Crash Recovery with Partial Amnesia Failure Model Issues

Qüestions sobre el Model de Fallades de Caiguda–Recuperació amb Amnèsia Parcial

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departament de Sistemes Informàtics i Computació

Tesi Doctoral

Presentada per:
Rubén de Juan-Marín

Dirigida per:
Dr. Luis Irún-Briz
Dr. Francesc Daniel Muñoz-Escoí

Juliol 2008, Valéncia

Agraïments

A Luis Irún Briz i Francesc D. Muñoz i Escoí, els meus directors de tesi, els vull donar les gràcies pel seu suport, col·laboració, assessorament i paciència al llarg d'aquest treball científic.

Vull agrair també als revisors d'aquest document pels seus comentaris que han ajudat a millorar i donar-li el format definitiu.

Al Institut Tecnològic d'Informàtica (ITI) li vull agrair l'oportunitat de poder fer recerca en el camp dels sistemes distribuïts i les facilitats oferides per fer el doctorat.

Als meus companys del grup de Sistemes Distribuïts els vull agrair per les discussions i comentaris que han servit per millorar i aprofundir el meu coneixement en el món dels sistemes distribuïts. No puc oblidar en aquests agraïments als companys de l'ITI pel magnífic ambient de treball en el qual he pogut desenvolupar aquest treball.

Als doctorants i professors de la Facultat d'Informàtica de la Universitat de Lugano els he de donar les gràcies per l'enriquidor ambient d'investigació en el que m'han permés fer part d'aquest treball, i en especial als membres del grup de sistemes distribuïts dirigit per Fernando Pedone.

Finalment, vull agrair-li a la meva familia el constant suport que m'han donat sempre, i que entre moltes coses m'ha permés dur a terme aquesta tesi.

Contents

List of Figures

List of Tables

Resum

Els sistemes replicats són una branca dels sistemes distribuïts que tenen per objectiu obtenir sistemes informàtics altament disponibles, tolerants a fallades i amb alts rendiments. Una de les últimes tendències en les técniques de replicació, gestionats per protocols de replicació, és fer ús de Sistemes de Comunicació a Grups i més especialment de la primitiva de comunicació difusió atòmica per obtenir protocols de replicació eficients.

Un aspecte important en aquests sistemes consisteix en com gestionen la desconnexió de nodes –que degrada el seu servei– i la connexió/reconnexió de nodes per mantenir el seu suport original. Aquesta tasca es delegada en els sistemes replicats al protocol de recuperació. Com treballa aquest depén especialment del model de fallades adoptat. Un model comunment utilitzat per sistemes amb gran estat és el de fallada-recuperació amb amnèsia parcial ja que proporciona periodes de recuperació curts. Però, l'assumpció d'aquest model implica l'aparició de diversos problemes. Molts d'aquests nous problemes ja han sigut solventats en la literatura: gestió de vistes, abortament de les transaccions locals iniciades en el node fallat –quan es parla de sistemes transaccionals–, o per exemple la reinclusió de nodes en el sistema replicat. Tot i això, hi ha un problema associat a l'assumpció d'aquest model que no ha sigut considerat completament: el fenomen de l'amnèsia. Fenomen que pot donar lloc a inconsistències si no és gestionat correctament.

Aquest treball presenta aquest problema d'inconsistència d'estat degut a l'amnèsia i el formalitza, definint les propietats que es deuen complir per evitar aquest problema i es presenten possibles solucions. A més a més, també es presenta i formalitza un problema d'inconsistència –degut a l'amnèsia– que apareix sota una seqüencia específica d'esdeveniments tolerada per la condició de progrés majoritària que implicaria parar el sistema, proposant també les propietats per evitar el problema i plantejant solucions. Com a conseqüència es proposa una forma alternativa de condició de progrés majoritària. Després es defineix més acuradament una de les solucions –que consisteix en persistir els missatges de forma atòmica en el procés d'entrega– i s'estudia el seu comportament baix diferents configuracions dels sistemes replicats. Finalment, s'inclou un estudi del sobrecost introduït per la utilització de la solució general plantejada, demostrant-se que si la memòria és suficientment ràpida aquest és assumible.

Resumen

Los sistemas replicados son una rama de los sistemas distribuidos que tienen por objetivo obtener sistemas informáticos altamente disponibles, tolerantes a fallos y con altos rendimientos. Una de las últimas tendencias en las técnicas de replicación, gestionadas por protocolos de replicación, es hacer uso de los Sistemas de Comunicación a Grupos y más especialmente de la primitiva de comunicación de difusión atómica para obtener protocolos de recuperación eficientes.

Un aspecto importante en estos sistemas consiste en cómo gestionan la desconexión de nodos –que degrada su servicio– y la conexión/reconexión de nodos para mantener su soporte original. Esta tarea es delegada en los sistemas replicados al protocolo de recuperación. Cómo trabaja éste depende especialmente del modelo de fallos adoptado. Un modelo comúnmente utilizado para sistemas con gran estado es el de fallo-recuperación con amnesia parcial ya que proporciona periodos de recuperación cortos pero asumir este modelo implica la aparición de diversos problemas. Muchos de ellos ya han sido solventados en la literatura: gestión de vistas, aborto de las transacciones locales en el nodo caído –cuando se habla de sistemas transaccionales–, o por ejemplo la reinclusión de nodos en el sistema replicado. De todos modos, hay un problema asociado a la asunción de este modelo que no ha sido considerado completamente: el fenómeno de la amnesia. Fenómeno que puede dar lugar a inconsistencias si no es gestionado correctamente.

