Optimizing Recovery Protocols for Replicated Database Systems

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Sistemas Informáticos y Computación

Tesis Doctoral

Presentada por:
Luis Héctor García Muñoz

Dirigida por:
Dr. Francisco Daniel Muñoz-Escoí
Dr. José Enrique Armendáriz-Íñigo

Julio 2013, Valencia

Agradecimientos

En primer lugar quiero agradecer a quien considero un gran profesor y amigo, mi director de tesis Dr. Francisco Muñoz, gracias Paco por todo tu apoyo y paciencia durante el desarrollo de este trabajo, gracias por mostrarme lo que la dedicación y empeño pueden lograr, gracias por todo el conocimiento que has compartido conmigo y tu gran capacidad para tener siempre una pronta y acertada respuesta.

Gracias a Enrique Armendáriz un gran compañero y también director de tesis. Gracias Enrique por esa gran capacidad para generar ideas y llevarlas a la práctica y por tu orientación en este interesante y algunas veces árido camino de la investigación.

Ambos son un gran ejemplo a seguir.

Gracias al Instituto Tecnológico de Informática por haberme permitido colaborar en el grupo de investigación de Sistemas Distribuidos. Gracias a todos los miembros del grupo, en particular quiero dar las gracias a Rubén de Juan Marín, Idoia Ruiz, Jerónimo Pla y Hendrik Decker con quienes he colaborado para la realización de los trabajos de investigación.

Gracias también a Pedro Blesa cuyas primeras instrucciones en el campo de la investigación fueron fundamentales.

Gracias a mis compañeros del Instituto Tecnológico de la Laguna, en especial a Roberto Dominguez por su apoyo invaluable.

En el apartado personal quiero dar las gracias a toda mi familia, a mis padres Luis y Lidia, a quienes debo todo lo que soy. A mis hermanos que siempre han creído en mí. A mis suegros José y Consuelo en quienes he encontrado unos segundos padres.

A Mayela, mi amada esposa, no encuentro palabras para agradecerte todo el apoyo que me has dado, siempre conmigo y alentándome cuando el camino parece indescifrable. A mis hijos por su cariño y comprensión. A Paty y Victor por su apoyo incondicional.

Contents

List of Figures

List of Tables

Abstract

Nowadays, information technology and computing systems have a great relevance on our lives. Among current computer systems, distributed systems are one of the most important because of their scalability, fault tolerance, performance improvements and high availability.

Replicated systems are a specific case of distributed system. This Ph.D. thesis is centered in the replicated database field due to their extended usage, requiring among other properties: low response times, high throughput, load balancing among replicas, data consistency, data integrity and fault tolerance.

In this scope, the development of applications that use replicated databases raises some problems that can be reduced using other fault-tolerant building blocks, as group communication and membership services. Thus, the usage of the services provided by *group communication systems* (GCS) hides several communication details, simplifying the design of replication and recovery protocols.

This Ph.D. thesis surveys the alternatives and strategies being used in the replication and recovery protocols for database replication systems. It also summarizes different concepts about group communication systems and virtual synchrony. As a result, the thesis provides a classification of database replication protocols according to their support to (and interaction with) recovery protocols, always assuming that both kinds of protocol rely on a GCS.

Since current commercial DBMSs allow that programmers and database administrators sacrifice consistency with the aim of improving performance, it is important to select the appropriate level of consistency. Regarding (replicated) databases, consistency is strongly related to the isolation levels being assigned to transactions.

One of the main proposals of this thesis is a recovery protocol for a replication protocol based on certification. Certification-based database replication protocols provide a good basis for the development of their recovery strategies when a snapshot isolation level is assumed. In that level readsets are not needed in the validation step. As a result, they do not need to be transmitted to other replicas. Additionally, these protocols hold a writeset list that is used in the certification/validation step. That list maintains the set of writesets needed by the recovery protocol. This thesis evaluates the performance of a recovery protocol based on the writeset list tranfer (basic protocol) and of an optimized version that compacts the information to be transferred.

The second proposal applies the compaction principle to a recovery protocol designed for weak-voting replication protocols. Its aim is to minimize the time needed for transferring and applying the writesets lost by the recovering replica, obtaining in this way an efficient recovery. The performance of this recovery algorithm has been checked implementing a simulator. To this end, the Omnet++ simulating framework has been used. The simulation results confirm that this recovery protocol provides good results in multiple scenarios.

Finally, the correction of both recovery protocols is also justified and presented in Chapter 5.

Resumen

En la actualidad, el uso de tecnologías de información y sistemas de cómputo tienen una gran influencia en la vida diaria. Dentro de los sistemas informáticos actualmente en uso, son de gran relevancia los sistemas distribuidos por la capacidad que pueden tener para escalar, proporcionar soporte para la tolerancia a fallos y mejorar el desempeño de aplicaciones y proporcionar alta disponibilidad.

Los sistemas replicados son un caso especial de los sistemas distribuidos. Esta tesis está centrada en el área de las bases de datos replicadas debido al uso extendido que en el presente se hace de ellas, requiriendo características como: bajos tiempos de respuesta, alto rendimiento en los procesos, balanceo de carga entre las replicas, consistencia e integridad de datos y tolerancia a fallos.

En este contexto, el desarrollo de aplicaciones utilizando bases de datos replicadas presenta dificultades que pueden verse atenuadas mediante el uso de servicios de soporte a mas bajo nivel tales como servicios de comunicacion y pertenencia. El uso de los servicios proporcionados por los sistemas de comunicación de grupos permiten ocultar los detalles de las comunicaciones y facilitan el diseño de protocolos de replicacion y recuperación.

En esta tesis, se presenta un estudio de las alternativas y estrategias empleadas en los protocolos de replicación y recuperación en las bases de datos replicadas. También se revisan diferentes conceptos sobre los sistemas de comunicación de grupos y sincronia virtual. Se caracterizan y clasifican diferentes tipos de protocolos de replicación con respecto a la interacción o soporte que pudieran dar a la recuperación, sin embargo el enfoque se dirige a los protocolos basados en sistemas de comunicación de grupos.

Debido a que los sistemas comerciales actuales permiten a los programadores y administradores de sistemas de bases de datos renunciar en alguna medida a la consistencia con la finalidad de aumentar el rendimiento, es importante determinar el nivel de consistencia necesario. En el caso de las bases de datos replicadas la consistencia está muy relacionada con el nivel de aislamiento establecido entre las transacciones.

Una de las propuestas centrales de esta tesis es un protocolo de recuperación para un protocolo de replicación basado en certificación. Los protocolos de replicación de base de datos basados en certificación proveen buenas bases para el desarrollo de sus respectivos protocolos de recuperación cuando se utiliza el nivel de aislamiento *snapshot*. Para tal nivel de aislamiento no se requiere que los *readsets* sean transferidos entre las réplicas ni revisados en la fase de cetificación y ya que estos protocolos mantienen un histórico de la lista de *writesets* que es utilizada para certificar las transacciones, este histórico provee la información necesaria para transferir el estado perdido por la réplica en recuperación. Se hace un estudio del rendimiento del protocolo de recuperación básico y de la versión optimizada en la que se compacta la información a transferir. Se presentan los resultados obtenidos en las pruebas de la implementación del protocolo de recuperación en el middleware de soporte.

La segunda propuesta esta basada en aplicar el principio de compactación de la informacion de recuperación en un protocolo de recuperación para los protocolos de replicación basados en votación débil. El objetivo es minimizar el tiempo necesario para transfeir y aplicar la información perdida por la réplica en recuperación obteniendo con esto un protocolo de recuperación mas eficiente. Se ha verificado el buen desempeño de este algoritmo a través de una simulación. Para efectuar la simulación se ha hecho uso del entorno de simulación Omnet++. En los resultados de los experimentos puede apreciarse que este protocolo de recuperación tiene buenos resultados en múltiples escenarios.

Finalmente, se presenta la verificación de la corrección de ambos algoritmos de recuperación en el Capítulo 5.

Resum

Actualment, l'ús de tecnologies d'informació i sistemes de còmput té una gran influència en la vida diària. Entre els sistemes informàtics actuals, els sistemes distribuïts són de gran importància per la capacitat que tenen per a escalar, proporcionar suport per a la tolerància a fallades, millorar el rendiment de les aplicacions i proporcionar alta disponibilitat.

Els sistemes replicats són un cas especial de sistema distribuït. Aquesta tesi està centrada en l'àrea de les bases de dades replicades, degut a l'ús estés que es fa d'elles, demanant característiques com: baixos temps de resposta, alt rendiment dels seus processos, equilibrat de càrrega entre les rèpliques, consistència i integritat de dades i tolerància a fallades.

En aquest context, el desenvolupament d'aplicacions utilitzant bases de dades replicades presenta dificultats que poden veure's atenuades utilitzant serveis de suport a més baix nivell, tals com els serveis de comunicació i de pertinença. L'ús dels serveis proporcionats pels sistemes de comunicació de grups permeten ocultar els detalls de les comunicacions i faciliten el disseny de protocols de replicació i recuperació.

En aquesta tesi es presenta un estudi de les aternatives i estratègies utilitzades en els protocols de replicació i recuperació en bases de dades replicades. També es revisen diferents conceptes sobre els sistemes de comunicació de grups i sincronia virtual. Es caracteritzen i classifiquen diferents tipus de protocols de replicació respecte a la interacció o suport que poden donar a la recuperació. No obstant això, l'enfocament es dirigeix als protocols basats en sistemes de comunicació de grups.

Com els sistemes comercials actuals permeten als programadors i administradors de sistemes de bases de dades renunciar en alguna medida a la consistència amb la finalitat d'augmentar el rendiment, és important determinar el nivell de consistència necessari. En el cas de les bases de dades replicades la consistència està fortament relacionada amb el nivell d'aïllament establit entre les transaccions.

Una de les propostes centrals d'aquesta tesi és un protocol de recuperació per a un protocol de replicació basat en certificació. Els protocols de replicació de bases de dades basats en certificació proporcionen una bona base per al desenvolupament dels seus respectius protocols de recuperació quan s'utilitza el nivell d'aïllament *snapshot*. Per a tal nivell d'aïllament no cal que els *readsets* siguen transferits entre les rèpliques ni revisats en la fase de certificació ja que aquests protocols mantenen un històric de la llista de *writesets* que s'utilitza per a certificar les transaccions. Aquest històric conté la informació necessària per a transferir l'estat perdut per la rèplica en recuperació. Es fa un estudi del rendiment del protocol de recuperació bàsic i de la versió optimitzada on es compacta la informació a transferir. Es presenten els resultats obtinguts en les proves de la implementació del protocol de recuperació sobre el middleware de suport.

La segona proposta està basada en aplicar el principi de compactació de la informació de recuperació en un protocol de recuperació per als protocols de replicació basats en votació dèbil. L'objectiu és minimitzar el temps necessari per a transferir i aplicar la informació perduda per la rèplica en recuperació obtenint així un protocol de recuperació més eficient. S'ha verificat el bon rendiment d'aquest algorisme mitjançant una simulació. Per a fer la simulació s'ha utilitzat l'entorn Omnet++. En els resultats dels experiments pot apreciar-se que aquest protocol de recuperació té bons resultats en múltiples escenaris.

Finalment, es presenta la verificació de la correcció d'ambdós algorismes de recuperació al Capítol 5.

Chapter 1

Introduction

1.1 Motivation

The technological development in the last decades has motivated the integration of technology with greater work capability in small and even more accessible devices. The number of computers and embedded systems has increased and they have become ubiquitous. These characteristics combined with the growing network and internet infrastructure, allows the integration of these devices in private and public computer networks. Because of this, everyday's life becomes to be more influenced by the use of complex computer systems, from recreational applications to critical life application systems.

These characteristics allow the integration of these devices into private and public communication networks, giving the possibility to forming distributed systems. The advantages of distributed systems are that they make easier to integrate different applications running on different computers into a single system. The system, if it is properly designed, can scale well with respect to the size of the underlying network. But, it may have a cost such as more complex software, performance degradation and weaker security.

The developments in the distributed system area in the last three decades, such as interprocess communication, remote invocation, cryptographic security, distributed file systems, data replication and distributed transaction mechanisms, provide the run-time infrastructure supporting today's networked computer applications.

1.1.1 Distributed Systems

A distributed system can be defined as a collection of autonomous computers, connected through a network, sharing the resources of the system and coordinating their activities, so that users perceive the system as a single, integrated computing facility.

Two aspects are relevant in distributed systems, the first one is the hardware: it is composed of multiple nodes. Those nodes are independent regarding their probability of failure. The second one is the software, through this one the users can see as if they are dealing with a single system. Depending on the underlying operating system, may be necessary to organize the system by means of a layer of software placed between the higher-level layer, i.e. applications and users, and a layer underneath consisting of the operating systems. This intermediate layer is a distribution middleware [8].

So, distributed systems are generally built by means of additional layers of software. This relies on middleware support through the use of software frameworks that provide abstractions such as distributed shared objects, and services including secure communication, authentication and access control, mobile code, transactions and persistent storage mechanisms.

Nowadays, distributed applications enable closer cooperation between users through replicated data and multimedia data streams, and will support user and device mobility using wireless and spontaneous networking.

At this time, distributed systems, particularly the Web-based and other Internet-based applications such as banking, flight booking, financial investment, etc., are of unprecedented interest and importance. Several of these applications make use of replication.

1.1.2 Replication

Replication is an increasingly important topic in many areas, especially in distributed systems and in database systems although with different purposes. In database systems, replication is used for performance and fault tolerance reasons. In distributed systems, some of the main objectives are to provide guarantees of message order and delivery, as well as providing fault tolerance, this last is provided by means of the replication. Replication is obtained maintaining multiple copies of the data in different computers.

However, replication introduces the consistency problem: whenever a replica changes it becomes different from the rest. So, we need to propagate all updates if we want to keep all replicas consistent. Nonetheless, it may degrade the system performance, especially in large-scale distributed systems, such as large replicated database systems.

1.1.3 Database Replication

Just about three decades, database replication has been an object of research, in to the first publications in this area we can find [56, 59].

In replicated databases, identical copies of data items are stored on different computers at different, possibly very distant sites. As a subarea of distributed systems and database theory and practice, the field of replication has acquired great relevance. It is increasingly used for supporting performance, fault tolerance and high availability.

Among all available replicas, clients can improve their throughput by transparently accessing the server replica that is closest to them. Suitable protocols cater for the mutual consistency of the data at each replica. Whenever a replica fails or the connection to it is broken, client transactions are redirected to other available servers. In order to maintain high availability, a need for efficient recovery procedures arises, for bringing failed or temporarily disconnected replicas back into the network of active servers as fully functional peers.

1.1.4 Replicated Database Recovery

In Replicated database systems it is very important for fault-tolerant and high availability purposes to recover a previously crashed replica and its state to be reconciled with the actual state from the rest of the replicas.

The recovery task basically consists in transferring the updates lost during failure from one or more active replicas to one or more recovering replicas, without impeding the overall system capability of providing normal application services. Since recovery must re-establish consistent data, the development of recovery protocols must take the idiosyncrasies of the used replication protocols into account. Under this premise, various recovery protocols have been proposed in the literature, among them [3, 12, 13, 33, 38, 39, 43].

Ideally, a good replication system should use mechanisms that are simple (so as to reduce overhead), cope well with network overload, maintain consistency, provide continuous operability and avoid transaction rollbacks [33]. Similarly, a good recovery protocol should be simple, efficiently distribute the recovery work among available replicas, and seamlessly allow for simultaneous concurrent transactions. All of this must be done reducing the temporal cost of the recovery process and reducing to a minimum amount the information to transfer for the recovery purposes. Additionally, both replication and recovery protocols must take into account the concurrency of transactions, which in many applications are required to comply with the ACID requirement [9], i.e., the atomicity, consistency, isolation and "durability" (a.k.a. persistence) of updates.

Typically, a synchronization mechanism for supporting the updating of alive and recovering replicas is deployed, since otherwise, recovery may become too complicated. A straightforward way to synchronize replicas would be to interrupt the ongoing application, but then, high availability is sacrificed. However, with a suitable Group Communication System (GCS) [15] and virtual synchrony [10], it is possible to generate synchrony points between failed and recovering replicas, taking the set of messages delivered to non-failed replicas into account.

The GCS provides a membership service and a reliable multicast. Membership services maintain a list of available, i.e., currently active and connected replicas, and implement the view concept [15], distinguishing between different states of the update history in which data items are seen by mutually isolated groups of servers.

1.2 Methodology

In this thesis we employ the methodology described on the next to deal with proposed improvements.

1. A bibliographical research must be done in order to set the background ideas and works existing about replication and recovery algorithms. This first work provides us with references and some related issues about problems and solutions. To this end, a survey is very helpful. Whenever a survey does not exist, it is highly convenient to generate one.

2. As a second step, new proposals must be formalized through a theoretical analysis.

3. To select a solution, if more than one is generated, discarding those that do not meet the requirements or have poor contribution. This is made checking the validity and correctness based on the theoretical analysis. We have to compare the solutions with the aim of discarding the less promising ones.

4. Measure the results through implementing the algorithms or simulation results that would provide its implementation, generating a cycle of improvements between the current step and the preceding steps in the event that the results obtained are not satisfactory.

5. Analysis of the performance of the new proposals.

6. Technical reports and papers containing the related information of all the precedent steps and the research results are written and sent for publication in specialized conferences of this area. Technical reports and papers must be improved according with the comments and suggestions from conference reviewers. In case of the work was not accepted for publication, the improved work may be presented in another conference. In order to ensure the quality of the work, papers are only sent to conferences which publish at IEEE-CS Press, Association of Computer Machinery (ACM) Press and Springer LNCS, or some included in the CORE classification available at http://www.core.edu.au/

For the design, development and implementation of the recovery protocols we must use the software development process detailed in the next:

- Analysis. The first step in software design and development is the analysis. Normally the analysis is divided in domain analysis and software elements analysis. Domain analysis is concerned with the possibility that the developers are not sufficiently knowledgeable in the subject area of the new software, the first task is to investigate the so-called ”domain” of the software.

  In the software elements analysis the developers try to establish the requirements for all system elements and then allocating some subset of these requirements to software. This is necessary because typically, people know what they want, but not what software should do.

- Design. This step involves two items, the specification and the software architecture. In the specification the goal is precisely to describe the software to be written. It is also done for understanding applications that were already developed. In the software architecture, the overall structure for the software and its nuances are defined in terms of the database design, the data structure design, etc. The software architecture refers to an abstract representation of that system. Architecture is concerned with making sure the software system will meet the requirements of the product.

- Prototyping/simulation. Once the requirement analysis and the design is done we can construct a prototype to make some tests and to give feedback to the developer who refines the product according to the exact expectation. Alternatively, we can construct a simulation model to make the tests. Actually, there are a large number of software simulation tools that help the developers to easily construct models for testing purposes. In order to evaluate the performance of the optimizations proposed in this work, we use the Omnet++ discrete event simulation environment [35].

- Implementation or code generation. In this step, the design is translated into a system with a machine-readable form. With a well-performed design, reducing it to code can be accomplished without complications.

- Testing. Once the code is generated we must test it with the objective of detecting faults and bugs that where committed in the previous steps.

1.3 About This Thesis

This thesis is situated in the distributed systems and database research field, so it has an inter-disciplinary nature. As stated in [61], the mechanisms and techniques used in both fields are very similar. Albeit, due to the many subtleties involved, mechanisms that are conceptually identical, end up being very different in practice and the results obtained in one field can not be applied directly in the other one.

We have centered this research in to the replicated database systems, and more specifically in the recovery process for crashed database replicas.

This research is enclosed inside the "CONDEP –From static to dynamic environments: CONsistency, Recovery and DEPendability Issues / CONFIA –De los entornos estáticos a los entornos dinámicos: cuestiones de CONsistencia, recuperación y FIAbilidad" research project (TIN2006-14738-C02), a Spanish research project co-developed by Instituto Tecnológico de Informática from the Universidad Politécnica de Valencia and the Universidad Pública de Navarra, Spain.

