctness of a concurrent failure-prone execution as one that is equivalent to some failure-free serial execution. An execution is serial if, for every pair of transactions, all of the operations of one transaction execute before any of the operations of the other. If the initial state of the database is consistent and if each transaction program is designed to preserve the database consistency if executed in isolation, then the execution is equivalent to a serial one that contains no transaction that observes an inconsistent database.

An execution is *serializable* if it produces the same output and has the same effect on the database as some serial execution of the same transactions. In a distributed database, *one-copy serializability* [BG83] holds when the interleaved execution of transactions is equivalent to a serial execution of those transactions on a centralized (one-copy) database. The theory of *serializability* [BHG87, Pap79] formally defines the requirements to achieve serializable executions. *Concurrency control protocols* [BHG87] are used to restrict concurrent transaction executions in a centralized and distributed database system only to executions that are serializable and one-copy serializable, respectively.

Isolation is the property of transactions which requires each transaction to see a consistent database at all times. In other words, an executing transaction cannot reveal its results to other concurrent transactions before its commitment.

There are a number of reasons for insisting on isolation. One has to do with maintaining the inter-consistency of transactions. If two concurrent transactions access a data item that is being updated by one of them, it is not possible to guarantee that the second will read the correct value. It is obvious that the issue of isolation is directly related to database consistency and is therefore the topic of concurrency control. The *two-phase locking* (2PL)

[EGLT76] concurrency control protocol is based in the principle that no transaction should request a *lock* on a data item after it releases one of its locks. Alternatively, a transaction should not release a lock until it is certain that it will not request another lock. As a consequence, during a transaction execution, all other transactions that request a lock on a data item that is locked, must wait until the lock is released. Therefore, the 2PL is said to be a blocking protocol. On the other hand, a multiversion [AK91, BC92] protocol ensures that a transaction never has to wait to read a database object, and the idea is to maintain several versions of each database object.

Durability refers to that property of transactions which ensures that once a transaction commits, its results are permanent and cannot be erased from the database. Therefore, the DBMS ensures that the results of a transaction will survive subsequent system failures. The durability property brings forth the issue of *database recovery*, that is how to recover the database to a consistent state where all the committed actions are reflected.

Transactions may be identified as belonging to various classes. Even though the fundamental problems are the same for different transaction types, the algorithms and techniques that are used to address them may be considerably different. It might, therefore, be of some value to discuss transactions briefly. It is possible to classify transactions according to several criteria.

The examples we have considered perform some task on stored data. This task is relatively simple: a data is at one location, and the transaction typically updates this data. We call this type of transaction *regular*. If the data is distributed, the management of the transactions becomes more involved and may require special measures. Transactions that operate on distributed data are commonly known as *distributed* transactions.

We now review the architectural transaction model used to process distributed transaction (see Figure 2.1). The distributed execution monitor consists of two modules: a *transaction manager* (TM) and a *scheduler* (SC). The transaction manager is responsible for coordinating the execution of the database operations on behalf of an application. The scheduler, on the other hand, is responsible for the implementation of a specific concurrency control algorithm for synchronizing access to the database.

A third component that participates in the management of a distributed transaction is the local recovery manager that exists at each node. Its function is to implement the local procedures by which the local database can be recovered to a consistent state following a failure.

Each transaction originates at one node, which we will call its *originating node*. The execution of the database operations of a transaction is coordinated by the TM at that transaction's originated node.

The transaction managers implement an interface for application programs which consists of five commands: BEGIN_TRANSACTION, READ, WRITE, COMMIT and ABORT. The processing of each of these commands in a replicated database is discussed below at an abstract level. For simplicity, we ignore the scheduling of concurrent transactions as well as the details of how data is physically retrieved by the data processor. These assumptions permit us to concentrate on the interface of the TM.

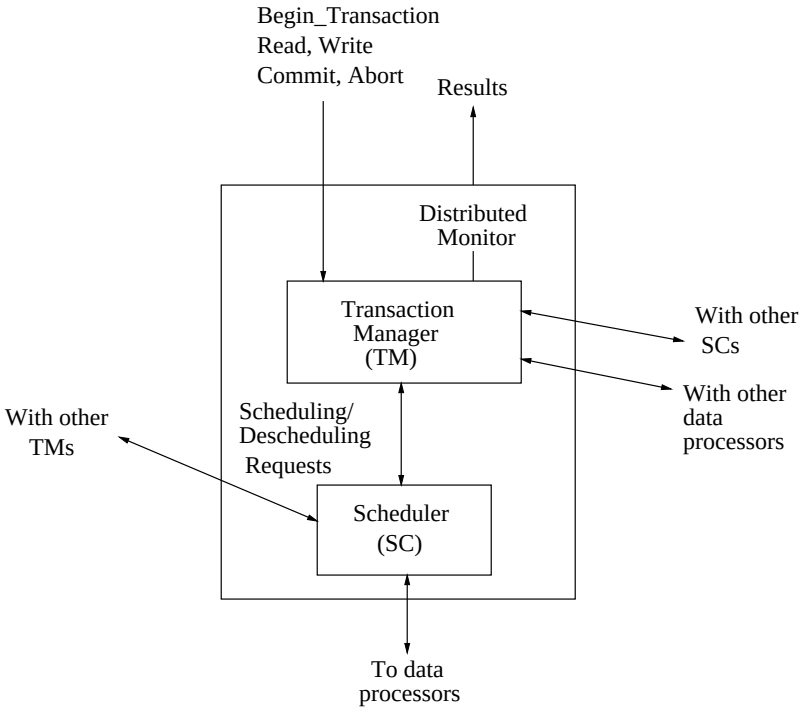


Figure 2.1: Distributed Execution Monitor

1. BEGIN_TRANSACTION. This is an indicator to the TM that a new transaction is starting. The TM does some bookkeeping, such as recording the transaction's name, the originating application, and so on.

2. READ. If the data item x is stored locally, its value is read and returned to the transaction. Otherwise, the TM selects one copy of x and requests its copy to be returned.

3. WRITE. The TM coordinates the updating of x's value at each node where it resides.

4. COMMIT. The TM coordinates the physical updating of all databases that contain copies of each data item for which a previous write was issued.

5. ABORT. The TM makes sure that no effects of the transaction are reflected in the database.

In providing these services, a TM can communicate with SC's and data processors at the same or at different nodes as depicted in Figure 2.1.

2.2.3 Failures

Any deviation of a system from the behavior described in the specification is considered a *failure*. We now identify the types of failures that may happen in a distributed DBMS. We focus mainly in the types of failures introduced by the fact of having a distributed system. Therefore, most types of failures we identify are related to communication failures. We first present the basic procedures to implement message exchange and we identify the various types of failures. Finally, we define synchronous and asynchronous networks since they provide the basis for implementing several recovery solutions.

The component nodes of a distributed DBMS communicate by exchanging messages. This communication approach is referred to as *message-passing* model. A message is a text of variable length. The key characteristics of a message-passing model are the type of communication network, communication failures that may occur, and the synchrony of the system. In the following, we introduce these concepts.

The type of network determines how a node communicates with other nodes [Puj97]. In a *broadcast channel* network, communication takes place over a single shared channel that connects all nodes. In a *point-to-point* network, communication takes place over links that connect pairs of nodes by sending and receiving messages, as described below.

Consider a link from node p to a node q. Associated with this link are two communication primitives, called SEND and RECEIVE. If p invokes SEND with a message m as a parameter, we say that p SENDS m to q. When it returns from that invocation we say that p *completes the sending of* m *to* q. When a node q returns from the execution of RECEIVE with message m as the returned value, we say that q RECEIVES m (from p).

Also associated with the link from p to q are an *outgoing message buffer* at p and an *incoming message buffer* at q. Informally, when p sends a message to q, p inserts m into

its outgoing message buffer, the link transports m to q's incoming message buffer, and m is then removed by q.

If a node or a link commits a failure, it is said to be *faulty* ; otherwise it is *correct*. Below, we list some types of relevant node and link failures [HT94].

Nodes Failures

A node failure may be characterized as follows.

Crash: A node does a *crash* failure if its own execution stops due to a node component failure such as hardware or power supply failure. An important result of this type of failure is that the database system is always assumed to loose the contents of main memory.

Send-omission: A node commits a *send-omission* failure on a message m if it completes the sending procedures of m, but m is not inserted into its outgoing message buffer. An important result of this type of failure is that m is not received at the target node.

Receive-omission: A process commits a *receive-omission* failure on a message m, if m is inserted into its incoming message buffer but it does not receive m. An important result of this type of failure is that m is not received at the target node.

Link Failures

A link l from a node p to a node q performs an *omission* failure on a message m, if m is inserted into p's outgoing buffer but link l does not transport m into q's incoming buffer.

We now present the concepts of synchronous and asynchronous networks. A point-to-point network is *synchronous* if:

- There is a known upper bound on the time required by any node to execute a processing step, for instance a read or write operation.
- Every node has a local clock with known bounded rate of drift with respect to real time.
- There is a known upper bound on message delay; which consists of the time it takes to send, transport and receive a message over any link.

The above properties are necessary for the use of *timeouts* to detect crash failures. In addition, synchronous networks are widely used to support several types of distributed database applications with time constraints, such as real time databases, in which message delay time is used as input parameter to the scheduling of tasks.

On the other hand, a point-to-point network is *asynchronous* if there are *no* timing assumptions whatsoever. In particular, there are no assumptions on the maximum message delay, clock drift, or the time needed to execute a step. This model is attractive and has

recently gained much attention for several reasons. It has simple semantics; applications programmed on the basis of this model are easier to port than those incorporating specific timing assumptions. In practice, variable or unexpected workloads are sources of asynchrony - thus synchrony assumptions are at best probabilistic.

2.3 Replicated Databases

In this section, we define replicated databases and we discuss the trade-offs between cost and availability/performance. We present the different types of replicated data that may be used within a replicated database. We also introduce data warehousing as an example of replicated database. We then present six classes of update propagation strategies, used to update replicated data. For each class, we give some examples of known update propagation strategies. Finally, we discuss the concepts of fault tolerance which are the key to replicated database consistency.

2.3.1 Definitions

Distributed database design addresses the problem of how data must be placed across the nodes of the distributed system. There are two basic alternatives for data placement: *partitioned* (or non replicated) and *replicated* [OV98, CP84]. In the partitioned scheme, the database is divided into a number of disjoint partitions each of which is placed at a different node. Replicated designs can be either *fully replicated* where the entire database is stored at each node, or *partially replicated* where each partition of the database is stored at one or more nodes. *Replicated database* refers to a distributed database application that uses a replicated design as the data placement solution. The resulting replication design is termed a *configuration*. In addition, replicated database is also used as a solution for masking failures and reconfiguring the system in response. As such, replication lies at the heart of any fault-tolerant computer architecture [HHB96].

More formally, a replicated database is composed of a set of replica copies which are stored at the distinct nodes of a distributed DBMS. Each copy is a replica of a specific data source object. For instance, relation R at node i is defined as a data source object and all replicas of R are called replica copies of R.

2.3.2 Assessment

We now present the main trade-offs introduced when using replicated databases. Replicated database have two major advantages [OV98, CP84]:

Improved Performance: Performance may be well improved by bringing the aggregate computing power of all nodes, to bear on a single load category. Reading the values of a replica copy instead of accessing remotely the source object enables as many read operations as there are replicas to be performed in *parallel*. Furthermore, whenever a

node does not store a replica copy it reads, it may read it from another node that stores it, thereby reducing communication cost. While the constraint of read-only appears to be restrictive, a large number of applications falls under this category, and hence can benefit from the performance enhancement.

Improved Reliability/Availability: If data is replicated so that it exists at more than one node, a crash of one of the nodes, or the failure of a communication link making some of these nodes inaccessible, does not necessarily make the data impossible to reach. Furthermore, system crashes or link failures do not cause total system inoperability. Even though some of the data may be inaccessible, the DDBS can still provide limited service.

On the other hand depending on the application and the configuration, replicated databases may, incure some overhead.

Cost: One of the costs of data replication is the duplication effort by having to carry out repetitive work, such as performing every replica copy update multiple times, once at each replica copy of the same source object. This is the largest cost of data replication. It can reduce the total update throughput by a factor equal to the number of replica copies of the same object that must be updated, as compared to a non-replicated system that uses the same amount of resources. Thus, the overhead increases with the number of replica copies.

Complex Design: Performance and availability must be analyzed within a multitude of other interacting factors. These factors include workload characteristics, application requirements and the update propagation strategy employed. Moreover, the overall performance and availability of a system are affected respectively by the performance and reliability characteristics of its constituent hardware and software subsystems. As such, system characteristics (e.g., individual node speed and reliability, multiprogramming level) and network characteristics (e.g., network topology, communication bandwidth and degree of reliability) must be taken into consideration when assessing the potential costs and benefits of increased replication with respect to performance and availability. These various factors make replicated databases design hard.

The decision as to whether replicate or not, and how many copies of any database or object to have, depends to a considerable degree on the user application. Note that replication causes problems in updating databases. Therefore, if user applications are predominantly update oriented, it may not be a good idea to have many copies of the data.

Assuming that data is replicated, the issue related to transparency that needs to be addressed is whether the users should be aware of the existence of copies or whether the system should handle the management of copies and the user should act as if there is a single copy of the data. When the responsibility of specifying that an action needs to be executed on multiple copies is delegated to the user, it makes transaction management simpler for distributed DBMS. On the other hand, doing so inevitably results in the loss of

some flexibility. It is not the system that decides whether or not to have copies and how many copies to have, but the user application. Any change in these decisions because of various considerations affects the user application and therefore reduces data independence considerably. Given these considerations, it is desirable that replication transparency be provided as a standard feature of distributed DBMS.

2.3.3 Replicated Data

In a replicated database configuration, replica copies may be characterized in three different ways [CHKS94], depending on their updating capabilities (also referred as their ownership).

Identical copies: A set of replica copies of a same object have the same rights, properties, or treatment. There is no source object. In this case, all replica copies are read/write and are said primary copies.

Primary/Secondary: One of the replica copies is selected as the primary copy, the other copies are secondary. A primary copy is a read/write replica copy and a secondary copy is read only. The primary copy is either indefinitely decided (fixed primary), or it may transfer its role to another copy (non-fixed primary). A primary copy is stored at a master node and a secondary copy at a slave node.

Snapshot: A snapshot is a view expression defined using primary copies as base relations. Periodically, they may be re-evaluated. Whenever this happens, the changes on each primary copy must be reflected in the snapshot. Snapshots are typically used in data warehouse applications. Similar to secondary copies, snapshots are read only.

2.4 Replication in Data Warehousing

Data warehousing uses replication for refreshing warehouse data. In this section, we introduce the concept of data warehouse its architecture and illustrate the value of replication techniques in that context.

2.4.1 Data Warehousing

One recent trend is the growing use of decision support systems for increasing organizations' competitiveness. Over the years, companies have built *operational* databases to support their day-to-day operations with On-Line Transaction Processing (OLTP) applications. OLTP applications, such as airline reservation or banking, are transaction oriented and update intensive. They need extensive data control and availability, high multiuser throughput and predictable, fast response times. The users are clerical. Operational databases are medium to large (up to several gigabytes). In effect, distributed databases have been used to provide integrated access to multiple operational databases.

Decision support applications have been termed *On-line Analytical Processing (OLAP)* [Cod95] to better reflect their different requirements. OLAP applications, such as trend analysis or forecasting, need to analyze historical, summarized data coming from operational databases. They use complex queries over potentially very large tables and are read intensive. Because of their strategic nature, response time is important. The users are managers or analysts. Performing OLAP queries directly over distributed operational databases raises two problems. First, it hurts the OLTP applications performance by competing for local resources. Second, the overall response time of the OLAP queries can be very poor because large quantities of data need be transferred over the network. Furthermore, most OLAP applications do not need the most current versions of the data and thus do not need direct access to operational data. The now popular solution to this problem is *data warehousing* [Inm96] which extracts and summarizes data from the operational databases in a separate database, dedicated to OLAP. Data warehousing is often considered an alternative to distributed databases, but in fact these are complementary technologies.

Data warehousing refers to a collection of technologies aimed at improving decision making. A data warehouse can be defined as a subject-oriented collection of data integrated from various operational databases. Information is classified by subjects of interest to business analysts, such as customers, products and accounts. Data warehouses are reminiscent of older mainframe-based reporting systems. However they are based on open systems and relational databases. Furthermore, a data warehouse can be directly accessed by end-users on powerful workstations via sophisticated, graphical analysis tools, thus eliminating the need to rely on skilled application programmers.

Warehouse information is historical in nature, reflecting OLTP transactions that have occurred in the past months or years. Thus, to facilitate access and analysis, warehouse data is typically summarized and aggregated in a multi-dimension way. OLAP operations manipulate the data along the multiple dimensions, for instance to increase or decrease the level of aggregation. Warehouse data is generally read-only.

2.4.2 Architecture

Figure 2.2 illustrates a simplified architecture of a data warehouse, with its various elements. One or more source databases, containing operational data updated by OLTP applications, are integrated in a single target database, or data warehouse. The target database is accessed through queries by desktop applications such as query and analysis, reporting and data mining tools. Popular desktop applications for data analysis are spreadsheet programs.

The metadata repository is a separate database that keeps track of the data currently stored in the data warehouse. Typical metadata include descriptions of target tables with their source definitions. The metadata repository is useful to isolate the data warehouse from changes in the schema of source databases. For instance, when a change occurs in a source database schema, the data warehouse administrator can simply update the repository and the change is automatically propagated to the target database and the OLAP applications.

The integration of multiple source databases in a data warehouse raises several issues. At design time, an integrated schema must be defined. Schema integration must deal with

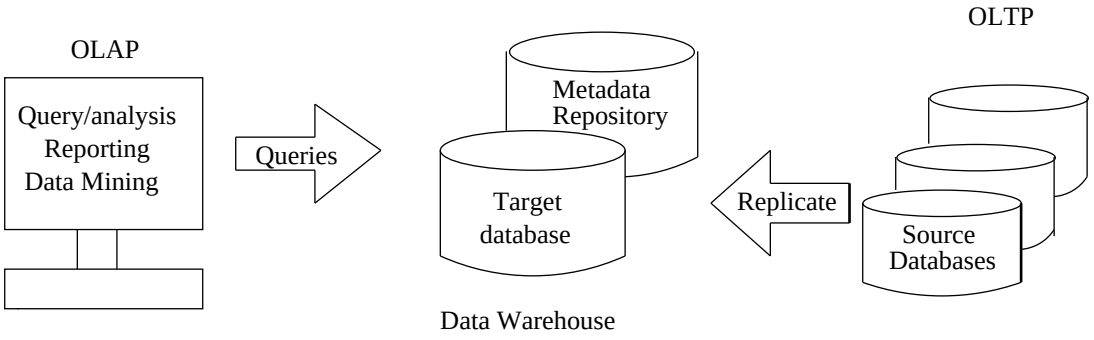


Figure 2.2: Data Warehouse Architecture

semantic conflicts across distinct, autonomous source databases. Schema integration in data warehousing typically assumes that each source database provides a relational interface. Thus schema integration can be done by defining relational views over the source tables [Rou98].

Populating the data warehouse relies on data extraction, cleaning and loading utilities [CD97]. Data extraction is implemented using gateways supporting standard interfaces such as Open Database Connectivity (ODBC) . Data is also usually cleaned to reduce inconsistencies like missing entries or invalid values. After extraction and cleaning, data is loaded in the data warehouse, with additional processing like summarization and aggregation. Loading data essentially means materializing the relational views of the integrated schema. Furthermore, data in the data warehouse is organized for efficient query processing with various kinds of indices.

After it has been populated, the data warehouse must be refreshed to reflect updates to the source databases. Refreshing is usually done periodically, for instance daily or weekly. It can also be done immediately after every update for OLAP applications that need to see current data. To avoid populating entire tables, only updates to the source data should be propagated to the data warehouse. This is done using asynchronous replication techniques that perform incremental maintenance of replicas from primary copies.

2.5 Update Propagation Strategies

Applications accessing replicated data seldom require that all replica copies must be always *mutually consistent* [BG83]. Mutual consistency holds when replica copies are equal among themselves. Replica copies may remain different among themselves for certain time intervals, or during specific operations, without compromising the applications semantics [BGM90a]. Therefore, several update propagation strategies have been developed for updating replicated data without strictly requiring that all copies be atomically and synchronously updated.

Following we present an overview of the most known update propagation strategies. We classify them by understanding their *principle*, *invariant condition* and show some examples based on [CHKS94].

2.5.1 Synchronous All

Principle: Whenever a transaction updates a replica copy all other ones of the same source object are updated inside the same transaction. Therefore, all replica copies are updated synchronously. The invariant condition of this class of strategy is that replica copies are always mutually consistent.

Example: *Two phase commit* (2PC) [Gra79] is a very simple and elegant protocol that ensures the atomic commitment of distributed transactions. It extends the effects of local atomic commit actions to distributed transactions by insisting that all nodes involved in the execution of a distributed transaction agree to commit the transaction before its effects are made permanent. There are number of reasons why such synchronization among nodes is necessary. First, depending on the type of concurrency control algorithm that is used, some schedulers may not be ready to terminate a transaction. For example, if a transaction has read a value of a data item that is updated by another transaction that has not yet committed, the associated scheduler may not want to commit the former. Of course, strict concurrency control algorithms that avoid cascading aborts would not permit the updated value of a data item to be read by other transactions until the updating transaction has terminated [HT88, BHG87].

Another possible reason why a participant may not agree to commit is due to deadlocks that require a participant to abort the transaction. Note that in this case, the participant should be permitted to abort the transaction without being told to do so. This capability is quite important and is called *unilateral abort*.

A brief description of the 2PC protocol that does not consider failures is as follows. The operation of the 2PC protocol (see Figure 2.3) between a coordinator and one participant in the absence of failures, where the circles indicate the states and the dashed lines indicate messages between the coordinator and the participants. The labels on the dashed lines specify the nature of the message. Initially, the coordinator writes a BEGIN-COMMIT record in its log, sends a PREPARE message to all participant nodes, and enters the WAIT state. When a participant receives a PREPARE message, it checks if it could commit the transaction. If so, the participant writes a ready record in the log, sends a VOTE-COMMIT message to the coordinator, and enters the READY state; otherwise, the participant writes an abort record and sends a VOTE-ABORT message to the coordinator. If the decision of the node is to abort, it can forget about that transaction, since an abort decision serves as a veto (i.e., unilateral abort). After the coordinator has received a reply from every participant, it decides whether to commit or to abort the transaction. If even one participant has registered a negative vote, the coordinator has to abort the transaction globally. So it writes an abort record, sends a GLOBAL-ABORT message to all participants nodes, and enters the ABORT state; otherwise, it writes a commit record, sends a GLOBAL-COMMIT message to all participants, and enters

the COMMIT state. The participants either commit or abort the transaction according to the coordinator's instructions and send back an acknowledgment, at which point the coordinator terminates the transaction by writing END-OF-TRANSACTION record in the log.

Note the manner in which the coordinator reaches a global termination decision regarding a transaction. Two rules govern this decision, which, together, are called the *global commit rule*:

1. If even one participant votes to abort the transaction, the coordinator has to reach a global abort decision.
2. If all participants vote to commit the transaction, the coordinator has to reach a global commit decision.

A few important points about the 2PC protocol that can be observed from Figure 2.3. are as follows. First, 2PC permits a participant to unilaterally abort a transaction until it has decided to register the affirmative vote. Second, once a participant votes to commit a transaction, it cannot change its vote. Third, while a participant is in the READY state, it can move either to abort the transaction or to commit it, depending on the nature of the message from the coordinator. Fourth, the global termination decision is taken by the coordinator according to the global commit rule. Finally, the coordinator and participant nodes enter certain states where they have to wait for messages from one another. To guarantee that they can exit from these states and terminate, timers are used. Each process sets its timer when it enters a state, and if the expected message is not received before the timer runs out, the process timeouts and invokes its timeout protocol.

Notice that certain failures of the coordinator or communication failures between it and the participants can cause the 2PC, even with the aid of the cooperative termination protocol, to *block* the transaction until the failure is repaired. Therefore, 2PC is said to be a blocking protocol [BHG87]. In addition, 2PC does not scale up because it requires a round trip of messages between the coordinator and every participant node. Therefore, performance degrades as the number of nodes increases. Finally, 2PC affects node autonomy since all participants must agree on the commitment of a transaction.

2.5.2 Synchronous Available

Principle: Whenever a transaction updates a replica copy, all other available replica copies are updated synchronously. Atomicity is guaranteed by 2PC. However, a write operation can take place even if some replica copies are not available. These replica copies will be updated later, using an asynchronous mechanism. The invariant condition of this protocol is that all available replica copies are up-to-date and are mutually consistent.

Examples: With *Read One Write All* (ROWA) [BG84, AD86, SN79], a read operation may be executed on an arbitrary copy. A write operation has to be executed on every copy. The underlying concurrency controller at each node synchronizes access to copies. On the other hand, with *ROWA Available* [GDC+83], a read operation may be executed on an arbitrary

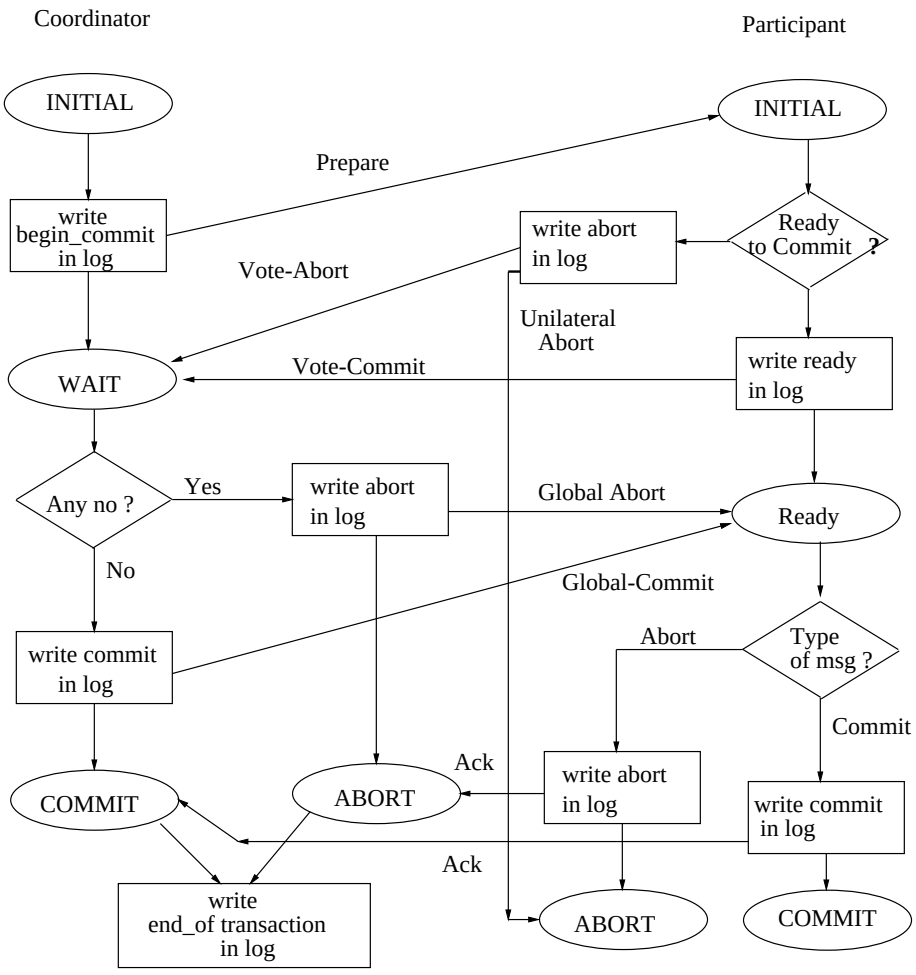


Figure 2.3: The Two Phase Commit Protocol

copy. A write operation does not write all copies of the item: it ignores any object that is unavailable. Thereby the problem of *not-up-to-date* copies is introduced. When a failed copy starts working again, it does not reflect the current database state. Transactions should be prevented from reading copies that have failed and recovered until these copies are brought up-to-date.