The main goals of this project are:

- Definition and design of a basic support system for distributed agreement on dynamic environments
- Dependability improvement on dynamic environments from distributed agreement
- Component development to improve dynamic systems dependability
- Improve a data replication middleware
- Flexible replication protocols development
- Efficient recovery/recociliation protocols development
- Use of fuzzy logic based techniques in distributed systems
- Support development for the evaluation of integrity constraints in replicated databases

Within this project, the goal of this thesis has been to provide optimizations for the developed recovery/reconciliation protocols, for improving its performance.

1.3.1 Objectives

The main research and thesis objectives are:

1. The first objective is to present a survey of alternative options and strategies employed by replication and recovery protocols developed in recent years. The main characteristics are summarized in a table. If we pretend to optimize the recovery protocols, it is mandatory to fully understand the currently proposed replication and recovery protocols. Then, this survey and this table are helpful to understand and to have "on one hand" the main characteristics of the surveyed protocols. There is not a previous work that surveys and compares these protocols.

2. Enhance the recovery process with compacting techniques that minimize the total amount of information to transfer and to process for recovery purposes. In a first proposal for optimization, trying to minimize the recovery information will help us to reduce the workload in both replicas and the communication network overload.

3. Improve the recovery process with compactness an considering the amnesia phenomenon [20].

4. Verification of the improvements presented through experimentation and simulation.

1.3.2 Publications

In order to accomplish with the requirements of the internal regulations of the Ph.D. Program of the Universitat Politècnica de València, some results related to this thesis and previously published in conference proceedings are listed in this subsection. These publications are original work and do not appear as part of any other Ph.D. thesis.

- Improving Recovery in Weak-Voting Data Replication. L. H. García-Muñoz, R. de Juan-Marín, J. E. Armendáriz-Íñigo, F. D. Muñoz-Escoí. 7th International Symposium on Advanced Parallel Processing Technologies (APPT), November 2007, Guangzhou, China. Lecture Notes in Computer Science, vol. 4847, pgs. 131-140, Springer, ISSN 0302-9743.
    - The conference "7th International Symposium on Advanced Parallel Processing Technologies (APPT)" in which this paper was published is included in ISI-PROCEEDINGS.
- Optimizing Certification-Based Database Recovery. J. Pla-Civera, M. I. Ruiz-Fuertes, L. H. García-Muñoz, F. D. Muñoz-Escoí. 6th International Symposium on Parallel and Distributed Computing (ISPDC), July 2007, Hagenberg, Austria. Proceedings, pgs. 211-218. ISBN 0-7695-2917-8. IEEE-CS Press.
    - As before, the international symposium where this paper was published, is also included in ISI-PROCEEDINGS.
- Recovery Protocols for Replicated Databases - A Survey. L. H. García-Muñoz, J. E. Armendáriz-Íñigo, H. Decker, F. D. Muñoz-Escoí. International Symposium on Frontiers in Networking and Applications (FINA), Track of The IEEE 21st International Conference on Advanced Information Networking and Applications (AINA-07), May 2007, Niagara Falls, Ontario, Canada. Proceedings, pgs. 220-227, IEEE-CS Press. ISBN 0-7695-2847-3.

  – As a track of the "IEEE 21st International Conference on Advanced Information Networking and Applications (AINA-07)", the international symposium where this paper was published has a CORE B classification.

- Associating Replication and Recovery Protocols for Replicated Databases. L. H. García-Muñoz, J. E. Armendáriz-Íñigo, F. D. Muñoz-Escoí. 15th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP) (work in progress track), February 2007, Naples, Italy. Proceedings, pgs. 5-6, SEA-Publications, ISBN 978-3-902457-12-7.

- Recovery Protocols for Replicated Databases - A Minimal Survey. L. H. García-Muñoz, J. E. Armendáriz-Íñigo, F. D. Muñoz-Escoí. 15th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP) (work in progress track), February 2007, Naples, Italy. Proceedings, pgs. 3-4, SEA-Publications, ISBN 978-3-902457-12-7.

  – As the same as the "International Symposium on Frontiers in Networking and Applications (FINA)", the "15th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)" where this last two papers were published has a CORE B classification.

1.4 Structure of the Thesis

The remainder of the thesis is structured as follows:

Chapter 2 presents a survey of database replication and recovery protocols, it classifies and analyzes every protocol according to the classification presented in [31, 60]. This Chapter is generated based on two previous publications [28, 29] about surveying replication protocols and associating them with recovery protocols.

Chapter 3 presents our first optimization proposal focused on certification-based replication protocols. Mainly, this optimization consists in dividing the recovery in two stages, reducing the certification load and the amount of information to be recovered in the second stage. The second technique scans and compacts the set of items to transfer, sending only the latest version of each item. We show that these techniques can be easily combined, reducing with this the time needed for the recoverer replica to transfer this information and avoiding network overload. Besides, it reduces the time needed for the recovering replica to apply the updates. This Chapter arises from a published work [16] in collaboration with Jerónimo Pla-Civera and Idoia Ruiz-Fuertes.

Chapter 4 introduces our second proposal for optimizing recovery protocols, This focused on Weak-Voting replication protocols. Based on the proposal made in [2], we review the functionality of the original recovery protocol and enhance it incorporating a compacting method for improving its performance and providing an accurate amnesia support. To this last, we consider some related works [21, 30, 23], and I want to thank to Rubén de Juan Marín for his extensive and valuable contribution to this work by providing the foundation basis for analysis and incorporation of a solution to the amnesia phenomenon in the recovery protocol implemented. As before, this Chapter is almost a transcription of the published article [30].

The Chapter 5 provides the correctness of the recovery algorithms detailed in the previous chapters.

Finally, Chapter 6 summarizes our conclusions from this work and outlines future research directions.

As can be seen, this thesis is based on a collection of previous works published, whose contents has been in force, and on minor adaptations of other previous works.

Chapter 2

Data Recovery

The aim of this chapter is to present a survey of alternative options and strategies employed by replication and recovery protocols in database systems developed in recent years. In that context, different concepts for Group Communication Systems and virtual synchrony are also reviewed. We characterize and classify different kinds of replication protocols in regard to their interplay with recovery. We narrow the focus on replication protocols based on group communication and discuss them broadly. Finally a conclusion for this chapter is given. We present a tabular comparison that summarizes and highlights significant characteristics as distinguished in Subsections 2.3.1 - 2.3.3.

2.1 Introduction

Nowadays, distributed systems are ubiquitous and provide support to many applications of several kinds, we can mention world wide web applications, client-server systems, databases, etc. This promising computing power is threatened by their susceptibility to failures. In recent years various techniques have been developed to integrate reliability and high availability to distributed systems.

As stated in [47] reliability, R(t), is the conditional probability that a system can perform its designed function at time t, given that it was operational at time t = 0. Thus R(t) is a function of the fault processes affecting the system, and of any mechanisms that prevent system failure when a fault occurs. Availability, A(t), is a useful measure for systems subject to failure and repair; it is defined as the probability that a system is operational at time t. Availability is often expressed as a steady-state value, either as the probability that the system is operational at any random time, or as a given amount of downtime over a specified interval. High availability is then reached when the system has a very short time of unavailability, e.g., less than 300 seconds per year.

Reliability can be improved with fault-tolerance. The fault-tolerance mechanisms should be directed to the most likely faults to obtain cost-effective system design. A system may be able to tolerate a given set of faults, but if the fault probabilities are high, may still not be sufficiently reliable for the application. Several mechanisms such as automatic fault-detection, diagnosis, repair and recovery could significantly reduce or eliminate downtime improving availability.

Fault-tolerance in distributed systems can be supported through redundancy of modules also known as replication. Replicated systems constitute a subset of distributed systems.

Replication has been the subject of study in several areas and has been implemented for various reasons. In distributed systems has been used primarily for fault-tolerance purposes and in replicated databases has been mainly used for performance reasons. When a replica fails or is disconnected from the replicated system, the client requests are forwarded to the non-failed replicas, thereby obtaining fault-tolerance and high availability. Performance can be improved redirecting client requests to the closest replica obtaining short response times and may be work load balancing.

2.2 General Recovery in Highly Available Systems

In order to maintain fault-tolerance and high availability, a need for efficient recovery procedures arises. Essentially there are two forms of recovery [58]:

- Backward recovery, the main goal is to bring the system from its present erroneous state back to a correctly previous one. For this purpose, it is necessary that each process periodically saves its state during failure-free execution on stable storage. The saved state contains sufficient information to restart process execution. Upon a failure, a failed process restarts from one of its saved states, thereby reducing the amount of lost computation. Each of the saved states is called a checkpoint [44, 18, 26].

  Checkpoint-based recovery protocols rely solely on checkpointing for system state restoration. Checkpointing is *uncoordinated* if each process takes its checkpoints independently. To recover from a failure or disconnection the construction of a consistent global state from these local states is needed, the best is the most recent distributed snapshot, also known as the recovery line, as shown in Figure 2.1.

  In pursuing of this recovery line, a problem known as the *domino effect* [50, 53] may occur especially in message-passing systems, because of messages induce inter-process dependencies during failure-free operation. Upon a failure in one or more processes in the system, these dependencies may force some non-failed processes to perform a rollback and could lead to the

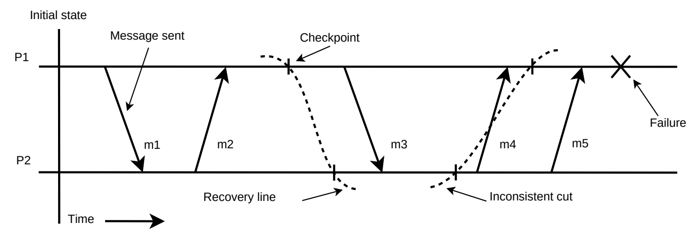


Figure 2.1: Consistent and inconsistent global states

phenomenon of rollback propagation. In order to illustrate this, consider the situation in which a process that sent a message m rolls back to a state prior to sending the message. The receiver of m must also make a roll back to the state that precedes the reception of the message m, otherwise the states of the processes would be inconsistent since one of them had received a message not yet sent, which should not be possible in a failure-free execution. In a more extreme scenario, rollback propagation may be extended to the initial system state, losing all the computational process performed, as depicted in Figure 2.2.

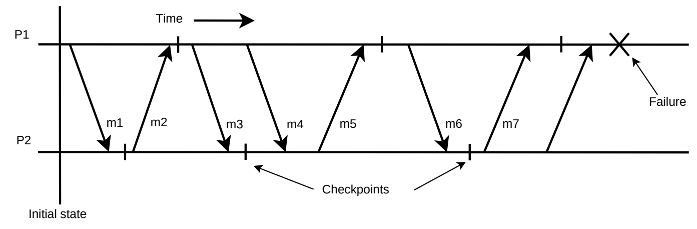


Figure 2.2: The domino effect

Consider the processes P1 and P2 that exchange messages during execution, upon a failure or disconnection, P1 is restored to the most recently local checkpoint. In the same fashion, P2 needs to be restored to its most recent checkpoint. Unfortunately, this two most recently saved states do not form a recovery line because the message *m7* has been received by P1. However, no other processes can be identified as sender of this message. So, P1 needs to be restored (rolled back) to an earlier local checkpoint. Nevertheless, even P1 was rolled back to an earlier previous state, this can not be considered as a recovery line, because P2 would have recorded the receipt of the message *m6* which could not identify the sender. Therefore, it is necessary to return P2 to a previous state. As same as before, when the process P2 is rolled back again to the next earlier local checkpoint, no sender can be identified for the message *m5*; and so on. For this example, in the searching for a consistent global state, the initial state is reached.

Many protocols have been proposed that selectively employ local checkpoints in order to eliminate the possibility of the domino effect (see the survey paper in [26]).

Checkpointing is *coordinated* if processes coordinate their checkpoints in order to record a consistent global state [14, 50, 53, 5]. So, the domino effect can be avoided bounding the rollback propagation by restarting the system from a consistent global state.

Checkpointing can be *communication-induced*, enforcing to each process to make checkpointing based on the information piggybacked on the application messages it receives from other processes. In addition to taking application-specific basic checkpoints, each process can also be asked by the protocol to take additional forced checkpoints, based on the piggybacked information as well as local control variables.

- Forward Recovery, in this case, an attempt is made to bring the system in a correct new state from which it can continue to execute. The main problem with this issue is that we must know in advance which errors may occur.

And the following categories of recovery schemes can be distinguished:

- Application Specific Recovery. In this kind of recovery, the recovery process is programmed in an explicit way and as part of the application. The resulting code involves both, knowledge about the application domain and the underlying hardware. The changes made at the underlying hardware or at the application level may require a substantial redesign of the recovery algorithms.

- Transaction Recovery. In the transaction model, computation is divided in transactions. In distributed environments, transaction commitment involves synchronous checkpointing ("force-writing") to stable storage by each of the processors at each transaction boundary.

  The protocols supporting the committing and aborting of transactions are easily extended to handle recovery from machine failures by treating failures as aborts. These protocols can be built into an operating system and can therefore be made transparent to applications.

  However, not all programs can be expressed as transactions. Transaction systems are based on several assumptions like required serializability, the probability that different transactions contend for the same data is low and finally, transactions are long enough, that is, involve enough computation to amortize the I/O delays of synchronous commits at each transaction boundary.

  So, transaction-based recovery is suitable for applications that are structured as transactions and that satisfy the above assumptions. This recovery is likely to be cost effective, since there is little additional overhead involved.

- Pessimistic Recovery [26]. The systems that synchronize communication and computation with checkpointing are called pessimistic, since they delay processing each message until both the state of sender and the state of the receiver have been checkpointed to avoid an inconsistency in the rare case of a failure. To avoid the substantial delays associated with checkpointing onto a stable storage medium such as mirrored disks, pessimistic systems typically use a backup process on another processor to hold checkpoints. All communication then requires a multiway synchronization of the primary and backup of both sender and receiver, and multiple failures can no longer be tolerated: If both the primary and the backup processors fail, recovery is no longer possible.

- Optimistic Recovery [57]. Optimistic recovery protocols ensure that the externally visible behavior of the system is equivalent to some failure-free execution. State is recovered by first restoring an earlier checkpoint from stable storage and then replaying logged messages, system never rolls back too far and the reason is that, in order to avoid domino effect [53] the extent of rollback in the optimistic recovery is controlled by replaying the correct number of messages.

  Optimistic recovery is a transparent recovery mechanism. This means that the applications can be written as if they were to be executed on an ideal failure-free machine.

  Transparent recovery will save programming effort and reduce the risk of introducing errors. Optimistic recovery applies to any system that can be viewed as a collection of recovery units communicating by message passing. It is not restricted, as transaction-based recovery is, to applications that can be structured as units of work accessing a global database to which concurrency control is applied.

  Unlike pessimistic recovery techniques, there is no synchronization required upon communication. Therefore, as long as the I/O bandwidth to the disk is sufficiently high, logging delays do not slow down computation. Because logging may be asynchronous, several log entries may be blocked into a buffer and written out in a single I/O operation. Provided that an input message is logged by the time the computation it engendered has completed and is ready to return a result to the external user, there is no response time delay.

Message logging and checkpointing can provide fault tolerance in distributed systems in which all process communication is through messages. Based on this last, Johnson and Zwaenepoel [40] propose a recovery algorithm using optimistic message logging and checkpointing for the distributed system recovery.

Damani and Vijay K. Garg in [19] propose an algorithm for recovering asynchronously from failures.

2.3 Replication and Recovery Protocols

As stated in [61] the mechanisms and techniques used to provide replication in both fields, distributed systems and databases are very similar, but due to the subtleties involved such as model, assumptions, guarantees provided and implementation, mechanisms that conceptually could seem equal, end up being very different in practice. So, it is very difficult to directly apply the results obtained from an area into the other one.

In replicated databases, identical copies of data items are stored on different computers at different, possibly very distant sites. As a subarea of database theory and practice, the field of replication is acquiring growing relevance. It is increasingly used for supporting performance, fault tolerance and high availability.

Among all available replicas, clients can improve their throughput by transparently accessing the server replica that is closest to them. Suitable protocols cater for mutual data consistency at each replica. Whenever a server site fails or the connection to it is broken, client transactions are redirected to other available servers. For maintaining high availability, a need for efficient recovery procedures arises, for bringing failed or temporarily disconnected nodes back into the network of active servers as fully functional peers.

The recovery task basically consists in transferring the updates lost during failure from one or more active nodes to one or more recovering sites, without impeding the overall system capability of providing normal application services. Since recovery must re-establish consistent data, the development of recovery protocols must take the idiosyncrasies of the used replication protocols into account. Under this premise, various recovery protocols have been proposed in the literature, among them [43, 33, 39, 38, 12, 13, 3].

Ideally, a good replication system should use mechanisms that are simple (so as to reduce overhead), cope well with network overload, maintain consistency, provide continuous service and avoid transaction rollbacks [33]. Similarly, a good recovery protocol should be simple, efficiently distribute the recovery work among available nodes, and seamlessly allow for simultaneous concurrent transactions. Additionally, both replication and recovery protocols must take into account the concurrency of transactions, which in many applications are required to comply with the ACID requirement [9], i.e., the atomicity, consistency, isolation and "durability" (a.k.a. persistence) of updates.

Typically, a synchronization mechanism for supporting the updating of alive and recovering replicas is deployed, since otherwise, recovery may become too complicated. A straightforward way to synchronize replicas would be to interrupt the ongoing application, but then, high availability is sacrificed. However, with a suitable Group Communication System (GCS) [15] and virtual synchrony [10], it is possible to generate synchrony points between failed and recovering sites, taking the set of messages delivered to non-failed sites into account. The GCS provides a membership service and a reliable multicast. Membership services maintain a list of available, i.e., currently active and connected sites, and implement the view concept [15], distinguishing between different states of the update history in which data items are seen by mutually isolated groups of servers.

This part of the work is focused on replication and recovery strategies designed for the primary partition model [51, 15]. It enforces that, in case of network partitioning, only the subgroup that has a majority of the system replicas (if any) can continue processing transactions. Thus, consistency is easily maintained since no other group of active replicas can cause conflicts in the recovery procedures. Hence, there is always a group of replicas that maintains an up-to-date database state, and any other subgroup can recover by obtaining such state from some replica of the majority subgroup. This model is typically assumed in recent works about database replication and recovery. Partitionable models have been assumed in the field of mobile databases, but we do not survey such kind of systems in this document.

2.3.1 Basic Questions for Replication Protocols

Since recovery is usually embedded in the replication process, the following questions must be answered (cf. [31, 51, 60, 61]). The various concepts as mentioned and labeled with acronyms below, are going to be explained in subsequent subsections.

1. *Server Architecture* (A): are transactions executed in *p*rimary-copy (P) or *u*pdate-anywhere (U) mode?
2. *Server Interaction* (I): is interaction between replica servers *c*onstant (C) or *l*inear (L)?
3. *Transaction Termination* (T): do transactions terminate by *v*oting (V) or *n*on-voting (N)?
4. *Update Propagation* (U): is it *e*ager (E) or *l*azy (L)?

Clearly, answers to some of the questions establish a distinction between various kinds of replication protocols. These are going to be used in Section 2.3.2 for classifying replication protocols that host the recovery protocols as addressed in Section 2.3.3.

Generally, the objectives listed below should be considered by the protocols.

1. *Enable and optimize transaction concurrency.* Two basic kinds of concurrency control mechanisms are distinguished as follows.
   - *Optimistic Concurrency Control.* This assumes that transaction conflicts are unlikely to occur when shared data are accessed. In that

case, remote server resources can remain largely untapped until transaction commit time. And if conflicts do occur, then transactions are aborted without further ado, so that they may be re-tried later.

- *Pessimistic Concurrency Control.* Conflicts are expected to occur, and remote resources must be ready to be tapped on demand at any moment during transaction time. Unless a deadlock occurs, pessimistic concurrency control makes sure that transactions will terminate successfully. Implementations of this pessimistic policy are the well-known Two Phase Locking (2PL), Strict 2PL and Timestamping (which, however, is occasionally used with optimistic control as well).

2. *Minimize transaction abortions.* This depends on the used concurrency control (as indicated in the previous point) and on the type of transactions, in the sense that, the more write operations there are and the longer the transactions last, the more conflicts are likely.

3. *Maintain replica consistency.* This is strongly though not inextricably related to concurrency control. In general, applications differ in their requirements of consistency, so that the isolation level of transactions may vary.

2.3.2 Classification of Replication Protocols

In [31], the following two modes for propagating updates to replicas are distinguished:

1. *Eager.* All replicas are updated during transaction execution time, i.e., no transaction is committed before all network nodes are updated. This guarantees a very high degree of replication consistency but increases transaction response times and thus slows down performance, due to the multiplicity of updates and message rounds. Moreover, eager updating is not a viable solution in mobile networks where nodes may be disconnected for extended periods.

2. *Lazy.* The updates of a transaction generally are executed in a single dedicated replica, typically the nearest one or the owner node of updated items. Updates are propagated to all remaining replicas asynchronously, which in general amounts to a separate transaction per node. Thus, lazy update propagation permits a wide variety of synchronization points, which however are needed because temporary inconsistencies may easily arise, due to the lack of synchrony.

According to [61], eager replication protocols can be classified along to the following three dimensions.

1. *Server Architecture*: It determines where transactions are executed in the first place. According to [31], the two main options are:
    - *Primary copy* [39]. Transactions always are directed to a designated node, which holds the "primary copy" of updated items. It is the only one to actively process updates solicited by transactions.
    - *Update anywhere* [41]. Any replica can directly process any transaction, i.e., transaction updates can be directed to and processed by any replica.
2. *Server Interaction.* The degree of communication among database servers at transaction time, i.e., the amount of network traffic generated by a given replication protocol, is measured by the number of interchanged messages. Two cases can be distinguished:
    - *Constant Interaction.* Independently of the number of operations in the transaction, a constant number of messages is used to synchronize the servers. Typically, protocols in this category group all operations of the transaction into a single message [1, 3, 12, 13, 33, 38, 39, 43].
    - *Linear Interaction.* Here, each operation of a transaction is dealt with separately. Operations can be sent either as SQL statements or as log records that contain the results of executing operations in particular servers [10, 33].
3. *Transaction Termination.* This is related to how atomicity is guaranteed. Two cases can be distinguished:
    - *Voting Termination.* Replicas are coordinated by an extra round of messages. It can be as complex as an atomic commitment protocol, or as simple as a single confirmation message [3, 10, 41].
    - *Non-voting Termination.* Nodes can decide on their own to commit or abort a transaction [25, 45].

In a later work, [63] distinguish the following five replication techniques. In all of them, a reliable total order broadcast is needed in order to propagate the updates:

1. *Active.* Based on the state machine replication approach [54], in this technique the delegate server receives a transaction requests from the client, the whole transaction is put into a message, and this message is broadcast to the servers (using total order) by the delegate server on behalf the client. All servers process the transaction and different transactions can use different delegate serves. A complete determinism in the execution of a transaction is required. All operations should be known at the start of the transaction. Because the entire transaction is sent and processed on all servers, read sets are also processed on all servers losing with this the benefit of load balancing for read operations.

2. *Certification-based.* Also known as "database state machine" [48]. In this replication technique the client sends the transaction to the delegate server. Unlike the active replication at this point nothing is broadcast, the delegate server process the transaction until the commit is required making the total order broadcast of both, the write set and the read set. So, all replicas share the same history of delivered messages and can detect conflicts, if any, with concurrent transactions. This allows to each replica to individually decide if the transaction commits or aborts. This technique can handle interactive transactions because write operations are deferred until the transaction commit is requested.

3. *Weak-voting.* This technique is similar to certification-based, except that when the transaction commit is required, the total order broadcast is only done for the writeset. When the writeset is delivered, the delegate server decides whether the transaction is committed or aborted in case of conflict with any previously committed transaction. Once the outcome is decided, it is communicated to other servers through a second broadcast that may not be totally ordered, but must be reliable. Note that two broadcasts per transaction are needed while in previously techniques only one is required.

4. *Primary copy.* In the primary copy replication all transactions are sent to a primary server, all other servers do not accept transactions and only apply the updates of the transactions processed in the primary server. A reliable broadcast is used for the primary server to communicate the updates. Abort or commit for the transactions are decided on the primary server, so as the serialization order.

5. *Lazy.* As described at the beginning of this section in lazy replication the updates of the transaction are sent to all other replicas once the transaction has committed. Even though this may allow faster transaction completion, when it is used with an update everywhere approach lazy replication can violate ACID properties.

As indicated before, it is also important to take the network's partition model into account [10, 15, 51]. Mostly, the primary partition model is employed. It enforces that, in case of network partitioning, only the sub-group that has a majority of alive nodes can continue to process transactions. Thus, consistency is maintained easily, since no other group of active servers can then cause conflicts. Hence, there always is a group of sites that maintains an up-to-date state of replicas, and any other sub-group can recover by obtaining this state from some replica of the majority sub-group.

Generally, eager update algorithms are preferable to lazy ones whenever replica consistency is achieved with update anywhere protocols. However, if performance is key and consistency can be compromised to some degree, better results are obtained with lazy update algorithms, mainly when a primary copy algorithm is used.

The oldest works [9, 11] did not consider the diffusion protocols based on GCSs, and they became on pessimistic and optimistic concurrency controls with voting techniques for transaction termination.

2.3.3 Recovery Protocols

Dealing with site failures is not an easy task, in most of cases some issues for the recovery are needed to be present in the replication protocol and during the failure free execution of transactions. We need to assume a partial amnesia crash failure model [17] so we can perform the recovery of a failed site. This model assumes that, at restart, some part of the state is the same as before the site crash, while the rest of the state is reset to a predefined initial state. For recovering a failed site, the actual state of the database needs to be transferred to it. Only after that is accomplished, the recovering site can again accept requests from other sites or from clients. To transfer the current state, three options can be distinguished: either to copy over the whole database, or to only transfer incrementally the last versions of all data items that were modified during the failure period, or to resend the update messages that did not reach the failed node.

Note that the checkpoint-based techniques described in Section 2.2 can not be used here due that backward recovery implies a violation to "Durability" property included in the ACID properties, so a recovery of this kind is not allowed.

Despite dismissing the above restriction, the amount of state that a database recovery protocol must manage necessarily involves the use of algorithms to "remember" everything that the recovering site has accepted and persisted and does not involve any kind of backtrack. Neither message logging nor checkpointing protocols consider the case in which the sites need to handle a large amount of state so they can hardly be adapted to this environment.

For classifying different kinds of recovery protocols, it is useful to answer the following questions. Answers to each of them are going to be addressed in more detail in subsequent subsections.

1. *Transfer Model* (TM): Is it a *full database transfer* (FT), or a version-based *incremental transfer* (IT) or are *lost* messages *resent* (LR)?

2. *Concurrency control during recovery*: (a) Regarding optimism (O), is it *optimistic* (O) or *pessimistic* (P)? (b) Considering the number of managers (M), does it use a *single* (S) or *multiple* (M) managers? (c) Is it *multi-versioned (V)* (Y(es)/N(o))?

3. *Recovery-work distribution* (W): is it *centralized* (C) or *distributed* (D)?

		Replication				Recovery				
		A	I	T	U	TM	O	M	V	W
[43]	Full DB transfer	U	C	N	E	FT	P	S	N	C
	Version number	U	C	N	E	IT	P	S	N	C
	Restrict set	U	C	N	E	IT	P	S	N	C
	Log filter	U	C	N	E	IT	P	S	Y	C
	Lazy transfer	U	C	N	L	IT	P	S	N	C
[33]	Bcast Writes Log Update	U	L	N	E	LR	P	S	N	C
	Bcast Writes Augm. Bcast	U	L	N	E	LR	P	S	N	C
	Delayed Bcast Log Update	U	C	N	E	LR	P	S	N	C
	Delayed Bcast Augm. Bcast	U	C	N	E	LR	P	S	N	C
	Single Bcast	U	C	N	E	1	P	S	N	C
[39]	Jiménez	P	C	N	E	LR	P	S	N	D
[38]	COLUP	U	C	V	2	IT	O	M	Y	D
[12]	CLOB	U	C	N	E	3	O	M	Y	C
[13]	FOBr	U	C	V	E	IT	O	M	Y	D
[3]	BRP	U	C	V	E	IT	O	M	Y	C
	ERP	U	C	N	E	IT	O	M	Y	C
	TORPE	U	C	N	E	IT	O	M	Y	C

Table 2.1: Protocol Classification.

1. Considers full database transfer, is needed if a site is new or if it is not in the record of views in the *logger*. Otherwise, it uses LR.
2. It is configurable, and may be hybrid.
3. IT for long-term failures, and LR for short-term ones.

Recovery Protocols by Kemme, Bartoli and Babaoglu

In [43] the authors propose solutions to transfer the state of the database to the recovering replicas without interrupting the transaction process in the rest of the system. To this end, they consider a replication model that applies eager update anywhere, with a constant interaction and non-voting transaction termination. Basically two ways for transferring the recovery information are discussed: (a) the GCS regular state-transfer when a view-change event arises, or (b) using a specially tailored recovery protocol. Option (a) is immediately discarded since the amount of state to be transferred is very big (the entire database) and this is impractical.

We describe on the sequel the five recovery alternatives presented in [43], plus the *enriched view synchrony*[6] mechanism assumed in that paper and used in all its recovery protocols.

Full Database Transfer. Despite discarding a database transfer when it is initiated by the common GCS behavior (GCS are usually employed in the active replication model; when a replica is added, they transfer the current state of the replicated object to such recovering replica), transferring the entire database still has sense in a few cases. Indeed, this management is mandatory for new replicas, but also attractive if the size of the database is small or if most of data has been changed during the failure interval. In this case we have a pessimistic and centralized concurrency control with a unique manager. The advantages of this method are its simple implementation and that it does not fully suspend the execution of the application, since the write operations are only delayed on the objects that are not yet transferred, and read transactions are allowed. The disadvantages are that it is made under a data transfer transaction schema that sets a read lock, which is released when the data has been read, and transferred to the recovering replica. Additionally, this could be highly inefficient in cases where the failure time for a replica was short.

Incremental Transfer Using Version Numbers. If the recovering replica was not active for a very short time, or the data updates were few, may be more advisable to determine which part of the database needs to be transferred. To do so, global identifiers for the transactions are used, so that the replica that will send the information for the recovery, can determine the last transaction that was correctly executed in the recovering replica and with this, the pending updates to send. It has the same replication model exposed before. For the recovery it uses the version-based transference model, with pessimistic concurrency control, a unique manager for the concurrency and the recovery work does not fully suspend the execution of the application, the write operations are only delayed on the objects that are not yet transferred, only the changed data are transferred to recovering replicas and the read lock can be released immediately on the not changed data. The disadvantages are that it is necessary to review the entire database to determine the objects to transfer, which can cause overload. An updated object is locked since the begin of the data transfer transaction until it is either transferred or considered non-relevant. Finally, note that not all the DBMSs can mark the objects with version numbers (i.e., with the identifiers of the transactions that generated their current values) as is required by this recovery protocol.

Reducing the Amount of Data to Check. Using a so-called "reconstruction table" can alleviate the disadvantages exposed in the previous subsection. It is a data structure to store information about recently updated data. A record in this table consists of a row identifier and a global identifier for the last transaction that updated the row. Each update is recorded in the reconstruction table, unless all sites have successfully performed the update.

In contrast to the row level locks of the previously discussed protocol options, this one only needs to set a single lock on the entire database. Once the incremental data set to be transferred is determined, that lock is replaced by fine-grained row level locks on the respective data items.

Replication is as before, and the incremental recovery is accomplished with pessimistic concurrency control and a unique centralized version-based manager site for controlling concurrency and the distribution of recovery tasks.

The main advantages are that version numbers are not needed. Hence, its implementation is more independent of the underlying DBMS, since labelling takes place modularly in a separate table that can be straightforwardly implemented as a relational table. Also a scan of the entire database is no longer needed, and non-relevant data are locked only for a very short period. The disadvantages are that the use of the reconstruction table demands additional space (that, however should be negligible even for very small devices, which nowadays also dispose of vast amounts of memory). And in spite of a relatively fast release of non-relevant data locks, the read-locking time span of relevant data could be considerable.

Filtering the Log. Up to now, sites have been supposed to set read locks for synchronizing the data transfer with concurrent transaction processing. In the previously discussed optimization, locking of non-relevant data is reduced, but locks on relevant data may still last long. To avoid locks, multiple versions of data can be used, i.e., the use of multi-version concurrency control, as in `PostgreSQL`, `Oracle` and optionally in `MS SQL Server`. In that case, transactions can continue to update the database while earlier versions that have been missed by the recovering site are transferred to it.

Recovery is version-based and incremental, concurrency control is pessimistic, and a unique centralized version-based manager site is used to distribute recovery tasks. Advantages are that transaction execution is not suspended, data transfer is not needed and locks are fully avoided. The disadvantage is that multiple data versions must be kept, but that can be left to the underlying DBMS, as in `PostgreSQL`, so that recovery is not burdened with that.

Lazy Data Transfer. Up to this point, all mentioned solutions use view changes as synchronization points. That is a simple approach but has several drawbacks:

1. The recovering site has to delay transaction processing on data that must be transferred (not necessary in version-based concurrency control).
2. If workload is high and data transfer takes long, then a recovering site might not be able to store all transaction messages delivered during data transfer, or it might not be able to apply these transactions fast enough to catch up with the rest of the system.
3. If the recoverer site fails, the recovering site needs to be reset so that recovery process can re-start all over again.

These drawbacks can be avoided if we decouple the synchronization point from the view change.

Initially the recovering site discards the messages delivered in the view change and the recoverer site starts the transfer. When the transfer is about to complete, the recoverer and the recovering sites determine a delimiter transaction to be delivered in the view change. The recoverer site then transfers all changes performed by transactions with an identifier that is smaller than the identifier of the delimiter transaction. The recovering site starts queuing transaction messages with identifier greater than the identifier of the delimiter transaction and finally applies these transactions once the data transfer is completed. The latter is done in several rounds. Only in the last round (when the delimiter transaction is determined), the transfer is synchronized with concurrent transaction processing by setting appropriate locks. The idea is to send in each round those data that were updated during the data transfer of the last round.

Again, the version-based incremental transfer mode is used for recovery, concurrency control is pessimistic, using a unique centralized multi-versioned manager to distribute recovery tasks. The significant variant of this protocol is that the transfer of the actual database state takes place in lazy mode. So, at least the failures at the recoverer site are handled more efficiently. The disadvantage is that this protocol requires a reconstruction table to maintain the information about recently modified data.

Enriched View Synchrony[6]. EVS makes use of an online reconfiguration of recovering nodes. Hence, the following key problem of all solutions for recovery as discussed so far requires even more attention: view changes can happen before reconfiguration is completed. View changes that occur during reconfiguration can cause considerable complications. For example, suppose that a site X acts as the recovering site of a node that joins the primary view and that X leaves the present view before reconfiguration is completed. At this point, only the node X and the recoverer node knows that reconfiguration has not been completed. All other nodes do not know whether the recovering node is qualified to process transactions nor which nodes need to continue with the reconfiguration process. At worst, this could lead to a primary view in which no member would be able to process transactions.

This complication is due to the fact that a member of a primary view is not necessarily an up-to-date member. In order to handle such situations, an extension of the traditional group communication abstraction is proposed in [43], the *enriched view synchrony* (EVS). Instead of ordinary views, EVS deals with so-called "enriched views", also called *e-views.* An e-view is a view with additional structural information. Sites in an e-view are grouped into non-overlapping sub-views that in turn are grouped into non-overlapping sub-view-sets. A view change then notifies about a change in the composition of the e-view (sites that appear to be reachable); such changes are performed automatically by the EVS. Additionally, EVS introduces e-view change events that notify about changes in the structure of the e-view in terms of sub-views and sub-view-sets. In contrast to view changes, e-view changes are requested by the application through dedicated primitives.

The characteristics of EVS can be summarized as follows: It maintains the structure of e-views across view changes. E-view changes between two consecutive view changes are totally ordered by all sites in the view. Finally, if a site installs an e-view Ev and then sends a message m, then any site that delivers

m delivers it after installing Ev. Note that the original definition of EVS does not consider total order and uniform delivery. However, accommodating these properties will be simple since they are orthogonal to the properties of EVS.

EVS provides simpler algorithms in regard to virtual synchrony. In particular, it provides the subsequently explained characteristics with respect to the incorporation of a site into the primary view (even though it is the DBMS who decides when to start the database transfer). When a site joins a primary view, it is done locally, i.e., it does not matter whether an operational primary view exists or not. When a recoverer site fails before terminating the data transfer to the recovering site, the remaining sites in the primary sub-view know that the recovering site is not updated, so it still is a member of their set of sub-views, but not of their sub-view. When a site enters the primary sub-view, all sites in that view know that the site now is updated and operational.

In short, with EVS, we are able to encapsulate the reconfiguration process, and the database system receives a more realistic picture of what is going on.

Recovery Protocols by Holliday

In [33], protocols called *Broadcast Writes*, *Delayed Broadcast* and *Single Broadcast* (the latter already presented in [1]) for recovery and replication are discussed. According to the classification in [60] as discussed in section 2.3.1, these are update-anywhere and non-voting protocols. Concurrency control is performed by the DBMS with strict two phase locking (*Strict 2PL*). These protocols use a GCS providing virtual synchrony. Virtual synchrony is used to ensure that messages are delivered in the same view in which they were broadcast and that two sites that pass to a new view have delivered the same set of messages in the previous view. These protocols use total order multicast primitives as provided by the GCS for controlling transactions. The explicitly stated objective of these recovery protocols is to minimize system downtime and disruption caused by failures.

Single Broadcast Recovery. When a site fails, the multicast subsystem detects the failure and the membership protocol creates a new view from which the failed site is excluded. The operational sites will receive a view change message. If a commit request message for a transaction T was delivered in the previous view, then it has been delivered to all sites that comprised that view and the transaction was committed or aborted by all the sites in the view. When the failed site that was excluded in the following view recovers locally, it will have obtained the effects of T and all transactions that committed in the view to which it belonged, but not later effects of any later view.

Some GCSs with virtual synchrony provide recovery mechanisms that log delivered messages, so that when a site recovers, missed messages can be executed at this site. That can be used when the Single Broadcast replication protocol is employed. If the GCS does not provide for global recovery, some sites can be assigned to be *loggers* for update messages. Note that messages can be logged intelligently. For example, none of the messages from aborted transactions need to be logged. Thus, the logger only stores view changes and operations of committed transactions. Also note that this change log is different from the recovery log maintained by the DBMS at each site and is used for *local* recovery. When a view change is indicated, the logger makes an entry in the change log, recording membership changes. Also delivered broadcast messages with transactional updates are added to the change log. If the transaction has read obsolete data, the corresponding entry is erased and the transaction aborted. Otherwise, the transaction is executed and committed.

The change log is used as follows. When the communication system detects a membership change and one or more sites are added to the view, no update transaction messages are delivered to it, or to any other site, until the new site has exchanged messages with one logger and the logger indicates that recovery is complete. The logger will then see a view change and a request of the new site to be updated, and will look for the last view in which the site to recover was present. If the site has been absent for a long time and the logger does not have registry of it or is a newly incorporated site, the full database must be transferred. Otherwise, the transactions that were committed after the last view in which the recovering site was a member, are sent to the site in their commit order.