2.5.3 Quorum-based

Principle: The ROWA family of protocols implicitly favors read operations by allowing them to proceed with only one copy, while requiring write operations to be carried at all the up nodes. This later condition also means that ROWA algorithms cannot permit write operations to succeed when it is not possible to communicate with a node because of a network failure. These two drawbacks, inflexible favoring of read availability and inability to tolerate communication failures, give rise to the Quorum-based protocols. In this case, updates occur only on a subset of replica copies, which form a so-called quorum. The replica copies which are not in the quorum are updated asynchronously. The invariant condition of this class is that a write quorum of a replica copy is up-to-date and mutually consistent.

Example: With *Quorum Consensus* [Tho79, Gif79], often termed *voting* methods, writes are allowed to be recorded only at a subset (a *write quorum*) of the up nodes, as long as reads are made to query a subset (a *read quorum*) that is guaranteed to overlap the write quorum. For example, if there are 10 replica copies, and 7 copies are written by update transactions, then at least 4 copies must be read. Each copy has a version number, and the copy with the highest version number is current. This technique is not attractive in most situations because reading an object requires reading multiple copies. In most applications, objects are read much more frequently than they are updated, and efficient performance on reads is very important. On the other hand, a great advantage of quorum consensus techniques is that they mask failures, with no need for intervention in order to resume operation after network partitions.

2.5.4 Lazy Master

Principle: Updates that are performed on a primary copy are first committed at the master node. Afterwards each secondary copy is updated *asynchronously*, in a separate transaction [GN95, Sta95]. Primary copies are updatable and secondary copies are read-only. In addition, there is a single primary copy for each secondary copy. The invariant condition of this class is that the primary copy is always up-to-date. Lazy master strategies [GHOS96, Lad90] have been widely implemented by the current available database systems. Before presenting the examples, we define some implementation solutions. For each solution we point out some systems that uses it. Table 2.1 summarizes the most relevant industrial solutions found in [Sta95, Smi95, Bob96, Dav94a, Dav94b, Dav94c].

Whenever the master node is the one that initiates update propagation then the update propagation strategy follows the *push* approach. The push approach makes it possible to per-

Sytems	*Capture*	*Update Propagation*	*Update Any Where ?*
Oracle	Trigger-based	Pull & Push	Yes
Sybase	Log-based	Push	No
Informix	Log-based	Push	No
Ingres	Trigger-based	Push	Yes
IBM	Log-based	Pull	No

Table 2.1: Replication Systems

form event-driven propagation that is discussed below. Sybase Replication Server, Informix and Ingres use the *push* approach. In contrast, when the slave requests update propagation, then it follows the *pull* approach. The pull approach is used by IBM Data Propagator and Oracle. The drawback of the pull approach is to make more difficult to do near-real-time update propagation of updates because a target node does not know when updates have occurred. The main advantage of the pull approach is that it provides scalability, reduces network load by propagating only the last value of data item or aggregated data instead of the whole sequence of updates performed at the master node.

The mechanism used to detect and select changes on a primary copy to propagate to its secondary copies is called *capture*. Capture is implemented in several ways such as follows:

Log Sniffing : The changes on a primary copy are detected by the monitoring of the contents of the history log. The major advantage of monitoring the log is that the capture procedures are completely autonomous to the transaction management and does not impact the performance of the underlying transaction manager. Transactions are processed as usual, and the capture component simply extracts the changes from the log as soon as detected. On the downside, the logging process may have to be modified to handle capture if the useful information for update propagation is not available in the log. IBM Data Propagator, Sybase Replication Server and Informix implement this solution.

Trigger-based: Whenever a primary copy is updated, the trigger engine fires and writes the updated data in corresponding propagation tables stored in the master database. In general, each propagation table is associated with a primary copy and more than one primary copy may be updated within a transaction. After the commitment of the updating transaction, update propagation starts towards the secondary copy nodes. In most implementations, the updates stored on each propagation table are propagated

together in a separate task. Therefore, whenever a transaction updates more than one primary copy, the ACID properties on the secondary copies are not guaranteed because the updates performed on a primary copy inside a single transaction are applied on each secondary copy in several transactions. Using triggers to propagate updates is generally easier and faster to implement than using log sniffing. The DBMS engine itself requires no modifications; propagation simply uses the existing trigger facilities to automatically implement the capture procedures. An interesting aspect of the method is that a condition may be defined to start update propagation (e.g. every hour). Oracle and Ingres are examples of systems that implement this method.

Snapshot: From the industrial point of view, snapshot is just a copy of the primary copy as it existed at some instant in time. In the snapshot approach, the capture procedure is invoked automatically by the master DBMS or the slave application program. In this case, the capture procedure takes a *snapshot* of the primary copy. Again, Oracle offers this capture solution

Propagating SQL Statements: In this case a local server (called a "virtual server") intercepts all SQL statements from the user application before they get to the database server. Read-only statements are passed on to the database server and write statements are sent to the central Broadcast Server. The Broadcast Server simultaneously broadcasts each write statement, in the same order, to all servers in the replicated environment. This method of replication has several advantages. It reduces network traffic by sending a single SQL statement instead of all the data or changed data from source to target. The virtual server provides update-anywhere replication without creating any conflicts since all write transactions go through the Broadcast Server and are applied everywhere in the same order. On the downside, the fact that every database server must be the same limits the flexibility and local autonomy at each node. Requiring all SQL statements to go through a central server creates a potential bottleneck. However, as long as network communication is very efficient because only SQL statements are transmitted instead of data, and the statements are broadcast concurrently across multiple servers, the above limitation may be not always true. Since local updates become remote, transaction performance may also be impacted. Another issue is whether it is realistic to assume that the same SQL statement will always produce the same results in all locations. Afic Technologies' Multi Server Option implements this solution.

Lazy Master replication can be used in different configurations (see Figure 2.4) such as:

Data Dissemination: involves replicating data from a single primary copy to multiple slave nodes. For example, a company might want to maintain a price list in headquarters and distribute it to each branch office every week.

Data Consolidation: consists of propagating and consolidating data from multiple master nodes to a single slave node. For example, a retail chain might have each store replicate its sales transactions to headquarters, where data are consolidated into a snapshot.

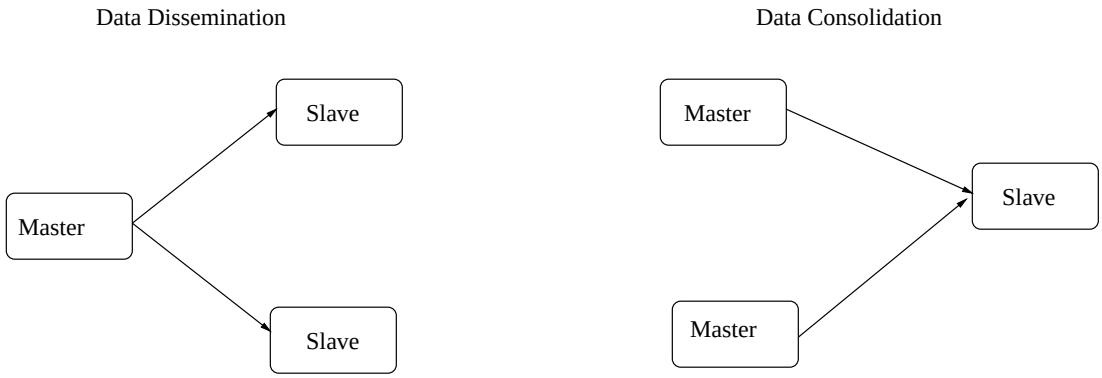


Figure 2.4: Configuration Examples

Examples: We present some lazy master examples. With *As Soon As Possible* (ASAP) write operations are executed on the primary copy. Committed writes are collected subsequently after the commitment of write operations and sent to all replica copies as independent transactions. The *Order-preserving* property holds when transactions are executed at the secondary copy following the same serial order as at the master node.

With *Quasi Copies* [ABGM90], information is controlled at a single central (or master) node, but the methods can also be applied in case of multiple central sites. Coherency conditions associated with a copy define the allowable deviations between an object and the copy. Coherency conditions can be related to time, version, or value. The central node, when operational, has to make sure that each remote node receives a message at least every s seconds. If a node does not receive any message for s seconds it assumes the central node to have failed.

Propagation of updates on the primary copy can be done in different ways :

Last minute : updates on the replicas are delayed up to the point when a coherency condition can be violated ;

Immediately : updates on the primary copy are propagated as soon as they occur ;

Early : updates on the primary copy can be propagated at any time before violation of the conditions ;

Delayed update : the installation of updates at the primary copy is delayed so that no condition is violated. The values are installed when convenient.

The main drawback of Quasi-Copies is that each time the primary copy is updated the coherency condition is evaluated, thereby introducing processing overhead that may degrade the performance of the master node.

Differential File [SL76] is similar to ASAP. However, a differential file is used to record the changes made on the primary copy and this differential file is used to update each secondary copy. The propagation start time depends on the policy, for instance, it can be as soon as possible, on user demand, or periodically.

2.5.5 Primary/backup

Principle: Updates are performed on the primary copy. One of the secondary copies is designated as a *backup copy*. This backup is responsible for recovery on a failure of the primary copy. Other secondary copies are updated asynchronously. The invariant condition of this class is that the primary copy is always up-to-date.

Examples: For instance with a *Remote Duplicate Database Facility* [Com87], the database system is assumed to consist of a primary node and a backup node. At the primary node, undo/redo log entries are written in a master log for every transaction. As this log is written, a copy is sent to a control process at the backup node. When a transaction commits it is assigned a ticket, which determines the order in which transactions must install their updates (i.e. the changes made) at the backup. To commit a transaction at the backup nodes, all the transactions with an earlier ticket must already have been committed. Other examples of Primary/backup strategies are *Remote Backup Procedure* [KHGMP91] and *Disaster protection* [GDC+83] .

2.5.6 Lazy Group

Principle: This class is also referred as *peer-to-peer*. All replica copies are potentially primary copies. Consequently, copies may become mutually inconsistent. Mutual inconsistency can be automatically detected and sometimes corrected [GHOS96]. The invariant condition is that no replica copy is guaranteed to be up-to-date.

Examples: For instance with the *Multi-Airline Reservation* strategy, updates are executed independently on different copies and conflicts have to be resolved later. In practice, several algorithms are used for this. One-copy serializability should be guaranteed by the reconciliation algorithm, which brings the database into a single consistent state.

A particular example of such an algorithm is the demarcation protocol [BGM90b]. It does not treat copies as true replicas anymore, but as independent copies. Local constraints are then formulated on each copy, which ensures that the independent copies can be merged again into a single logical copy later on. For instance, when there are 100 empty seats on an airplane and two independent copies, each copy is allowed to increase the number of seats sold by at most 50. The *demarcation protocol* assumes operations that change the value of a data item to be commutative.

Update Any Where is implemented in several available systems. In this case, copies can exchange information about their updates, thereby integrating update information from

other nodes. Thus, the database may be brought to a fully consistent and up-to-date state. Again it is the reconciliation algorithm that should guarantee one-copy serializability.

Timestamps are commonly used for reconciliation. Each data item carries the timestamp of its most recent update. Each replica update carries the new value and is tagged with the old data item timestamp. Each node detects incoming replica updates that would overwrite earlier committed updates. The node tests if the local replica's timestamp and the updates old timestamp are equal. If so, the update is safe. The local replica's timestamp advances to the new transactions timestamp and the data item value is updated. If the current timestamp of the local replica does not match the old timestamp seen by the root transaction, then the update may be *dangerous*. In such cases, the node rejects the incoming transaction and submits it for *reconciliation* [GHOS96]. One of the reconciliation techniques consists in undoing committed updates and applying them in correct timestamp order. Oracle provides update conflict detection and resolution, including time-based, commutative arithmetic and user-defined methods. In Ingres, the options for resolving conflicts can be the following: the initial update has priority, the latest update has priority, the conflict has user-specified priority, or replication is halted and conflicts are manually resolved.

2.6 Fault Tolerance Protocols

To tolerate failures, update propagation strategies need large amounts of communication to keep the replicas *consistent*. Consistent here means that the principle of the update propagation strategy is correctly enforced. Note that this communication reaches all nodes that store replicas copies in a well-defined order and that agreement exists on which nodes are up and which are down. For such communication an extensive variety of *broadcast* protocols [HT94] have been established during the last decade. *Reliable Broadcast* is the weakest type of broadcast protocol that guarantees three properties:

- all correct nodes agree on the set of messages they deliver,
- all messages broadcast by correct nodes are delivered,
- no *malicious* messages are ever delivered.

While these properties may be sufficient for some applications, Reliable Broadcast imposes no restriction on the *order* in which the messages are delivered. In some applications, this order is important. There is a collection of stronger broadcasts, differing in the guarantees they provide on message delivery order.

Informally, *FIFO Broadcast* is a Reliable Broadcast that guarantees that messages broadcast by the same sender are delivered in the order they were broadcast. *Causal Broadcast*, a strenghtening of FIFO Broadcast, requires that messages be delivered according to the *causal precedence relation*. Roughly speaking, if the broadcast of M causally precedes the broadcast of M', then M must be delivered before M'. If two messages are not causally related, however, different nodes can deliver them in different orders. *Atomic Broadcast* prevents this undesirable behavior by requiring nodes to deliver all messages in the same order.

Finally, *FIFO Atomic Broadcast* combines the requirements of FIFO Broadcast and Atomic Broadcast, *Causal Atomic Broadcast* combines the requirements of Causal Broadcast and Atomic Broadcast.

Protocol designers tend to distinguish between *broadcast* and *multicast*. Broadcast means sending to all nodes on the network; multicast means sending to a selected subset of nodes. In practice, replication strategies use multicast protocols. The designers of such strategies, however, usually reason from a universe that only contains the members of the multicast group, so it makes sense to call them broadcast protocols.

2.7 Current Trends in Data Replication

Replicated databases is a mature research field. However, since it has been widely used in several new types of distributed database applications such as data warehousing, mobile computing and others, several new problems are introduced. In this section, we present the most important challenges in replicated databases related to specific types of application scenarios. We first present some open problems in the data warehouse domain. Next, we identify the challenges of using replication in heterogenous databases and mobile environments. Finally, we describe some of the issues related to replication in systems consisting of large numbers of computing nodes used frequently in electronic commerce applications.

2.7.1 Data Warehousing

Because data warehouse has been a fast growing market for software products and services, practice has been and still is preceding research. All the leading software vendors, in particular relational DBMS vendors, are offering data warehouse systems. However, most of the current products present severe limitations in terms of flexibility, efficiency and scalability. To overcome these limitations, important research issues must be addressed.

In particular, data warehouse management is getting difficult with the deployment of decentralized datamarts as an alternative to the centralized data warehouse approach. For operational reasons, these datamarts are fairly autonomous and tend to grow and duplicate information without global consistency control within the organization. Architectural issues should be addressed to avoid uncontrolled data duplication and yield flexibility and scalability. This requires a better combination of data warehouse and replicated database technologies.

Data integration including data extraction, cleaning, loading and refreshing still presents serious challenges. In addition to schema inconsistencies traditionally studied in heterogeneous data integration, data cleaning should emphasize data inconsistencies. Another problem is *change detection* which detects and propagates the changes in the source data to the data warehouse. [Wid95] classifies the data sources according to their ability for change detection. *Cooperative sources* provide trigger capabilities which ease the programming of automatic notifications of changes. *Logged sources* maintain a log from which changes can be extracted. *Queryable sources* can be queried by the data warehouse. Finally, *snapshot*

sources can only be copied off-line. The WHIPS data warehousing project at Stanford [HGMW+95, LZW+97] addresses the problem of automatic change detection and incremental data integration in the data warehouse.

Another problem related to data integration is improving the freshness of warehouse data. Replication techniques have been used successfully for refreshing the data warehouse periodically [HHB96]. But for some applications, e.g., on-line financial analysis, very high freshness is crucial and traditional replication solutions do not work [GHOS96]. Asynchronous replication techniques that support near real-time constraints [PSdM98] are necessary here.

2.7.2 Replication in Heterogeneous Databases

Fault-tolerance and reliability in heterogeneous database systems (HDBS) is a relatively open area. Methods to achieve reliability through recovery protocols have been discussed in the literature, but they do not address availability. The problems of exploiting redundancy of data and achieving resiliency against node and communication failures have only recently been addressed.

In contrast to explicit replication in a DDBS, with the purpose of achieving fault-tolerance, data redundancy in a HDBS may possibly be a pre-existing fact. Much of the data existing across the various platforms in an enterprise may be redundant, either by replication or by some logical relationship. To make use of this redundant information as a global resource in a cost-effective manner, interdependencies between data objects need be identified, specified and then manipulated from a level that is global to all the disparate systems [WQ90, RSK90, SLE91, DEKB93]. Furthermore, it is desirable to tap this redundancy as a means to increase the overall availability of the enterprise-wide data management system.

2.7.3 Replication in Mobile Environments

Recent advances in hardware technologies such as portable computers and wireless communication networks have led to the emergence of mobile computing systems. Examples of mobile systems that exist today include portable computers that use cellular telephony for communications and trains and airplanes that communicate location data with supervising systems. In the near future, tens of millions of users will carry a portable computer and communicator that uses a wireless connection to access a worldwide information network. The mobile environment is constrained, however, by the limited communication bandwidth, frequent mobile unit disconnection and limited mobile unit battery lifetime, among other factors.

In such a mobile environment, the replication of data and services will be essential [DPS+94, DH95, BI92]. This is because users and their transactions move from one cell to another while accessing the database. Unless the data moves with the user in the form of a cache or a mobile disk storage, data will need to be re-transmitted back and forth. Retransmission wastes the limited communication bandwidth and slows down transaction

processing because of retransmission delays. Moreover, re-transmission/reception results in faster power consumption of the limited battery source.

Mobile computing in general is still a wide open field with many competing control theories and architectures paradigms. Replication control relies on many other functions such as concurrency control, addressing, search, time management, security, etc. Most of these are still unsettled in the mobile environment.

The Bayou storage system [TTP+95] provides an infrastructure for colaborative applications that manages the conflicts introduced by concurrent activity while relying only on the weak connectivity available for mobile computing. To cope with arbitrary network partitions, the system is built around pair-wise client-server and server-server communications. To provide high availability, Bayou employs lazy group replication where clients are able to connect to any available server to perform Reads and Writes. Support for automatic conflict detection and resolution enables applications to deal with concurrent updates effectively. The system guarantees eventual consistency by ensuring that all updates eventually propagate to all servers, that perform updates in a global order, and that any update conflicts are resolved in a consistent manner at all servers.

Complementary to Bayou storage system, [ADN+97] presents Bayou's protocol for lazy group replications. Three basic design decisions went into Bayou's anti-entropy protocol: the model of pair-wise reconciliation between peer replicas, the exchange of write operations stored in per-replica logs that are compactly characterized using version vectors, and the propagation of writes between replicas in an order that is closed with respect to the write's accept, causal or total order.

2.7.4 Replication in Large-scale Systems

There are also issues related to replication in systems consisting of large numbers of computing nodes. An example of a typical application involving large-scale replication is an electronic shopping network. Customers of such a system are able to order merchandise from nodes scattered across the country ; thus the merchandise database is massively distributed. Customer transactions mainly involve read operations, but data updates occur when orders are placed. Data replication must be used to meet the availability requirements of the commercial network. The Internet is another, obvious, example.

Synchronous protocols perform poorly in such large-scale systems. This is partially due to the considerable volume of message traffic that is generated to synchronize the large number of replicas placed on a widely-dispersed network. It is also due to the wide fluctuation in communication delays that are common in wide area networks. In addition, the massive number of components (nodes, communication links and software processes) impose a mode of continuous partial failure on the system.

A few successful scalable replication methods have been introduced in the literature. The common approach taken by these methods is to relax mutual consistency and use lazy master or group update propagation strategies to achieve higher performance and availability. Grapevine [BLNS82] is one of the earlist systems that used relaxed consistency approach. The Coda file system [SKK+90] addresses issues of node scalability.

Related to large-scale systems is the work proposed in [ADN+95] *Severless* file systems distribute file system server responsabilities across large numbers of cooperating machines. This approach eliminates the central server bottleneck inherent in today's file system designs to provide improved performance, scalability, and availability.

2.8 Conclusion

Data replication is a distributed database solution used to improve performance and availability for several types of distributed applications such as data warehousing and on-line financial transactions. Replication techniques also address fault-tolerance activities of masking failures, and reconfiguring the system in response. In this chapter , we defined replicated databases in the context of distributed databases, paying special attention to distributed transactions and failures since they are key for the understanding of update propagation properties.

The contributions of this chapter are the following. First, we gave the architectural framework to define distributed databases and transactions. In addition, we identified the main failures that may happen in a DBMS. Then we defined replicated databases and, in addition, we showed how replicated databases fits in a data warehouse scenario. Next, we identified six classes of update propagation strategies used to update replicated data: Synchronous All, Synchronous Available, Quorum-based, Lazy Master, Primary/backup and Lazy Group. Each class has an invariant condition and is suited for specific types of applications. For instance, the 2PC is suited for applications in which mutual consistency must be assured. We gave special attention on Lazy Master solutions since several terms and concepts are used in the remainder of this thesis. We presented the industrial solutions for data replication. It is clear that most of the available systems use *log based* capture methods to implement update propagation and that the major advantage of the *push* approach is to make possible to achieve *near-real-time* update propagation. Next, we introduced the concept of fault tolerance protocols used to keep replica copies consistent.

Finally, we presented the current trends in data replication showing that it is used as a solution for different types of distributed applications such as data warehousing, mobile computing and large-scale systems.

Chapter 3

Framework

A lazy master replication design defines the replication solutions used to improve the performance of a specific replicated database scenario in which updates performed on a primary copy are first committed at the master node, afterwards each secondary copy of a primary copy is updated *asynchronously*, in a separate transaction using a specific *update propagation* strategy. It typically includes the definition of the data to be replicated, the number of replica copies, the nodes at which replica copies must be placed, the nodes that may update replica copies and others characteristics. A variety of terms that may be used to define a replication design were informally presented in Chapter 2. However, several terms are ambiguous and some others are still missing to make clear the characterization of a replication design. In this chapter, we define a formal framework that may be used for a lazy master replication design that we consider in this thesis.

Our framework is defined using five basic parameters: *ownership, configuration, transaction model, propagation* and *refreshment*. The three first ones, ownership, configuration and transaction model, were informally presented in Chapter 2. The last two ones, propagation and refreshment, define when updates must be propagated from a primary copy and applied to a secondary copy, respectively. The configuration parameter is related to the placement of replicated data among the nodes of the distributed system. We consider three basic configurations that correspond to data dissemination, data consolidation and logical partition [Dav94a] since they are most used in practice. In addition, we consider a hybrid configuration that can be built using basic configurations.

For each configuration, we define a *correctness criteria*. Our correctness criteria are similar to the guarantees provided by fault tolerant protocols. For instance, updates are applied to each secondary copy of a same primary copy in the same order in which they were executed on the primary copy. Finally, we briefly present a motivating example of a replication design using our framework.

The contributions of this chapter are the following:

1. We formally define a framework for lazy replication.

2. We introduce two new replication concepts that are used to define a replication framework: propagation and refreshment.

3. We define three basic correctness criterias that may be used to define the correctness criteria of hybrid configurations.

4. We present a motivating application example using our framework.

This chapter is organized as follows. Section 1 present the five parameters we use to establish a replication scheme. Section 2 introduces the correctness criterias for the configurations we consider. Section 3 shows an application example that is suited to our framework. Finally, Section 4 concludes.

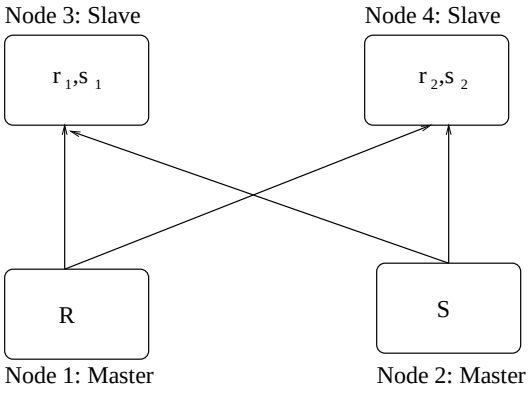


Figure 3.1: An example of a mMaster-nSlave configuration

3.1 Parameters

In this section, we introduce the five parameters that characterize our framework. We focus on lazy master replication, that is, replication designs that uses lazy master update propagation strategies. However, the parameters we present are also valid for lazy group replication designs. We use the term *lazy master replication scheme* to refer to a replication design that fits in our framework. Therefore, whenever the five parameters are set, a replication scheme is established. We do not consider how to choose the data to be replicated since we consider that this involves the knowledge of the semantics of the distributed application.

3.1.1 Ownership

The *ownership* parameter [GHOS96] defines the node capabilities for updating replica copies. If a replica copy is updatable it is called a *primary* copy (denoted by capital letters), otherwise it is called a *secondary* copy (denoted by lower-case letters). Each relation has a single primary copy. For each primary copy, for instance R, there is a set of secondary copies $r_1, r_2, ...$ We refer to the set of primary and secondary copies as *replica copies*. In addition, we sometimes use the term *replicated data* instead of replica copies. We identify three types of nodes: **Master, Slave** and **MasterSlave**. Whenever a node stores only primary copies, it is be referred to as a master node. Similarly, whenever a node stores only secondary copies it is called a slave node. Finally, a node that stores both primary and secondary copies is called a MasterSlave node. We assume that each primary copy of a MasterSlave node is computed using the values of a subset of secondary copies stored at the same node. A node i *is a master* of node j when node i stores a set of primary copies that are replicated at node j. In this case, node j *is a slave* of node i.