Here, the transfer mode clearly is either FT (cf. Section 2.3.3) or, based on the log, incremental. Concurrency control during recovery is not needed because no transaction is processed until the recovery is complete. Recovery is centralized in a site that also acts as logger. The advantage is that clear decision criteria can be applied for determining whether full database transfers are really necessary and when they can be avoided by sending only the messages lost since the last view to which the failed site belonged. The disadvantages are that no transaction is processed until recovery is completed. Moreover, data versioning is required so that write and commit messages can take notice of stale data reads, in which case the transaction is aborted.

Delayed Broadcast Recovery. The *delayed broadcast* replication protocol decouples the writeset broadcast from the commit broadcast for any transaction. This behavior raises some problems when recovery is being considered. It might happen that the recovering site was able to deliver the writeset for a particular transaction, but not its commit or rollback message. So, that writeset was lost when the site failed and should be retransmitted now by the recoverer site if its commit message was delivered whilst the recovering site was crashed; i.e., messages delivered in a view where the recovering site was up and running might have to be remembered and resent by the recoverer.

Two possible solutions for the problems caused by the writeset-commit decoupling are presented:

1. *Log Update Method*. The loggers must examine their logs or the state of the

database to determine if there exists on progress transactions in the sites without failure. If there are, the logger should mark these transactions so that when the commit or abort message is delivered, if the commit was successful, the Logger will find the record containing the writes for that transaction and copy it to the view change record. So when a previously failed site rejoins to the group, the logger begins with the execution of the writeset of the transactions that were in progress when the site failed, following with the operations of the transactions that were originated and committed while the site was failed. The commit order is the same for all non-aborted transactions. The operations of the aborted transactions are not included in the log since their effects are undone in the sites without failure.

The transfer model for the database update used here is log-based, during the recovery a pessimistic concurrency control is used with a single manager, the recovery work distribution is centralized in a unique site. This protocol has the advantage that it does not need the data versioning used in the single broadcast protocol. The flow of messages is executed in the recovering sites in the original order, recreating with this the same conditions than in the non-failed sites. As a disadvantage we have that the loggers must maintain the logs of previous views whether or not a site fails in case of there were write messages from the terminated transactions in the those views. Additional work is done by the sites that behave as loggers.

2. *Augmented Broadcast Method*. This second method gives additional process to the sites of the on-going transactions and requires a change in the recovery lock manager algorithm. If a site S_j has any transaction in course when a new view is installed (assuming that such a view change implies that a replica has rejoined the system), it modifies the commit protocol in a way that the writesets are included in the commit messages for all transactions that broadcast their writesets in a previous view; the sites that have been operating through the change of view will ignore the writesets and will directly process the commit messages. The sites that are loggers will log the augmented message. This extension is only needed by on-going transactions; i.e., not for those that are started once the recovery process is finished.

   Similar to the previous method, the transfer model for the database update used here is log-based, during the recovery a pessimistic concurrency control is used with a unique manager, the recovery work distribution is centralized. The advantages of this protocol are that data versioning is not needed, the messages are executed in the recovering sites in the original order, recreating with this the same conditions than in the non-failed sites as the previous protocol, but tries to avoid the overload in the Loggers by distributing it towards the sites that have on-going transactions. As disadvantages we have that additional work is done by sites of transactions in course and requires a change to the recovery lock manager algorithm: the write requests are included in the commit request.

Broadcast Writes Recovery. When transactions are long, the Broadcast Writes protocol has a clear advantage over replica update protocols that do not use multicast. We can assume that when a view change occurs, there will be many on-going transactions and it is better not to abort all of them at each view change. Due to this, using database sites as Loggers instead of relying on the recovery mechanism provided by the multicast system could be of significant benefit. The Augmented Broadcast global recovery method presented for the Delayed Broadcast protocol could be used for Broadcast Writes. In Augmented Broadcast, only the final broadcast for a transaction, the commit request, is affected by the need to augment it with the writeset. The method then works as it does for Delayed Broadcast. When the Log Update method is used with Broadcast Writes, the Logger must be careful to remove messages from the log for a transaction that is aborted for any reason. In the case of Delayed Broadcast, only transactions that were aborted at the time of termination request had to be removed from the log. However, with the Broadcast Writes protocol, transactions can be aborted by sites because of deadlocks. If the write requests of two or more transactions cause a deadlock, all operational sites will abort one of the transactions (and the same transaction is aborted at each site). The writes of the aborted transaction are not included in the update portion of the view change record. However, the last write of the transaction to be aborted could be logged and replayed to the recovering site.

In these two last recovery protocols, the update transfer mode is log-based. During recovery, pessimistic concurrency control with a unique manager based on 2PL is used; the distribution of recovery tasks is centralized. The advantages of these protocols are that they are capable of supporting the most general transaction types in a distributed database, without the need of data versioning. Moreover, the second protocol tries to balance the work among *loggers* and the other sites. The disadvantages of the second protocol are the same as for *Delayed Broadcast*, i.e., additional work is burdened upon on-going transaction processing sites, and a change of the recovery lock manager algorithm is needed: write requests are included in the commit request message so that this information is entered into the log. The disadvantage for the case of *Log Update Method* is that loggers must take care of clearing messages from the log for a transaction that is aborted for whatever reason and not only those with an explicit abort message or whose commit message is rejected.

Parallel Recovery by Jiménez, Patiño and Alonso

The proposal for doing the recovery task in a parallel way exposed in [39] is based on a model that consists in a set of database replicas in an asynchronous system. This model is extended with a failure detector. Sites interchange messages through a reliable channel, and no Byzantine failures are considered.

The system is structured in two layers. The first layer has the replication middleware and relies on a GCS. In this middleware the replication and recovery protocols are implemented. Its GCS provides membership service, reliable multicast and the notion of view. The second layer contains the data being replicated, it is assumed that the data is divided into disjoint partitions (or classes) and each one has a master site. The transactions that access data in a given partition should be local to the partition master site; i.e., if a transaction requests processing in a non-master site, this site forwards the request to the partition master site. A site executes only its local transactions; for remote transactions only installs their updates. The transactional system supports strict two phase locking.

The aim of the recovery protocol is to identify the missed transactions in a failed but now recovering site, obtaining these transactions from an active site and applying them in such recovering site. Recovery is made on a partition basis, i.e., each partition is recovered independently from other ones. A partition can be in one of the next states: (a) *online*: those partitions that are working normally; (b) *crashed*: when its master site has failed; (c) *recovering*: when such master site is restarted; (d) *pre-online*, when the recovering has completed its first steps but is not yet *online*.

A partition can be elected as *recoverer* and it changes to that state. The recovery procedure terminates with a forwarding phase during which the partition is in *forwarding* state. A partition can not process transactions from clients during the crashed, recovering or pre-online states, in which only can process transactions associated with the recovery. When a recovering site joins to a working group a view change is performed. As part of this procedure, the recovering site indicates the *log sequence number* (LSN) of the last committed transaction. Once a site is elected as recoverer site, it sends the recovery information to the recovering site. The recoverer site is able to process transactions even in the recovery process.

This protocol can be extended to support parallel recovery in several sites. Thus, the same partition master site is able to multicast missed transactions to multiple recovering sites (if more than one site are restarted at once). Additionally, when a site is recovering, its missed transactions are sent to it from all the master sites that have any transaction to be recovered. So, recovery parallelism is improved from both of these sides.

This recovery protocol assumes a replication protocol based on a primary-copy server architecture, with constant server interaction, non-voting transaction termination, and eager update. The recovery protocol is log-based with a pessimistic concurrency control with a single manager, and with recovery work distribution.

The main advantages of this protocol are that when the recovery task is performed in a parallel form supposes an optimization in the transfer time and load balancing. The single period in which the transactions are not processed is during a view change, when the sites are blocked. This protocol presents the disadvantages of processing the transactions solely in the partition master site, and when failure periods are long the information to transfer may be abundant.

The COLUP Recovery Protocol

In [38] a configurable eager/lazy replication protocol with a lazy recovery protocol is proposed. This replication protocol, called *Cautious Optimistic Lazy Update Protocol* (COLUP), uses the concept of node role, given special importance to a node where a particular object is created. Such node is referred to as the *owner* for all objects created by its local applications. This owner node will be consulted during the voting phase performed at commit time. In this way the owner is the manager for the object accesses and is responsible for coordinating the propagation of the last versions of the object. An identifier for the owner node is included in the identifiers of the objects. For any object, a set of nodes will maintain synchronous copies; i.e., consistent replicas of its state. The other nodes that have a replica of the object constitute the set of asynchronous nodes. In these nodes the updates to the object will be eventually received, once the updates have been committed in synchronous replicas.

Conflicts between transactions are solved in an optimistic way, using object versions and reviewing them during the commit phase. As a result, a transaction is aborted if it has read obsolete values that were updated by other concurrently committed transactions. Thus, access to the objects is allowed with no need of locks. A disadvantage of the lazy updates is that the probability of aborting transactions gets increased. It is necessary then to establish a threshold for the probability of aborting a transaction accessing obsolete object values. Thus, when a transaction tries to access an object, this probability is calculated and compared with the established threshold. As result, the algorithm obtains an updated version for the objects that might be obsolete. Using a high threshold the number of requested updates is minimized, and the number of transactions executed in the system is increased since the used resources for update propagation are decreased. But this may cause an increase in the number of aborted transactions because the number of objects with obsolete values is also increased. This can degrade the system productivity, so it is convenient the use of an algorithm to dynamically adapt the threshold value to an optimal value.

The recovery protocol considers the existence of a membership monitor that is executed on each node. The monitor observes a preconfigured set of nodes, and notifies its local node about the changes in this set. When the membership monitor detects a failed node a notification is sent to each node that remains in the system. This causes an update in the "list of alive nodes". During the execution of a transaction a number of messages must be sent to the different owners of the objects. If a message must be sent to a failed owner, then it will be redirected to a new owner of that object. Each node sends a message with the *previous grants* conceded to the objects by the previous owner. The new owner can process the requests as if it was the original owner node of the object.

When an original owner node recovers from a failure, every alive node is notified by its membership monitor. Then, further messages must be sent to the original owner node. In addition, the recovering node sends a message to the node that managed its owned objects and with this, synchronizes the activity in both nodes. A recovering node may receive requests for objects that were updated during the failure interval. In order to handle this situation, the recovering node must consider each object of which it is owner like an asynchronous replica until it is updated by a synchronous replica. To ensure that a recovering node achieves a correct state for its owned objects, an asynchronous low priority process is executed. This process sends an update request for all non-synchronized objects including the new objects created during the failure period.

The replication protocol is eager update-anywhere, with constant server interaction and voting transaction termination. Recovery uses a version-based transfer model. The concurrency control is optimistic with a distributed manager and multiversioned.

As advantages offered by the recovery protocol we can find that the recovery task is totally supported by the hybrid replication protocol, so the recovery is part of the basic algorithm and it is not necessary to add more code. Another advantage is that the updates are deferred until the recovering node accesses obsolete data. With object versioning the use of locks is not necessary and the rate of aborted transactions is reduced. As disadvantage we found that the time of COLUP for processing transactions is usually greater than in pure lazy replication protocols.

CLOB: Short-Term Failure Recovery

CLOB (Configurable LOgging for Broadcast protocols) described in [12] is defined as a framework for reliable broadcast protocols that are used as a basis for database replication. Its aim is to manage the logging of missed messages in the broadcast protocol core, providing with this automatic recovery for short-term failures, but discarding the log and notifying the database replication protocol modules in case of long-term outages. This kind of support can be easily combined with version-based recovery protocols. To this end, once a failure is detected the database replication protocol must follow its traditional version-based management for recovery purposes, but it will be discarded if the replica is able to rejoin the system soon. In this case, CLOB automatically propagates the missed update messages to the recovering replica, which receives and applies them avoiding any additional waiting time both in the source and destination replicas. On the other hand, if the outage period exceeds a given threshold, the reliable broadcast service will notify the replication protocol about that, discarding the message logs maintained by CLOB and delegating the recovery management to the upper-layer components.

The replication protocol is eager update-anywhere, with constant server interaction but would have to consider some additional parameters to decide when the logged messages can be eliminated. This protocol applies voting transaction termination. The basic support for the recovery based on logs will be identical if the transaction termination is voting or non-voting. During the recovery, the transfer of the state of the database is version-based in long-term failures, but is log-based in short-term failures.

As advantages we can mention that it combines version-based and log-based transfer of information for the recovery depending on which is more advisable, without restricting to a single model of transfer and being able to take advantage of each one in its case. A minimum blocking time for replicas that participate in the recovery is also obtained when the log-based recovery is used. As disadvantages, it is necessary to maintain the related information to both recovery methods, and the transaction service time is increased even with no failures because all messages must be saved in persistent storage.

The FOBr Recovery Protocol

The recovery protocol explained in [13] FOBr, is designed as a complement for the replication protocol FOB (Full Object Broadcast), which is an optimistic eager update-anywhere protocol and makes use of a GCS [15] membership service. In this protocol the concept of replica role is used and it can be:

1. Owner node: Initially it is the node where the object was created; this is what we call physical ownership. However, the node where a set of objects was created might have crashed and the ownership migrates (logical ownership). This owner node is the manager of *access confirmation requests* (ACR) for that object.
2. Synchronous nodes: These nodes did not create the object but are considered up-to-date replicas of it. They provide us with fault tolerance.

Several transactions can be grouped in a session. Since an ACR management is used, the session identifiers (SIDs) include information about the node identifier where it was initiated. The objects are identified in a similar way to the sessions.

Objects are identified similarly as Sessions with object identifiers (OIDs). These OIDs hold several information, including the owner node, that identify them univocally through all the nodes. Besides the OID, the consistency protocol may need (FOB does) to maintain extra information associated to each OID such as version numbers, timestamps,...

This metadata information will also need to be transferred when a recovering node receives its updated information. When the user initiates a commit, the protocol performs several operations before it is effectively applied into the database:

1. It collects the transaction writeset and groups its OIDs by their owner node.

2. An ACR is sent to each owner node of these writeset objects. The owner nodes decide then whether to grant or revoke the access to these OIDs. This sending is performed sequentially and in ascending order of node identifiers in order to avoid multiple abortions.
3. The node receives the ACR responses and:
    - If any ACR is revoked, the transaction must abort. If any of the other ACRs was granted, a message must be sent to that node in order to release the grants.
    - If they are all granted, the transaction is propagated with a reliable broadcast and when delivered, it is committed in all the nodes. When a node receives this broadcast: (i) It aborts other locally conflicting sessions that are still in early phases of operation; (ii) It applies the changes into the database; and, (iii) It releases the ACRs granted to the finished transaction.

The recovery protocol has two phases:

1. Collection phase: It includes all the events that happen since the moment a node fails until the moment it joins to the system. Two steps are taken:
    - The remaining alive nodes decide in a deterministic way, which ones of them inherit the ownership of the faulty node objects.
    - A structure is created in each alive node in order to hold the *OIDs* of all objects that will be updated while these nodes are not present. The structure needs to be stored persistently in order to allow recovery in a total system failure. In this structure a recovery list is also stored, and it saves the updated *OIDs* that any site lost during a view change.
2. Recovery phase: it includes all the steps followed by the nodes of the system when a failed node initiates operations again. In the recovery phase we distinguished two roles for the participant nodes: the recovery node that is the node that is trying to join the system and needs to update its database, and the previously active node that is the node that has the information to help the recovering nodes to join the system.

   The recovery phase begins when the previously-active nodes receive a notification, by the membership service, about the recovery of some previously considered failed node. This notification has two parameters, the recovering nodes list and the actual view number. Then, the following steps are taken:
    - The previously-active nodes build a `JOIN_UPDATE` message to update the currently owned objects that they know the recovering nodes have missed. This `JOIN_UPDATE` message is built following this procedure:

(i) The recovery list is checked to obtain the set of updated *OIDs* that the node currently owns. (ii) The set of *OIDs'* states is retrieved from the database and it is included in the message in order to update the recovering node database. (iii) The set of missed *OIDs* and network views is also included, because the recovering node needs to hold recovery information until the system is complete. However, this information is not transferred if the currently recovering node is the latest one; i.e., no other faulty node exists when it has finished its recovery.

- The recovering node waits until it has received the `JOIN_UPDATE` message from all previously-active nodes. As soon as a `JOIN_UPDATE` message arrives, the recovery list is reconstructed with the information provided by the message. The recovering node will have created a transaction to apply all the updates that it had to receive. Once committed, the recovering node sends a **MERGED** message to all the previously-active nodes and waits for a **NO ACT**(**NO ACT** stands for "node active") response.
- When the previously-active nodes receive the **MERGED** message they know that the recovering node has applied all the remaining updates. If the **MERGED** receiver was not the inheritor of the recovering node objects, it simply assumes that the recovering node has recovered the ownership. If the receiver is the inheritor, it has to migrate this ownership packing the *ACR* granted locks into a **NO ACT** message and send it to the recovering node.
- Finally, when the recovering node receives the `NO ACT` message, will be able to manage its objects and the recovery is completed.

According to this, the replication protocol is eager update-anywhere, with constant server interaction and with voting transaction termination. The recovery protocol has a version-based transfer model, the concurrency control is optimistic with a distributed manager and with multiversion. The recovery work is distributed.

The advantages offered by the recovery protocol are that minimizes the amount of data to transfer, balances the recovery work, and allows the execution of transactions during recovery time. It has low space requirements. Its disadvantages are: (a) it complements a replication protocol (FOB) with a non-standardized isolation level; (b) for each transaction that commits, we must explore its write-set (and save the *OIDs* contained in it) if there was any failed node.

Recovery Protocols by Armendáriz

In [3] three eager update replication protocols are considered, and a recovery protocol that can be applied to all of them is proposed. The first replication protocol called Basic Replication Protocol (BRP) is based on the optimistic 2PL

(O2PL). As a result of the addition of improvements and variations to protocol BRP, is presented the second protocol called Enhanced Replication Protocol (ERP). This replication protocol reduces response times and transaction abortion rates by removing the Two Phase Commit (2PC) rule and the use of queues. Finally, the third replication protocol, named Total Order Replication Protocol with Enhancements (TORPE), that makes use of the total order multicast primitive provided by the GCS to ordering the transactions executed by the system. The main idea for the recovery proposed in [3] once a node re-joins the network after failure, an alive recoverer node is appointed. It informs the joining node about the updates it has missed during its failure. Thus a dynamic database partition (hereafter DB-partition) of missed data items, grouped by missed views, is established, in recovering and recoverer nodes, merely by some standard SQL statements. The recoverer will hold each DB-partition as long as the data transfer of that DB-partition is going on. Previously alive nodes may continue to access data belonging to the DB-partition. Once the DB-partitions are set in the recovering node, it will start processing transactions, which are, however, blocked when trying to access a DB-partition. Once the partitions are set in the recoverer, it continues processing local and remote transactions as before. It will only block for update operations over the DB-partition.

The three replication protocol are eager update-anywhere with constant server interaction. BRP has voting transaction termination, whilst ERP and TORPE have non-voting termination. The recovery protocol is version-based, the concurrency control during the recovery is optimistic with a distributed manager and with multiversion. The recovery work is distributed.

The main advantages are that recovery is distributed, the DB-partition in a recoverer site can be released even when the recovery process is not concluded, and that transactions can be accepted and committed in recoverer sites if they do not interfere with the DB-partitions being recovered. The disadvantage is that if DB-partitions are defined on the basis of each view modified items, an object may be transferred several times, to avoid this we must "compact" the DB-partitioning.

2.4 Conclusions

This Chapter provides a detailed review of several replication techniques and recovery protocols suitable to these replication protocols.