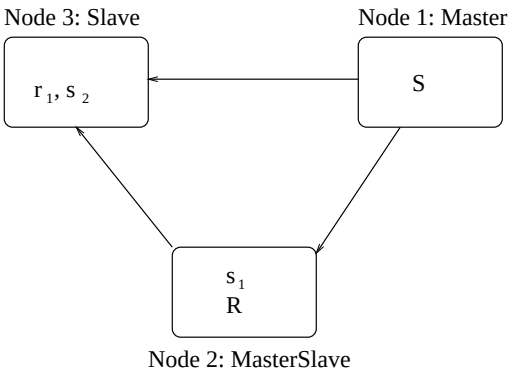


Figure 3.2: An Example of a Master-MasterSlave-Slave Configuration

3.1.2 Configuration

The *configuration* parameter defines the component nodes of a replication scheme. We focus on three basic configurations:

1Master-nSlaves: a replication scheme with a *single* master node, i, and n slaves of i.

mMaster-nSlave: a replication scheme with m master nodes and n slave nodes. Each slave node is a slave for the m master nodes. Figure 3.1 shows an example of a **2Master - 2Slave** configuration in which nodes 1 and 2 are masters that store R and S, respectively. Node 3 stores r_1 and s_1, and node 4 stores r_2 and s_2. Both nodes 3 and 4 are slaves of the master nodes.

Master-MasterSlave-Slave: a replication scheme with three types of nodes: master, MasterSlave and slave. Each MasterSlave node j is slave of all master nodes. Each slave node k is slave for a MasterSlave, j, and all j's master nodes. Figure 3.2 shows an example of a **Master-MasterSlave-Slave** configuration. Node 1 is a master node that stores S, node 2 is a masterslave node that stores R as well as s_1. Note that the values of R are computed using the values of s_1. Node 3 is a slave that stores r_1 and s_2. Therefore, node 3 is slave of both node 1 and node 2.

Hybrid configurations can be obtained by combining the three basic configurations. For instance, a **mMaster-nSlave** and Master-MasterSlave-Slave denotes a configuration that has the capabilities of both mMaster-nSlave and Master-MasterSlave-Slave configurations.

3.1.3 Transaction Model

The *transaction model* defines the properties of the transactions that access the replica copies at each node. In our framework, we fix the properties of the transactions. We focus

on three types of transactions that read or write replica copies: *update transactions*, *refresh transactions* and *queries*.

An update transaction (denoted T) updates a set of primary copies $R, S, ...$ For a Master node, T is a transaction that may read from local non-replica data to update at least one primary copy. In addition, for a masterslave node, T is a transaction that reads at least one secondary copy to update exactly one primary copy. Each time a transaction is committed, a timestamp (denoted C) is generated. For all transactions, we use the subscript to denote a replica copy identifier being read or written and, whenever necessary, the superscript denotes the transaction serial number. In the example shown in Figure 3.2, T_S denotes a transaction that updates S , $T_{s_1,R}$ denotes a transaction that reads the values of s_1 to write R and $T_S^1, T_S^2, ...$ is the serial execution order of the transactions that update S.

A refresh transaction (noted RT) is composed by the serial sequence of write operations performed by an update transaction T used to update (henceforth refresh) at least one secondary copy. RT_{s_1} denotes a refresh transaction that refreshes s_1.

Finally, a query, Q, consists of a sequence of read operations on replica copies. In Figure 3.2, Q_{r_1,s_2} denotes a query that reads the values of r_1 and s_2 in this order, Q_{R,s_1} denotes a query that reads the values of R and s_1 and Q_{s_1} denotes a query that reads only s_1. We assume that once a transaction is submitted for execution to a local transaction manager at a node, all conflicts are handled by the local concurrency control protocol.

3.1.4 Propagation

The propagation parameter defines "when" the updates to a primary copy must be multicast towards the nodes storing its secondary copies.

The multicast of a message is done using the following primitive:

$Multicast(Source, M, \{Target_1, Target_2, ...\})$**:** A *Source* node that may be a Master or MasterSlave node multicasts a message M to the set of *Target* nodes that may be Slave or MasterSlave nodes.

In the example of Figure 3.3, the result of $Multicast(2, M_S, \{3, 4\})$ is the multicast of message M_S to nodes 2 and 3.

We focus on two types of propagation: *deferred* and *immediate*. When using a *deferred* propagation strategy, the serial sequence of writes on a set of primary copies performed by an update transaction T is multicast together within a message M, after the commitment of T. We use the lower index to denote the set of primary copy identifiers related to the sequence of operations carried by the message, and whenever necessary, the upper index denotes the update transaction serial number that is related to the operations carried by the message. For instance M_S denotes a message that carries the operations performed by T_S and $M_S^1, M_S^2...$ denote the messages that carry the serial sequences of operations of $T_S^1, T_S^2...$, respectively.

When using an *immediate* propagation, each write operation performed by a transaction, for instance T_S, is immediately multicast inside a message m_S, without waiting for the

commitment of the original update transaction T_S. We alternatively use o_S to denote the operation carried by m_s. Whenever necessary, we use the first upper index to denote the serial order of the update transaction related to the operation carried by m_S, and the second upper index to denote the serial order of the operations with a transaction. Therefore, $m_S^{1,1}$, $m_S^{1,2}$... (or $o_S^{1,1}$, $o_S^{1,2}$...) denotes the serial sequence of operations performed by T_S^1.

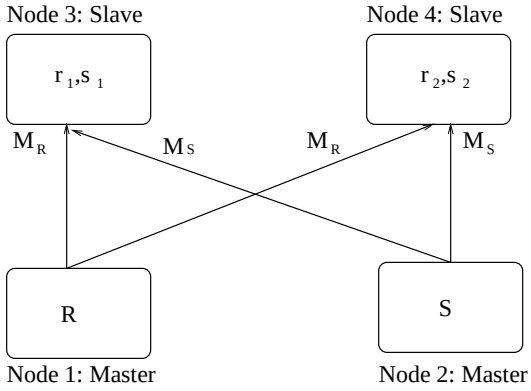


Figure 3.3: mMaster-nSlave configuration with message Multicast

3.1.5 Refreshment

Without loss of generality, the *delivery* of a message (M or m) at node k is interpreted, in our lazy master scheme, as the submission of a refresh transaction to the local transaction manager. Therefore, whenever there is no ambiguity we use message delivery instead of refresh transaction delivery.

Formally, the delivery of a message is done using the following primitive:

Deliver($Secondary, M, Target$)**:** is interpreted as the submission of a refresh transaction (related to M), that updates a $Secondary$ copy, to the local transaction manager of a $Target$ node, that may be a Slave or MasterSlave node.

In the example of Figure 3.3, the result of $deliver(s_1, M_S, 3)$ is the submission of RT_{s_1} to the local transaction manager of node 3.

The *refreshment* parameter is defined by the *triggering* and *ordering* components. The triggering parameter component defines when the delivery of a refresh transaction starts with respect to a propagation strategy. We consider on three trigger modes: *deferred*, *immediate* and *wait*. The couple formed by the propagation and the trigger mode determines a specific update propagation strategy. For instance, with a *deferred-immediate* strategy propagation involves the sending of a RT towards each slave node and, as soon as a RT is received at a

Propagation	*Refreshment (Triggering mode)*	*Update Propagation*
Deferred	Immediate	Deferred-Immediate
Immediate	Immediate	Immediate-Immediate
Immediate	Wait	Immediate-Wait

Table 3.1: Update Propagation Strategies

slave node, it is submitted for execution. With *immediate-immediate*, propagation involves the sending of each write performed by a T towards each slave node, without waiting for the commitment of the original update transaction. At a slave node a refresh transaction is started as soon as the first write operation is received from the master or MasterSlave node. Finally, *immediate-wait* is similar to *immediate-immediate*, however, a refresh transaction is submitted for execution only after the complete reception of all write operations of the original update transaction. Table 3.1.5 shows the update propagation strategies we focus on.

The *ordering* parameter component defines the order of message delivery at each node and defines to the correctness criteria for each configuration. In the the example of the mMaster-nSlave in Figure 3.3, whenever node 3 receives M_R and M_S it must order message delivery following some criteria. Similarly, in the example of the Master-MasterSlave-Slave configuration shown in Figure 3.4 ,whenever node 3 receives M_R and M_S, it must also order them following some ordering criterias.

3.2 Correctness Criteria

We now define the correctness criterias for each configuration presented in Section 3.1.2. Whenever the correctness criteria for a specific configuration is enforced we say that replica copies in that configuration are *consistent*.

3.2.1 1Master-nSlave

In a 1Master-nSlave configuration, the message delivery order at each slave node must follow the same delivery order that would be obtained when using a FIFO reliable multicast protocol. A FIFO reliable multicast protocol requires that all correct nodes deliver the same set of messages and if a node multicasts a message M_S^1 before it multicasts M_S^2 then no correct node delivers M_S^2 unless it has previously delivered M_S^1.

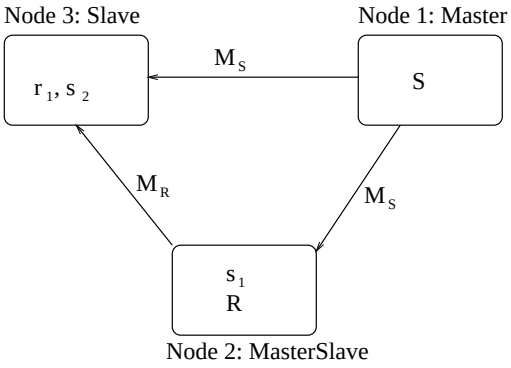


Figure 3.4: Master-MasterSlave-Slave Configuration with message multicast

3.2.2 mMaster-nSlave

In a mMaster-nSlave configuration, the message delivery order at each slave node must follow the same delivery order that would be obtained when using a FIFO reliable total or simply FIFO atomic multicast protocol [HT94] as the underlying protocol to implement an update propagation strategy. With a FIFO Atomic multicast protocol, if two correct nodes l and k both deliver M_R and M_S, then node l delivers M_S before M_R if and only if node k delivers M_S before M_R. If in addition, the delivering order at all nodes follows the messages timestamp order, then the multicast protocol is called FIFO Chronological. [GM97]. Note that the correctness criteria for a mMaster-nSlave configuration includes the correctness criteria for the 1Master-nSlave configuration.

Let us consider the example in Figure 3.3. In this example, there are two Master nodes, node 1 and node 2, that store R and S, respectively. The updates performed on R are multicast within M_R towards nodes 3 and 4. In the same way, the updates performed on S are multicast within M_S towards nodes 3 and 4. With a FIFO reliable protocol, the messages coming from the same master are FIFO ordered. However, there is no ordering among the messages coming from different masters. Therefore, it might happen that M_R and M_S are delivered following this order at node 3 and in reverse order at node 4. With a FIFO atomic multicast protocol, M_R and M_S are delivered in the same order at both nodes 3 and 4.

3.2.3 Master-MasterSlave-Slave

In a Master-MasterSlave-Slave configuration, the message delivery order at each slave node must follow the same delivery order that would be obtained when using a Causal Atomic multicast protocol [HT94] as the underlying protocol to implement an update propagation strategy. A Causal Atomic multicast requires that if the multicast of a message, for instance

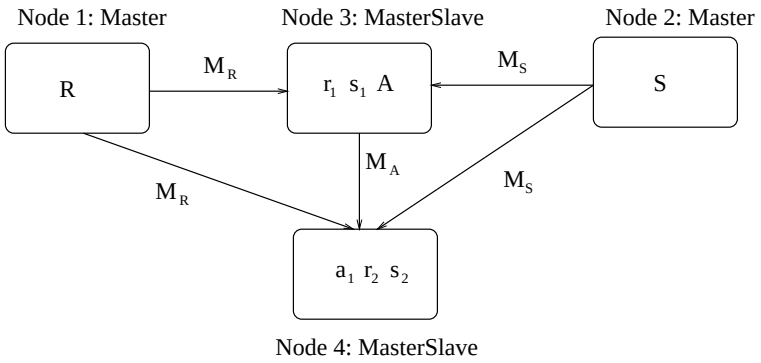


Figure 3.5: Master-MasterSlave-Slave Configuration with message multicast

M_S, causally precedes the multicast of another message, for instance M_R, then no correct node in the group of nodes that constitutes a Master-MasterSlave-Slave delivers M_R unless it has previously delivered M_S. In addition, Total order is also provided. Note that the correctness criteria for a Master-MasterSlave-Slave configuration also includes the correctness criteria for a 1Master-nSlave configuration.

In the first example, of master-MasterSlave-slave configuration in Figure 3.4, S is updated and M_S is multicast towards node 2 and 3. At node 2, R is updated after the refreshment of s_1. The updates on R are multicast towards node 3 inside M_R. Finally, at node 3, r_1 and s_1 are eventually refreshed. With a causal atomic multicast protocol, M_S and M_R are delivered at node 3 following this order.

In the above example, Total order is not required. However, let us consider the second example of a master-MasterSlave-slave configuration that appears in the example of Figure 3.5. In this example, node 1 and node 2 are master nodes of R and S, respectively. Node 3 is a masterslave nodes that stores r_1, s_1 and A. The primary copy A is updated, each time r_1 or s_1 is updated. Node 4 is a slave node that stores a_1, r_2, s_2. At node 1, R is updated and M_R is multicast towards node 3 and 4. At node 2, S is updated and M_S is multicast towards node 3 and 4. At node 3, M_R is delivered and r_1 is updated. Afterwards, A is updated in a separate transaction. Therefore, node 3 propagates M_A towards node 4. In this example, the causal order does not impose any ordering criteria among M_R and M_S at nodes 3 and 4. With Causal Atomic protocol M_R and M_S are delivered in the same order at nodes 3 and 4.

3.2.4 Hybrid Configuration

A Hybrid configuration is composed by at least two of the three basic configurations. A replica copy may be involved in more than one configuration at the same time. However, each component configuration is orthogonal to each other. Therefore, the correctness criteria

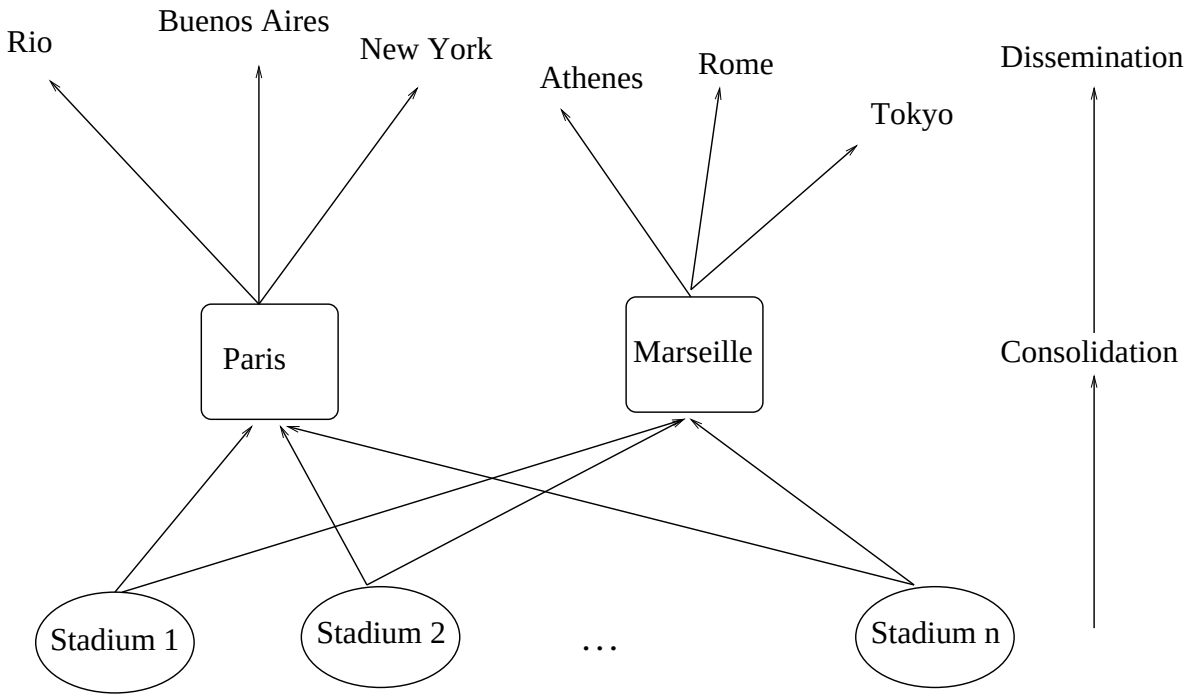


Figure 3.6: An Example of a Replication Design

of each component configuration must be enforced so that each replica copy is consistent with respect to the configurations it may be involved. In the following section we provide an example of a hybrid configuration and discuss its correctness criteria.

3.3 Application Example

The idea is to disseminate on-line relevant information about games in real time to journalists spread all around the world. As a result, interesting statistics and correlations may be generated. In the following, we present the replication scheme implemented.

The objective is to provide access to consistent statistical information as: the number of yellow cards by game, the fastest goal, the score of a game and others. It is important to recall that more than one game may occur at the same time. Lazy master replication was the solution used to handle this distributed application. The replication configuration that was implemented for the World Cup is close to the one shown in Figures 3.6. This configuration is hybrid and combines the properties of both *Data Consolidation* and *Data Dissemination* configurations. The former correspond to our $mMaster-nSlave$ configuration and the later

to the $1Master-nSlave$ configuration. In the following we present how these configurations are combined.

The $mMaster-nSlave$ is composed by a set of ten stadiums placed in different cities such as: Lyon, Toulouse, etc. Each stadium corresponds to a master node. Each master node stores a primary copy that may store several attributes for a specific game as:

$$< game\_id, nscore\_game, yellow\_cards, red\_cards, numb\_attacks, date >$$

where $game\_id$ is the identification of a game, $score\_game$ corresponds to the game score , $yellow\_cards$ is the number of yellow cards, red_cards is the number of red cards, and $numb\_attacks$ is the number of attacks performed by each team and $date$ is the day in which the game happened. Each primary copy is updated periodically by an update transaction. Since the events of a game are unpredictable transactions may vary from short to long ones. Paris and Marseilles are slave nodes. Each slave node stores the set of secondary copies related to each primary copy that is used to consolidate the relevant information queried by the French journalists. An example of a query is:

select $numb\_attacks$ from $r, s...$
where $numb\_attacks \geq 5$ and $date \leq today$

The above query searches for the teams that performed more than five attacks that are playing at the moment the query is processed and that happen in the past. In addition, the average number of yellow cards during the Cup, is an example of a statistic. The solution of using two data consolidation nodes was chosen for two reasons. The first one is to always have available a back-up node in case of failure. The second one is to reduce the propagation overhead at Paris and Marseille.

One important requirement of this application was that the committed updates performed at each master node should be applied in the same order in Paris and Marseilles to produce a *consistent* view of the world cup for all journalists. That is, suppose that the games Brazil x Marroc and England x Argentina happens at the same time. If journalists in Paris knows that Brazil performed a goal and afterwards Argentina performed a goal, then journalists at Marseille should know these two events in the same order. Consistency is provided by ensuring the correctness criteria defined for a mMaster-nSlave is enforced.

The second component of the hybrid configuration is a 1Master-nSlave. The set of secondary copies in Paris and Marseilles is used each one as primary copy. Again, the updates performed in these primary copies must be disseminated towards the most relevant cities in the world as fast as possible. Note that the correctness criteria of the 1Master-nSlave component is orthogonal to the correctness criteria of the mMaster-nSlave component and both correctness criterias must be enforced to enforce consistency for the hybrid configuration.

An important requirement of this application was to consolidate and disseminate data as fast as possible around the world so that international journalist may query *fresh* data. Therefore, the *deferred-immediate*, *immediate-immediate* or *immediate-wait* strategies could be the candidates update propagation strategies used to refresh replicated data in the two components of the hybrid configuration.

There are several applications that ask for *fresh* data and consistency enforcement such a financial applications, warehouses and others. The example we show in this section, yet simple, covers all relevant aspects of the replication framework defined in this Chapter, showing its utility.

3.4 Conclusion

Our replication framework provides the elements necessary to define a replication scheme and analyze its properties. In this chapter, we defined a lazy master replication framework that is composed by five basic parameters: ownership, configuration, transaction model, propagation and refreshment. The ownership specifies the rights of updating replica copies. The transaction model is related to the properties of transaction that manages replica copies. In this thesis, we use a fixed transaction model for simplicity. However, different transaction models may be used. The propagation parameter defines when updates on primary copies must be propagated towards its secondary copies. Furthermore, the refreshment parameter establishes when and in which order updates must be applied to secondary copies. The combination of a propagation strategy and refreshment parameters defines an update propagation strategy.

The configuration parameter describes how replicas copies are placed at the nodes of the distributed system. The choice of a configuration may be very complex. For simplicity, we fixed three basic configurations that were based on the data dissemination, consolidation and logical partitions presented in the previous chapter. For each configuration, we identify its correctness criteria based on the guarantees provided by some fault tolerance protocols. Therefore, each correctness criteria establishes a delivering order criteria that reflects the order in which updates must be applied to refresh replicated data at each node.

Finally, we presented an application example to illustrate the use of our framework and to motivate the research done in this thesis. The application example we presented was inspired in a real example of the World Cup 98 which uses an hybrid configuration that is composed by a $mMaster - nSlave$ and $1Master - nSlave$ basic configurations.

Chapter 4

Update Propagation Strategies

One central goal of replicated database applications is to guarantee a high level of performance. In our replication example presented in Chapter 3, it is crucial to have fast access to replicated data from any journalist location. In addition, the relevant events that happen at each stadium (master node), recorded within update transactions, must be propagated as fast as possible to all other locations to refresh secondary copies. We can mention several other applications examples such as telecommunication systems [GHK+97], data warehouses [Inm96] and on-line financial transactions [Sha97] where performance is critical. Recall that the *two-phase-commit* protocol (henceforth 2PC) is known not to perform well since it requires several round trip messages between the node where the update is initiated and every other node having a replica of the same data. Therefore, performance degrades as the number of nodes increases. Furthermore, 2PC is a blocking protocol in the case of network or node failures [BHG87].

With a *deferred-immediate* strategy, which is a lazy master update propagation strategy, each time a transaction updates a primary copy at some master node, all the updates are propagated towards the secondary copies of the same object, and each secondary copy is updated in a separate refresh transaction. As a consequence, this scheme looses the mutual consistency property assured by 2PC. Furthermore, the interval of time between the execution of the original update transaction and the corresponding refresh transactions may be large due to the time needed to propagate and execute the refresh transactions. The degree of *freshness* indicates the proportion of updates that are reflected by a given secondary copy but have nevertheless been performed on the primary copy.

In this chapter, we address the problems of freshness improvement and correctness enforcement in lazy master replication schemes. We first present a lazy master system architecture. With respect to freshness improvement, we propose a propagation strategy, briefly introduced in Chapter 3 called *immediate-propagation*. It works as follows: updates to a primary copy at some master node are immediately propagated towards the other secondary copies held by slave nodes without waiting for the commitment of the original update transaction. We propose two update propagation strategies to support immediate-propagation:

immediate-immediate and *immediate-wait*. With *immediate-immediate*, a refresh transaction is started at a slave node as soon as the first update operation is received from the master node. With *immediate-wait*, a refresh transaction is started at a slave node after the complete reception of all updates (of the same transaction) from the master node. In addition, we also present the algorithm that implements the *deferred-immediate* for our architecture.

In the second part of this chapter, we focus on correctness enforcement. Correctness enforcement is closely related to message delivery. In our framework, message delivery is equivalent to refresh transaction execution. In a given lazy master configuration, correctness may be enforced by using the underlying multicast protocol defined to guarantee a specific correctness criteria. For instance, in a 1Master-nSlave configuration, correctness is enforced when the message delivery order at each slave node follows the same delivery order that would be obtained using a FIFO reliable multicast protocol. With a *deferred-immediate* strategy this means that refresh transactions execution order in a slave node follows the execution order of their original update transactions at the master node.

Now, let us consider that this protocol is used to enforce correctness for the 1Master-nSlave configuration and the replication scheme evolves to a mMaster-nSlave configuration. In this case, correctness is enforced when the message delivery order at each slave node follows the same delivery order that would be obtained using a FIFO reliable total or simply FIFO atomic multicast protocol [HT94] instead of only the FIFO reliable protocol. With a *deferred-immediate* strategy, this means that at each slave node refresh transactions must be executed in the same order. In this case, a solution for enforcing correctness would be to up-grade the underlying multicast protocol to a FIFO atomic protocol. However, this type of solution is not generic and requires adjusting the multicast protocol whenever the configuration changes. As a consequence, the replication design looses flexibility.

In our solution, we propose to fix a reliable FIFO multicast protocol and to adapt and extend a scheduling algorithm [GM97] to our replication architecture which has the following principle: refresh transactions are scheduled for execution using their original update transaction timestamp values. In other words, the first update transaction committed at a node must have a corresponding refresh transaction that is first executed at a master or masterslave node.

We show that this single algorithm enforces correctness for all configurations we consider (1Master-nSlave, mMaster-nSlave, Master-MasterSlave-Slave and Hybrids). In our framework, we use the term *refresher* to refer to the scheduling algorithm. We give three variants of the refresher algorithm. The first one uses the *deferred-immediate* update propagation strategy. The second and third variants use *immediate-immediate* and *immediate-wait*, respectively. Finally, we discuss the main procedures used for node recovery.

The main contributions of this chapter are the following:

1. We propose a lazy master replication architecture.

2. We show how *deferred-immediate* update propagation is implemented in our architecture.

3. We propose a solution for freshness improvement by introducing two new update propagation strategies: *immediate-immediate* and *immediate-wait*.

4. We propose a solution for correctness enforcement through the refresher algorithm and its three variants.

5. We present the recovery procedures for the update propagation strategies we consider in this chapter.

The remainder of this chapter is structured as follows. Section 1 presents the general node system architecture used by our update propagation strategies. Section 2 describes update propagation strategies. Section 3 presents the refresher algorithm and its three variants. Section 4 describes node recovery. In section 5 we compare our solution with related work. Finally, Section 6 concludes.