The use of different replication techniques involves the use of different data structures. The required information to implement this replication varies from one protocol to another, so the information available for recovery depends directly on the replication protocol used. We have emphasized that a replication protocol should not be considered complete without contemplating how the recovery will take place. It would be desirable that alongside the development of the replication protocol, the recovery protocol was also developed. However, in the literature has not always been so. The use of the guarantees that a group communication system may provide contribute to the development of more efficient and better structured replication and recovery protocols, yet there are several opportunity areas in this scope. So, we should consider the hardware evolution as well as the use of new paradigms, such as the increased use of portable devices that suggest more customer mobility and the cloud computing.

As a concluding remark we advise to consider recovery algorithms that use version-based management and that distribute the recovery work among available replicas to balance the workload during the recovery process. Very few replicated database recovery systems are capable to combine these techniques to reduce recovery times. When it has been partially possible (as in [38, 39]), it was because replication protocols had some special characteristic (the use of a primary copy schema in [39], that reduces flexibility and might compromise fault tolerance; the use of lazy updates in [38], that may compromise consistency). The work presented in [3] could be a good exception, but it has not presented performance measurements that confirm its good theoretical performance. This analysis will be used as a basis for designing new recovery protocols trying to combine the advantages of the protocols analyzed.

Chapter 3

Optimizing Certification-Based Database Recovery

Certification-based database replication protocols are a good basis to develop replica recovery when they provide the snapshot isolation level. For such isolation level, no readset needs to be transferred between replicas nor checked in the certification phase. Additionally,these protocols need to maintain a historic list of writesets that is used for certifying the transactions that arrive to the commit phase. Such historic list can be used to transfer the missed state of a recovering replica. We study the performance of the basic recovery approach –to transfer all missed writesets– and a version-based optimization –to transfer the latest version of each missed item, compacting thus the writeset list–, and the results show that such optimization reduces a lot the recovery time.

3.1 Introduction

Replication has been the regular solution for achieving high availability. But such level of availability requires that crashed replicas were recovered. Database replication is a special kind of highly-available service since in this case replica recovery implies the application of the missed updates, being inefficient a complete state transfer since it needs a long time to be completed. Even transferring only the missed updates, there is no easy way to complete such recovery in a short time.

There have been many good works devoted to database replication recovery [3, 12, 33, 39, 43], but almost none of them has provided a rigorous performance study of the proposed approaches. This chapter is focused to show that replica recovery is not easy when the load of the replicated system is not light, and that some optimizations can partially overcome such problem. To this end, we have tried to select the database replication kind [63] that provides the best support for developing an easy recovery: certification-based replication. In this replication variant a historic list of the applied writesets needs to be maintained in order to certificate transactions (i.e., validate and locally decide in each replica about the success of each terminating transaction). Such a historic writeset list can be stored and used for transferring the missed updates to recovering replicas. Additionally, the resulting replication protocol does not need any voting termination [60] and provides very good performance if the conflicting rate is low [63]. Moreover, for the snapshot isolation level, a certification-based replication protocol is the natural solution, since it does not demand readset transfers. So, such kind of replication protocol provides an ideal basis to research on replica recovery and a basic recovery protocol can be easily developed.

But such a basic recovery protocol does not provide good performance (i.e., a short recovery time). So, some optimizations are needed in order to get acceptable results. To this end, we have combined a version-based approach, similar to those proposed by other research groups (e.g., in some of the recovery variants of [43]) and in some of the previous papers written by the SiDi group [12, 13] but specifically adapted to a certification-based replication protocol. Such optimization introduces a negligible overhead and shortens the recovery time, as shown in Section 3.6.

The rest of this chapter is structured as follows. Section 3.2 presents the assumed system model. Section 3.4 describes the replication protocol taken as the basis for our recovery proposals. Section 3.5 thoroughly explains the recovery strategies. Section 3.6 discusses the performance results. Finally, Section 3.7 presents some related work and Section 3.8 gives the conclusions.

3.2 System Model

We assume a partially synchronous distributed system –where clocks are not synchronized but the message transmission time is bounded– composed by N nodes where each one holds a replica of a given database; i.e., the database is fully replicated in all system nodes. These replicas might fail according to the partial-amnesia crash failure model proposed in [17], since all already committed transactions are able to recover but on-going ones are lost when a node crashes. We consider this kind of failures as we want to deal with node recovery after its failure.

Each system node has a local DBMS that is used for locally managing transactions. On top of the DBMS a middleware is deployed in order to provide support for replication. More information about our MADIS middleware can be found in [37, 46]. This middleware also has access to a group communication service (GCS, on the sequel).

A GCS provides a communication and a membership service supporting virtual synchrony [15]. The communication service features a total order multicast for message exchange among nodes through reliable channels. Membership services provide the notion of view (current connected and active nodes with a unique view identifier). Changes in the composition of a view (addition or deletion) are delivered to the recovery protocol. We assume a primary component membership [15]. In a primary component membership, views installed by all nodes are totally ordered (there are no concurrent views), and for every pair of consecutive views there is at least one process that remains operational in both views. The GCS groups messages delivered in views [15]. The uniform reliable multicast facility [32] ensures that if a multicast message is delivered by a node (faulty or not) then it will be delivered to all available nodes in that view. All these characteristics permit us to know which writesets have been applied in the context of an installed view. In this work, we use Spread [55] as our GCS.

We use a replication protocol based on certification [63], which does not require any kind of voting in order to decide how a transaction should be terminated (either committing or aborting).

3.3 Certification-Based Replication

This replication technique is described in [48] as "database state machine", with a similar approach with technique A4 described in [34] and called *certification-based* by [63]. As stated in section 2.3.2 the delegate server receives transactions from clients and executes these transactions, nothing is broadcast until the commit time when the writeset and the readset are broadcast to all servers using a total order broadcast. Upon delivering this message, a certification phase is locally executed to decide if the transaction commits or aborts. Because all replicas share the same history of delivered messages, they can detect conflicts, if any, with concurrent transactions. Only one total order broadcast per transaction is needed.

3.4 Replication Protocol

We have selected the SIR-SBD protocol (see Figure 3.1) described in [46] for a case study of our recovery mechanisms, because it is a good sample of a certification-based [63] database replication protocol, providing the snapshot isolation level [7] and thus avoiding the transfer of transaction readsets.

This protocol uses an atomic multicast [32], i.e., a reliable multicast with total order delivery, and thus it ensures that the writesets being multicast by each replica at commit time are delivered in all replicas in the same order. It uses two data structures for dealing with writesets: `ws_list`, which stores all the writesets known (i.e., delivered) until now, and `tocommit` queue, which holds

```
Initialization:
  1. lastvalidated_tid := 0
  2. lastcommitted_tid := 0
  3. ws_list := $\emptyset$
  4. tocommit_queue_k := $\emptyset$
I. Upon operation request for $T_i$ from local client
  1. If select, update, insert, delete
    a. if first operation of $T_i$
      - $T_i$.start := lastcommitted_tid
      - $T_i$.priority := 0
    b. execute operation at $R_k$ and return to client
  2. else    /* commit */
    a. $T_i$.WS := getwriteset($T_{ik}$) from local $R_k$
    b. if $T_i$.WS = $\emptyset$, then commit and return
    c. $T_i$.priority := 1
    d. multicast $T_i$ using total order
II. Upon receiving $T_i$ in total order
  1. obtain wsmutex
  2. if $\exists$ $T_j \in$ ws_list : $T_i$.start $<$ $T_j$.end $\wedge$
    $T_i$.WS $\cap$ $T_j$.WS $\neq \emptyset$
    a. release wsmutex
    b. if $T_i$ is local then abort $T_i$ at $R_k$ else discard
  3. else
    a. $T_i$.end := ++lastvalidated_tid
    b. append $T_i$ to ws_list and tocommit_queue_k
    c. release wsmutex
III. $T_i$ := head(tocommit_queue_k)
  1. if $T_i$ is remote at $R_k$
    a. begin $T_{ik}$ at $R_k$
    b. apply $T_i$.WS to $R_k$
    c. $\forall$ $T_j$ : $T_j$ is local in $R_k$ $\wedge$ $T_j$.WS $\cap$ $T_i$.WS $\neq \emptyset$
       $\wedge$ $T_j$ has not arrived to step II
       (this is analyzed by our conflict detector,
       concurrently with the previous step III.1.b)
       - abort $T_j$
  2. commit $T_{ik}$ at $R_k$
  3. ++lastcommitted_tid
  4. remove $T_i$ from tocommit_queue_k
```

Figure 3.1: SIR-SBD algorithm at replica R_k

those writesets locally certified but not yet applied in the local database replica. Moreover, for each transaction, the attributes `start` and `end` hold something similar to the transaction start and commit timestamps, respectively. Due to the total order multicast and the behaviour of the protocol, the second counter is the same for a system transaction in all the replicas, i.e., all the replicas identify with the same commit timestamp a system transaction –this will be handy when studying the performance graphs.

Note that we have tacitly assumed that the underlying database system is supposed to be able to check for conflicts, and to abort transactions whose access patterns violate the snapshot isolation level rules.

This protocol is also based on the existence of a block detection mechanism [46]. We have assigned the following priorities to the transactions. All transactions are initialized with a 0 priority level. They get level 1 when they are multicast in their local node or when their writeset is delivered in their remote nodes. This ensures the correctness of this alternative, since our blocking detection mechanism aborts a transaction only if all of these conditions are satisfied. Otherwise, no particular action is taken:

- The transaction to be aborted is local.
- It has not locally requested its commit; i.e., its writeset has not been multicast.
- The transaction that causes its abortion has been generated for applying a remote writeset.

This approach satisfies the correctness criteria of the snapshot isolation level, since the writeset above mentioned is associated to a transaction that has successfully passed its global validation phase. It already has a commit timestamp which of course is in the range of the [start, commit] interval of the local transaction, since the latter has not yet requested its commit.

3.5 Recovery Strategies

We describe a basic recovery in Section 3.5.1 and its optimized version in Section 3.5.2. The optimization consists in compacting the list of missed writesets, maintaining only the last version of each missed item.

3.5.1 Basic Recovery

As a general overview of the main goal of our recovery protocol, let us say that one node (recoverer) will transfer the missed writesets to the recovering node arranged by their respective versions. This means that user application transactions executed on the recovering node will run under GSI [25] in a slower replica. As it may be seen there are no restrictions to execute user transactions in the replica and transactions executing at other replicas will behave as if nothing happens in the system.

A recovering replica R_i joins the group, triggering a view change. As part of this procedure, the recovering protocol instance running in R_i multicasts an *ask-for-help* message indicating the $version_i$ of its last applied writeset –this version corresponds to the commit timestamp of the last transaction applied in that node. No message activity in the recovering node is done –all messages delivered are ignored– until this message is delivered. At this moment, the recovering node starts to enqueue the total order delivered messages –with writeset information about other transactions in the system sent by the rest of the replicas– to be processed later.

In parallel to this, a deterministic procedure takes place to choose a recoverer replica. The recoverer replica (R_j), after receiving the *ask-for-help* message, starts a recovery thread that sends a point-to-point message with all the missed writesets starting from $version_i + 1$, i.e., the recoverer node sends the portion of its `ws_list` that covers from $version_i + 1$ to the end of the `ws_list` at that moment. Note that this `ws_list` is one of the elements on which the replication protocol algorithm is based, and it can also be used for our recovery purposes, as it contains all the information we need. This way, we reuse the data maintained by the replication protocol, minimizing the overhead introduced by the recovery support in normal operation (i.e. no additional data collection is needed to support possible recovery processes). Note also that this approach guarantees that the recovering node will receive, on one hand, the writesets from $version_i+1$ to the last known version of the recoverer at the moment of the *ask-for-help* message reception. On the other hand, the writesets delivered in total order in the system after the *ask-for-help* message will be enqueued in the recovering message buffer. This way, it is ensured that the recovering node will not miss any writeset.

When this point-to-point recovery message is delivered to the recovery protocol, it stores this information in both the `ws_list` and the `tocommit` queue, as all these writesets were already certified in the recoverer node. Then, the replication protocol is ready to directly apply in the database the writesets in the `tocommit` queue and to start certifying its own enqueued total order messages –delivered after the *ask-for-help* message. Note that the certification of the enqueued messages must wait for the recovering information to be stored in the `ws_list`, as this structure is used in the certification process, but it is not necessary to wait to the application of these missed writesets in the database. In other words, just after the storage of the transmitted writesets in both data structures, the recovering node can act as in normal mode.

This kind of recovery inherits the main ideas of the second approach described in [43] ("Data transfer within the database system") and, up to our knowledge, had been already implemented and studied in other projects (e.g., GlobData, in order to add recovery capacity to the protocols presented in [52], but its performance results were only described in an internal project report).

3.5.2 Compacting

This basic procedure can be enhanced by compacting the point-to-point message in order to minimize the transmission and application time. The point-to-point recovery message has to provide all the changes in the database made from $version_i$+1 to the current version of the recoverer node. This information can be sent in a raw mode, i.e., sending the writesets of all the transactions committed during this period of time. Then, in the recovering node, each writeset is applied in a new transaction –like any other replica does in normal function. This is the way used in the basic recovering protocol explained before.

All this procedure can be enhanced if the recoverer replica elaborates a special writeset composed by the last version of each modified object in all the transactions committed during the crash time, i.e., if the same object was modified by more than one transaction, only the last version of it would be transmitted along with its corresponding `end` timestamp. This special writeset, built only in the recoverer replica at recovery time, would be applied by the recovering replica in a single transaction, which can greatly improve the committing time, not only for being just one –although possibly big– transaction, but also for avoiding useless updates of the same object. This way, compacting will reduce both the transmission and the checking time as we will see later in the performance results. The time needed by the recoverer node to prepare this compacted message is not negligible, but we will see in the graphs that it does not imply any noticeable overhead.

Note also that the regular function of the replication protocol is not compromised by this optimization. Indeed, the recovering replica can start processing transactions immediately. The writesets transferred in the recovery message are not needed by the recovering replica in order to certify any new local writeset, since such new writesets should be certified against the writesets regularly delivered in the new view in which such recovering replica has rejoined the group. However, such compacted writesets can be needed for certifying remote transactions in such recovering replica, but its compacted version is enough for such kind of certification. Note that can exist long remote transactions that have started before the recovery process started, and their [start, commit] interval might overlap the end of some transactions included in the compacted missed writesets. Since at least the latest version of each missed updated item is present in such compacted set, all conflicts detectable with the original writeset list will be detectable with such compacted sequence. For instance, assume that there were N transactions T_1, T_2, ...,T_N in the original missed writeset list and that each writeset contained M items a_{11}, a_{12}, ...,a_{1M}, ..., a_{N1}, a_{N2}, ..., a_{NM}, and each of these transactions has a consecutive logical commit timestamp (t_i for T_i, being t_{i+1}=$t_i + 1$). Without generalization loss, let us assume that there are only M/2 items per writeset that have not been updated in any of its successive writesets (except in the last writeset of such compacted list that is the single one that cannot be compacted –their updated items are their trivially latest versions in the recovery transfer set–), being a_{iK_i} those items (where $K_i \subset \{j \in \mathbb{N} : 1 \leq j \leq M\}$ and $|K_i| = M/2$). So, if a given "future" transaction T_j was started between, e.g. T_1 and T_2, its writeset should be checked against all writesets in the range [T_2, T_{j-1}]. Note that T_j has been terminated after T_N, and as a result of this, all items updated by all transactions in the range [T_2, T_{j-1}] are also included in its "compacted variant" since $N < j - 1$, and our compacting process guarantees that only items rewritten during the [T_1,T_{N-1}] interval are removed from the T_1..T_N writeset sequence (but if any item has not been rewritten, it appears in such compacted sequence, and this guarantees that exactly one version of all original writeset elements appears in the compacted version). Additionally, we have the advantage of a boost in the checking time, since instead of having the complete sequence of [T_1, T_N] writesets, we only have a compacted item sequence $a_{1K_1}..a_{NK_N}$, as assumed above (i.e., half of the items, in this hypothetical example).

Our optimization shares some of the characteristics of the fifth recovery strategy described in [43] ("Restricting the set of objects to check") but further optimizes that technique. To this end, our compacting is able to restrict the objects being checked without needing any additional table where the objects are being recorded during the crash interval. Additionally, it still shares the advantage of getting such set of items to be transferred without requiring any read lock nor global read operation on the items stored in the regular database tables. But, on the other hand, it is partially dependent on the replication protocol approach (certification-based), and can not be easily adapted to all other database replication variants (e.g., the active and weak-voting variants [63] do not need any historic writeset log).

3.6 Performance Study

In this work we intend to measure several aspects of our recovery implementation:

- Under which circumstances (work load and crash length) a failed node can recover and reach the state of the other replicas.
- How long does it take to reach the state of the other replicas.
- Compacting impact.

To accomplish the comparison, we use PostgreSQL [49] as the underlying DBMS, and a database with a single table with two columns and 10000 rows. One column is declared as primary key, containing natural numbers from 1 to 10000 as values.

All protocols have been tested using our MADIS middleware with 4 replica nodes. Each node has an AMD Athlon(tm) 64 Processor at 2.0 GHz with 2 GB of RAM running Linux Fedora Core 5 with PostgreSQL 8.1.4 and Sun Java 1.5.0. They are interconnected by a 1 Gbit/s Ethernet. In each replica, there is a varying number of concurrent clients (from 4 to 12). Each client executes an endless stream of sequential transactions, each one accessing a fixed number of 20 items for writing, with a fixed pause of 500 ms between each consecutive transaction. Each test begins with the execution of 500 global transactions, after that, a failure occurs in a random replica (the failure of a replica consists in the termination of its process). The failure lasts for a period in which a varying number of global transactions is executed by the other replicas. After this time, the failed node restarts and begins the recovery process until it reaches the state of any of the other replicas. The test continues once the recovery ends, until the completion of 500 more global transactions, when the experiment finishes.

In the figures we show the evolution of nodes in committing transactions in the system. All the transactions have a global identifier –the end counter– and must be committed locally in each replica. This way, one global transaction requires a local transaction in each replica, and we can know how quick a node goes by seeing the last committed global identifier at that node (see the vertical axes in both Figure 3.2 and Figure 3.3). This way, each graph shows this evolution in three nodes in the system: the failed, the recoverer and another node. The bigger the slope of that curve, the faster the node goes committing global transactions.

The results obtained show that the basic recovery technique was very poor in comparison with the compacting approach. Both recovery techniques were tested allowing the immediate start of new local transactions in the recovering replica.

The results obtained without compacting (see Figure 3.2.a) and light load show that the recovering node can easily reach the current state of the system. We can see in the figure that all the replicas have a linear evolution and when the failure occurs, the failed node does not make any advance –and so its line is horizontal. Then, when the recovery process begins, the recovering node starts to progress with more slope than the other nodes, i.e., it commits more transactions per second, and thus it can reach the global state and continue with the same previous linear behavior.

With medium and heavy loads (Figures 3.2.b and 3.2.c) the recovery trend is too slow and the recovering replica is not able to cope with its intended work. Note that our MADIS middleware is not a commercial prototype and performance is not our main goal. So, the load parameters considered are relative to the current capabilities of our middleware. This way, with the previously described conditions and 12 clients per node, the system has to deal with near 20 transactions per second, which is our middleware saturation point. Because of this, this load parameter combination has been called heavy.

Obviously, the recovering node can catch the other replicas as long as the current load in the system provides enough idle time (due to no clients to attend) to apply the missed writesets. Local transactions starting in the recovering node during the recovery process are very likely to abort because of their outdated snapshot. These local transactions will delay the application of the missed writesets due to the conflicts arisen in the underlying database. Thus, the lighter the load of local transactions in the recovering node, the faster these missed writesets will be applied. To sum up, the system load affects in two ways: it determines the idle time available to reduce the gap with the other replicas, and also the amount of local transactions in the recovering node possibly delaying the application of these missed writesets. In the light environment, replicas have an important amount of this idle time and the recovering node has no problem to catch the rest. In the medium one, the replicas have little idle time, and the amount of local transactions make impossible to reduce the gap. And in the heavy case, there is no idle time and local transactions even broaden this gap.