4.1 System Architecture

The underlying idea of our architecture is to maintain the autonomy of each node. This means that neither the local transaction management protocols nor query processing are changed to support a lazy master replication scheme. Each node, whether master or slave, is equipped with three components, in addition to the database system. Figure 4.1 shows the architecture of a node. The first component is the Replication Module, which itself consists of three components: *Log Monitor, Propagator and Receiver*. The second component is the *Refresher*, which provides different qualities of service by implementing different refreshment strategies. The third component is the *Deliverer*, which manages the submission of refresh transaction to the local transaction manager. The last component, the *Network Interface*, is used to propagate and receive messages. For simplicity, it does not appear in Figure 4.1 and is not discussed in this thesis. We detail the functionality of the main modules:

Log Monitor: It implements *log sniffing* [SKS86, KR87, Moi96], which is a procedure used to extract the changes to a primary copy by continuously reading the content of a local History Log (noted H). We do not consider any particular commercial log file format in our study. However, we safely assume (see Chap. 9 of [GR93]) that a log record contains all the information we need such as $timestamp, primary\_id$, and other relevant attributes that will be presented in the next section. When the log monitor finds a write operation on R, it reads the corresponding log record from H and writes it into a stable storage called *Input Log* that is used by the Propagator. We do not deal with conflicts between the write operations on the History Log and the read operations performed by the Log Monitor since this procedure is well known and available in commercial systems [Moi96].

Receiver: It implements message reception at the slave node. Messages coming from different masters are received through a network interface that stores them in a *Reception Log*. The receiver reads messages from the *Reception Log* and stores them in pending queues. The contents of these queues form the input to the *Refresher* . A slave node detects its master node failure using *time-out* procedures implemented in the network interface. As soon as a failure is detected by the receiver, it signals it to the Refresher. At a slave recovery, the receiver uses the *reception log* to check out the last received message. Afterwards, the receiver signals its recovery to its master node using its propagator.

Propagator: It implements the propagation of messages that carry log records issued by the Log Monitor or reception of recovery signals issued by the Receiver. Both types of messages are written in the *Input Log*. The propagator continuously reads the records of the *Input Log* and propagates messages through the network interface. A master node is able to detect failures of its slave nodes using the network interface. As soon as a failure is detected, the propagator deactivates propagation towards the failed node. Propagation is only reactivated when the master receives the recovery signal coming

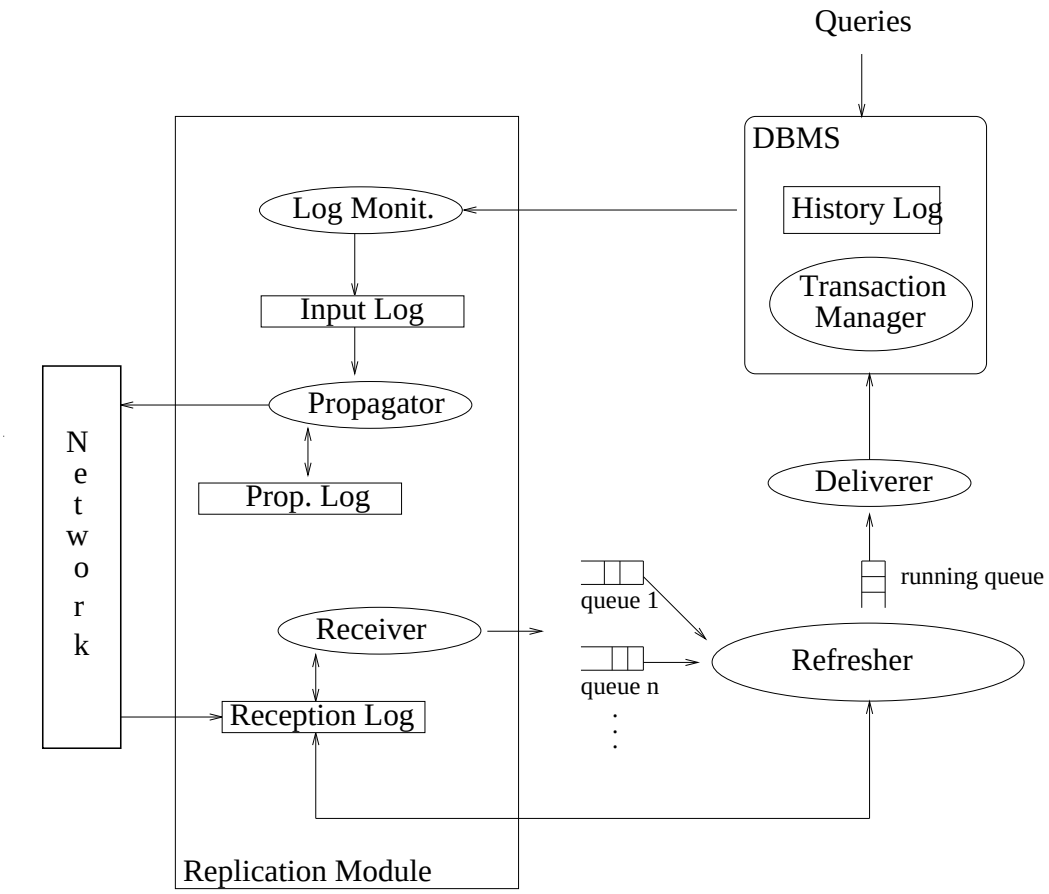


Figure 4.1: Architecture of a node: square boxes represent persistent data repositories and oval boxes represent system components.

from the failed node. During a master recovery, the propagator uses the *propagation log* to restart propagation.

Refresher: It implements refreshment procedures related to refresh transaction delivery. It reads the contents of the pending queues, one for each master node, and based on some update propagation strategy, it orders refresh transactions using a *running queue*. The running queue contains all ordered refresh transactions not entirely executed.

Deliverer: It implements message delivery at a Slave or MasterSlave node through the submission of refresh transactions. A message M is delivered whenever the effects of the related refresh transaction are made visible (or *committed*) at the Slave or MasterSlave node. The Deliverer reads the contents of the running queue in FIFO order and submits each write operation to the *local transaction manager* to update at least one secondary copy.

4.2 Update Propagation Strategies

Recall that the *propagation* parameter defines "when" the updates to a primary copy must be multicast towards the nodes storing its secondary copies and the *refreshment* parameter is defined by the *triggering* and *ordering* components. Furthermore, an update propagation strategy is defined when the propagation and refreshment parameters are set. We consider three update propagation strategies: *deferred_immediate*, that is based on the common approach used by existing lazy master replication schemes [Moi96], and the *immediate_immediate* and *immediate_wait* strategies which we propose. In this section, we present them with respect to our architecture. We consider only the *triggering* component of the refreshment parameter. Therefore, the algorithms we present here are for a 1Master-1Slave configuration. The *ordering* component will appear in the following section when we present the refresher algorithm. We first present the deferred and immediate propagation strategies that establish the basis for other strategies.

4.2.1 Propagation

The Log Monitor reads log records from H in the same order as they were written so as to preserve serializability. It is important to mention that the algorithm used to manage H in the local database system is orthogonal to our strategies and has no impact on the way they function. The propagator also reads log records from the Input Log in the same order as they were written and propagates them in serial order. The Input Log stores log records and possible signals issued by the slave receiver. Each log record stored in both H and the Input Log carries the information necessary to perform an operation, that is the following attributes (see [GR93]): $timestamp, primary\_id, tuple\_id, field\_id, operation, new\_value$.

Each node implements a clock inside the local transaction manager. A transaction *timestamp* is the transaction's commitment time. The master identification, *primary_id*, identifies the primary copy R that is updated at the master node. Tuples are identified by

```
Propagator
input: Input Log
output: messages sent to slave nodes
variables:
    o: a record read from the Input Log
    m: carries o
    M^i: carries a seq. of o associated with the same transaction
begin
        repeat
            read(Input Log,o);
            if propagation = immediate
                        wrap o into a message m;
                        propagate (m);
            else /* propagation = deferred */
                if o = write for new transaction T
                            create a message M associated with T;
                if o = write for transaction T
                            add o to M;
                if o = commit
                            propagate(M);
                else if o = abort
                            discard(M);
        for ever;
end.
```

Figure 4.2: Propagator Algorithm

their primary key. In addition, the updated field within a tuple is identified by $field\_id$. Next, *operation* identifies the type of operation (update, delete, insert, abort and commit) performed. In case of update operations, new_value contains the new value of the field being updated. When there is no ambiguity, we sometimes use the term operation in place of log record.

The propagator implements the algorithm given in Figure 4.2. With *immediate* propagation, each write operation, w_i, of R, by transaction T is read from the Input Log and afterwards, a *propagate* function forwards a message m_i containing the input log record to the slave holding a copy r. Thus, for every slave node, there are as many messages as write operations in T. The performance impact of the possibly large number of messages generated is analyzed in the following chapter. It is important to note that concurrent update transactions at the master produce an interleaved sequence of write operations in H for different update transactions. However, write operations are propagated in the same order as they were written in the input log to preserve the master serial execution order. Update transaction's *abort* and *commit* at a master are also detected by the Log Monitor.

With *deferred* propagation, the sequence $w_1, w_2, ...w_n$ of operations on R done by transaction T is packaged within a single message, noted M_i, that is propagated to the slave holding r, after reading T's *commit* from the *Input Log*. The message M will form the body of the refresh transaction RT that updates r at the slave node. In case of an *abort* of T, the corresponding M_i under construction is discarded using the *discard* function.

4.2.2 Reception and Refreshment

Each Receiver reads the messages M^i in their reception order from the Reception Log. In addition, each Receiver checks the master identification *primary_id* field to select in which queue to store the log record(s) that each message carries. We assume that when a message carries a sequence, this sequence is stored as a single record in the queue.

To update a secondary copy r_i, the Refresher continously reads the queues seeking for new incoming records. With *deferred-immediate*, the Refresher reads a sequence $w_1, w_2, ..., w_n$ within a single message from a queue and subsequently writes the entire sequence into the running queue to be submited by the Deliverer, as a refresh transaction, to the local transaction manager. Note that the effect of the serial execution order of update transactions $T_1, T_2..., T_k$ performed at the master is preserved at the slave because the corresponding refresh transactions $RT_1, RT_2..., RT_k$ are performed in the same order.

With *immediate-immediate*, each time an operation is read from a queue, it is subsequently written into the running queue to be submitted by the Deliverer to the local transaction manager as part of some refresh transaction. Since write operations are propagated in the same order as they were written in the input log at the master node and, in addition, the underlying multicast protocol is a FIFO one, which preserves propagation order, then each message that carries a write operation is received following the propagation order. Therefore, the sequence of message stored in a running queue follows the serial order in which they occured at the master. Hence the effect of the serial execution order of the update transactions performed at the master is preserved at the slave node. When an abort operation for a transaction T is read by the Refresher it is also submitted to the local transaction manager to abort RT.

With *immediate-wait*, refreshment is done in two steps. In the first step, a message is read from q_r exactly as with *immediate-immediate*. However, each operation associated with a transaction T is stored into an auxiliary data structure, called a Reception Vector (noted RV). Thus, there is one vector per transaction and each element of a vector is an operation. When a *commit* operation for a transaction T is read from queue q_r, then this operation is appended to the corresponding vector RV and a refresh transaction RT is formed with the sequence of operations contained in RV, and submitted by the Deliverer to the local Transaction Manager.

The period of time delimited by the reading, in a queue q_r, of the first operation for transaction T, and the reading of the *commit* operation for T, is called the *wait period* for T. Figure 4.3 summarizes the algorithm executed by the Refresher for a given queue q_r using *immediate-wait*.

```
Immediate_Wait
input: a queue q_r
output: submit refreshment transactions
variables:
    o: the content of a message stored in q_r
    RV: vector of write operations for RT
begin
        repeat
            read(q_r,o);
            if o corresponds to a new transaction
                        Create a new vector RV;
            if (o ≠ commit) and (o ≠ abort)
                        add o to its associated vector RV;
            if o = commit
                        add o to its vector RV;
                        submit RV as an RT;
            if o = abort
                        discard the vector associated with o;
        for ever.
end.
```

Figure 4.3: Immediate-Wait Algorithm for queue q_r

For all the three strategies, when a refresh transaction RT is committed, the Deliverer marks all the messages M^i or m^i in the reception log as *processed*.

4.2.3 Optimization for Immediate-Wait

We propose an extension of the *immediate-wait* strategy that takes advantage of the wait period to optimize the execution of refresh transactions. In many applications, such as a data warehousing [FMS97], replicas are used in a slave node as the operand relations of materialized views. Essentially, the replication mechanism is used to propagate incremental changes from a data source (i.e. a master node) to a data warehouse (i.e. a slave node). Thus, the transactions that refresh a replica in the slave are then propagated to refresh the materialized views, thereby entailing the execution of new *view-refresh* transactions.

We suppose that every refresh transaction to r generates a *view-refresh* transaction to a materialized view V_r defined on r; which is the case for some data warehousing applications, also called Data Stores [FMS97].

In fact, incremental view refreshment algorithms take as input the *net changes* to their operand relations [GMS93]. Our first optimization is to compute, for each RT_i, the next change replica during the wait period. This is performed using the RV_i. For instance, if a sequence of writes to a same data item occurs in RV_i, only the last one is kept.

The second optimization is to start the *view-refresh* transactions generated by a RT_i as soon as the RT_i is started, directly using the RV_i, that is, without waiting for the RT_i *commit* execution at the slave. Futhermore, the computation of the changes to be made to a view V_r can be computed from the RV_i as soon as records are entered into RV_i.

4.3 Refresher Algorithms

An update propagation strategy is composed by the combination of the propagation and refreshment parameters. These two parameters were set for the three update propagation strategies presented previously. However, the *ordering* component of the refreshment parameter was ommited for pedagogigal reasons. We now set the *ordering* component for each update propagation strategy and present the refresher procedures for the each strategy: *deferred-immediate Refresher*, *immediate-wait Refresher* and *immediate-immediate Refresher*. We show that the ordering choice we make is sufficient to enforce correctness for any lazy master configuration.

Preliminaires

Following the network model presented in [GM97], we assume that the reliable FIFO multicast has a known upper bound Max and that all nodes have synchronized clocks, such that the difference between any two clocks is not higher than the precision ϵ. The timestamp of a message corresponds to the timestamp (henceforth noted C) of the update transaction that originated that message. The timestamp of T corresponds to the real time value at T's commitment time.

The Principle of the FIFO Ordering Algorithm

The principle of the FIFO ordering algorithm we use for the three strategies works as follows. A refresh transaction RT is executed at a slave or MasterSlave node the latest at real time $C + Max + \epsilon$, where C is the timestamp of RT's original update transaction.

FIFO here means that the first update transaction commited at a node must have a corresponding refresh transaction that is first executed at a slave or MasterSlave node. Therefore, whenever clocks are synchronized, the effects of updates on secondary copies follow the same chronological order in which there corresponding primary copies were updated.

4.3.1 Deferred-Immediate Refresher

The *deferred-immediate* Refresher implements the FIFO ordering algorithm for the *deferred-immediate* strategy (see Figure 4.4). The top of each pending queue $q_1, ...q_n$, one for each master or MasterSlave node, may store a message M. Recall that with the FIFO reliable multicast protocol, the sequence of messages $M_1, ...M_n$ is stored in a specific pending queue following their timestamp order (or commitment order). Without loss of generality, we use M and M' to denote messages coming from distinct masters nodes.

Whenever a message arrives in a queue that was empty, the Refresher performs the scheduling procedures by selecting among the top messages of $q_1 ... q_n$ the one which shows the smaller timestamp C. Afterwards, the Refresher calculates the deliver time (noted $deliver\_time$) for M. The $deliver\_time$ is the time in which M must be put available for delivery. This means that the refresh transaction carried by M must be written into the running queue. The $deliver\_time$ of M and is calculated using the following formula:

$$deliver\_time = C + Max + \epsilon.$$

Once the $deliver\_time$ is calculated, the Refresher needs to check how much it has to wait to deliver M. The waiting time for M (noted $time\_out$) is calculated by subtracting the $deliver\_time$ value from the the local clock time (noted $local\_time$) value, as follows:

$$time\_out = deliver\_time - local\_time.$$

The Refresher then sets a local $timer$ with the $time\_out$ value for M.

$$time\_out = deliver\_time - local\_time.$$

Whenever the time expires, it means that the local clock value is equal to $deliver\_time$ and that M is ready to be delivered. The Refresher then writes M into the running queue. It is important to note that while the timer is activated for M, if another message M' with $C' < C$ arrives, the Refresher calculates both the $deliver\_time'$ and $time\_out'$ for M' and changes the value of the timer to $time\_out'$. This means that whenever $time\_out'$ expires, M' will be equally written into the running queue. When M' is delivered a new $time\_out$ value is calculated for M since its delivery time does not change. We use the terms C, $deliver\_timer$, $local\_time$ and $time\_out$ to refer to the scheduling attributes of a message M. The scheduling attributes of a message M are calculated by the procedure Calculate-Attributes (see Figure 4.5).

Let consider an example in which there is a slave node that manages two pending queues q_i and q_j. Suppose that M_R is read from q_i at $15:10$ and that $Max = 10min$ and $\epsilon = 1min$. Therefore, the Refresher calculates the scheduling attributes for M_R:

$C = 15:05$,
$deliver\_timer = 15:16$ $(15:05 + 10 + 1)$,
$local\_time = 15:10h$,
$time\_out = 6min$.

Thus, the timer is set to 6 min. Two minutes after, M_S arrives in q_j and the Refresher reads it and calculates the scheduling attributes for M_S:

$C = 15:03$,
$deliver\_time = 15:14$,
$local\_time = 15:12$,
$time\_out = 2min$.

Since the timestamp of M_S is smaller than the timestamp of M_R the Refresher interrupts the timer for M_R and re-activates it with the value of 2 minutes for M_S.

Deferred-Immediate Refresher
input: pending queues $q_1 ... q_n$, one per master node
output: write refreshment transaction into running queue
variables:
M: stores the contents of a message from a pending queue q_i;
M': stores the contents of a message from a pending queue $q_j \neq q_i$;
C: timestamp of a message M;
timer: stores the state of the timer: active or inactive
begin
timer = inactive;
$M = M' = \emptyset$;
repeat
for $i = 1..n$
if the top of $q_i \neq \emptyset$
select the message on top of $q_1 ... q_n$
with min C
into M';
if $M' \neq \emptyset$ and $M' \neq M$
$M = M'$;
Calculate-Attributes(C);
if timer is active and is expired
write(M, running queue);
delete(q_i,M);
timer = inactive;
for ever.
end.

Figure 4.4: The Deferred-immediate refresher algorithm

Calculate-Attributes
input: a timestamp value C associated with a message M
output: the time is set with a $time\_out$ value
variables:
C: the timestamp of M
$local\_time$: the clock time
$deliver\_time$: deliver time for M
$time\_out$: time_out for M
timer: stores the state of the timer (active or inactive)
begin
if $deliver\_time$ for $M = null$
$deliver\_time$ for $M = C + Max + \epsilon$;
$time\_out = deliver\_time - local\_time$;
set local-timer with $time\_out$;
timer = active;
end.

Figure 4.5: The procedure Calculate-Attribute

Two minutes later (at 15:14) no new message arrives, the timer expires and M_S is written into the running queue. Afterwards, the Refresher reads M_R again from q_i and calculates only the new $time\_out$ value for M_R, since the $deliver\_time$ was already calculated. In our example, the new $time\_out$ value for M_R is:

$$deliver\_time(15:16) - local\_time(15:14) = 2min$$

Two minutes later, no new message arrives, the timer expires and M_R is written into the running queue.

If several messages have equal timestamps, they are selected following their master identification priority. Notice that a message with a timestamp less than or equal to a timestamp C always comes from a different master because messages are propagated from the same master or masterslave node in FIFO order.

4.3.2 Properties

We now show that the properties of the *deferred-immediate* Refresher are sufficient to enforce correcteness for any lazy master configuration. Its important to notice that the principle of the FIFO ordering algorithm is the same for the three update propagation strategies. Therefore the properties shown here are equally valid for *immediate-wait* and *immediate-immediate* Refreshers.

Property 1: Total Order is Enforced

Proof: Messages whose timestamps differ more than the clock precision are scheduled chronologically at each node because since the maximun transmission delay is Max and

clocks are synchronized within ϵ, then at $C + Max + \epsilon$ a node is sure that it will never receive any messages with timestamp $\leq C$. Otherwise those messages would have suffered a transmission delay higher than Max, a contradiction with our network model. Consequently, at time $C + Max + \epsilon$ the Refresher can write M in the running queue. Notice that because of non perfect clocks, the timestamps given by the master nodes can differ from the timestamps in the absolute time.

We can show that if $C - C' > \epsilon$ on the absolute time, then M and M' are scheduled according to the timestamp order seen in the absolute time. Let us assume that an external observer first sees C and after a delay $d > \epsilon$ sees C'. The timestamps C and C' given by the master nodes always meet $C < C'$ even if the clock of the node that generates C is in advance of ϵ with regard to the clock of the node that generates C'.

However, if $C - C' \leq \epsilon$ then M and M' are scheduled following their master node priority, but the same order for all nodes. Therefore, total order is always enforced.

Property 2: Causal Atomic Order is Enforced

Proof: Consider a master-masterslave-slave configuration and suppose that M and M' are two messages multicast by a master node i and a masterslave node j, respectively. In addition, suppose that M causally precedes M' and that the slave node uses the FIFO scheduler algorithm to order message delivery. It is obvious that if M causally precedes M' then $C < C'$ because M' is generated at node j after the multicast and the delivery of M. Therefore, $C' \simeq C + Max + \epsilon + \Delta$, where Δ is the time spent to process M at node j. This allows us to conclude that $C - C' > \epsilon$ is always true. Therefore, total chronological order is sufficient to enforce causal order and total order. Hence, causal atomic order is enforced.

Lemma 1: Correctness is enforced for 1Master-nSlave Configurations

In a 1Master-nSlave configuration, correctness is enforced when the message delivery order at each slave node must follow the same delivery order that would be obtained when using a FIFO reliable multicast protocol. The underlying multicast protocol is a FIFO reliable protocol, therefore correctness is enforced.

Lemma 2: Correctness is enforced for mMaster-nSlave Configurations

Proof: In a mMaster-nSlave configuration correctness is enforced whenever messages are delivered following the delivery order that would be acheived when using a FIFO Atomic multicast protocol. Atomic order is guaranteed by property 1 and FIFO order is guaranteed because we use assume that the underlying multicast protocol is FIFO reliable one. Therefore, the effects of a FIFO Atomic multicast protocol are enforced .

Lemma 3: Correctness is enforced for mater-masterslave-slave Configurations

Proof: In a master-masterslave-slave configuration correctness is enforced whenever messages are delivered following the delivery order that would be acheived when using a Causal

Atomic multicast protocol. Atomic order is guaranteed by property 1 and Causal order is guaranteed by property 2. Therefore, the effects of a Causal Atomic multicast protocol are enforced .

Lemma4: Correctness is enforced for Hybrid Configurations

Proof: In a hybrid configuration, correctness is enforced when the correctness criterias for 1Master-nSlave, nMaster-nSlave and master-masterslave-slave configurations. The is guaranteed by Lemma1, Lemma2 and Lemma3 .

4.3.3 Immediate-wait Refresher

The *immediate-wait* refresher implements the FIFO ordering algorithm for the *immediate-wait* strategy (see Figure 4.7). At a slave or MasterSlave node, the top of each pending queue $q_1, ...q_n$ may store a message m that carries an operation o of some refresh transaction. The Refresher reads, in round robin, the top message m of each queue. Each time a new transaction is detected in q_i the Refresher creates a reception vector (denoted RV) to store all operations from q_i associated with the incoming transaction. We use the term *complete* RV to refer to a RV that already stores the complete sequence of operations $w_1, ...commit$ related to some refresh transaction. This vector may also be interpreted as a message M because, in fact, it stores the sequence of operations that M would carry when using the *deferred-immediate* strategy. Whenever there is no ambiguity we use M instead of RV. Again, recall that with the FIFO reliable multicast protocol, messages are stored in each pending queue in their timestamp order. In the example of Figure 4.6, q_i stores a sequence of write operations of two distinct update transactions, T_R and T_S, coming from the common master node of R and S. w_R^1 denotes the first operation on R and w_S^1 denotes the first operation on S. Similarly, $commit_R$ and $commit_S$ correspond to the commit operations of w_R^1 and w_S^1, respectively. The complete reception vectors that will be generated by the Refresher are: $RV_S = [w_S^1, commit_S]$ and $RV_R = [w_R^1, commit_R]$. Notice that in this case RV_S corresponds to M_S and RV_R to M_R. Futhermore, M_S must be delivered after M_R because $commit_R$ is stored in q_i after $commit_S$.

To manage message scheduling, we define a set of *working queues* $wq_1...wq_n$, each one is related to a pending queue q_i. In the example of Figure 4.6, wq_i is related to q_i and stores the contents of each reception vector RV (or message M) in timestamp order. The Refresher reads the top messages of $wq_1...wq_n$ to perform the scheduling procedures.

Whenever a complete RV related to q_i is generated, the Refresher first checks whether wq_i is empty. If wq_i is empty, besides writing M into wq_i, the Refresher also selects the message that shows the smallest timestamp among the top messages of $wq_1...wq_n$. If M is selected then the $deliver\_time$ and $time\_out$ values for M are calculated and the timer is set with $time\_out$. Otherwise, if the timer was active for another message, the timer continues normally its activities. When the timer expires for some message M, the Refresher detects it and writes M into the running queue and deletes M from its working queue. Afterwards, the Refresher selects the next message to be scheduled.

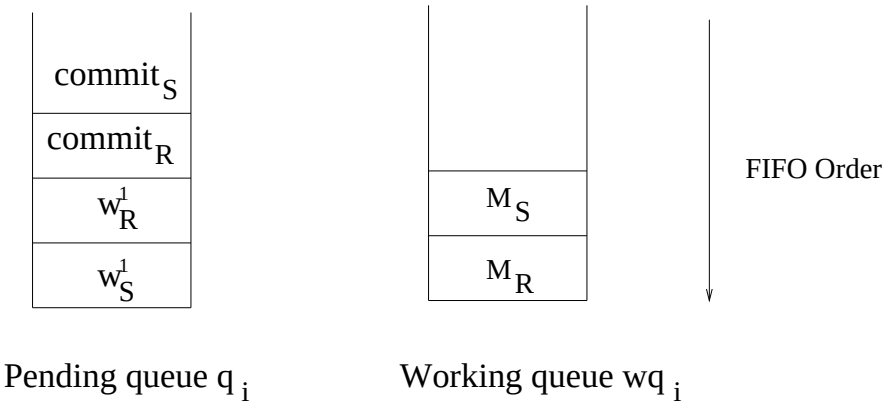


Figure 4.6: Example of a pending queue q_i and its corresponding working queue wq_i.

Notice that the *immediate-wait* Refresher algorithm shows the same propreties shown by the *deferred-immediate* Refresher algorithm one because since a RV can be considered a message M, and each message M is stored in a working queue wq_i associated with q_i, than the proofs of correcteness for the *deferred-immediate* Refresher are equally valid for the *immediate-wait* Refresher.

4.3.4 Immediate-immediate Refresher

The *immediate-immediate* refresher implements the FIFO ordering algorithm for the *immediate-immediate* strategy (see Figure 4.9). At a slave or masterslave node, the top of each pending queue $q_1, ...q_n$ may store a message m that carries an operation o of some refresh transaction. The Refresher reads, in round robin, the top message of each queue $q_1...q_n$ and if m does not correspond to a *commit* message, then the Refresher writes it into the running queue. Therefore, the operation carried by m is immediately submitted for execution to the local transaction manager. In the example of Figure 4.8, the sequence $< w_S^1, commit_S >$ that corresponds to RT_S and $< w_R^1, commit_R >$ that corresponds to RT_R are both stored in q_i, and $< w_T^1, commit_T >$, that corresponds to RT_T, is stored in q_j. In this example, w_S^1, w_T^1 and w_R^1 may be read in this order and written in the same order in the running queue.

Therefore, the operations coming from different masters may be written in the running queue in any order. In our example, let us consider that the correct refresh transaction delivery order is: RT_R, RT_T, RT_S. With the *immediate-immediate* Refresher, the delivery order of a refresh transaction is defined when its *commit* is written into the running queue, because even if the incoming operations of all queues are previously written in any order, the execution of a commit by the local transaction manager defines the serial refresh transaction execution order. Figure 4.8 shows the contents of the running queue that is generated by

Immediate_Wait Refresher
input: pending queues $q_1 ... q_n$, one per master node
output: write refreshment transaction into running queue
variables:
M: stores the contents of a message from a pending queue q_i;
M': stores the contents of a message from a pending queue $q_j \neq q_i$;
C: the timestamp of M
o: the content of a message m stored in q_i
$wq_1 ... wq_n$: working queues associated with each $q_1 ... q_n$
RV: vector of operations for a RT associated with a wq_i

```
begin
M = M' = ∅;
begin
    repeat
        for q_1...q_n do
            if q_i ≠ ∅
                read(q_i,o);
                if o corresponds to a new transaction
                        Create a new vector RV;
                if (o ≠ commit) and (o ≠ abort)
                        add o to its associated vector RV;
                if o = commit
                        add o to its vector RV;
                        write(wq_i,RV);
                        select the message in top wq_1...wq_n
                        with min C into M';
                        if M' ≠ ∅ and M' ≠ M
                           M = M';
                           Calculate-Attributes(C);
                if o = abort
                           discard the vector associated with o;
                delete o from q_i;
            if timer is active and expired
                write(M, running queue);
                delete(wq_i, M);
                M = ∅;
                timer = inactive;
                select the message in top wq_1...wq_n
                with min C into M;
                if M ≠ ∅
                        Calculate-Attributes(C);
        end-for
    for-ever.
end.
```

Figure 4.7: The Immediate-Wait Refresher Algorithm

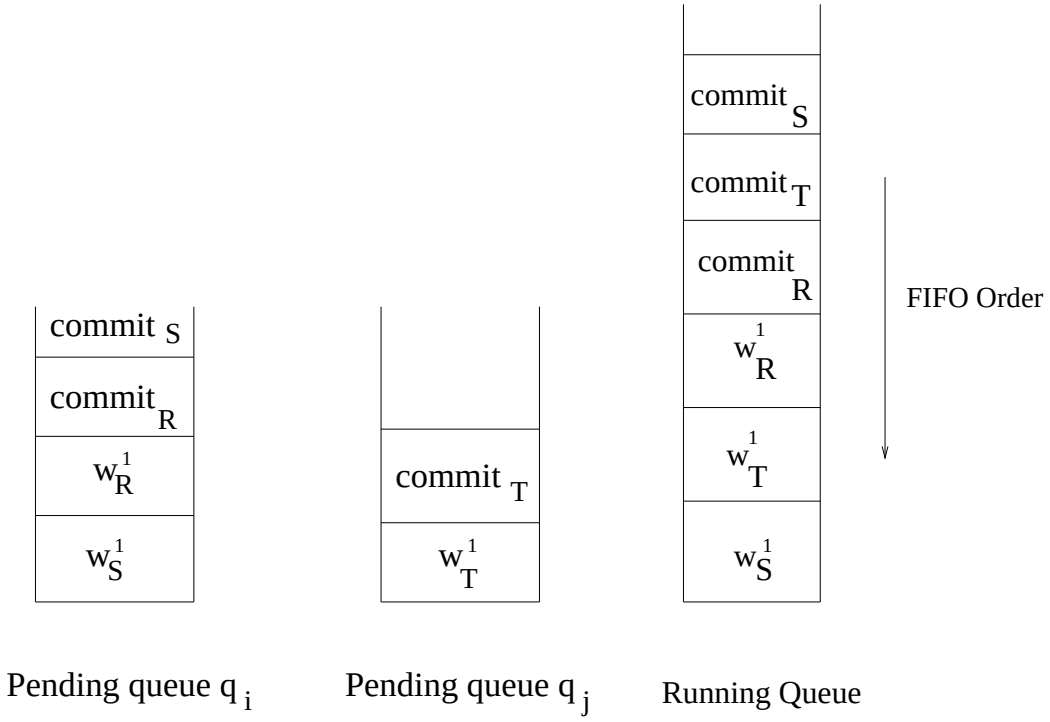


Figure 4.8: Example of *immediate-immediate* Refresher execution.

```
Immediate_immediate Refresher
input: a queue q1...qn
output: write refresh transaction into running queue
variables:
   o,m, m': stores the content of a message stored in q1, ...qn
begin
      m = m' = ∅;
      timer = inactive;
      repeat
         for i = 1...n do
            if qi ≠ ∅
                  read(qi,o);
                  if o ≠ commit
                     write(o, running queue);
                     delete(o,qi);
                  else
                  if o = commit
                     select commit operation in top of q1...qn
                     with min C into m';
                     if m' ≠ m
                        m = m';
                        Calculate-Attributes(C);
            if timer is active and expired
                  write(commit, running queue);
                  delete(commit,qi);
                  timer = inactive;
                  select commit operation in top of q1...qn
                  with min C into m;
                  if m ≠ ∅
                     Calculate-Attributes(C);
         end-for.
      end-repeat
end.
```

Figure 4.9: The Immediate-immediate refresher algorithm

the Refresher to guarantee the correct delivery of RT_R, RT_T, RT_S. Therefore, the ordering procedures necessary to guarantee FIFO order must be performed by the Refresher whenever a *commit* operation is read from a pending queue.

When a *commit* operation is read from the top of q_i, the Refresher selects the *commit* message among $q_1...q_n$ that shows the smallest timestamp. Notice that the *commit* record carries a timestamp value C. Afterwards, the Refresher calculates the $deliver\_time$ and the $time\_out$ values for the associated refresh transaction RT using the same formulas presented previously, and sets the timer. If meanwhile a new commit operation is read for another

refresh transaction RT' from queue q_j, the scheduling attributes of RT' are calculated and if necessary, the timer is interrupted and re-activated with RT' $time\_out$ value, using the same scheduling principle presented for the deferred-immediate strategy. When the timer expires for some refresh transaction RT, it detects it and write RT's *commit* operation into the running queue. Afterwards, it selects the next *commit* operation to be scheduled.

Notice that the *immediate-immediate* Refresher algorithm shows the same properties shown by the *deferred-immediate* and *immediate-wait* strategies because the message delivery order is defined by the commit timestamp order. Therefore, the scheduling algorithm guarantees FIFO order. Thus, the proofs for correcteness for all configurations shown for the *deferred-immediate* Refresher are equally valid for the *immediate-immediate* Refresher.

4.4 Dealing with Failures

We now present the connection and node recovery protocols used in our replication scheme. They are based on those used for transaction monitoring recovery (see [GR93]). Therefore, we introduce only the additional features needed for our replication scheme.

To start update propagation towards a slave node, a master node must first initiate a connection. When update propagation is done, the master can close the connection. In order to preserve the autonomy of each DBMS, connection and disconnection requests are captured by our replication module through the Local History Log (noted H). The result of a connection or disconnection request at each node is done using the *connection table* that is described below. Connection and disconnection are requested using the following functions:

Connect($Master\_id$**,** $Slave\_id$**,** $Replica\_id$**):** $Master\_id$ requests a connection to $Slave\_id$ in order to start update propagation on $Replica\_id$. A corresponding log record is generated and written in the Local History Log of $Master\_id$ with the following information:

$$<'' Connect'', Master\_id, Slave\_id, Replica\_id >.$$

Disconnect($Master\_id$**,** $Slave\_id$**,** $Replica\_id$**):** $Master\_id$ requests a disconnection to $Slave\_id$ in order to stop update propagation on $Replica\_id$. A corresponding log record is generated and written in $Master\_id$ Local History with the following information:

$$<'' Disconnect'', Master\_id, Slave\_id, Replica\_id >.$$

Each node i keeps control of its connections using a relational *connection table* stored in the local DBMS. The main attributes of this table are $Node\_id$, $Replica\_id$ and $Status$. $Node\_id$ identifies a node that is connected to node i and $Replica\_id$ identifies the local replica copy involved in a specific connection. If the node is a master node then the replica copy is a primary copy, otherwise it is a secondary copy. The *status* attribute indicates the

current status of the connection. In a master node, the Log Monitor reads the connection table to check which primary copies it must monitor. It also uses the connection table to detect slave failures and perform master recovery. A slave node uses the connection table to identify its masters in order to detect master failure and perform its own node recovery. A connection table may be updated by the Receiver and the Propagator, and may be read by the local DBMS users.

Node initialization starts by creating a connection table a each node. At a slave node, each pending queue is forced to store a single *Disconnect* record for all its possible masters. Thus, a *Disconnect* record indicates to the Refresher that each queue is not available for refreshment.

Connection and disconnection are handled as follows. When the Log Monitor reads a *Connect* (or *Disconnect*) record from H, it writes it in the Input Log. Then, the Propagator reads a Connect (or Disconnect) record from the Input Log, and propagates the Connect (or Disconnect) message towards $Slave\_id$. In case of connection, it creates a new connection entry in the local connection table. Otherwise it deletes the corresponding connection from the connection table. When a slave node receives the Connect message, the Receiver creates a new connection entry in the local connection table and then enqueues the contents of the Connect message in the correct pending queue. Thus, a Connect record indicates to the Refresher that the queue is available for refreshment. When a slave node receives the Disconnect message, the Receiver deletes the corresponding connection entry from the local connection table and then enqueues the contents of the Disconnect message the correct pending queue. Thus, a Disconnect record indicates to the Refresher that a queue is not available for refreshment.

Disconnection can be granted only after all messages sent by the master have been received by the slave. This is easy because messages are propagated using a FIFO reliable multicast protocol. Thus, when a slave receives a Disconnect message, it is obviously the last one for the corresponding connection.

Slave failure is managed as follows. When a slave node fails, all the connected masters detect the failure through their Propagator which subsequently interrupts propagating activities towards the failed slave node. We assume that the master Propagator periodically checks for its slaves node availability using the network interface. A slave recovery is performed in two steps. First, the Receiver writes in each pending queue a *Reconnect* record which carries the following information:

$$<\text{"}\ Reconnect\text{"}, Master\_id, Slave\_id, Replica\_id, Message\_id >.$$

$Message\_id$ indicates the last received message M_i or m_i (this is easily read by the receiver based on the *reception log*) and is used as re-synchronization point to restart the master propagation and the slave refreshment activities. The Refresher reads a Reconnect record and is aware that the connection with $Master\_id$ is being re-established. Therefore, in the second step, the slave node tries to re-establish its pending connections. The Receiver signals reconnection to the Propagator by writing the Reconnect record inside the Input Log. Whenever the slave propagator reads the Reconnect record from the Input Log, it propagates

it towards $Master\_id$. In parallel with Reconnect propagation, the slave Receiver re-starts reception activities for $Master\_id$. It does so by searching in the Reception Log for all messages received from $Master\_id$ but not yet processed by the Refresher, and stores each message in the correct queue q. Therefore, the refreshment activities on queue q are also re-started.

Master failure is handled as follows. When a master node fails, all the connected slave nodes detect the failure through their Receiver. We assume that the slave Receiver periodically checks for its masters node availability using the network interface. As soon as a master failure is detected, the slave Receiver writes a $Fail$ record of the form:

$$< ''Fail'', Master\_id >$$

in the correct pending queue. This record indicates the Refresher of $Master\_id$ failure. Whenever a master recovers, it tries to re-establish its connections by propagating a Reconnect message towards each slave node it was connected to. In this case, the $Message\_id$ field is nil. When the slave node receives a Reconnect message, it stores its contents in the correct queue q. Thus, the Reconnect record indicates the Refresher of $Master\_id$ recovery. Afterwards, the slave receiver writes the Reconnect record in the Input Log with $Message\_id$ correctly set. Whenever the slave Propagator reads a Reconnect record from the Input Log, it propagates it towards $Master\_id$. In parallel with Reconnect propagation, the receiver re-starts reception activities for $Master\_id$. As in any database system, the Propagation Log and Reception Log are supposed to have enough disk space to deal with recovery.

Our node recovery protocol is non-blocking because in case of a master failure, the slave refreshment activities on other secondary copies are not interrupted. Using the connection table, a user can detect the failure of a master and make a choice on how to proceed.

4.5 Related Work

In this section, we relate the algorithms proposed in this section with other similar approaches. In addition, we show their applicability to refresh data in a data warehouse.

In [SKS86] a system is described that uses timestamp-based concurrency control and proposes to apply updates to replicated data in their arrival order, possibly restoring inconsistencies when arrivals violate the timestamp ordering of transactions. This mechanism acheives consistency by undoing and re-executing updates which are out-of-order. This approach differs from ours since with our algorithms inconsistency is not permited.

In [Gol92], a timestamp message delivery protocol that implements eventual delivery is proposed. The approach uses periodic exchanges of messages between pairs of principals to propagate messages to group of nodes. Incomming messages are stored in a log and later delivered to the application in a defined order. With our refresher algorithms, data freshness is crucial, therefore, eventual delivery algorithms are not suited for the applications we are interested on. The use of the history log for propagation of updates was suggested in [KR87].

Incremental agreement is a strategy [CHKS95] that focus on managing network failures in replicated databases and do not address the problem of improving freshness neither the existence of different configurations. Refreshment is performed using the slave log instead of by the local transaction manager as done be the algorithms proposed in this chapter.

In the *epidemic* model [AAS97], update operations are executed locally at any single node. Later, nodes communicate to exchange up-to-date information. In this way updates pass through the system like an infectious disease, hence the name epidemic. Thus, users perform updates on a single node without waiting for communication, and the system can schedule communication at a later convenient time. Epidemic algorithms are designed to satisfy a level of consistency weaker than *serializability*. Primarily, they preserve the causal order of update operations. For some applications this is sufficient for correctness. Otherwise, designers optimistically assume that conflicts will be rare and can be handled using application specific compensation. Epidemic techniques are useful but are too weak for supporting transaction processing. Fundamentally, the goal of epidemic algorithms is to ensure that all replicas of a single data item converge to a single final value.

We now show in more details how data replication may be used to refresh data in a warehouse. A warehouse is a repository of integrated information drawn from remote data sources. Since a warehouse effectively implements materialized views, the views must be maintained as the data source are updated.

Unfortunately, existing materialized view maintenance algorithms [ZGMW95] fail in a warehousing environment. Existing approaches assume that each source understands view management and knows exactly what data is needed for updating the view. However, this is not always true. Sources can inform the warehouse when an update occurs, e.g., a new employee has been hired, or a patient has paid her bill. However, they cannot determine what additional data may or may not be necessary for incorporating the update into the warehouse views. When a simple update information arrives at the warehouse, it may discover that some additional source data is necessary to update the views. Thus, the warehouse may have to issue queries to some of the sources making refreshment a slow task. A solution to improve performance in the above scenario is to use data replication.

In this context, the $mMaster - nSlave$ is the appropriate configuration to be used for warehouses and views are built on top of replicated data instead of source data. Therefore, the materialized views are refreshed whenever replicated data is refreshed and the problem of querying data sources disapears. It is important to note that the correctness criteria defined for the $mMaster - nSlave$ is indirectly applied to the materialized views. That is, the order in which secondary copies are refreshed will determine the order in which updates are applied to refresh a view.

4.6 Conclusions

In this chapter, we addressed the problems of improving data freshness and enforcing correctness in lazy master schemes. We presented a system architecture for master and slave nodes that is used by our update propagation strategies. With respect to freshness im-

provement, we proposed two new *immediate* strategies called: *immediate-immediate* and *immediate-wait*. With *immediate-immediate*, updates on a primary copy at some master node are immediatly propagated towards the other secondary copies and a refresh transaction is started at a slave node as soon as the first write operation is received from the master node. Similarly, with *immediate-wait*, the propagation procedure follows the same principle. However, a refresh transaction is started at a slave node after the complete reception of all updates (of the same transaction) from the master node. In the next chapter, the freshness improvements of our immediate strategies are analyzed.

In the second part of the chapter, we have presented a FIFO ordering algorithm used by the Refresher to order message delivery. This algorithm orders chronologicaly refresh transaction execution using their original update transaction timestamp values. Therefore, whenever clocks are synchronized, the effects of updates on secondary copies follows the same order in which their corresponding primary copies were updated. We presented three variants of the Refresher algorithm for our architecture, *deferred-immediate* Refresher, *immediate-immediate* Refresher and *immediate-wait* Refresher. Each variant is suited for a specific update propagation strategy. We presented the properties of the Refresher algorithm and showed that it is sufficient to enforce correctness for the 1Master-nSlave, mMaster-nSlave, Master-MasterSlave-Slave and hybrid configurations. We presented a non-blocking protocol for master and slave nodes recovery. Finally, we related our work with others in the field and showed how it can be used as a solution to refresh a data warehouse.

Chapter 5

Validation

5.1 Introduction

To validate the strategies proposed in Chapter 4, we need to demonstrate their performance improvement. Performance evaluation of update propagation strategies is a difficult task since several factors such as node speed, multiprogramming level, network bandwitch and others, have a major impact on performance results. Some update propagation strategies have been evaluated analytically [GN95, GHOS96]. However, analytical evaluation is typically very complex and hard to understand. In addition, the results may not reflect the real behavior of the strategies under various workloads. In this thesis, performance evaluation is crucial to assess and compare the freshness improvements brought by our *immediate-immediate*, *immediate-wait* and *deferred-immediate* strategies. Therefore, instead of doing performance analysis of our strategies, we prefer to use a simulation environment that reflects as much as possible a real replication context.

To our knowledge, there is no simulation tool that provides both network and database parameters which we need. In this chapter, we propose a simulation environment to validate and measure the performance of the update propagation strategies as well as the scheduling procedures necessary to enforce correctness. This environment is simple, yet taking into account all relevant factors. Our performance evaluation takes into account a data consolidation configuration. We choose this configuration because it is widely used in data warehouse applications. Futhermore, it allows the evaluation of our strategies when the number of master nodes increases (we consider up to 8 master nodes).

The contributions of this chapter are:

1. We introduce a simulation environment used for the performance evaluation.
2. We present the most relevant implementation solutions for a slave node.
3. We propose a performance model used to run the experiments.

4. We analyze the results of four experiments that show the gains and drawbacks of each update propagation strategy in different workload scenarios.

The organization of this chapter is as follows. Section 2 presents the simulation environment used to validate and evaluate *immediate-immediate*, *immediate-wait* and *deferred-immediate* strategies. Section 3 presents the performance model used in our evaluation. In Section 4, we explain and discuss our 4 experiments and their results. Finally, Section 5 concludes.

5.2 Simulation Environment

There are many factors that may influence the performance of our strategies, for instance the processing capability of the slave nodes and the speed of the network medium. However, we isolate the most important factors such as: (i) transaction execution time, reflected by the time spent to log the updates made by a transaction into the history log (noted H), (ii) refresh transaction (noted RT) propagation time, that is the time needed to propagate the necessary messages from a master to a slave, and (iii) refresh transaction execution time. Consequently, our simulation environment only focuses on the components of a node architecture that determine these three factors. In the following, we present the component modules of the simulation environment. We assume familiarity with object orientation and operation system concepts that are well documented in [BGV97]. In particular, we use the OMT notation [Rou92] to graphically present the static and functional models for the simulator modules.

5.2.1 Modules

Figure 5.1 portrays the simulation modules: *Master*, *Network*, *Slave* and a database server. The Master module simulates all relevant functionalities of a master node such as log monitoring and message propagation. The Network module implements and simulates the most significant factors that may impact our update propagation strategies such as the delay to propagate a message. The Slave module implements the most relevant components of the slave node architecture such as Receiver, Refresher and Deliverer. In addition, for performance evaluation purposes, we add the Query component into the slave module, which implements the execution of queries that read replicated data. We do not consider node failure in our performance evaluation because we focus in freshness and not in recovery evaluation. Therefore, the reception log as well as the recovery procedures are not taken into account. Finally, a database server is used to implement refresh transactions and query execution.

Our environment is implemented on a Sun Solaris using Java/JDBC as the underlying programming language. Java is a pure object language intended for network applications. It supports the basic principles of the object approach, that is encapsulation, inheritance, polymorphism and dynamic binding. Multithreading is also provided: several tasks can be executed in parallel and synchronized in a single application. In addition, Java offers a set of predefined classes such as Thread and DatagramSocket. The Thread class contains all the methods necessary to manage a process such as *Start*, *Stop*, *Wait* and others. The DatagramSocket class contains all the methods necessary to manage a socket in datagram mode. Java uses JDBC to communicate with different database systems. In particular we use Oracle 7.3 to implement the database functionalities such as transaction execution, query processing and others. These and other interesting aspects of Java make it good choice as the underlying platform of our simulation environment.

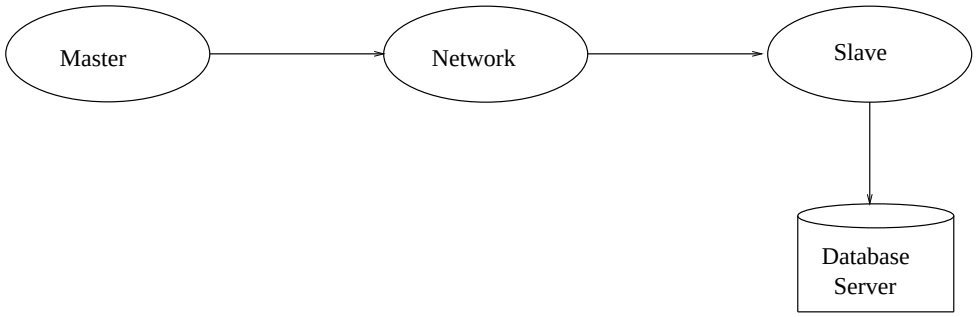


Figure 5.1: Simulation Modules

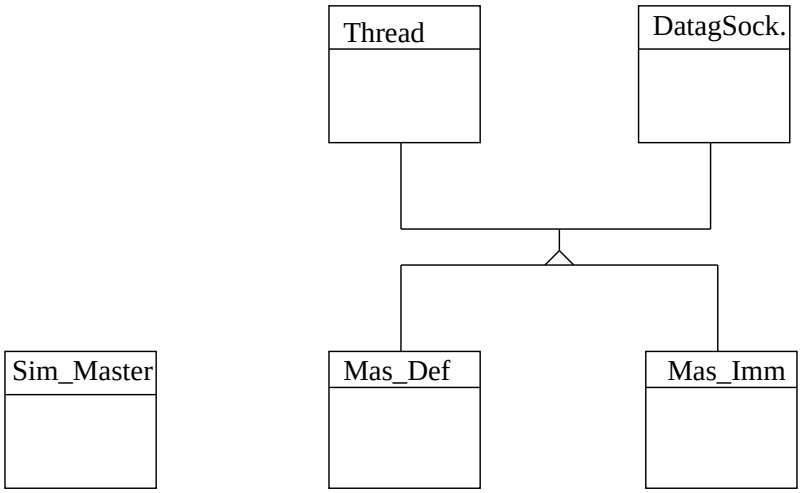


Figure 5.2: Master Module Classes

5.2.2 Master Module

The Master module (see Figure 5.2) is composed of three classes: *Sim_ Master*, *Mas_ Def* and *Mas_ Imm*. The Sim_Master class implements all the procedures necessary to start and finish a simulation execution. The Sim_Master class interacts with the two other classes (see Fig 5.3), Mas_Def and Mas_Imm. Each class simulates the deferred and immediate propagation strategies, respectively, and inheritates from both Thread and DatagramSocket classes. Therefore, each instance of one of these classes (see Figure 5.3) corresponds to a master node (henceforth master process). The communication between each master process

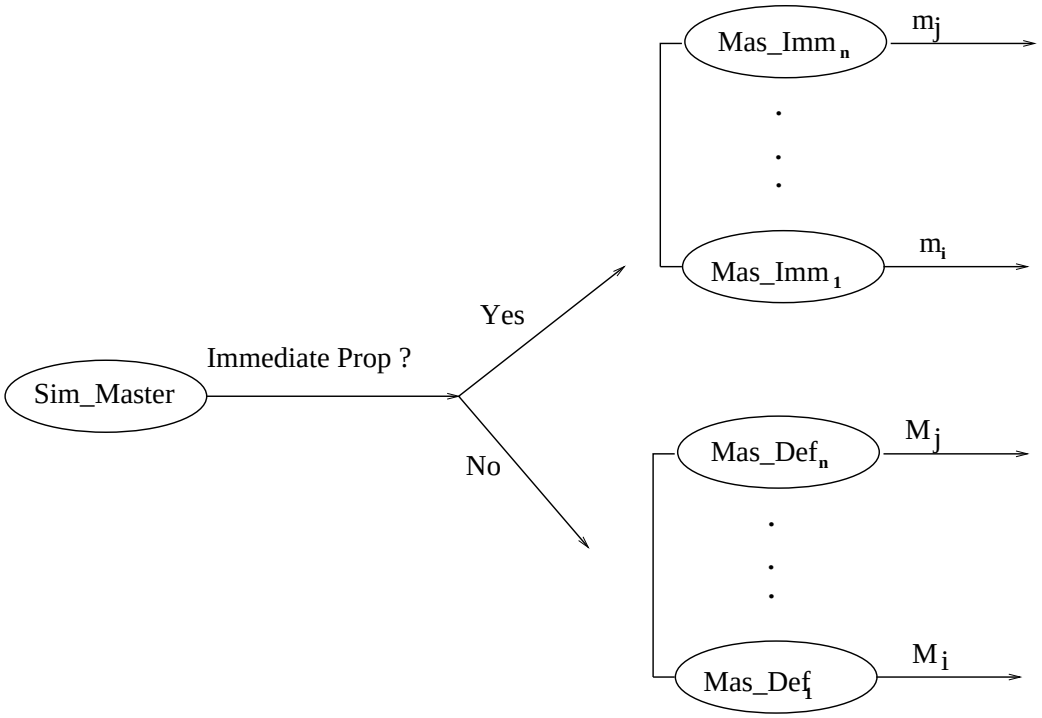


Figure 5.3: Functional Model of the Master Module

and the network module is implemented through sockets. Propagation is simulated within each instance *Mas_ def* and *Mas_ imm* and works as follows. Each master process monitors a different primary copy. The simulation of the log monitoring activities is implemented by the occurrence of two types of events: *write detection* and *transaction detection*. The write detection event establishes the occurrence of a write, and the transaction detection event establishes the start of an update transaction. Write detection and transaction detection events have an arrival rate that may have a *poisson* or *exponential* behavior. The generation of the time interval between two events is done using a *generator* program. The generator program asks for the *mean* value and the number of events. The mean value is used to compute each time interval for the number of specified events. The *generator* progam produces two files. The first one, *twrite*, contains the time intervals between each two writes of T_i. The second file, *tstart*, contains the time interval between each two update transactions T_i and T_{i+1}.