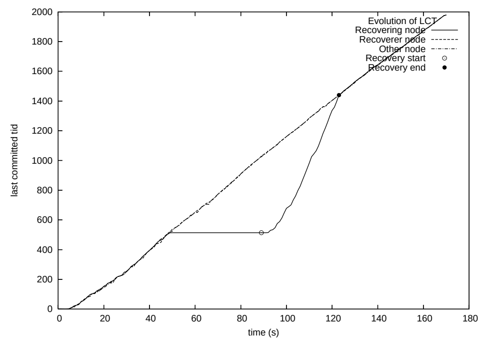
As it can be seen in the graphs, the compacting technique (see Figure 3.3) allows the failed replica to quickly achieve a state close to those of the other replicas. In all the tests, the recovering replica is able to do so without too much delay. Specifically, when the load and the crash time are small, we can observe that the evolution of the recovering node after the crash is not as progressive as in the basic technique, but it has two phases. The first one is a big step towards the global state due to the application of the compacted writeset; and the second, the final evolution during the application of the enqueued messages delivered in the meanwhile. Comparing with the previous basic technique, it can be noticed that the recovery process lasts quite less with the compacting approach. Indeed the recovery time is only 30 seconds with this approach, while it was 42 seconds with the basic recovery; i.e., almost a 28.6% reduction.

The next graph (Figure 3.3.b) shows the behavior of the system when both the load and the crash time are medium. The shape of the curves is similar to that of a light load and the recovery process is much more faster than with the basic technique. In the example provided in our figures, the recovery takes only 72 seconds with the compacting optimization, whilst the recovery was not possible using the basic recovery strategy.

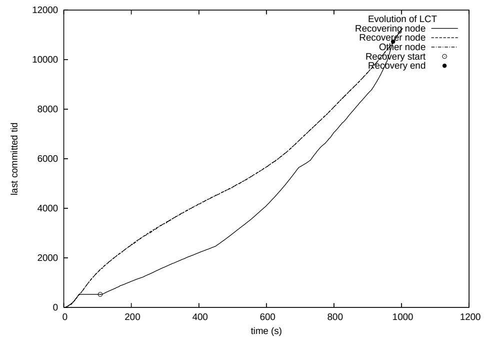
Finally, when the load and crash time are maximum (Figure 3.3.c), the optimized technique increases its completion time, as expected, but it is still able to complete the recovery in an acceptable time (92 seconds).

Note that in these two last cases –with a medium and heavy load–, the time needed for completing the recovery is quite long –72 and 92 seconds, respectively–, but the non-recovering replicas and the recoverer one have been able to process new transactions at a regular pace; i.e., their availability is not compromised by the recovery of another replica. Additionally, the recovering replica has accepted new transactions as soon as possible, and this introduces a non-negligible delay in its recovery, but also shortens a lot the interval where the service in such replica is not available.

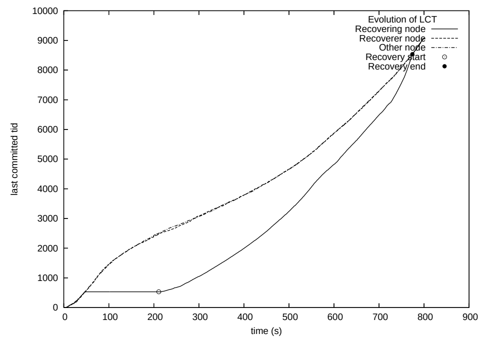
More tests were performed disabling the start of new local transactions in the recovering replica in order to quantify the improvement when both techniques



a) Light load (4 clients), short crash (500 trans)

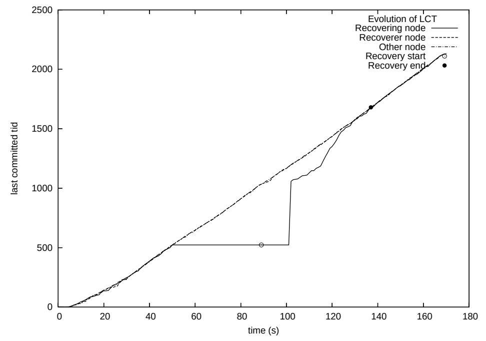


b) Medium load (8 clients), medium crash (1000 trans)

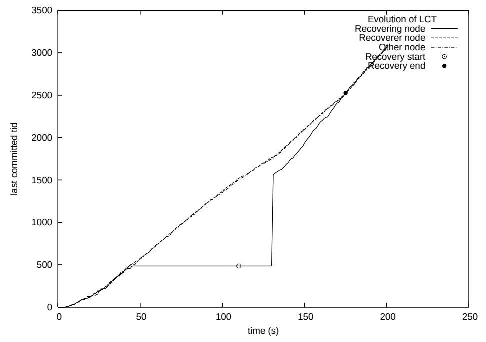


c) Heavy load (12 clients), long crash (2000 trans)

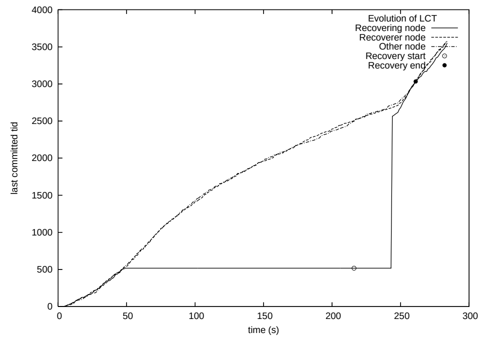
Figure 3.2: Recovery without compacting



a) Light load (4 clients), short crash (500 trans)



b) Medium load (8 clients), medium crash (1000 trans)



c) Heavy load (12 clients), long crash (2000 trans)

Figure 3.3: Recovery with compacting

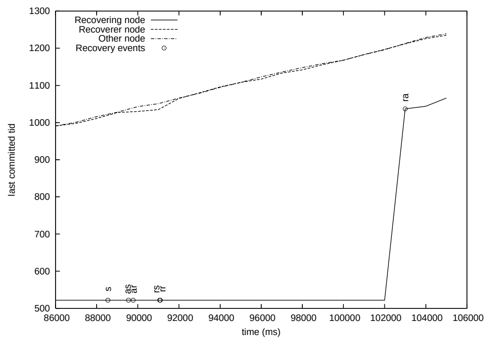


Figure 3.4: Recovery events

achieve the completion of the recovery process. The results show that, in a medium loaded environment the compacting technique achieves the completion of the process in 36 seconds while the basic one lasts 59 seconds (38.98% of improvement). In the heavy loaded environment case, the reduction obtained is up to 60.82% (97" versus 38"). Note that the compacting technique, even allowing new local transactions in the recovering node, is faster than the basic one without these new local transactions.

In order to analyze more deeply the cost related to each step of the recovery process, Figure 3.4 has been included showing the recovery events from the starting point of the recovery process to the compacted writeset application in the case of a light load environment. These events are the following: *s*, the starting point of the recovery (when the failed replica restarts); *as*, the *ask-for-help* message is sent by the recovering node; *ar*, this *ask-for-help* message is received in the recoverer node; *rs*, the recoverer node sends its compacted point-to-point message; *rr*, this point-to-point message is completely received and preprocessed in the recovering node (the information contained is provided to the necessary components); *ra*, the recovering node finishes the application of the writesets contained in the recovery message. At this point, the recovery algorithm is done, but the recovery process will last until the the recovering node, already in a normal function, processes all its buffered messages and the incoming ones until it reaches the state of the rest of the replicas.

This way, the interval between *s* and *as* includes the start of the MADIS replica and its node recovery process (regarding global issues independent of the repositories and protocols used), as this is not in the scope of this work, no further analysis is necessary. The interval between *as* and *ar* is the time the group communication toolkit takes to deliver the *ask-for-help* message. The interval *rs*-*rr* is the time between the end of the submission and the end of the preprocessing of the recovery message; this time is, in all cases, negligible as the implementation used overlaps the creation, sending, receiving and processing of the message (the recovery message is transmitted via a stream). The two interesting intervals are, therefore, *ar*-*rs* and *rr*-*ra*. The first one is the time needed to create and send the compacted message in the recoverer node and it can be seen in the graph that during this time, the recoverer node decreases its performance drastically –indeed, it is unable to process more transactions in such interval– but only momentarily due to the effort required to compact the data. Note that such compacting period lasts approximately two seconds in this example, but the regular pace of other nodes is retaken by the recoverer in less than one additional second. The second important interval is the time needed to apply locally all the updates contained in the compacted message. This is the greatest interval, as expected, as it implies interaction with the underlying database. As showed, the compacting technique introduces an extra step (only a light overhead compared with the total processing time) but achieves notable reductions in the recovery time.

The enlarged graphs corresponding to the other two cases show similar results to the above exposed.

To sum up, the optimization presented in this work has been able to reduce the recovery time to a 71.4% of the original recovery time in a lightly loaded environment. Moreover, with 1000 missed transactions, the basic recovery technique failed to complete the recovery, since the recovering replica was not able to process the queue of received writesets on time, and its receiving queue was continuously growing, whilst the optimized version does not get overloaded with ten times such load.

3.7 Related Work

The use of version-based recovery protocols –the same approach taken as the basis of our proposed optimization– had been already suggested in the fourth and fifth recovery variants of [43], but in both cases still demanded a lot of effort for maintaining the set of versions to be transferred to each crashed replica. Either a version-based DBMS was assumed or a special additional table needs to be managed and updated each time a transaction commits. Our research group used the latter solution in [12, 13] but in both papers such protocols were designed as a recovery approach for a replication system that did not provide any standard isolation level. Those solutions were developed in our COPLA system [27], and such middleware was targeted to provide an object-relational translation, with an object-oriented programming interface where the traditional isolation levels did not match. In this work, we have optimized the version-based approach taking as its basis a certification-based [63] replication variant in order to support the snapshot [7] isolation level.

Only in [12] and [38] there are some performance analyses of database recovery protocols. But, as already said, [12] is penalized by its non-standard features (non-standard API and non-standard isolation level), whilst the replication protocol assumed in [38] was hybrid (could be configured either as eager or lazy, but always with a lazy core) and this introduced a high abortion rate that was partially compensated with an outdateness estimation function. In all cases, the advantages of both approaches –and both were developed by our research group– have been improved by the solution presented now (shortest recovery time, and lowest abortion rate).

There have been many other works devoted to database replica recovery [3, 13, 33, 39, 43] but, up to our knowledge, none of them has presented a performance study of their proposed solutions.

3.8 Conclusions

We have presented a first basic recovery approach for certification-based recovery protocols, analyzing its recovery time when the system load varies. Up to our knowledge this is the first performance study for such kind of recovery techniques in the field of database replication. Although certification-based replication protocols provide a good basis for developing recovery protocols, this first basic approach can be easily improved. A possible optimization based on a missed update compacting has also been presented. The performance study shows that the overall recovery time can be reduced up to a 68% of the recovery time of the basic approach, in the less favorable configuration for the optimized technique.

Chapter 4

Improving Recovery in Weak-Voting Data Replication

Nowadays eager update everywhere replication protocols are widely proposed for replicated databases. They work together with recovery protocols in order to provide highly available and fault-tolerant information systems.

In this chapter we provide a compacting mechanism to the recovery information. With this, we try to minimize the time needed to transfer and apply the missed information at the recovering replica. The idea of this enhancement is to obtain a more efficient recovery protocol. Through a simulation, we verify the good performance of the algorithm with the compacting improvement.

For simplicity's sake the Omnet++ simulation environment is briefly described. After that, we explain the developed simulation program. The experimental parameters are also specified, as well as several considerations related with the experiments.

4.1 Introduction

As previously stated in Chapter 2 replication protocols can be designed for eager or lazy replication [31], and for executing updates in a primary copy or at all node replicas [60]. With eager replication we can keep all replicas exactly synchronized at all nodes, but this could have an expensive cost. With the lazy alternative we can introduce replication without severely affecting performance, but it can compromise consistency. Many replication protocols are based on eager update everywhere with a *read one, write all available* (ROWAA) approach

[9]. As we have briefly highlighted before, these replication protocols provide high availability. However, only a few of them deal with the possible reconnection of the failed node, which is managed by recovery protocols [12, 39, 43, 2].

The aim of the recovery protocols is to bring failed or temporarily disconnected nodes back into the network as fully functional peers, by reconciling the database state of these recovering nodes with that of the active nodes. This could be done by logging transactions and transferring this log to recovering nodes so they can process missed transactions, or transferring the current state of the items that have been updated in the database since the recovering node failed.

This chapter is focused in the recovery protocol for eager update everywhere replication protocols, proposing one optimization to the work presented in [2]. This enhancements includes amnesia support [20] and a better performance reducing the amount of data to save in the actions done before recovering and the amount of data to transfer at recovering time. The main idea in the last case is to compact recovery data eliminating redundant information.

In addition, we provide a table with the results of a simulation, where the advantages of the compacting approach are verified.

The rest of this chapter is distributed as follows. Section 4.2 provides the system model. Section 4.3 briefly describes the weak-voting replication technique. Section 4.4 deals with the basic recovery protocol. Section 4.5 explains the necessary actions for the amnesia support. Next, Section 4.6 relates the process of compacting recovery information. Later, Section 4.7 describes the simulation environment, parameters and shows the simulation results followed by the related works in Section 4.8. In the final Section 4.9, we provide our conclusions.

4.2 System Model

The basic recovery protocol has been designed for database replicated systems composed by several replicas –each one in a different node–. These nodes belong to a partially synchronous distributed system: their clocks are not synchronized but the message transmission time is bounded. The database state is fully replicated in each node.

This replicated system uses a group communication system (*GCS*) [15]. Point-to-point and broadcast deliveries are supported. The minimum guarantee provided is a FIFO and reliable communication. A group membership service is also assumed, that *knows* in advance the identity of all potential system nodes. These nodes can join the group and leave it, either explicitly or implicitly by crashing, raising a *view change event*. Therefore, each time a membership change happens, i.e. any time the failure or the recovery of one of the member nodes occurs, it supplies consistent information about the current set of reachable members as a view. The group membership service combined with the *GCS* provides *Virtual Synchrony* [15] guarantees. A *primary component* [15] model is followed in case of network partitioning.

The replicated system assumes the *crash-recovery with partial-amnesia* [17] model. This implies that an outdated node must be recovered from two "lost of updateness": forgotten state and missed state. This assumption supports a more realistic and precise way to perform the recovery process. So the assumed model allows to recover failed nodes from their previous crashing state maintaining their assigned node identifiers.

Consequently, when a node crashes, every active node must abort any transaction started by the failed node whose commit messages have not been yet delivered. A similar behavior is adopted when the system can not go on because the progress condition has been lost. In this situation, the nodes in minority (e.g. disconnected) must also abort the started transactions whose commit message has not been yet delivered. Thus, the whole activity that was not committed during the working life is aborted.

4.3 Weak Voting Replication

This replication technique is described in [42] as the "Serializability Protocol", and is presented by [63] with the name of Weak Voting. As we mention in Subsection 2.3.2 the delegate server executes the transactions issued by the clients delaying the write operations, when the transaction commit is required, a total order broadcast is only done for the writeset to all servers. Once delivering the writeset, the delegate sever can determine if conflicting transactions have been committed and decide if the current transaction is commited or aborted.

This last is communicated to the other servers with a new broadcast that may not be totally ordered, but must be reliable. The voting is said to be weak as only the delegate server can decide on the outcome of the transaction. Other servers cannot influence this decision and must abide by the delegates decision.

4.4 Basic Recovery Protocol

Our basic proposal is inspired in the recovery protocol presented in [2]. It has been designed for *eager update everywhere* database replication protocols and proposes the use of *DB-partitions* (see below). It was originally designed for providing recovery support for the *ERP* and *TORPE* [2] replication protocols. Such protocols use a voting termination approach [60], and can be considered as weak voting replication protocols [63]. This basic recovery protocol can be outlined as follows:

- The system has a database table named $MISSED$, which maintains all the information that will be needed for recovery purposes. Each time a new view is installed a new entry is inserted in the $MISSED$ table if there are failed nodes. Each entry in $MISSED$ table contains: the view identifier, the identifiers of crashed nodes in this view –$SITES$–, and the identifiers list of data items modified during this view –OID_LIST–. The two first ones are set at the beginning of the view, while the last one grows as long as the view passes.

- When a set of crashed nodes reconnects to the replicated system, the recovery protocol will choose one node as the *recoverer* with a deterministic function. Then in a first step the *recoverer* transfers the metadata recovery information to all reconnected nodes. This metadata information contains: the identifiers of modified items, and the crashed node identifiers in each view lost by the oldest crashed node being recovered. The per-view metadata generates a DB-partition during the recovery process; i.e., such items will be blocked while they are being transferred to the recovering node, logically partitioning the database. These *DB-partitions* are also used in order to block in each replica the current user transactions whose modified items conflict with its *DB-partitions*. Subsequently, the *recoverer* starts to recover each *recovering* node view by view. For each lost view, the *recoverer* transfers the state of the modified items during this view. And, once the view has been recovered in the *recovering* node, it notifies the recovery of this view to all alive nodes. The recovery process ends in each *recovering* node once it has updated all its lost views.

- As a transaction broadcast is performed spreading two messages –*remote* and *commit*–, it is possible that a reconnected node receives only the second one, without any information about the updates to be committed. In this case the replication protocol will transfer the associated writesets to these nodes. This behavior implies that transaction writesets are maintained in the sender node until the *commit* message is broadcast.

But this recovery protocol presents the following two problems:

- Amnesia phenomenon. Although we are assuming the *crash-recovery with partial amnesia* [17] failure model, many systems do not handle it in a perfect way. This problem arises because once the replication protocol propagates the *commit* message associated to one transaction, and it is delivered, the system assumes that this transaction is being committed locally in all replicas. But this assumption even using strong virtual synchrony [15] is not always true. It is possible that a replica receives a transaction *commit* message, but before applying the commit the replica crashes, as it is commented in [62] –the basic idea is that message delivery does not imply correct message processing–. The problem will arise when this crashed node reconnects to the replicated system, because it will not have committed this transaction and the rest of the system will not include among the necessary recovery information the updates performed by this transaction, arising then a problem of replicated state inconsistency.

- Large $MISSED$ table and redundant recovery information. If in the system there are long-term crashed nodes –meaning nodes failed during many views– and there are also high update rates it is possible that the $MISSED$ table enlarges significantly with high levels of redundant information, situation that is strongly discouraged. Redundant recovery information will appear because it is possible that the same item has been modified in several views where the crashed nodes set is very similar. In this case if an item is modified during several views, only knowing the last time –meaning the last view– it was updated is enough. Therefore, it will be interesting to apply algorithms that avoid redundant recovery information, because the larger $MISSED$ tables the greater the recovery information management overhead becomes.

In the following section we will present an approach for solving these problems improving the basic recovery protocol.

4.5 Amnesia Support

In order to provide amnesia support different approaches can be considered. These approaches can be classified depending on which recovery information they use. On one hand, there are the ones using the broadcast messages –log-based– [12, 39] and, on the other hand there are the ones using the information maintained in the database –version-based– [43, 2].

But before describing how the amnesia support can be provided in the basic recovery protocol, it must be considered how this amnesia phenomenon manifests. In [23], it is said that the amnesia phenomenon manifests at two different levels:

- *Transport level.* At this level, amnesia implies that the system does not remember *which messages have been received.* In fact, the amnesia implies that received messages non-persistently stored are lost when the node crashes, generating a problem when they belong to transactions that the replicated system has committed but which have not been already committed in the crashed node.

- *Replica level.* The amnesia is manifested here in the fact that the node “forgets” *which were the really committed transactions.*

The information maintained in order to perform the amnesia recovery process will be the broadcast replication messages. In this replication protocol two messages for each propagated transaction: *remote* and *commit.* The amnesia recovery must be performed before starting the recovery of missed updates –the latter will be done by the basic recovery protocol–. The amnesia recovery process will consist in reapplying the messages belonging to non really committed transactions.

A transport-level solution consists in each node storing persistently the received messages, maintaining them as long as the associated transaction, t, has not been committed and discarding them as soon as t its really committed in the replica. But, the message persist process must be performed atomically inside the delivery process as already discussed in [62] with its "successful delivery" concept. Moreover, messages belonging to aborted or rolled-back transactions must be also deleted.

Once the amnesia phenomenon is solved at transport level, it is necessary to manage the amnesia problem at replica level. At this level the amnesia implies that the system can not remember which were the really committed transactions. Even for those transactions for which the "commit" message was applied, it is possible for the system to fail *during* the commit. Then the amnesia recovery process in a replica will consist in reapplying (and immediatly deleting, in the same transactional context) the received and persistently stored messages in this replica that have not been already deleted, because it implies that the corresponding transactions have not been committed in the replica. These messages are applied in the same order as they were originally received.

It also must be noticed, that in this process is not needed to apply the *remote* messages whose associated *commit* messages have not been received, because it implies that they have been committed in the subsequent view, and therefore their changes are applied during the recovery of its first missed view.

Finally, once the amnesia recovery process ends, the basic recovery protocol mechanism can start.

4.6 Compacting Recovery Information

In order to increase the performance at the moment of determining and transferring the necessary information for the synchronization of recovering nodes, we propose some modifications based on packing information that enhance the basic recovery protocol described in [2]. This could be done by compacting the records in the $MISSED$ table, and with this, minimize the items to transmit and to apply them in the recovering node, reducing thus the transmission and synchronization time.

Originally the $MISSED$ table stores in each record, i.e. view, all nodes that remain in failure in the view, being able to repeat several times –one by each view– the identifiers of the nodes that were in failure in previous views. Similar to crashed nodes, the identifiers of missed updated objects can be repeated in different $MISSED$ view entries for being modified these objects in two or more views where there were failed nodes.

In regard to the failed nodes identifiers the compressing solution relies on the idea that it is enough to know the first view in which the node was failed and the view when it reconnected to the replicated system. Therefore, the recovery protocol must transfer the updates performed from the first view it was failed until the view it reconnected. Then, it is only necessary to store the identifier node in the first view it was crashed.

The item identifiers can be packed due to the fact that the recovery information only maintains the identifiers of updated items. The state of these items is retrieved by the *recoverer* from the database at recovering time. Moreover, if a *recovering* node, k, has to recover the state of an item modified in different views lost by k it will receive as many times the item value, but transferring its state only once is enough. As a consequence, it is not relevant to repeat the identifier of an updated item across several views, being only necessary to maintain it in the last view it was modified and can be erased, if it is, in other previous views.

During DB-partition generation, as user transactions are blocked, there is no compacting process going on in the system. Hence, possible generation of non-correct DB-partitions is avoided. Once this metadata has been transferred, establishing the *DB-partitions*, the compacting process is restarted. This blocking process is not necessary if the whole set of failed nodes in the previous view is contained in the current set of failed nodes. In fact, it must be remarked that this work behavior is already provided by the original recovery protocol due to the established *DB-partitions*, which block any update access.

Whenever one (or more than one) node fails, the recovery protocol starts the execution of the actions to advance the recovery of failed nodes. To this end,

- A new view is installed, and a new record in the $MISSED$ table is inserted, containing the new view identifier and the identifiers of set of nodes that was present in the previous view and are no longer present in this new view, as an initial packing for the $MISSED$ table, only the identifiers of the recently failed nodes are saved in the field $SITES$ of this record.
- When a transaction commits, the field which contains the identifiers of the updated items, OID_LIST, will be updated in the following way:
    1. For each item in the $WriteSet$, the OID_LIST is scanned to verify if the item is already included in it or not. If it is not, it is included and is looked for in previous views OID_LIST, eliminating it from the OID_LIST in which it appears, compacting thus the OID_LIST, i.e. the information to transfer when a node recovers.
    2. If as a result of this elimination, an OID_LIST is emptied, the content of the field $SITES$ is included into the field $SITES$ of the next record, and the empty record in the table $MISSED$ can be eliminated.

When a node reconnects to a replicated system, the new view is installed and some actions for local recovery may be performed at the recovering node. The other nodes know who is the recovering node, and every one performs locally the next actions:

1. A recoverer node is elected with a deterministic procedure, according to the original protocol, the oldest one with the bigger identifier.
2. The *MISSED* table is scanned looking for the recovering node in the field *SITES* until the view that contains the recovering node is found. The items for which the recovering node needs to update its state are the elements of *OID_LIST* of this view and the subsequent views.
3. At the recoverer node, the recovery information is sent to the recovering node according to the basic protocol.
4. Once the recovering node has confirmed the update of a view, the node is eliminated from the *SITES* field in this view, and if it is the last item, also the record that contains this view is eliminated.
5. If a recoverer node fails during the recovering process, then another node is elected to be the new recoverer, according to the basic protocol. And it will create the partitions pending to be transferred, according to the previous points, and then it will perform the item transfer to recovering nodes, again as in the basic protocol.

It is important to note that in a view change consisting in the join and leave of several nodes, we must first update the information about failed nodes, and later execute the recovery process. As a final remark, this compacting process will help the recovery protocol to minimize the needed recovery information to be transferred. However, its compression rate will depend on the user application. If replication updates concentrate in few data items among several views the compacting will have high rates, but if these changes are highly scattered the compacting rate values will be low.

4.7 Recovery Simulation

With the aim of evaluating the performance of the compacting enhancement for the basic protocol described in Section 4.4 we have developed a simulation program and execute various experiments analyzing the results.

4.7.1 Simulation Environment

The simulation program is written in C++ and is based on the Omnet++ simulation environment [35]. OMNeT++ has been developed by Andrs Varga based on the previous work by Dr. Gyrgy Pongor, Omnet written in Object Pascal at the Technical University of Budapest, Department of Telecommunications.

OMNeT++ is a discrete event simulation environment. Its primary application area is the simulation of communication networks, but because of its generic and flexible architecture, is successfully used in other areas, such in our case.

This simulation environment also provides a component architecture for models. Components (modules) are programmed in C++, then assembled into larger components and models using a high-level language called NED. As an additional remark, Omnet++ has also an extensive graphics user interface – GUI – support, mainly used for debugging purposes.

4.7.2 Simulation Model

The simulation system model is written in C++, and it is compound of hierarchically nested modules. Modules communicate through message passing. The modules at the lowest level of the module hierarchy encapsulate behaviour.

We have considered three replicated scenarios with 5, 9 and 25 nodes each one. The replicated database has 100000 data items. All simulations start having all replicas updated and alive. Then, we start to crash nodes one by one –installing a new view each time a node crashes–, until the system reaches the minimum primary partition in each scenario. At this point two different recovery sequences are simulated. In the first one, denoted as order 1, the crashed nodes are reconnected one by one in the same order as they crashed, while in the second, denoted as order 2, they are reconnected one by one but reversing their crash order. In both cases, each time a node reconnects a new view is installed, and immediately the system starts its recovery, ending its recovery process before reconnecting the following one. In any installed view we assume that the replicated system performs 250 transactions successfully, and each transaction modifies 20 database items. All simulation parameters are described in Table 4.1.

The items in the writeset are obtained randomly with a uniform distribution. We have not used neither a hot spot, as in other previous works [41], nor typical workloads as TPC-W or TPC-C [36]. In both cases, they would be more favorable environments for the compacting method than a uniform distribution, since they suppose more frequent access to a set of items of the database, removing a big amount of items in the compacting process. We have also assumed a fast network, and this reduces the performance difference between the normal and compacting recoveries, since it only depends on the amount of transferred items. If we had a slow network, such difference would have been bigger.

Parameter	**Value**
Items in the database	100.000
Time for a read	4 ms
Number of servers	5, 9, 25
Time for a write	6 ms
Transactions per view	250
Time for an identifier read	1 ms
Transaction length	20 modified items
Time for an identifier write	3 ms
Identifier size	4 bytes
CPU time for an I/O operation	0,4 ms
Item size	200 bytes
Time for point to point message	0,07 ms
Maximum message size	64 Kbytes
Time for broadcast message	0,21 ms
CPU time for network operation	0,07 ms

Table 4.1: Simulator Parameters.

4.7.3 Simulation Results

We have made one hundred repetitions for every experiment obtaining with this, the guarantees of a low dispersion (see Table 4.2).

This simulation has not considered the costs of managing the recovery information compacting because this work is performed online, therefore its associated overhead penalizes only the replication work performance, but not the recovery.

The simulation results show that the more views a crashed node loses the better the compacting technique behaves, which is a logical result. In fact, when more updates a crashed node misses the probability of modifying the same item increases. Both in the Table 4.2 and in the Figure 4.1 we can observe the same behavior. When a crashed node has lost only one view the compacting technique does not provide any improvement because it has been unable to work. But, as long as the crashed node misses more views the compacting technique provides better results.

It must be also noticed that the basic recovery protocol could arrive to transfer a greater number of items than items has the original database. This occurs because it transfers for each lost view all the modified (and created items in this view) independently they are transferred when recovering other views where these items have been also modified. This situation is avoided by our recovery protocol enhancement. And in the worst case the proposed solution will transfer the whole database because during the inactivity period of the recovered node all the items of the database have been modified.

Order	*Nodes*	*Views*	*Basic*		*Compacted*	
			Avg	*StdDev*	*Avg*	*StdDev*
1	5	2	165.8	0.23	161.7	0.21
2	5	1	82.8	0.20	82.9	0.18
2	5	3	248.7	0.18	236.7	0.16
1	9	4	331.6	0.19	308.1	0.18
2	9	1	82.9	0.17	82.9	0.20
2	9	3	248.7	0.17	236.7	0.18
2	9	5	414.5	0.18	376.0	0.17
2	9	7	580.4	0.18	501.9	0.17
1	25	12	995.1	0.18	767.2	0.12
2	25	1	82.9	0.20	82.9	0.19
2	25	3	248.7	0.19	236.7	0.16
2	25	5	414.6	0.18	376.1	0.15
2	25	7	580.4	0.19	502.1	0.14
2	25	9	746.2	0.19	616.0	0.12
2	25	11	912.0	0.19	719.2	0.11
2	25	13	1077.9	0.19	812.6	0.11
2	25	15	1243.8	0.19	897.1	0.10
2	25	17	1409.6	0.19	973.6	0.10
2	25	19	1575.5	0.19	1042.9	0.09
2	25	21	1741.3	0.19	1105.4	0.08
2	25	23	1907.2	0.18	1162.0	0.07

Table 4.2: Recovery times in seconds.

4.8 Related Work

For solving the recovery problem [9] database replication literature has largely recommended the crash recovery failure model use as it is proposed in [12, 39, 13, 43, 2] while process replication has traditionally adopted the fail stop failure model. The use of different approaches for these two areas is due to the fact that usually the first one manages large data amounts, and it adopts the crash recovery with partial amnesia failure model in order to minimize the recovery information to transfer.

The crash-recovery with partial amnesia failure model adoption implies that the associated recovery protocols have to solve the amnesia problem. This problem has been considered in different papers as [62, 23, 22] and different recovery protocols have presented ways for dealing with it. The *CLOB* recovery protocol presented in [12] and the *Checking Version Numbers* proposed in [43] support amnesia managing it in a log-based and version-based way, respectively.

In regard to the compactness technique, [16] uses it in order to optimize the database recovery. In this case, this technique is used to minimize the information size that must be maintained and subsequently transferred in order to

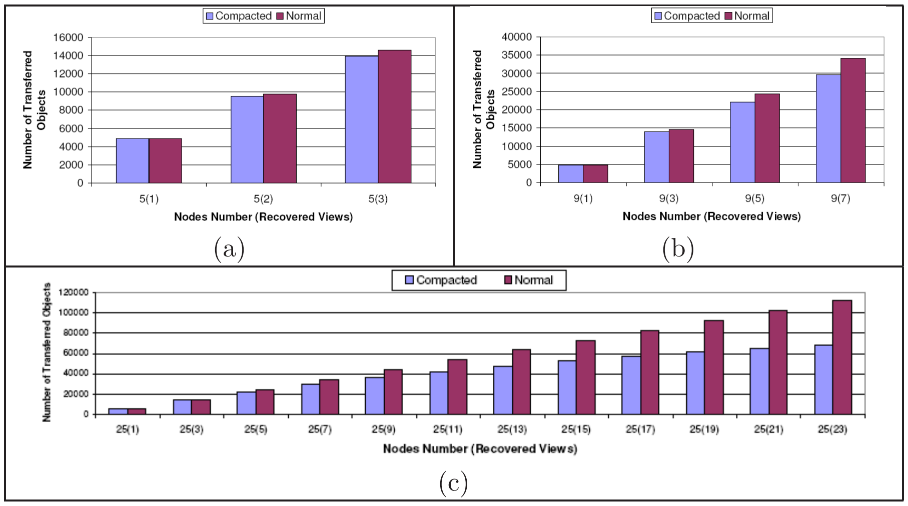


Figure 4.1: Item Compactness: (a) 5 nodes, (b) 9 nodes, (c) 25 nodes.

perform the recovery processes. Such paper also presents experimental results about the benefits introduced by using this technique, reaching up to 32% time cost reductions.

The background idea of our compacting technique is very similar to the one used in the recovery protocol presented in [43] under the "Restricting the Set of Objects to Check" title. This protocol maintained in a database table the identifiers of the modified objects when there were failed nodes. Each one of these object identifiers was inserted in a different row, storing at the same time the identifier of the transaction which modified the object. Therefore, when an object was modified the system checked if its identifier was already inserted in this table. If it has not, the protocol created a new entry where inserted the identifier object and the transaction identifier. If it already existed an entry with this object identifier, the protocol simply updated in this entry the transaction identifier. So, this recovery protocol also avoids redundant information, but it uses a more refined metadata granularity –transaction identifier– than our enhanced protocol –view identifier–.

4.9 Conclusions

In this Chapter we have reviewed the functionality of a recovery protocol for weak-voting data replication. We have enhanced it providing an accurated amnesia support and incorporating a compacting method for improving its performance.

The amnesia support has been improved using a log-based technique which consists in persisting the messages as soon as they are delivered in each node, in fact they must be persisted atomically in the delivery process.

Our compacting technique avoids that any data object identifier appears more than once in the *MISSED* table. Then, this mechanism reduces the size of recovery messages, both the ones that set up the DB-partitions and the ones which transfer the missed values.

Obviously, we must say that the improvement provided by our approach depends on the replicated system load activity, the update work rate, and the changed items rate. For the first two ones, we can consider in a general way that when higher they are better our compacting technique behaves. This is because the probabilities of modifying the same item in different views increase. This consideration drives us to the changed items rate, which is really the most important parameter. It tells us if the performed updates are focused in few items or not. Then for our technique it is interesting that changes are focused in as few items as possible. In fact, the worst scenario for our technique will be the one in which all the modifications are performed in different items.

With the tests made with the simulation model the advantages of the enhanced recovery protocol have been verified when comparing the results of both protocols. The obtained results have pointed out how our proposed compacting technique provides better results when the number of lost views by a crashed node increases. Thus, our compacting technique has improved the recovery protocol performance for recoveries of long-term failure periods.

As final conclusion, we can say that our enhanced recovery protocol works better in some of the worst scenarios from a recovery point of view: when the crashed node has lost a lot of updates and the changed items rate is not very high.

Chapter 5

Correctness

This chapter provides the correctness of the recovery algorithms detailed in the previous chapters. We have to extend the correctness criteria for replication protocols to recovery ones. Hence, for safety the ordering of transactions applies also to the recovering processes. Regarding liveness, we require that processes which (re)join the system could recover the missed changes and reach a state where those transactions were accepted by the replication protocol as if the process had never crashed.

5.1 Introduction

In order to prove the correctness of the algorithms used, we use the correctness criteria proposed in [4]. So, both safety and liveness properties are required. Informally, a safety property specifies that nothing “bad” will happen, ever, during the execution of a system. Besides, a liveness property provides that something “good” will eventually happen. In order to ensure the system consistency, all transactions must commit in the same order at all available replica sites. Beside this safety property, a data replication protocol must guarantee other liveness properties such as the atomicity of a transaction:

- If a transaction commits at a site, it will eventually commit at all sites.
- If a transaction aborts at a site, it will eventually abort at all the sites where such transaction had started or it will be discarded at all the sites where the transaction had not yet started.

Our recovery protocols do not violate the correctness criteria of the replication protocols and we argue the correctness of these recovery protocols assuming that there is always a primary component and that at least one replica, maintaining all the metadata needed for the recovery, transits from one view to the next one.

As stated in subsection 3.4, due to the total order multicast and the behaviour of the protocol, all the replicas identify with the same commit timestamp a system transaction, this timestamp can be used for both, the replication and recovery protocols as a medium to uniquely identify the writesets of a transaction. This last is very valuable in the recovery protocol because it can be used to deal with the amnesia phenomenon as established in chapter 3. Additionally, the next considerations will be helpful for the correctness proof. We assume a recovering replica called R_i, which joins to the primary partition in the current view $\mathcal{V}_n$ indicating the last $version_i$ of its last applied writeset, i.e., the commit timestamp of the last transaction applied by it. Assume as well, a recoverer replica selected by a deterministic procedure inside the recovery protocol, on the sequel R_j.

5.2 Correctness of Certification-Based Database Recovery

Correctness proof for the certification-based recovery protocol can be argued if it can be proved that the replication protocol is deterministic and the consistency with the rest of the replicas is achieved by the recovering replica. Furthermore, the protocol achieves the same isolation level as the replication protocol. To this end, through Lemma 1, we argue that no updates are lost while no view changes are present. Within Lemma 2, we assert that no updates are lost even when view changes occur during the execution. Finally, in the theorem 3 the consistency of the recovered replica is justified.

Lemma 1 (Absence of Lost Updates in Executions without View Changes)**.** *If no failures and view changes occur during the recovery procedure, and the recovery procedure is executed until completion, a replica can resume transaction processing in that partition without missing any update.*

Proof. (Outline). Considering the original recovery protocol, i.e., without compacting the writesets, let us consider a set of recovery transactions executed only at R_i associated to each set of tuples in *ws_list* structure to be applied. Thus, the set of tuples to be delivered will be the writesets from $version_i + 1$ to the last known version of the writesets in R_j at the moment of the ask_for_help message is delivered. On the other hand, we must consider the set of concurrent transactions that commit during the recovery process. The writesets of committed transactions before the ask_for_help message delivery will be included in the recovery transactions transferred from R_j to R_i. The writesets delivered in total order after the ask_for_help message delivery will be enqueued in R_i message buffer.

Recovery transactions are sequentially executed at the recovering replica R_i according to the order they were stored in *ws_list* structure at recovering replica R_j. Concurrent execution of transactions from the begining of the current view until delivering of ask_for_help message were already certified by the replication protocol in the rest of the replicas, therefore they are included in the frame portion of *ws_list* that should be transferred to R_i. Transactions after the ask_for_help message delivery can be certified by R_i itself, after updating the *ws_list* structure and even before writing these values in the database.

All of these, ensures that R_i will receive all writesets from $version_i + 1$ until last version established at R_j when the ask_for_help message is delivered. The writesets delivered in total order after the ask_for_help message delivery will be buffered at R_i. This way, it is proved that R_i will not miss any writeset.