Log monitoring and message propagation within Mas_Def is implemented using a single loop. Each time a loop is executed, a time value is read from *tstart* and used to set a clock.

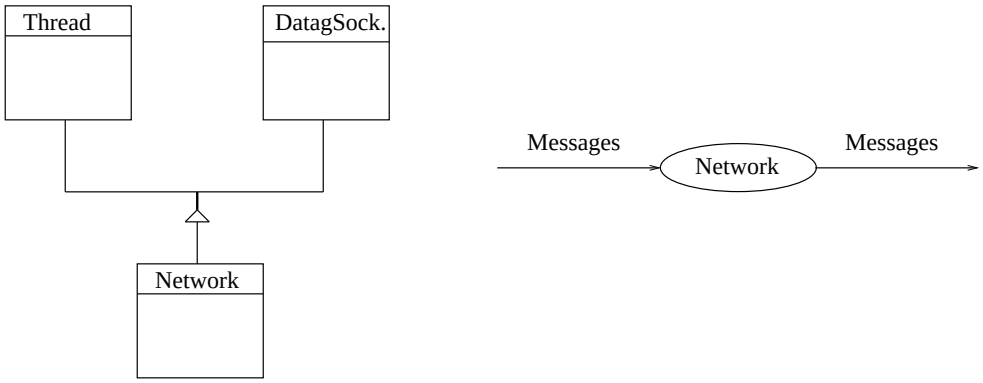


Figure 5.4: Network Module Class and Functional Model

Whenever the clock times out, it indicates that it is time to propagate a refresh transaction message. Each propagated message (M_i) carries the relevant data such as $master\_id$, TID, and the description of the list of operations that composes a refresh transaction. This data is concatenated in a string and encapsulated inside a message, which is written into a socket towards the network module.

Log monitoring and message propagation within Mas_Imm is implemented using two nested loops. In the internal loop, a value of *twrite* is read to determine when the next write occurs. In the external loop, a value of *tstart* is read to determine when the next update transaction occurs. Each time a loop is executed, a time value is read and used to set a clock value. Whenever the clock times out, it indicates that it is time to propagate an operation. Each propagated message carries the relevant data such as $master\_id$, TID, and the description of the executed operation. This data is concatenated in a string and encapsulated inside a message, which is written into a socket towards the network module.

5.2.3 Network Module

The Network module (see Figure 5.4) is implemented within the *Network* class which inherits from both Thread and DatagramSocket classes. The Network class implements the reading of messages that were propagated by each Master process. Messages are read in FCFS (First Come First Served) order. The size of each message is used to calculate the network delay introduced to propagate a message. Afterwards, a clock is set with the network delay value and whenever the clock time expires, the message is written in reception order into a socket towards the slave module.

5.2.4 Slave Module

The slave module is implemented within three sub-modules which implements each one the *Receiver*, *Refresher*, *Deliverer* components. In addition, the *Query* sub-module implements query activities. In the following, we present each sub-module.

Receiver

The Receiver sub-module is implemented as a class which inherits from both Thread and DatagramSocket classes (see Figure 5.5). It implements the reading of messages propagated by the network module. Messages are read from a socket in FCFS order. In addition, the Receiver class implements all the methods necessary to read, insert and delete messages from the pending queues. Each time a message is read from the socket, it is written into the correct pending queue. Each pending queue is implemented using a relational table of the underlying database system. Hence, the receiver class uses JDBC to manage the database. Each tuple of a pending queue table stores relevant information that describes an operation or a sequence of operations. The schema of a pending queue table has the following attributes:

$$TID, OID, < sequence\_op >, Master\_id$$

The TID corresponds to update transaction T's identification which corresponds to $T's$ *start time* value and OID is the operation identification which corresponds to the time value at which an operation was executed at its master node. $sequence\_op$ is a list of which each element contains the following sub-attributes:

$$op\_type, tuple\_id, field\_id, new\_value$$

When op_type refers to a *commit* operation, the OID value is used to store T's timestamp value. The order of the elements of $sequence\_op$ corresponds to the serial order in which writes were performed at the master. The choice to use tables to implement pending queues is in accordance with the transaction monitoring solutions presented in [GR93].

With *immediate* propagation, messages originated from a common update transaction T are easily identified because they all have the same transaction identifier value. In addition, the $sequence\_op$ will always contain a single operation that may be a *write*, *commit* or *abort*. Finally, even if the pending queue is implemented as a relational table, it is managed as a queue because the *select* command is used to select the next message with minimun TID value when using *deferred* propagation or minimun OID when using *immediate* propagation. Since messages are propagated using a reliable FIFO broadcast protocol, it is guaranteed that messages with earlier TID or OID values are not in their way to be received whenever a select command is issued.

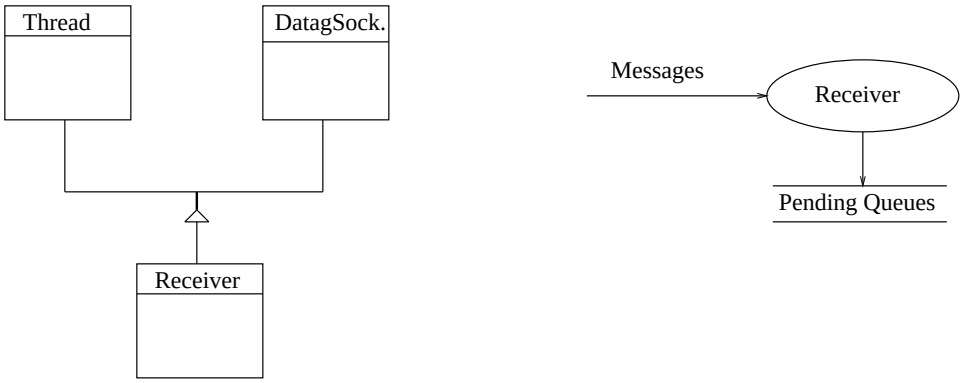


Figure 5.5: Receiver Sub-Module Class and Functional Model

Refresher

Figure 5.6 shows the set of classes used to implement Refresher sub-module. The Refresher class inheritates from the Thread class and invokes three other classes: *Def_Imm*, *Imm_Imm* and *Imm_Wait* (see Figure 5.7). Each class implements the scheduler algorithm for the *deferred_immediate*, *immediate_immediate* and *immediate_wait* strategies, respectively. Each one of these classes uses JDBC to manage the underlying database system to read and delete tuples from the pending queues tables.

Within the Def_Imm class, the selection of the next message to be treated is done using the select command over the union of the sets of pending queues. Whenever a new tuple (or RT) is selected from the head of a pending queue, the scheduling procedure is activated. The scheduling of RT is implemented as follows. If RT is the most recent refresh transaction i.e, its *commit* time value is the earliest one compared to the *commit* time value of all other unscheduled refresh transactions, the timer is set for RT with Max value, otherwise RT waits to be scheduled. We show how Max is computed in the following section. Whenever the timer expires for RT, the tuple that describes the sequence of operations of RT is inserted into the running queue table and deleted from its pending queue table. Similarly to the pending queue tables, the running queue is also implemented using a relational table that is defined using the same table scheme. Therefore, tuple insertion, deletion and selection are easily implemented.

The *Imm_Imm* class is implemented as follows. The selection of the next message to be treated is done using the select command over the union of the sets of pending queues. Recall that each tuple corresponds to the description of a write operation. If *op_type* is not *commit* or *abort*, the selected tuple is subsequently inserted into the running queue table. When *op_type* is *commit* for a RT, the tuple is read from a pending queue and the scheduling procedure for RT is activated. Recall that the *commit* write message carries

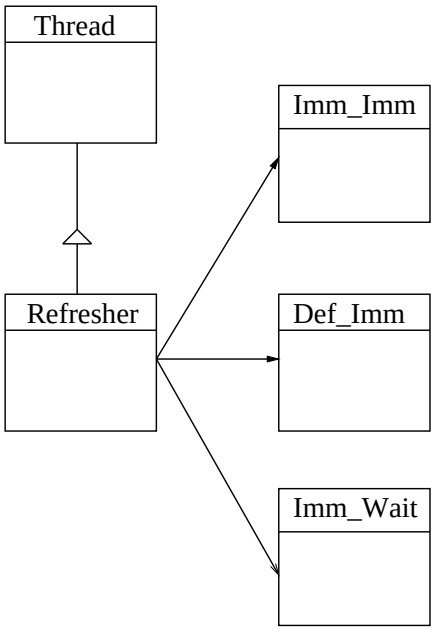


Figure 5.6: Refresher Sub-Module Classes

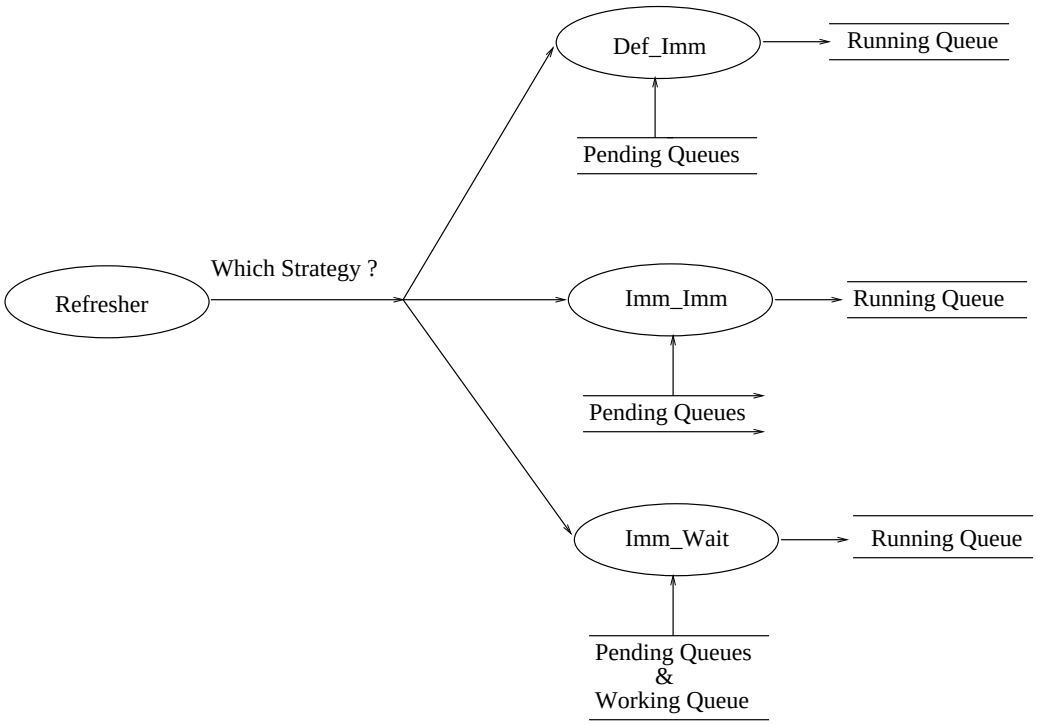


Figure 5.7: Functional Model of the Refresher Module

RT's timestamp necessary to compute Max. When the timer expires for *RT*, the *commit* tuple is inserted into the running queue table. If an *abort* is read, its tuple is simply inserted into the running queue table.

Within the Imm_Wait class, the next message to be treated is also selected using the select command over the union of the set of pending queues. If *op_type* is not a *commit* or *abort* operation; the selected tuple is written into a reception vector (noted RV). Recall that each element of this vector stores the description of a write operation related to a *RT*. The index order of the elements of RV corresponds to the serial order in which the sequence of writes were performed in the master. Whenever *op_type* is a *commit* for *RT*, the tuple content is inserted in the corresponding RV. Then, RV is written as a single tuple in the correct working queue. Each working queue is also implemented using a relational table, similarly to a pending queue. In this case, each tuple corresponds to a refresh transaction description and whenever a new tuple is inserted into a working queue table, it indicates that a complete refresh transaction *RT* was received. In addition, the scheduling activities are activated for a specific refresh transaction each time a new tuple is selected from the head of a working queue.

Deliverer

Figure 5.8 shows the set of classes used to implement the Deliverer sub-module. The Deliverer class inherits from the Thread classes and works as follows (see Figure 5.9). It keeps selecting the next operation or refresh transaction to be submitted for execution. It interacts with two other classes: the *Delv_RT* and *Delv_Op* classes. The former implements the submission of a sequence of write operations of complete refresh transaction and the later implements the submission of operations. Within Delv_RT, the next refresh transaction to be submitted corresponds to the tuple of the running queue table with minimun timestamp value. Whenever a new tuple is selected, its content is read to build a *RT* that is subsequently submitted to the local database server for execution. After the commitment of *RT*, its corresponding tuple is deleted from the running queue table. This class uses JDBC to interact with the underling database server. Notice that JDBC makes possible the connection to different database servers.

The Delv_Op class implements the procedures necessary to submit a write operation to the local database server. Since the write execution order is irrelevant among refresh transactions coming from distinct master nodes, the next operation to be submitted corresponds to the tuple of the running queue table with the minimun OID value. This selection criteria guarantees that commit operations are executed in FIFO order. In this case, refresh transaction execution is done in parallel since the order among them is defined following the order in which commit operations are executed. Therefore, each incoming *RT* may be performed using an exclusive database connection. A new refresh transaction is detected each time a new TID value is selected from the running queue table.

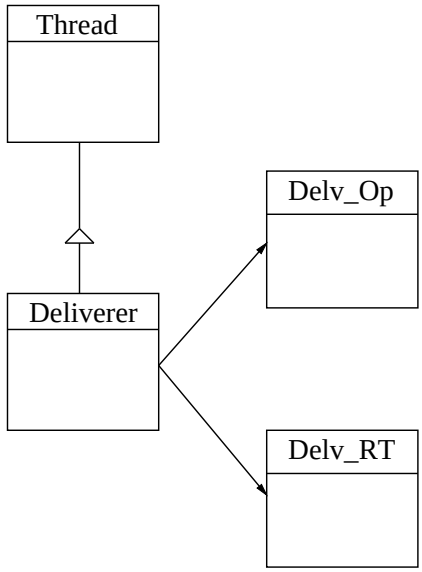


Figure 5.8: Deliverer Sub-Module Classes

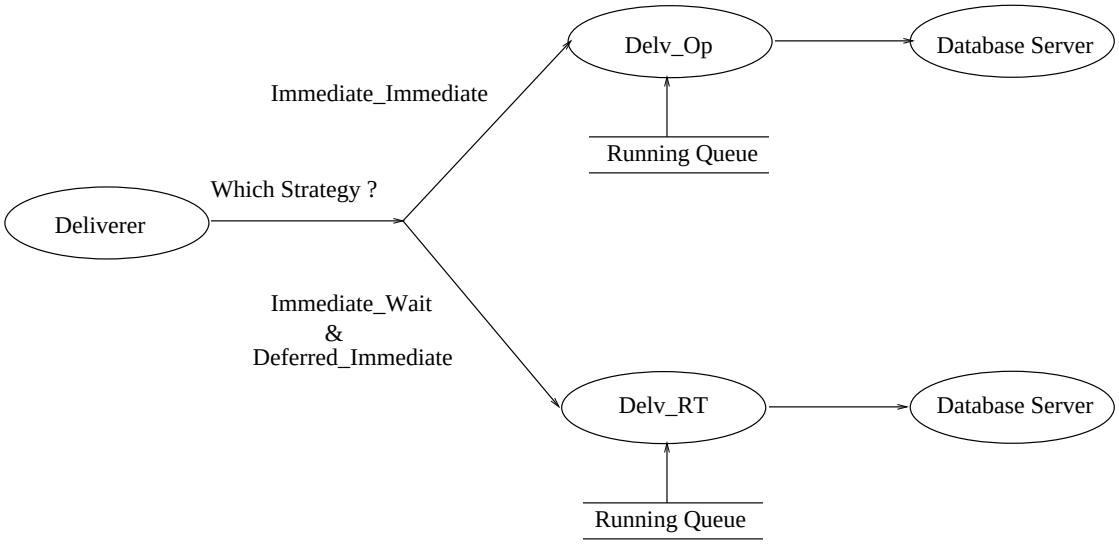


Figure 5.9: Deliverer Functional Model

Query

Finally, the Query sub-module is implemented using the Query class which inheritates from the Thread class and works as follows. Each query reads the complete set of secondary copies. The query expression we use is *Select attr from r, s ... where attr* $\geq c_1$ *and attr* $\leq c_2$, where c_1 and c_2 are fixed. The reason why we chose this expression is because we want to measure freshness when the number of master node increases. Therefore, the selection of the corresponding set of secondary copies gives the average behavior.

Similarly to the Master process, query execution is implemented in a *loop*. The sequence of intervals of time between each query is generated and stored in the *tquery* file. Each query is processed inside a loop and the next time interval is read from the *tquery* file. Whenever the clock expires, a new query is submitted again.

5.2.5 Initialization

Simulation is initialized as follows. The Master, Network and Slave modules are activated in parallel. The Sim_Master reads the master parameters from a *parameter* file (simulation parameters are the subject of the next section). Then, Sim_Master creates *nbmaster* instances of either *Mas_Def* or *Mas_Imm*. *nbmaster* corresponds to the number of master nodes. The Receiver class reads the slave simulation parameters and activates the Refresher, Deliverer and Query thread classes. The Refresher and Deliverer classes interact with the correct classes used to run a specific update propagation strategy. A simulation ends when each master process sends a message indicating the end of simulation. The Receiver exits whenever it has read the complet set of messages. It signals this event to the other processes of the slave module. The Receiver stops its execution only after the complete termination of all the processes it started.

5.3 Performance Model

In this section, we formalize the concept of *freshness* needed for our performance evaluation. Then we present the parameters we considered in our experimentations as well as the terms used to explain our results.

Freshness is formalized as follows. We may have different transactions sizes but we assume that their occurrence is uniformly distributed over time. Using this assumption of uniformity, the $shift$ of r at time t with respect to R is the difference between the number of committed update transactions of R, noted $n(R)$, and the number of committed refresh transactions on r, noted $n(r)$, at time t:

$$shift(t, r) = n(R) - n(r)$$

We define the *degree of freshness*, f, as:

$$f = 1 - shift(t, r)/n(R); f \in [0, 1]$$

Therefore, a degree of freshness close to 0 indicates that data freshness is bad, while a degree of freshness close to 1 indicates that data freshness is excellent. In the remainder of this chapter, we use the terms freshness and degree of freshness interchangeably.

Update transactions may be executed in different ways. The *density* of an update transaction T is the average interval of time (noted ϵ) between write operations in T as it is reflected in the history log. If, on average, $\epsilon \geq c$, where c is a predefined system parameter, then T is said to be *sparse*; otherwise, T is said to be *dense*. We focus on *dense* update transactions i.e., transactions with a small time interval between writes. Therefore, the time interval between writes is small. In addition, we vary transactions arrival rate distribution (noted λ_t) in history log. Updates are done on the same attribute (noted *attr*) of a different tuple. In addition, we take into account that transactions may *abort*. Therefore, we define an abort transaction percentage (noted *abr*) of 0, 5%, 10%, 20%, that corresponds to the percentage of transactions that aborts in an experimentation. For instance if $abr = 10\%$, this means that 10% of the update transactions that execute aborts after the execution of half of its write operations.

Network delay is calculated by $\delta + t$, where δ is the network delay introduced to propagate each message and t is the *on-wire* transmission time. In general, δ is considered to be non significant, and t is calculated by dividing the message size by the network bandwidth [CFLS91]. In our experiments, we use a short message transmission time (noted t_{short}), which represents the time needed to propagate a single log record. In addition, we consider that the time spent to transmit a sequence of log records is linearly proportional to the number of log records it carries. The network overhead delay to propagate each message is modeled by the system overhead to read and write from sockets. The *Total propagation time* (noted t_p) is the time spent to propagate all log records associated with a given transaction. Thus, if n represents the size of the transaction with immediate propagation, $t_p = n \times (\delta + t_{short})$, while with deferred propagation, $t_p = (\delta + n \times t_{short})$. *Network contention* occurs when δ increases due to the increase of traffic on the network. In this situation, the delay introduced by δ may impact the total propagation time, specially with immediate propagation.

We define as *refreshment time* the time spent to execute an RT and the *update propagation time* is defined as the time delay between the commitment of RT at the slave and the commitment of its corresponding T at the master. Query arrival rate distribution (noted λ_q) is exponential and query size is fixed to 5.

To measure freshness, we fix a 50% conflict rate for each secondary copy because it gives high chances to have conflicting queries and refresh transactions. This means that each refresh transaction coming from distinct master nodes update 50% of the tuples, of each seconday copy, that are read by a query. Each time an update transaction commits, a database variable, called *version_master*, is incremented. Similarly, each time a refresh transaction commits another database variable, called *version_slave*, is incremented. For each query, the degree of freshness is computed by subtracting *version_slave* from *version_master*.

We compare the impact of using two concurrency control protocols on query response time. One protocol is the strict two phase locking protocol (henceforth, S2PL) which may

ϵ	Density of a T_i
λ_t	mean time interval between Trans.
λ_q	mean time interval between Queries
$nbmaster$	Number of Master nodes
$\|Q\|$	Query Size
$\|RT\|$	Refresh Transaction Size
ltr	Long Transaction Ratio
abr	Abort Ratio
Protocols	Concurrency Control Protocols
t_{short}	Prop. Time of a single record
δ	Net. Delay to prop. a message
t_p	Total propagation Time

Table 5.1: Definition of parameters

Parameters	*Values*
ϵ	$mean = 100ms$ (dense)
λ_t	*low: (*$mean = 10s$*)*,
	bursty: ($mean = 200ms$*)*
λ_q	Exponential: *low (*$mean = 15s$*)*
$\|Q\|$	5
$\|RT\|$	5; 50
$nbmaster$	1 to 8
Conflict	50%
Protocols	S2PL, Multiversion
t_{short}	20ms and 100ms
ltr	0; 30%; 60%; 100%
abr	0; 5%; 10%; 20%

Table 5.2: Performance Model

increase response time when a query conflicts with a refresh transaction. Since, Oracle 7.3 does not implement Strict Two Phase Lock (S2PL) we simulate it by using Oracle's *select for where* command followed by *for update*.

The second protocol is a multiversion protocol, which is implemented in different ways [BHG87] by several commercial database systems. The main idea of a multiversion protocol is to increase the degree of concurrency between transactions through a mechanism that permits the execution of both a query and a transaction in a conflict situation. Here we focus on the *Snapshot Isolation* based multiversion protocol available in Oracle 7.3. In this protocol, a transaction T executing a read operation on a data item always reads the most recent version of that data item that has been committed before the beginning of T, later called *Start_ Timestamp* of T. Therefore, T reads a *snapshot* of the database as of the time it started. Updates performed by transactions that are active after the time *Start_ Timestamp* are invisible to T. It is important to notice that using a *Snapshot Isolation* multiversion protocol, queries never conflict with refresh transactions.

We define two types of update transactions. Small update transactions have size 5 (i.e., 5 write operations), while long transactions have size 50. To understand the behavior of each strategy in the presence of short and long transactions, we define four scenarios. Each scenario determines a parameter called *long transaction ratio* (noted *ltr*). We set *ltr* as follows:

- in scenario 1, $ltr = 0$ (all update transactions are short),
- in scenario 2, $ltr = 30$ (30 % of the executed update transactions are long),
- in scenario 3, $ltr = 60$ (60 % of the executed update transactions are long),
- in scenario 4, $ltr = 100$ (all transactions are long).

Whenever $ltr > 0$, the value of Max is calculated using the average time spent to propagate a long transaction ($50 * t_short$). On the other hand when $ltr = 0$, the value of Max is calculated using the average time spent to propagate a short transaction ($5 * t_short$). Refresh transaction execution is performed on top of an Oracle 7.3 system using C/SQL. For simulation purposes, each write operation corresponds to an UPDATE command that is submitted to the server for execution. The definition and values of the parameters of the performance model are summarized in Tables 5.1 and 5.2. The results are average values obtained for the execution of 40 update transactions.

5.4 Performance Evaluation

The goal of our experimentations is to understand the behavior of the three propagation strategies on a *low* and *bursty* scenarios since these workloads are typical of the applications introduced in Chapter 2. The first experiment presents our results for the *low* workload and the second one for the *bursty* workload. In both scenarios, for pedagogical reasons, we

Experiment	*Measure*	*Vary*	*Worklaod*	*Num. Masters*
1	Freshness & Query Response Times	ltr	Low	Up to 8
2	Freshness & Query Response Times	ltr	Bursty	Up to 8
3	Freshness	abr	Bursty	1
4	Freshness & Query Response Times	Net. Delay	Bursty	1

Table 5.3: Experiments Goal

sometimes present our results for a 1Master-1Slave configuration first then for a 2Master-1Slave, 3Master-1Slave, etc, up to 8 masters. In the third experiment, we study the impact on freshness when update transactions abort. In the fourth experiment, we verify the freshness improvement of each strategy when the network delay to propagate a message increases. Finally, we discuss our results. Table 5.3 summarizes each experiment.

5.4.1 Experiment 1

The goal of this experiment is to analyze the query response times and the degree of freshness obtained for a *low* update transaction arrival rate at each master node.

Freshness (1Master-1Slave)

We now analyze the degree of freshness obtained for different values of ltr (long transaction ratio). As despicted in Figure 5.10, for 1 master when $ltr = 0$, the degree of freshness is almost 1, i.e., replicas are almost mutually consistent with the three strategies. The reason is that on average, $\lambda_t \simeq t_p$, that is the time interval between the execution of a T_i and a subsequent T_{i+1} is sufficiently high to enable completion of T_i's update propagation before the commitment of T_{i+1}. However, for higher ltr values, $\lambda_t < t_p$ for the three strategies. Thus, during T_i's update propagation, some transactions $T_{i+1}...T_{i+n}$ may be committed, thereby decreasing the degree of freshness. For all values of ltr, the degree of freshness obtained with *deferred_immediate* and *immediate_wait* are close because the refreshment time is near equal for these two strategies. Furthermore, the total propagation times are also close since there is no network contention.

With *immediate-immediate*, refreshment time of a RT_i is greater than that of other strategies because the time interval between the execution of two write operations, w_j and w_{j+1}, of RT_i, is impacted by t_{short}, δ and ϵ, slowing down refreshment time. However,

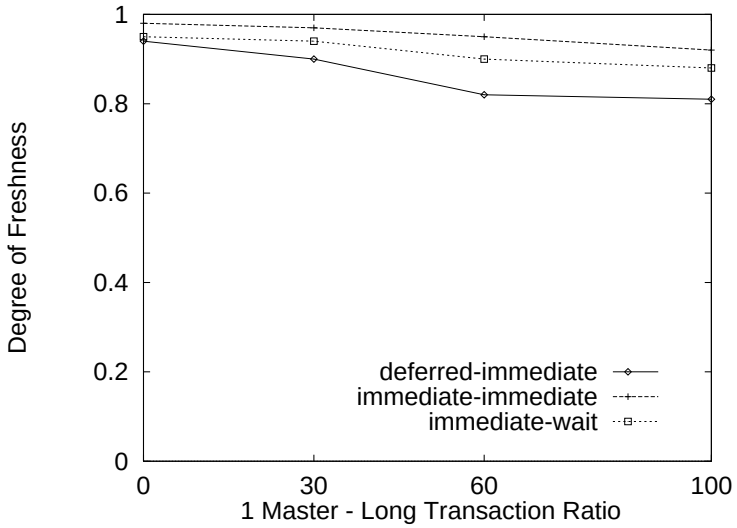


Figure 5.10: Low Workload - Degree of Freshness

immediate-immediate update propagation time is smaller, compared to *immediate-wait* and *deferred-immediate*, because propagation and refreshment are done in parallel. That is, RT_i's execution starts after the reception of the first write done by T_i. Therefore, *immediate-immediate* is the strategy that always yields the best degree of freshness. For all strategies, the degree of freshness does not vary linearly with *ltr* since we are mixing transaction sizes and our freshness measure is based on transaction size.

Freshness (nMasters-1Slave)

We now analyze the degree of freshness when there are more than one master node. As shown in Figure 5.11, for two masters the impact on the degree of freshness is already noticed with $ltr = 0$. The reason is that the number of messages in the network increases due to the increase of master nodes, therefore network contention may increase the total propagation time. As a consequence, $\lambda_t < t_p$ for all master nodes and during a RT_i's update propagation, some transactions $T_{i+1}...T_{i+n}$ may be committed at each master node, thereby decreasing the degree of data freshness for each secondary copy. This behavior is perceived when $nbmasters = 2$, $nbmasters = 4$, $nbmasters = 6$ and $nbmasters = 8$ (see Figure 5.11).

For all experiments, it is clear that *immediate-immediate* performs better than the two other strategies because the effects of network contention are reduced due to the parallelism

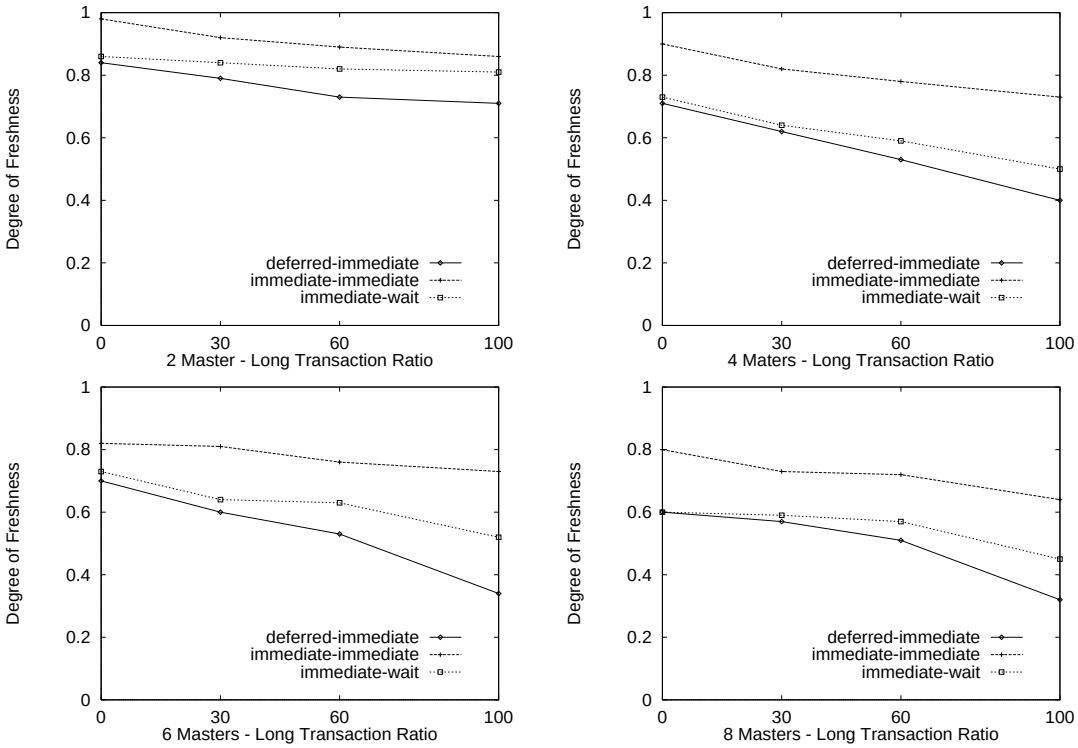


Figure 5.11: Low Workload - Degree of Freshness

of log monitoring, propagation and refreshment. In addition, *immediate-wait* always performs clearly better than *deferred-immediate*. For instance, with 2 masters and $ltr = 30$, the degree of freshness is 0.78 when using *deferred-immediate*, 0.8 when using *immediate-wait* and 0.93 when using *immediate-immediate*. With 4 masters, the values of freshness obtained with $ltr = 100$ is 0.78 with *immediate-immediate*, 0.56 with *immediate-wait* and 0.41 with *deferred-immediate*. In this case, *immediate-immediate* performs almost *two* times better than *deferred-immediate*. With 6 masters and $ltr = 100$, *immediate-immediate* may perform 2.5 better than *deferred-immediate* and *immediate-wait* may perform 1.7 better than *deferred-immediate*. With 8 masters and $ltr = 100$ the values of the degree of freshness is 0.68 with *immediate-immediate*, 0.45 with *immediate-wait* and 0.32 with and *deferred-immediate*.

As the number of masters increases and the values of ltr grows, the degree of freshness decreases much more with both *deferred-immediate* and *immediate-wait* compared with *immediate-immediate*. The reason is that network contention increases due to the higher number of master nodes and the increase of transaction sizes. Therefore, total propagation time also increases for the three strategies. However with *immediate-immediate*, this effect is reduced due to the parallelism of tasks.

For all experiments, as the number of masters increases, for $ltr = 60$ and $ltr = 100$, the improvements introduced by *immediate-wait* begin to be significant compared to *deferred-immediate*. For instance, with 6 masters for $ltr = 100$, the degree of freshness is 0.33 for *deferred-immediate* and 0.52 for *immediate-wait*. The reason for improvement is that the time spent to log monitor a complete update transaction with *deferred-immediate* increases, thereby increasing the average total propagation time. This type of effect does not happen with *immediate-wait* because log monitoring and propagation are done in parallel.

Response Time (1Master-1Slave)

We now analyze the response time for a single master node. With *immediate-immediate*, query response time may increase whenever a query conflicts with a refresh transaction because propagation and refreshment are done in parallel. Therefore, refreshment time is impacted by the total propagation time. However, for low workloads, the chance of conflicts is reduced because $\lambda_t \simeq \lambda_q$. That is the reason why the mean query response times are not seriously affected.

Response Time (nMaster-1Slave)

We now report the impact on query response times when the number of master nodes is greater than one. As the number of master nodes increases, the number of concurrent refresh transactions updating distinct secondary copies augments and the chances of conflicts between queries and refresh transaction also increases. Thus a query that reads the complete

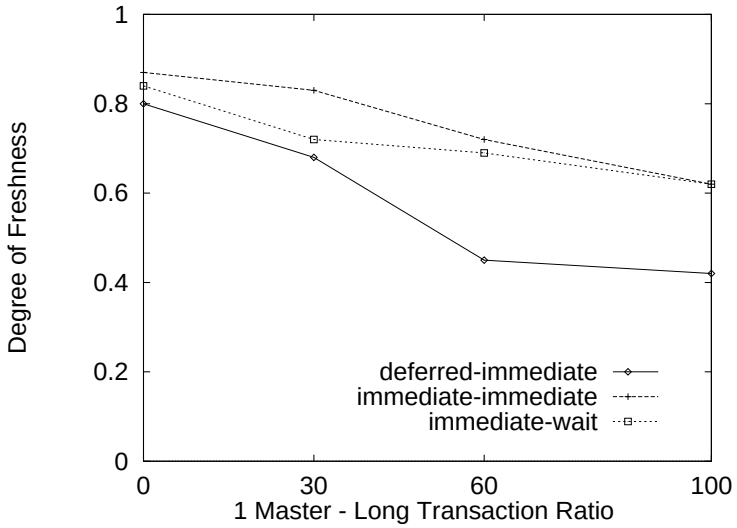


Figure 5.12: Bursty Workload - Degree of Freshness

set of secondary copies may be blocked during a significant period of time. In addition, the total propagation time augments due to the increase in the number of messages towards the same slave node. Therefore, with *immediate-immediate*, the query response times may be significantly affected. For instance, with 6 and 8 master nodes, the response times obtained with *immediate-immediate* are almost two times the response times obtained with *immediate-wait* and *deferred-immediate*.

5.4.2 Experiment 2

The goal of this experiment is to analyze the degree of freshness and the query response times obtained for the three strategies for a *bursty* transaction arrival rate.

Freshness (1Master-1Slave)

As despicted in Figure 5.12, when $ltr = 0$ (short transactions), the degree of freshness is already impacted because on average, $\lambda_t < t_p$. Therefore, during T_i's update propagation, $T_{i+1}, T_{i+2}...T_{i+n}$ may be commited. Notice that even with the increase of network contention, due to the type of workload, *immediate-immediate* yields a better degree of freshness than both *deferred-immediate* and *immediate-wait* for the same reasons mentioned for the low workload.

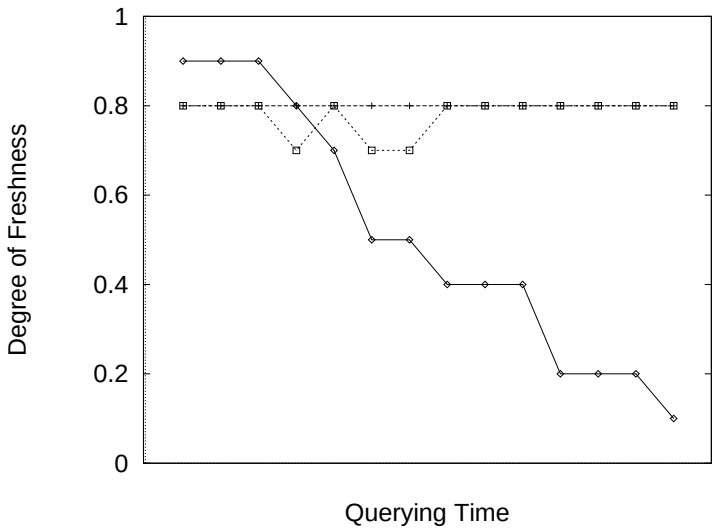


Figure 5.13: Bursty Workload - Freshness Behavior ($ltr = 100$)

When long update transactions are executed, the degree of freshness decreases because t_p increases and $\lambda_t << t_p$. Notice that *immediate-wait* begins to improve and becomes clearly better than *deferred-immediate* when $ltr = 60$ and $ltr = 100$. This is because q_r is quickly filled with a large number of operations such that, when the Refresher reads q_r, seeking for a new RT_i, it may happen that, all operations associated with RT_i may have been already received and stored in q_r. In this case, the additional wait period of *immediate-wait* may be reduced to 0. This is cleary seen when $ltr = 100$ (all refresh transactions are long). Figure 5.13 shows the values of the degree of freshness for sequences of query executions when $ltr = 100$ using the three strategies. The degrees of freshness obtained with *immediate-immediate* and *immediate-wait* are almost equal. On the other hand, the degree of freshness of *deferred-immediate* decreases rapidly because there is no parallelism of tasks as with the immediate strategies. So, when update transaction sizes increase, update propagation times rise much more compared to the immediate strategies. This impacts the degree of freshness much more seriously.

Freshness (nMaster - 1Slave)

We now repport on the results obtained when the number of masters is greater than one. For small values of ltr (0, 30), *immediate-wait* and *deferred-immediate* perform similarly, as shown in Figure 5.14. Notice that *immediate-wait* may even give a worse degree

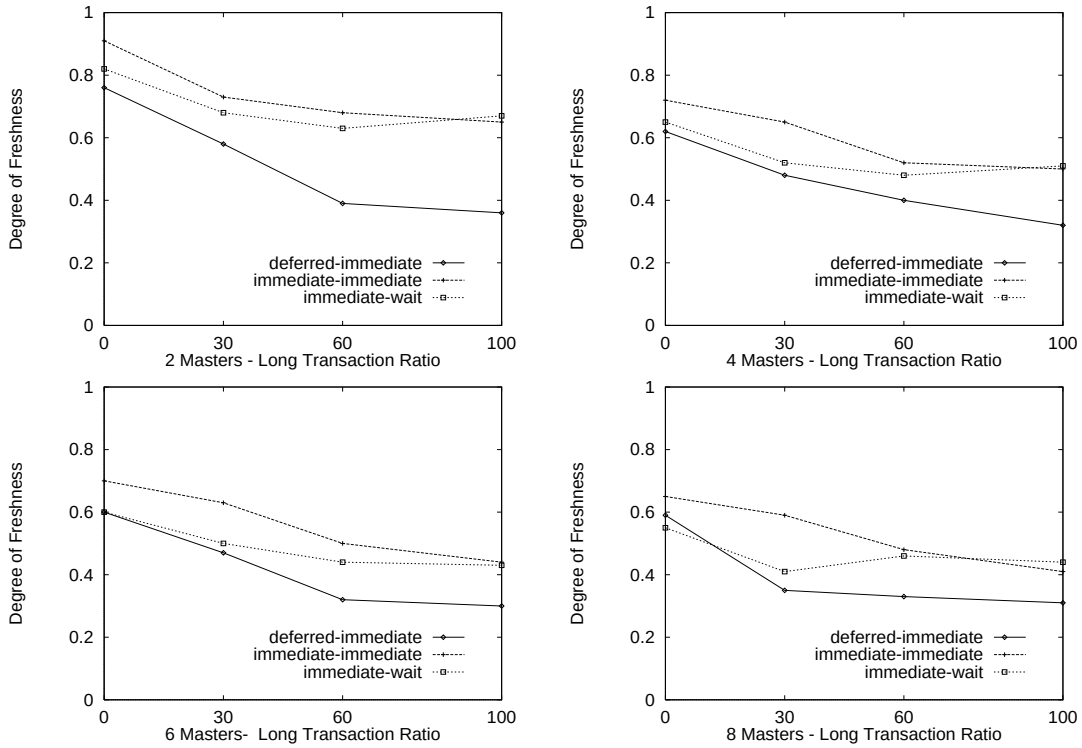


Figure 5.14: Bursty Workload - Degree of Freshness

of freshness compared to *deferred-immediate* because δ increases sufficiently to increase the *immediate* total propagation time. Therefore, the total propagation time of a short transaction using *deferred* propagation may be less than the total propagation time using *immediate* propagation. An example of this situation happens with 8 master nodes when $ltr = 0$.

With the increase of the values of ltr and the number of master nodes, *immediate-wait* begins to perform much better then *deferred-immediate*. For instance, with 4 masters with $ltr = 30$, the degree of freshness is 0.62 for *immediate-immediate*, 0.51 for *immediate-wait* and 0.39 for *deferred-immediate*. With 6 masters and $ltr = 60$, the degree of freshness is 0.55 for *immediate-immediate*, 0.48 for *immediate-wait* and 0.31 for *deferred-immediate*. The reason why *immediate-wait* performs better is the same as before. That is, each pending queue is quickly filled with a large amount of messages and whenever the refresher reads the first operation of a refresh transaction RT, all the other operations of RT may be already stored inside the corresponding pending queue. Therefore, the time spent to wait for the commit message, necessary to start RT execution, may be reduced or even become 0.

Notice that *immediate-wait* may even perform better than *immediate-immediate*. For instance, with 2, 4, and 8 masters with $ltr = 100$, the results shown in *immediate-wait*

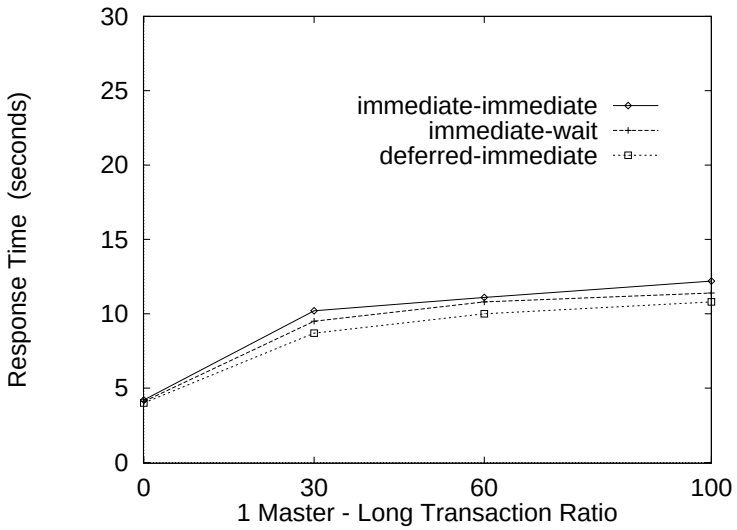


Figure 5.15: Bursty Workload - Response Time

are better then those shown by *immediate-immediate*, as despicted in Figure 5.14. The reason for this improvement is the following. With *immediate_ wait* the sequence of write operations carried by a refresh transaction RT is submitted within a single submission request to the local database server. Instead, with *immediate-immediate*, each time a write operation is received at a slave node, a submission request is issued by the deliverer to the local transaction manager. In a bursty workload with $ltr = 100$, the number of request submissions augments because λ_t is small and the number of operations is high. In addition, as the number of master nodes increases, more submission requests are done in parallel, instead of serially. Hence the refreshment load of the slave may be very high and the time to have a submission request serviced augments. Therefore, with *immediate-immediate* the time spent to execute a refresh transaction may increase, diminishing the degree of freshness for each secondary copy.

Response Time

With one master node, response times are not significatly impacted because the chances of conflict are reduced (see Figure 5.15). As the number of master nodes augments, response time may increase even more because the chances of conflicts are higher due to the increase of concurrency bewteen refresh transactions updating several secondary copies and queries reading them (see Figure 5.16), in addition, $\lambda_t << \lambda_q$. Therefore, the probability of having

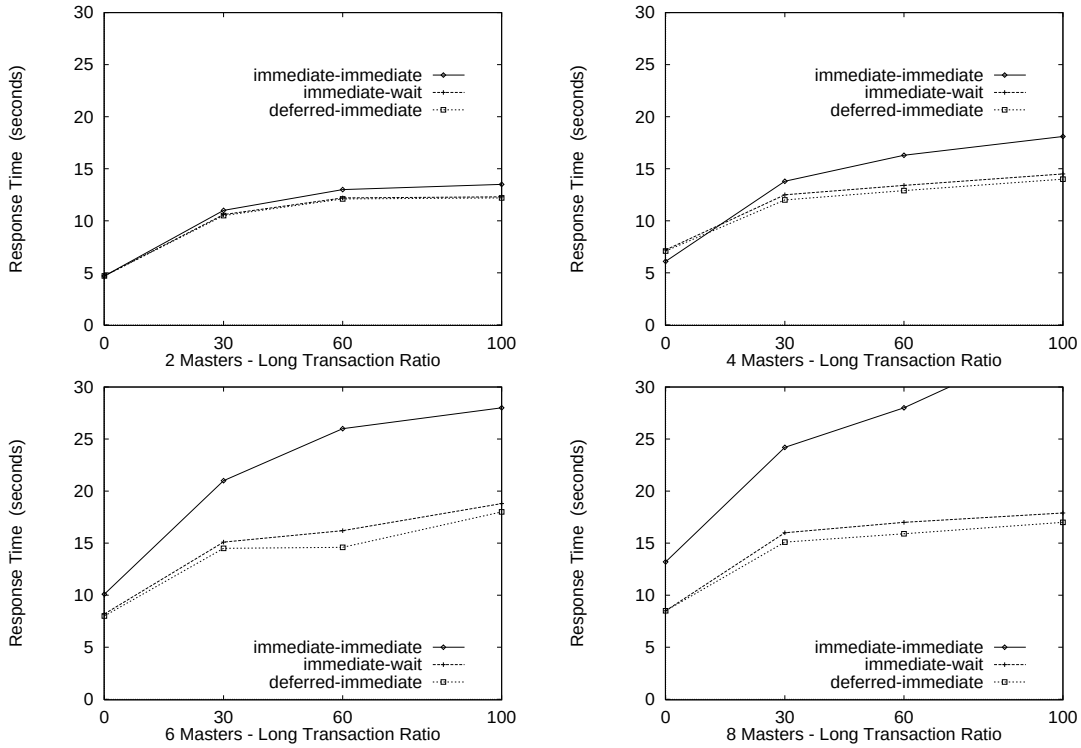


Figure 5.16: Bursty Workload - Response Time

a query blocked increases. Here the effects of the increase of refreshment time introduced by *immediate-immediate* are strongly perceived. With 8 masters for $ltr = 0$, $ltr = 30$, $ltr = 60$ and $ltr = 100$ a query may take 14s, 24s, 26s and 35s to be processed.

5.4.3 Experiment 3

The goal of this experiment is to show the effects of transactions *aborts* on the degree of freshness values. We consider a single master node in a bursty workload.

As shown in Figure 5.17, for $ltr = 0$ and various values of *abr* (5,10,20), the decrease of freshness introduced by update transactions that abort with *immediate-immediate* is insignificant. In the worst case, it achieves 0.2. This behavior is the same for other values of ltr (30, 60, 100). The same behavior is also observed for *immediate-wait*. These results show that the time spent to discard the reception vectors and *undo* refresh transactions at a slave node are insignificant compared to propagation time. Therefore, for the immediate strategies, we may safely state that freshness is not affected in case of update transaction aborts.

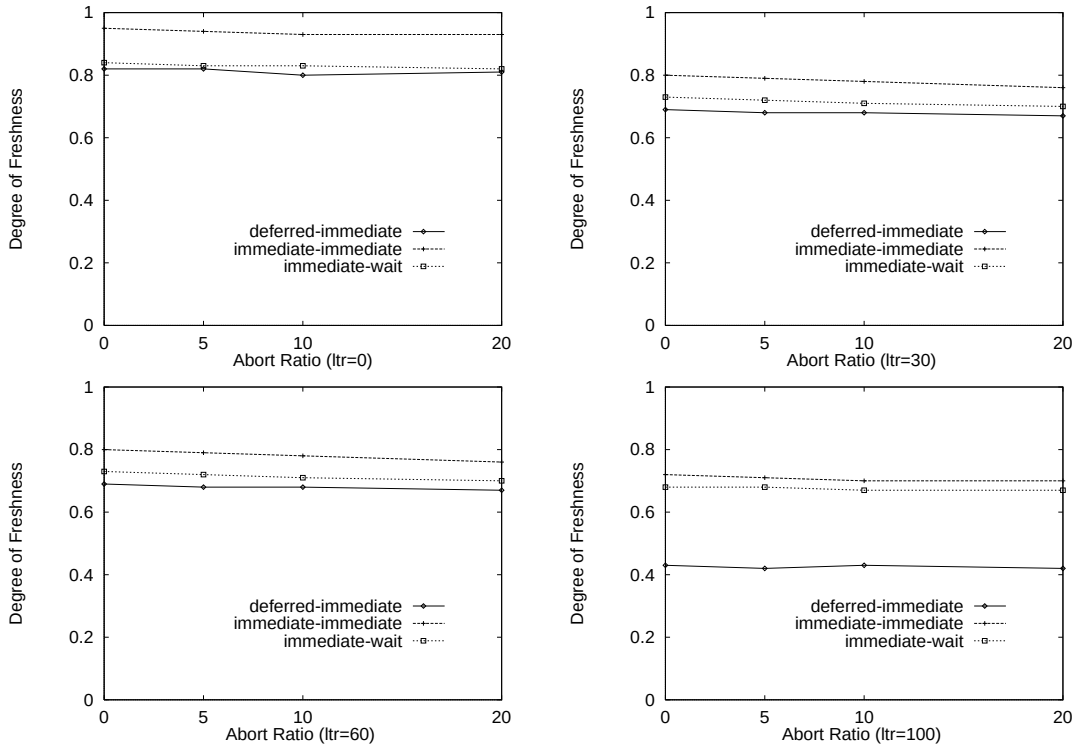


Figure 5.17: Bursty Workload - *Abort* Effects

With *deferred-immediate*, the degrees of freshness may even increase because no processing is initiated at the slave node until the complete commitment and propagation of an update transaction. Therefore, while a transaction is aborting at the master node, the slave node may be catching up.

5.4.4 Experiment 4

The goal of this experiment is to show the impact on the degree of freshness and query response times values when the network delay necessary to propagate a message increases. We consider a single master node in a bursty workload.