Let us consider now the optimization proposed in the recovery protocol providing compactness when the recovery transactions are constructed at R_j. This is done constructing the recovery transactions to be transferred to R_i, considering only the last version of the objects included in the *ws_list* structure, in the specified interval. So, we are considering all the updated objects, can be seen that this compacting operation does not affect this argument. □

Lemma 2 (Absence of Lost Updates after a View Change)**.** *A recovering replica R_i in view $\mathcal{V}_n$ that transits to view $\mathcal{V}_{n+1}$ resumes the recovery process without missing any update.*

Proof. (Outline). We have to consider the cases when the recoverer replica R_j fails. Otherwise, the recovery process remains unaffected. If R_j fails, it will force the selection of another recoverer replica R_k, restarting the recovery of R_i from scratch. However, the $version_i$ being communicated to R_k might be grater than that communicated to R_j. Note that R_j might have transferred some prefix of the list of missed writesets to R_i. Hence, no updates will be missed since we will be under the circumstances of Lemma 1. Assuming that the time length of installed views is stable enough to eventually achieve the recovery of a replica without continuously failing recoverers. □

Theorem 3 (GSI Recovery)**.** *Upon successful completion of the recovery procedure, a recovering replica reflects a state compatible with the GSI execution that took place.*

Proof. (Outline). According to [24] it is sufficient to show that if a given replication (or recovery) protocol using SI replicas provides global atomicity and commits update transactions in the same order at all replicas it provides GSI. To prove that this implementation is deterministic and obeys GSI rules, we need to show two properties. The first property is that at the certification of a transaction, all replicas have the same *ws_list* and *version*. From the replication protocol point of view, as we are assuming a replication protocol that ensures

GSI, the concurrent committed transactions during the recovery process are applied in the same order at all alive replicas. On the other hand, by Lemmas 1 and 2, we have shown that the recovery does not produce lost updates and missing updates in the recovering replica (nor at any other available node, since the certification process remains the same at the rest of replicas) are applied in the same order they are committed. □

5.3 Correctness of Weak-Voting Based Database recovery

Correctness proof for this recovery protocol can be argued if the consistency with the rest of the replicas is achieved by the recovering replica, and at the end of the recovering the serialization order is not opposed with the serialization order of the replication protocol. To prove this, in the theorem 6 the resulting serialization order is stated to be compatible trough Lemma 4 and Lemma 5, where we argue that no updates are lost while no view changes are present neither when view changes occur during the execution respectively.

Lemma 4 (Absence of Lost Updates in Executions without View Changes)**.** *If no failures and view changes occur during the recovery procedure, and the recovery procedure is executed until completion, a replica can resume transaction processing that DB-partition without missing any update.*

Proof. (Outline). Let ν_{id} denote the last installed view identifier at the recovering node R_i before it crashed and allow $\{\mathcal{P}_{\nu_{id+1}}, \mathcal{P}_{\nu_{id+2}},..., \mathcal{P}_{\mathcal{V}_i.id-1}\}$ represent the set of recovery DB-partitions. Let us denote $\{t_{r\nu_{id+1}}, t_{r\nu_{id+2}},...,t_{r\mathcal{V}_i.id-1}\}$ as the set of recovery transactions associated to each previous DB-partition that must be applied. In the same way, let us denote t_1,..., t_f as the set of generated transactions during the recovery process, assume they are ordered by the time they were firstly committed. Let us denote by t_{rec}, as the last committed transaction before the chosen recoverer switches from the *alive* to the *recoverer* state. The set of concurrent transactions with the recovery process are: $\{t_{rec+1},..., t_f\}$. Therefore, the sequence of transactions can be divided as follows:

- Subsequence $\{t_1, ..., t_{rec}\}$. Transactional atomicity is guaranteed by the underlying transactional system. Hence, upon restart, none of these transactions may be lost and the effects of uncommitted transactions do not appear in the system, by means of the procedure responsible for creating DB-partitions. These transactions are the ones that have attempted to update a DB-partition. No transaction has been issued by the recovering node. These committed transactions are applied at all alive nodes in the same order since they are total-order delivered.
- Subsequence $\{t_{rec+1}, ...t_f\}$. All DB-partitions are set. Transactions committed in this interval comprised data belonging to data not contained in

DB-partitions or yet recovered. As these transactions are totally ordered by the GCS, the serialization is consistent at all nodes. Thereby, and since no failures occur, the subsequence $\{t_{rec+1}, ...t_f\}$ is applied to the underlying system in its entirety at all alive nodes.

- Subsequence $\{t_{r_{\nu_{id+1}}}, t_{r_{\nu_{id+2}}},...,t_{r_{\mathcal{V}_i.id-1}}\}$. Each one of these transactions encloses a set of user transactions issued in each missed view. These transactions will be committed as soon as they are received (freeing its associated DB-partition). Again, as there are no failures all missed updates will be applied at node R_i and it will switch to the alive state.

- Subsequence $\{t_{f+1},...\}$. These transactions correspond to all alive nodes in the alive state and will be governed by the replication protocol, which guarantees the application of updates serially at all alive nodes.

Since no transaction can be lost in any subsequence, the lemma is proved. □

Lemma 5 (Absence of Lost Updates after a View Change)**.** *A recovering replica R_i not yet fully recovered, in view $\mathcal{V}_i$ that transits to view $\mathcal{V}_{i+1}$ resumes the recovery process without missing any update.*

Proof. During the processing of the view change under consideration, if the recoverer has crashed, a new one is elected. Otherwise, the recoverer from the previous view continues as recoverer. In either case, the recoverer in view $\mathcal{V}_{i+1}$ will start the recovery thread, which will multicast the missed updates from the last installed view of the recovering node R_i, referred as ν_i.

- First case. In this case, none has received the recovery metadata message, so a new node is elected as the recoverer and sends a message indicating the start of recovery to the recovering node which, this last restarts the recovery process.

- Second case. The failed node is R_j or R_i not yet fully recovered. The former recoverer R_j has sent the DB-partitions to be set, but it failed during the data transfer of missed updates. At worst, for all DB-partitions to be sent. This may imply that some missed views have been transferred by R_j but its application has not finished at R_i and hence they are multicast twice. However, this second message will be discarded. Hence, the recovery process will continue by the new recoverer, as soon as its DB-partitions are set, with the data transfer of left missed updates. As it has been seen, recovery is resumed without missing any update thereby proving the Lemma.

□

It is important to note that due to the non-total-ordering nature of the recovery transaction generation and the application of missed updates, R_i will not read 1CS values, during the recovery process. This is the price to pay to maintain a higher degree of concurrency and availability. However, once all updates are applied our recovery protocol guarantees that it is 1CS.

Theorem 6 (1-Copy-Serializable Recovery)**.** *Upon successful completion of the recovery procedure, a recovering node reflects a state compatible with the actual 1CS execution that took place.*

Proof. According to Lemmas 4 and 5, a recovering node that resumes normal processing at transaction t_{f+1}, reflects the state of all committed transactions. The recovering node applies transactions in the delivery order. The recoverer sends committed transactions (they were totally ordered by the 1CS replication protocol) grouped by views. Moreover, this order is the same as they were originally applied at the recoverer in the given view. Moreover, metadata is total order consistent by the total order delivery of installed views. Hence, the serialization order at the recovering node cannot contradict the serialization order at the recoverer node. Since we are assuming the recoverer node is correct, the state resulting after the recovery procedure is completed at the recovering node is necessarily 1CS. □

5.4 Conclusion

In this Chapter, we have discussed the correctness for the recovery protocols proposed in chapters 3 and 4. We have established safety and liveness correctness criteria.

Database replication techniques based on group communication normally rely on total order broadcast primitive. Total order broadcast ensures that messages are delivered in the same order on all replicas in a reliable way. In both cases, certification and weak-voting based replication and recovery protocols, takes the advantages of this approach. So, while normal processing, writesets are fully applied in the same order at all replicas. After a replica fails, information about failed nodes and object identifiers are updated. Then, for a rejoining replica this information can be used for recovery purposes having for sure that this rejoining replica had applied the writesets of the transactions delivered until the last view they were present, providing its last view identification or last committed transaction identification.

Chapter 6

Conclusion

Databases are designed primarily for human service, at enterprises or home environments. These databases try to solve a greater number of needs that arise in modern societies. Among these needs, highlights the mobility of users and the wide geographical location that they may have. More over, increased reliance on computer systems.

Globalization is one of the greatest influences. The processes that characterize better these trends are:

- Increased mobility, associated with people, and information of several kind with more or less relevance, but always required by users.
- Concurrency, everyone has to be available anywhere and at any time. Information and products of large corporations have to be available in all countries for users.
- Web-based systems and other Internet-based applications such as banking, flying reserves, financial investment, etc., are of unprecedented interest and importance.

It is in this context that this thesis should be located. Specifically and by way of summary, the next section presents our work.

6.1 Summary

In this Thesis we have reviewed the most relevant publications about the subject of this work: Database replication and recovery. We have related, when it was possible by constraints imposed by the replication methodology, recovery protocols with the adequate replication protocols. This work lead us with references and some related issues about recovery problems and some possible solutions.

As a result and because none was found a survey was elaborated presenting a useful comparison table, various alternative options and strategies employed by replication and recovery protocols developed in recent years was summarized. This was very helpful to fully understand the currently proposed replication and recovery protocols and to make new proposals.

Two enhancements for the recovery protocols were made:

- In the first one, we have taken advantage of the historic list of write-sets that is used for certifying the transactions that arrive to the commit phase using it for the recovery purposes. More over,we optimize the recovery process with compacting techniques minimizing the total amount of information to transfer and to process for recovery protocol. We minimize the recovery information managed, reducing with this the workload in the recoverer and recovering replicas and reducing the communication network overload.

  A performance study was done, analysing the recovery time when the system load varies. Up to our knowledge this is the first performance study for such kind of recovery techniques in the field of database replication.

- In the second one, additionally to the compacting techniques we have enhanced it providing an accurate amnesia support. For this last, we have used a log-based technique which persists the messages as soon as they are delivered in each node.

  The tests was made with a simulation model on the Omnet++ tool. The advantages of the enhanced recovery protocol have been verified when comparing the results of both protocols. The obtained results have pointed out how our proposed compacting technique provides better results when the number of lost views by a crashed node increases. Thus, our compacting technique has improved the recovery protocol performance for recoveries of long-term failure periods.

In both cases, our compacting technique reduces the size of recovery messages, we must say that this improvement provided by our approach depends on the replicated system load activity, the update work rate, and the changed items rate. For the first two ones, we can consider in a general way that when higher they are better our compacting technique behaves. The worst scenario for our technique will be the one in which all the modifications are performed in different items.

As final conclusion, we can say that our enhanced recovery protocols work better in some of the worst scenarios from a recovery point of view: when the crashed node has lost a lot of updates and the changed items rate is not very high.

6.2 Future Research Direction

A major limitation in current recovery protocols is the lack of parallelism. Some works [39, 43] attempt the recovery using multiple sources for the recovery information without suspend the system service. Recovery can be a slow and blocking process, [39] intended to alleviate these problems by providing multiple recoverers based on partitioning the database by type of conflicts. It is convenient to perform an analysis of the convenience of implementing parallel recovery by dividing the data in a deterministic way and considering the data structure for recovery algorithms used in previous chapters.

Bibliography

[1] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. *LNCS*, 1300:496–503, 1997.

[2] J. E. Armendáriz, F. D. Muñoz, H. Decker, J. R. Juárez, and J. R. González de Mendívil. A protocol for reconciling recovery and high-availability in replicated databases. *21st International Symposium on Computer Information Sciences, Springer*, 4263:634–644, November 2006.

[3] José Enrique Armendáriz. *Design and Implementation of Database Replication Protocols in the MADIS Architecture*. PhD thesis, Univ. Pública de Navarra, Pamplona, Spain, February 2006.

[4] José Enrique Armendáriz-Iñigo, José Ramón González de Mendívil, José Ramón Garitagoitia, and Francesc D. Muñoz-Escoí. Correctness proof of a database replication protocol under the perspective of the I/O automaton model. *Acta Inf.*, 46(4):297–330, 2009.

[5] Özalp Babao, Keith Marzullo, and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In *Distributed Systems*, pages 55–96. Addison-wesley, 1993.

[6] Özalp Babaoglu, Alberto Bartoli, and Gianluca Dini. Enriched view synchrony: A programming paradigm for partitionable asynchronous distributed systems. *IEEE Trans. Computers*, 46(6):642–658, 1997.

[7] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10, 1995.

[8] Philip A. Bernstein. Middleware: A model for distributed system services. *Commun. ACM*, 39(2):86–98, 1996.

[9] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[10] Kenneth P. Birman. *Reliable Distributed Systems Technologies, Web Services, and Applications*. Springer, 2005.

[11] Michael J. Carey and Miron Livny. Conflict detection tradeoffs for replicated data. *ACM Trans. Db. Syst.*, 16(4):703–746, 1991.

[12] F. Castro, J. Esparza, M. I. Ruiz, L. Irún, H. Decker, and F. D. Muñoz. CLOB: Communication support for efficient replicated database recovery. In *PDP*, pages 314–321, 2005.

[13] F. Castro, L. Irún, F. García, and F. D. Muñoz. Fobr: A version-based recovery protocol for replicated databases. In *PDP*, pages 306–313, 2005.

[14] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[15] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. In *ACM Comp. Surv. 33(4)*, pages 1–43, 2001.

[16] Jerónimo Pla Civera, Maria Idoia Ruiz-Fuertes, Luis H. García-Muñoz, and Francesc D. Muñoz-Escoí. Optimizing certification-based database recovery. Technical report, ITI-ITE-07/04, Instituto Tecnológico de Informática, 2007.

[17] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[18] Flaviu Cristian and Farnam Jahanian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *SRDS*, pages 12–20, 1991.

[19] Om P. Damani and Vijay K. Garg. How to recover efficiently and asynchronously when optimism fails. In *ICDCS*, pages 108–115, 1996.

[20] Rubén de Juan Marín. *Crash Recovery with Partial Amnesia Failure Model Issues*. PhD thesis, Universidad Politécnica de Valencia, Valencia, Spain, September 2008.

[21] Rubén de Juan-Marín, Luis H. García-Muñoz, José Enrique Armendáriz-Iñigo, and Francesc D. Muñoz-Escoí. Reviewing amnesia support in database recovery protocols. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (1)*, volume 4803 of *Lecture Notes in Computer Science*, pages 717–734. Springer, 2007.

[22] Rubén de Juan-Marín, Luis Irún-Briz, and Francesc D. Muñoz-Escoí. Recovery strategies for linear replication. In *ISPA*, pages 710–723, 2006.

[23] Rubén de Juan-Marín, Luis Irún-Briz, and Francesc D. Muñoz-Escoí. Supporting amnesia in log-based recovery protocols. In *ACM Euro-American Conference on Telematics and Information Systems*, Faro, Portugal, May 2007. ACM Press.

[24] J.R. González de Mendívil, J.E. Armendáriz-Iñigo, F.D. Muñoz-Escoí, L. Irún-Briz, J.R. Garitagoitia, and J.R. Juárez. Non-blocking rowa protocols implement gsi using si replicas. Technical report, ITI-ITE-07/10 , Instituto Tecnológico de Informática, 2007.

[25] Sameh Elnikety, Fernando Pedone, and Willy Zwaenopoel. Database replication using generalized snapshot isolation. In *SRDS*, 2005.

[26] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[27] J. Esparza, A. Calero, J. Bataller, F. Muñoz, H. Decker, and J. Bernabéu. COPLA: A middleware for distributed databases. In *3rd Asian Workshop on Programming Languages and Systems (APLAS '02)*, pages 102–113, 2002.

[28] Luis H. García-Muñoz, J. Enrique Armendáriz-Iñigo, and Francisco D. Muñoz-Escoí. Associating replication and recovery protocols for replicated databases. In *Work in Progress Session of the Euromicro PDP*, Naples, Italy, Feb. 2007.

[29] Luis H. García-Muñoz, J. Enrique Armendáriz-Iñigo, and Francisco D. Muñoz-Escoí. Recovery protocols for replicated databases - a minimal survey. In *Work in Progress Session of the Euromicro PDP*, Naples, Italy, Feb. 2007.

[30] Luis H. García-Muñoz, Rubén de Juan-Marín, José Enrique Armendáriz-Iñigo, and Francesc D. Muñoz-Escoí. Improving recovery in weak-voting data replication. In Ming Xu, Yinwei Zhan, Jiannong Cao, and Yijun Liu, editors, *APPT*, volume 4847 of *Lecture Notes in Computer Science*, pages 131–140. Springer, 2007.

[31] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD Conference*, pages 173–182, 1996.

[32] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. ACM Press, 2nd edition, 1993.

[33] JoAnne Holliday. Replicated database recovery using multicast communication. In *NCA*, 2001.

[34] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. The performance of database replication with group multicast. In *FTCS*, pages 158–165, 1999.

[35] http://www.omnetpp.org/. Omnet++ a discrete event simulation environment.

[36] http://www.tpc.org/. The transaction processing performance council.

[37] L. Irún, H. Decker, R. de Juan, F. Castro, J. E. Armendáriz, and F. D. Muñoz. MADIS: a slim middleware for database replication. In *11th Intnl. Euro-Par Conf.*, pages 349–359, Monte de Caparica (Lisbon), Portugal, September 2005.

[38] Luis Irún, F. Castro, F. García, A. Calero, and Francisco Muñoz. Lazy recovery in a hybrid database replication protocol. In *XII Jornadas de Concurrencia y Sistemas Distribuidos*, 2004.

[39] Ricardo Jiménez, Marta Patiño, and Gustavo Alonso. An algorithm for non-intrusive, parallel recovery of replicated data and its correctness. In *SRDS*, pages 150–159, 2002.

[40] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *J. Algorithms*, 11(3):462–491, 1990.

[41] Bettina Kemme. *Database Replication for Clusters of Workstations*. PhD thesis, Swiss Federal Inst. of Technology, Zurich, Switzerland, 2000.

[42] Bettina Kemme and Gustavo Alonso. A suite of database replication protocols based on group communication primitives. pages 156–163, 1998.

[43] Bettina Kemme, Alberto Bartoli, and Özalp Babaoglu. Online reconfiguration in replicated databases based on group communication. In *DSN*, pages 117–130, 2001.

[44] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. In *FJCC*, pages 1150–1158, 1986.

[45] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD Conf.*, 2005.

[46] Francesc D. Muñoz-Escoí, Jerónimo Pla-Civera, María Idoia Ruiz-Fuertes, Luis Irún-Briz, Hendrik Decker, José Enrique Armendáriz-Iñigo, and José Ramón González de Mendívil. Managing transaction conflicts in middleware-based database replication architectures. In *SRDS*, pages 401–410. IEEE-CS Press, October 2006.

[47] Victor P. Nelson. Fault-tolerant computing: Fundamental concepts. *Computer*, 23(7):19–25, 1990.

[48] Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[49] PostgreSQL. Web site. Accessible in URL: http://www.postgresql.org, 2007.

[50] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1:220–232, 1975.

[51] A. Ricciardi, A. Schiper, and K. Birman. Understanding partitions and the 'no partition' assumption. In *4th FTDCS Workshop*, pages 354–360, September 1993.

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

[53] David L. Russell. State restoration in systems of communicating processes. *IEEE Trans. Software Eng.*, 6(2):183–194, 1980.

[54] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[55] Spread. The Spread communication toolkit. Accessible in URL: http://www.spread.org, 2007.

[56] Michael Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on software Engineering*, SE(5):188–194, 1979.

[57] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.

[58] Andrew S. Tanembaum and Maarten Van Steen. *Distributed Systems, principles and paradigms*. Prentice Hall, 2002.

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

[60] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *SRDS*, pages 206–215, 2000.

[61] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *ICDCS*, pages 464–474, 2000.

[62] M. Wiesmann and A. Schiper. Beyond 1-Safety and 2-Safety for replicated databases: Group-Safety. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT2004)*, 2004.

[63] Matthias Wiesmann and André Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Trans. Knowl. Data Eng.*, 17(4):551–566, 2005.