Figure 5.18 compares the freshness results obtained when $\delta = 100ms$ and $\delta = 20ms$ (δ denotes the network delay to propagate a message). When $\delta = 20$ and $ltr = 100$ *immediate-immediate* improves 1.1 times better than *deferred-immediate* and when $\delta = 100$, *immediate-immediate* improves 5 times better. The improvements brought by *immediate-wait* compared to *deferred-immediate* are close to these two results. Similar results are acheived when $ltr = 60$. In addition, in the presence of long transactions, the decrease of freshness when

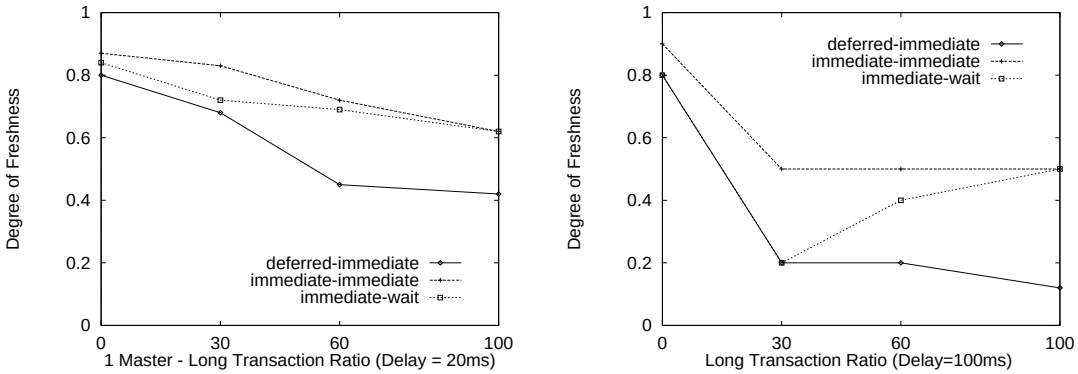


Figure 5.18: Increase of Network Delay - Degree of Freshness

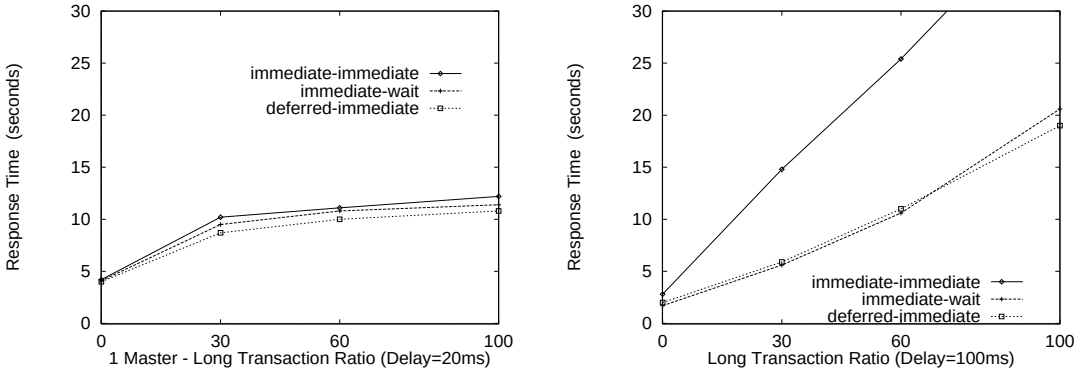


Figure 5.19: Increase of Network Delay - Response Time (1 Master)

$\delta = 100ms$ is significantly higher for *immediate-wait* and *deferred-immediate* compared to the results obtained when $\delta = 20ms$. This is because the higher the value of δ is, the longer it takes to propagate a refresh transaction and in average the values of t_p may be much higher than λ_t. Finally, these experiments confirm the benefits of having tasks being performed in parallel when using *immediate_ immediate*.

Figure 5.19 compares the response times results obtained when $\delta = 100ms$ and $\delta = 20ms$. Whenever the value of δ increases, the locking holding time in conflict situations augments because the refreshment time is proportional to propagation time. Notice that when $\delta = 100ms$ the increase of query response is high when $ltr = 30$ and when $\delta = 20ms$ this impact may be insignificant.

5.4.5 Discussion

We now summarize and discuss the major observations of these experiments. With low workloads, the degree of freshness for one master node is slightly impacted if update transactions are dense and long. In this case, *immediate-wait* and *deferred-immediate* give similar degrees of freshness because their total propagation times are close. *Immediate-immediate* is the strategy that gives the best degree of freshness because propagation and refreshment are done in parallel. *Immediate-immediate* is the only strategy that may introduce an increase in query response times because propagation and refreshment are done in parallel. Therefore, in conflict situations, the increase of query response times may depend on t_{short}, ϵ and δ. However, the mean query response time is not seriously affected because $\lambda_q > \lambda_t$. Therefore, the chances of conflicts are small.

In low workloads, with the increase of the number of master nodes and *ltr*, *deferred-immediate* is the strategy that degrades the most from network contention because there is no parallelism of tasks. In this case, *immediate-wait* may improve by 15% over *deferred-immediate*. Finally, the improvements brought by *immediate-immediate* do not degrade significantly as the number of masters increases because the workload is low and the effects of network contention is reduced.

When update transactions arrive in bursts, the degree of freshness decreases much more in the presence of long transactions. The *immediate-immediate* strategy still yields the best degree of freshness even with network contention. When the value of *ltr* grows, *immediate-wait* gives results close to those of *immediate-immediate*. When all transactions are long, *immediate-wait* performs like *immediate-immediate* because each pending queue is quickly filled with operations due to the immediate propagation. Thus, the effect of the *wait period* of *immediate-wait* may be eliminated because the operations that compose a refresh transaction RT may be already available in q_r whenever the refresher seeks for a new refresh transaction. With *deferred-immediate* the degree of freshness is much more impacted, compared to *immediate-immediate* and *immediate-wait* because there is no parallelism advantage in improving the degree of freshness.

With the increase in the number of master nodes and *ltr* values, the delay introduced by network contention may increase significantly the propagation time for the three strategies. Again, *immediate-wait* performs like *immediate-immediate* and when $ltr = 100$, *immediate-wait* may even perform better then *immediate-immediate*. The reason is that the overhead to submit each refresh transaction operation with *immediate-immediate* increases due to the number of secondary copies to be refreshed. Therefore, refreshment time may increase.

With *immediate-immediate*, the mean query response time may be seriously impacted by the parallelism of propagation and refreshment. When $\lambda_q >> \lambda_t$, the chances of conflicts increase much more compared to the case of low workload. Query response times are much lower with *immediate-wait* because there is no parallelism between propagation and refreshment. Here, the network delay to propagate each operation has an important role and the higher its value, the higher are the query response times in conflict situations. In any case, the use of a multiversion protocol on the slave node may significantly reduce query response times, without a significant decrease in the degree of freshness.

Workload	*Transaction Size*	*Best Degrees of Freshness*
Low	Short	Immediate-Immediate
Low	Long	Immediate-Immediate
Bursty	Short	Immediate-Immediate
Bursty	Long	Immediate-Immediate & Immediate-Wait

Table 5.4: Most Relevant Results

The *abort* of an update transaction with *immediate-immediate* and *immediate-wait* does not impact the degree of freshness since the delay introduced to undo a refresh transaction or discard a reception vector, respectively, are insignificant compared to the propagation time. Finally, the improvements brought by *immediate-immediate* and *immediate-wait* are much more significant when the network delay to propagate a single operation augments. This clearly shows the advantages of having tasks being executed in parallel.

5.5 Conclusion

In this chapter, we described the simulation environment used to validate and analyze the performance of our *immediate_ immediate*, *immediate_ wait* and *deferred_ immediate* update propagation strategies for a data consolidation configuration. Note that short transaction sizes correspond to long ratios (*ltr*) values of 0 and 30, and long transactions correspond to the other values of *ltr* (60,100). The scheduling algorithm for each strategy was also validated. The simulation environment consists of 3 modules; Master, Network and Slave, and is implemented using Java/JDBC and Oracle 7.3. We detailed the simulation/implementation solutions for each module. The performance model we proposed isolates the main factors that may impact the degree of freshness such as update transactions sizes and arrival rates, network delay and query arrival rate.

We performed four experimentations that reveal the improvements of the *immediate* update propagation strategies compared to the *deferred* strategy. The principle of our experimentation was to exhaustively evaluate the three strategies for bursty and low workloads by varying the transaction arrival rates and check the impacts on the degree of freshness and query response times.

The main performance evaluation results are summarized in Table 5.4. The *immediate-immediate* is the strategy that always improves freshness even with network contention. For low workloads, *immediate-immediate* shows significant improvements compared to the

other two strategies as the number of masters increases. For instance with 8 masters, it may improve two times over *deferred-immediate* and 1.5 over the *immediate-wait*. In bursty workloads, even with a few master nodes, *immediate-immediate* already shows significant improvements. On the other hand, in low workloads, *immediate-wait* begins to perform significantly better than *deferred-immediate* as the number of master nodes and the transaction size increase. Query response times may be affected when using *immediate-immediate* because refresh transaction execution time is proportional to its propagation time. This effect may be dramatic for long distance networks. However, the use of a multiversion concurrency control protocol can easily solve this problem.

We showed that update transaction aborts do not degrade the degree of freshness for the three strategies because the time spent to discard the reception vectors and *undo* refresh transactions at a slave node are insignificant compared to propagation time. Therefore, we may safely state that freshness is not affected in case of update transaction aborts.

The increase of network delay may make *immediate-immediate* improve much better than *deferred-immediate* (by a factor 5) because the total propagation time increases, thereby giving time for the commitment of several update transactions. This result is also true for *immediate-wait* when update transactions are long.

Finally, *immediate-immediate* shows the best freshness results in all experiments and is the strategy that should always be used by applications that need to acheive high degrees of freshness, i.e., when the difference between the number of commited updates performed at a primary copy and the number of commited updates performed at a secondary copy must be minimal.

Chapter 6

Conclusion

Many distributed database applications replicate data to improve data availability and query response times. Several existing update propagation strategies may be used to refresh replicated data. Some of them guarantee mutual consistency of replicated data but do not provide good performance. Furthermore, they may be blocking in case of failures. Lazy replication has been used as an alternative solution in several types of applications such as on-line financial transactions and telecommunication systems and others where mutual consistency may be relaxed

In this case, replicated data may remain different for certain time intervals, or during specific operations without compromising the application semantics. However, for several types of applications even with improved performance users want to have data as *fresh* as possible. This means reducing the deviation between the updates committed on a primary copy and the updates committed on its secondary copies.

In this thesis, we focused on a special lazy replication scenario called lazy master. In this case, for each replica copy, there is only one primary copy. We concentrated on the problems that appear in lazy master replication schemes with requirements of high performance and fresh data. These problems have been receiving increased attention in different domains such as Internet, mobile computing, data warehousing and others, and a deeper understanding of how replication techniques may be used to solve some open problems is missing. Each chapter of this thesis has contributed to address these problems.

In chapter 2, we gave the architectural framework to define distributed databases and transactions. In addition, we identified the main failures that may happen in a database. Then, we defined replicated databases and showed how replicated databases fit in a data warehouse scenario. Next, we identified six classes of update propagation strategies used to update replicated data: Synchronous All, Synchronous Available, Quorum-based, Lazy Master, Primary/backup and Lazy Group. Each class has an invariant condition and is suited for specific types of applications. In addition, we paid special attention on lazy master solutions since several terms and concepts have been used in the subsequent chapters of this thesis. We presented the industrial solutions for data replication. It is clear that most of

the available systems use *log based* capture methods to implement update propagation and that the major advantage of the *push* approach is to make possible to achieve *near-real-time* update propagation. Next, we introduced the concept of fault tolerance protocols used to keep replica copies consistent. Finally, we presented the current trends in data replication showing that it is used as a solution for different types of distributed applications such as data warehousing and mobile computing.

In Chapter 3, we defined a lazy master replication framework that is composed of five basic parameters: ownership, configuration, transaction model, propagation and refreshment. The ownership specifies the rights of updating replica copies. The transaction model is related to the properties of transaction that manages replica copies. In this thesis, we used a fixed transaction model for simplicity. However, different transaction models may be used. The propagation parameter defines when updates on primary copies must be propagated towards its secondary copies. Furthermore, the refreshment parameter establishes when and in which order updates must be applied to secondary copies. The combination of a propagation strategy and refreshment parameters defines an update propagation strategy.

The configuration parameter describes how replicas copies are placed at the nodes of the distributed system. The choice of a configuration may be very complex. For simplicity, we fixed three basic configurations that were based on the data dissemination, consolidation and logical partitions presented in Chapter 2. For each configuration, we identified its correctness criteria based on the properties provided by some fault tolerance protocols. Therefore, each correctness criteria establishes a delivering order criteria that reflects the order in which updates must be applied to refresh replicated data at each node.

In Chapter 4, we addressed the problems of improving data freshness and enforcing correctness in lazy master schemes. We presented a system architecture for master and slave nodes that is used by our update propagation strategies. With respect to freshness improvement, we proposed two new *immediate* strategies called: *immediate-immediate* and *immediate-wait*. With *immediate-immediate*, updates on a primary copy at some master node are immediately propagated towards the other secondary copies and a refresh transaction is started at a slave node as soon as the first write operation is received from the master node. Similarly, with *immediate-wait*, the propagation procedure follows the same principle. However, a refresh transaction is started at a slave node after the complete reception of all updates (of the same transaction) from the master node.

In the second part of Chapter 4, we presented a FIFO ordering algorithm used by the Refresher to order message delivery. This algorithm orders chronologically refresh transaction execution using their original update transaction timestamp values. Therefore, whenever clocks are synchronized, the effects of updates on secondary copies follow the same order in which their corresponding primary copies were updated. We presented three variants of the Refresher algorithm for our architecture, *deferred-immediate* Refresher, *immediate-immediate* Refresher and *immediate-wait* Refresher. Each variant is suited for a specific update propagation strategy. We presented the properties of the Refresher algorithms and showed that it is sufficient to enforce correctness for the 1Master-nSlave, mMaster-nSlave, Master-MasterSlave-Slave and hybrid configurations. We presented a non-blocking protocol

for master and slave node recovery. Finally, we related our work with others in the field and showed how it can be used as a solution to refresh a data warehouse.

In Chapter 5, we proposed implementation solutions and a simulation environment to validate and measure the performance brought by our *immediate-immediate* and *immediate-wait* update propagation strategies compared to the *deferred-immediate* strategy as well as the scheduling procedures necessary to enforce correctness. Our environment was implemented using JAVA Database Connectivity (JDBC) and Oracle 7.3 as the database server at the slave node to implement all slave database procedures.

Our performance evaluation takes into account a data consolidation configuration. We chose this configuration because it is widely used in data warehouse applications. Furthermore, it allows the evaluation of our strategies when the number of master nodes increases (we consider up to 8 master nodes). We exhaustively showed the impact of different types of parameters such as the transaction workload at each master node, the transaction sizes, the network delay to propagate a message and abort ratios. The *immediate-immediate* is the strategy that always improves freshness even with network contention. For low workloads, *immediate-immediate* shows significant improvements compared to the other two strategies as the number of masters increases. For instance with 8 masters, it may improve by two times over *deferred-immediate* and 1.5 over the *immediate-wait*. In bursty workloads, even with a few master nodes, *immediate-immediate* already shows significant improvements. On the other hand, in low workloads, *immediate-wait* begins to perform significantly better than *deferred-immediate* as the number of master nodes and the transaction size increase. The increase of network delay may make *immediate-immediate* improve much better than *deferred-immediate* (by a factor 5). This result is also true for *immediate-wait* when update transactions are long.

We conclude this chapter by giving directions for future work on data replication. Fault-tolerance and reliability in heterogeneous database systems is a relatively open area. Methods to achieve reliability through recovery protocols have been discussed in the literature, but they do not address availability. The problems of exploiting redundancy of data and achieving resilience against node and communication failures have only recently been addressed. Data replication in this case is used to manage fault-tolerance and reliability.

Mobile computing in general is still a wide open field with many competing control theories and architecture paradigms. Replication control relies on many other functions such as concurrency control, addressing, search, time management, security, etc. Most of these are still unsettled in the mobile environment.

The Bayou storage system [TTP+95] provides an infrastructure for collaborative applications that manages the conflicts introduced by concurrent activity while relying only on the weak connectivity available for mobile computing. To cope with arbitrary network partitions, the system is built around pair-wise client-server and server-server communications. To provide high availability, Bayou employs lazy group replication where clients are able to connect to any available server to perform Reads and Writes. Support for automatic conflict detection and resolution enables applications to deal with concurrent updates effectively. The system guarantees eventual consistency by ensuring that all updates even-

tually propagate to all servers, that perform updates in a global order, and that any update conflicts are resolved in a consistent manner at all servers.

Complementary to the Bayou storage system, [ADN+97] presents Bayou's protocol for lazy group replication. Three basic design decisions went into Bayou's anti-entropy protocol: the model of pair-wise reconciliation between peer replicas, the exchange of write operations stored in per-replica logs that are compactly characterized using version vectors, and the propagation of writes between replicas in an order that is closed with respect to the write's accept, causal or total order.

In this context, future work following on this thesis could be the reformulation of our framework, architecture and algorithms for lazy group replication schemes. In this case, all replica copies are primary copies. The idea would be to consider new types of configurations and transaction models and extend our algorithms to guarantee correctness. In addition, one may consider hybrid schemes, which aggregates both the characteristics of lazy master and lazy group replication schemes.

Bibliography

[AAS97] D. Agrawal, A. Ell Abbadi, and R. C. Steinke. Epidemic algorithms in replicated databases. *ACM Symp. on Principles of Database Systems (PODS)*, Tucson, May 1997.

[ABGM90] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, 15(3):359–384, September 1990.

[AD86] P.A. Alsberg and J.D. Day. A principle for resilient sharing of distributed resources. *International Conference on Software Engineering*, Los Angeles, February 1986.

[ADN+95] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Severless network file systems. *ACM Symposium on Operating Principles (SIGOPS)*, Colorado, December 1995.

[ADN+97] T. E. Anderson, M. D. Dahlin, J. M. Neefe, David A. Patterson, D. S. Roselli, and R. Y. Wang. Flexible update propagation for weakly consistent replication. *ACM Symposium on Operating Systems Principles (SIGOPS)*, Saint-Malo, December 1997.

[AK91] D. Agrawal and V. Krishnaswamy. Using multiversion data for non-interfering execution of write-only transactions. *ACM SIGMOD Int. Conf. on Management of Data*, Denver, May 1991.

[BC92] P. M. Bober and M. J. Carey. On mixing queries and transactions via multiversion locking. *Int. Conf. on Data Engineering*, Tempe, May 1992.

[BG83] P.A. Bernstein and N. Goodman. The failures and recovery problem for replicated databases. *ACM Symp. on Principles of Distributed Computing*, August 1983.

[BG84] P.A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery. *ACM Transactions on Database Systems*, 9(4), 1984.

[BGM90a] D. Barbara and H. Garcia-Molina. The case controlled inconsistency in replicated data. *Int. Workshop on Management of Replicated Data*, Houston, November 1990.

[BGM90b] D. Barbara and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *Int. Conf. on Extending Data Base Technology (EDBT)*, Venice, March 1990.

[BGV97] Mokrane Bouzeghoub, Georges Gardarin, and Patrick Valduriez. *Les Objets*. Eyrolles, Paris, 1997.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[BI92] B. R. Badrinath and T. Imielinski. Replication and mobility. *IEEE Workshop on Management of Replicated Data*, November 1992.

[BLNS82] A. Birrell, R. Levin, R.M. Needham, and M.D. Shroeder. Grapevine: An exercise, in distributed computing. *Communications of the ACM*, 25(4), April 1982.

[Bob96] S. Bobrowski. Oracle 7 server concepts, release 7.3. *Oracle Corporation*, Redwood City, CA, 1996.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *ACM SIGMOD Record*, 26(1), March 1997.

[CFLS91] M.J. Carey, M.J. Franklin, M. Livny, and E.J. Shekita. Data caching tradeoffs in client-server DBMS architectures. *ACM SIGMOD Int. Conf. on Management of Data*, Denver, June 1991.

[CHKS94] S. Ceri, M.A.W. Houtsma, A.M. Keller, and P. Samarati. A classification of update methods for replicated databases. Technical report, Standford University, 1994.

[CHKS95] S. Ceri, M.A.W. Houstma, A.M. Keller, and P. Samarati. Independent updates and incremental agreement in replicated databases. *Distributed and Parallel Databases*, 3(3):225–246, July 1995.

[Cod95] E. Codd. Twelve rules for on-line analytical processing. *Computerworld*, April 1995.

[Com87] Tandem Computers. Remote duplicate facility (rdf) system management manual. *ACM SIGMOD Int. Conf. on Management of Data*, March 1987.

[CP84] S. Ceri and G. Pelegatti. *Distributed Database Systems: Principles and Systems*. New York : McGraw-Hill, 1984.

[Dav94a] J. Davis. Data replication. *Distributed Computing Monitor*, 1994.

[Dav94b] J. Davis. Oracle delivers. *Open Information Systems*, July 1994.

[Dav94c] J. Davis. Sybase system 10. *Open Information Systems*, March 1994.

[DEKB93] W. Du, A. Elmagarmid, W. Kim, and O. Bukhres. Supporting consistent updates in replicated multidatabase systems. *Int. Journal on VLDB*, 2(2), 1993.

[DH95] M. Dunham and A. Helal. Mobile computing and databases: Anything new ? *ACM SIGMOD Record*, 24(4), December 1995.

[DPS+94] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The bayou architecture: Support for data sharing among mobile users. *Workshop on Mobile Computing and Applications*, 1994.

[EGLT76] K.P Eswaran, J.N. Gray, R. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11), November 1976.

[FMS97] F. Fabret, M. Matulovic, and E. Simon. State of the art: Data warehouse refreshment. *DWQ - Report*, 1997.

[GDC+83] N. Goodman, D.Skeen, A. Chan, U.Dayal, S.Fox, and D. Ries. A recovery algorithm for a distributed database system. *ACM SIGACT-SIGMOD Symp. on Database Systems*, Atlanta, March 1983.

[GHK+97] T. Griffin, R. Hull, B. Kumar, D. Lieuwen, and G. Zhou. A framework for using redundant data to optimize read-intensive database applications. *Int. Workshop on Real-Time Databases*, Burlington, September 1997.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The danger of replication and a solution. *ACM SIGMOD Int. Conf on Management of Data*, Montreal, June 1996.

[Gif79] D.K. Gifford. Weighted voting for replicated data. *ACM-SIGOPS Symp. on Operating Systems Principles*, Pacific Grove, December 1979.

[GM97] L. George and P. Minet. A fifo worst case analysis for a hard real-time distributed problem with consistency constraints. *Int. Conf. on Distributed Computing Systems (ICDCS97)*, Baltimore, May 1997.

[GMS93] H. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining views incrementally. *ACM SIGMOD Int. Conference on Management of Data, Washington, DC*, Washington, DC, May 1993.

[GN95] R. Gallersdorfer and M. Nicola. Improving performance in replicated databases through relaxed coherency. In *Int. Conf. on VLDB*, Zurich, September 1995.

[Gol92] R. Goldring. Weak-consistency group communication and membership. *PhD Thesis - University of Santa Cruz*, 1992.

[GR93] J.N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan-Kaufmann, 1993.

[Gra79] J. Gray. Notes on database operating systems. *Lecture Notes in Computer Science*, 1979.

[HGMW+95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The stanford data warehousing project. *Data Engineering, Special Issue on Materialised Views on DataWarehousing*, 18(2):41–58, June 1995.

[HHB96] A.A. Helal, A.A. Heddaya, and B.B. Bhargava. *Replication Techniques in Distributed Systems.* Kluwer Academic Publishers, 1996.

[HT88] V. Hadzilacos and S. Toueg. A theory of reliability in database systems. *Journal of ACM*, 35(1), January 1988.

[HT94] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. *Technical Report TR 94-1425, Dept. of Computer Science, Cornell University, Ithaca, NY 14853*, 1994.

[Inm96] W.H. Inmon. *Building the Data Warehouse.* John Wiley, 1996.

[KHGMP91] R.P. King, N. Halim, H. Garcia-Molina, and C.A Polyzois. Management of remote backup copy for disaster recovery. *ACM Transactions on Database Systems*, 16(2), June 1991.

[KR87] B. Kahler and O. Risnes. Extending logging for database snapshot refresh. *Int. Conf on VLDB*, Brighton, September 1987.

[Lad90] R. Ladin. Lazy replication: Exploiting the semantics of distributed services. *Int. Workshop on Management of Replicated Data*, Houston, November 1990.

[LZW+97] W. Labio, Y. Zhuge, J. Wiener, H. Gupta, H. Garcia-Molina, and J. Widom. The whips prototype for data warehouse creation and maintenance. *ACM SIGMOD Int. Conf. on Management of Data*, Tucson, May 1997.

[Moi96] A. Moissis. *Sybase Replication Server: A Pratical Architecture for Distributing and Sharing Corporate Information*, 1996.

[OV98] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems.* 2nd Edition, Prentice Hall, 1998.

[Pap79] C. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(2), October 1979.

[Pap86] C. Papadimitriou. The theory of concurrency control. *Computer Science Press*, Rockville, Maryland 1986.

[PS98] E. Pacitti and E. Simon. Update propagation strategies to improve data freshness in lazy master schemes. *14e Journées de Données Avancées (BDA98)*, Hammamet, October 1998.

[PSdM98] E. Pacitti, E. Simon, and R. de Melo. Improving data freshness in lazy master schemes. *Int. Conf. on Distributed Computing Systems (ICDCS98)*, Amsterdam, May 1998.

[Puj97] G. Pujolle. *Les Réseaux*. Eyrolles, Paris, 1997.

[PV98] E. Pacitti and P. Valduriez. Replicated databases: concepts, architectures and techniques. *Network and Information Systems*, 1(3), December 1998.

[Rou92] N. Roussopoulos. *Object-oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[Rou98] N. Roussopoulos. Materialized views and data warehouse. *ACM SIGMOD Record*, 27(1):21–26, March 1998.

[RSK90] M. Rusinkiewicz, A. Sheth, and G. Karabatis. Specifying interdatabase dependencies in a multidatabase environment. *IEEE Computer Special Issue on Heterogeneous Distributed Databases*, 24(12), December 1990.

[Sha97] D. Shasha. Lessons from wall street: case studies in configuration, tuning and distribution. *ACM SIGMOD Int. Conf on Management of Data*, Tucson, June 1997.

[SKK+90] M. Satyanarayanan, J.J Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C Steere. Coda: A highly available file system for a distributed workstation environment. *ACM Transactions on Database Systems*, 39(4), April 1990.

[SKS86] S. K. Sarin, C. W. Kaufman, and J. E. Somers. Using history information to process delayed database updates. *Int. Conf. on VLDB*, Kyoto, August 1986.

[SL76] D. G. Severance and G. Lohman. Differential files: their application to the maintenance of large databases. *ACM Transactions on Database Systems*, 1(3), September 1976.

[SLE91] A. Sheth, Y. Leu, and Elmagarmid. Maintaining consistency of interdependent data in multidatabase systems. *Technical Report CSD-TR-91-016*, March 1991.

[Smi95] G. Smith. *Oracle7 Symmetric Replication*. Oracle Corporation, 1995.

[SN79] M. Stonebraker and E. Neuhold. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on Software Engineering*, 3(3), May 1979.

[Sta95] D. Stacey. Replication: Db2, oracle, or sybase ? *ACM Sigmod Record*, 24(4), 1995.

[Tho79] R.H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2), June 1979.

[TTP+95] D. B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *Symposium on Operating System Principles (SIGOPS)*, Colorado, December 1995.

[Wid95] J. Widom. Research problems in data warehousing. *Int. Conf. on Information and Knowledge Management*, Maryland, November 1995.

[WQ90] Gio Wiederhold and X. Qian. Consistency control of replicated data in federated databases. *Int. Workshop on Management of Replicated Data*, Houston, November 1990.

[ZGMW95] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. View maintenance in a warehouse environment. *ACM SIGMOD Int. Conf. on Management of Data*, 1995.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399