Este trabajo presenta este problema de inconsistencia debido a la amnesia y lo formaliza, definiendo las propiedades que se deben cumplir para evitarlo y plantea posibles soluciones. Además, también se presenta y formaliza un problema de inconsistencia –debido a la amnesia– que aparece bajo una secuencia específica de eventos tolerada por la condición de progreso mayoritaria que implicaría parar el sistema, proponiendo también las propiedades para solventarlo y planteando soluciones. Como consecuencia se propone una forma alternativa de condición de progreso mayoritaria. Después se define de forma más precisa una de las soluciones –que consiste en persistir los mensajes de forma atómica en el proceso de entrega– y se estudia su comportamiento bajo diferentes configuraciones de los sistemas replicados. Finalmente, se incluye un estudio del sobrecoste introducido por la utilización de la solución general planteada, demostrando que si la memoria es suficientemente rápida éste es asumible.

Abstract

Replicated systems are a kind of distributed systems whose main goal is to ensure that computer systems are highly available, fault tolerant and provide high performance. One of the last trends in replication techniques managed by replication protocols, make use of Group Communication System, and more specifically of the communication primitive atomic broadcast for developing more efficient replication protocols.

An important aspect in these systems consists in how they manage the disconnection of nodes –which degrades their service– and the connection/reconnection of nodes for maintaining their original support. This task is delegated in replicated systems to recovery protocols. How it works depends specially on the failure model adopted. A model commonly used for systems managing large state is the crash-recovery with partial amnesia because it implies short recovery periods. But, assuming it implies arising several problems. Most of them have been already solved in the literature: view management, abort of local transactions started in crashed nodes –when referring to transactional environments– or for example the reinclusion of new nodes to the replicated system. Anyway, there is one problem related to the assumption of this second failure model that has not been completely considered: the amnesia phenomenon. Phenomenon that can lead to inconsistencies if it is not correctly managed.

This work presents this inconsistency problem due to the amnesia and formalizes it, defining the properties that must be fulfilled for avoiding it and defining possible solutions. Besides, it also presents and formalizes an inconsistency problem –due to the amnesia– which appears under a specific sequence of events allowed by the majority partition progress condition that will imply to stop the system, proposing the properties for overcoming it and proposing different solutions. As a consequence it proposes a new majority partition progress condition. In the sequel there is defined in a more accurate and precise way one of the solutions – that consists in persisting messages atomically in the delivery process– and it is studied its behaviour under different replicated systems configurations. Finally, it is included a study of the overhead introduced when using this generic solution, demonstrating that if the memory is fast enough the overhead can be tolerated.

Chapter 1

Introduction

1.1 Replicated Systems

Replicated systems are a special subset of distributed systems focused on providing good performance levels, high availability and fault tolerance. These are successful key points for information systems. Good performance levels refer to the requirement that applications would provide short response times to client requests. High availability implies that the system responsiveness is up and working during a high percentage of time. Usually, when talking about high availability it is said that systems are built for providing service during the: 99.999%, 99.9999%, 99.99999% –also known as five, six and seven nines respectively– of the time. When talking about fault tolerance it is meant that the service is provided even in the presence of failures or crash node occurrences. Thus, in order to provide all these characteristics, replicated systems are composed of several replicas interconnected through a network where the state is partially or wholly replicated.

On the one hand, performance can be improved when clients access the closest replica to them [94, 99, 88], or by using load-balancing algorithms [104, 49, 2]. On the other hand, replicas may fail or may disconnect; therefore, fault tolerance and high availability are reached forwarding client requests to non-failed nodes in a transparent way.

Replicated systems can be arranged attending to different characteristics, but one of the most important classifications distinguishes these systems; those which perform their work outside the boundaries of transactions and those whose work is performed inside. First ones are usually denoted as *process replication* (but in order to avoid ambiguities here, they will be named "non transactional") systems while the second ones are known as *replicated transactional* systems (e.g. replicated databases). Therefore, the main difference between these systems is that the second ones are constrained by ACID [10] or ACID-like properties while

the first ones are not. An important aspect on both cases is to determine the replicated consistency level they need and to ensure that it is guaranteed. This desired state consistency can vary among different replicated systems depending on their respective necessities and their respective robustness or tolerance in front of replicated state inconsistencies. In the case of transactional replicated systems consistency is highly related to the isolation level established among transactions.

Regardless of the replication type, latest trends in replication techniques –managed by replication protocols [94, 99, 88, 104, 49, 2]– make use of a Group Communication System (GCS for short) [24] as it is detailed in [121]. These GCSs offer different services to the systems built on top of them. They provide several communication primitives, such as the atomic broadcast [69] allowing a more efficient implementation of replication protocols. These trends were started in [100], where authors demonstrated that the atomic broadcast primitive could be used instead of atomic commitment –either two phase commit [61] or three phase commit [66, 116]. Moreover, GCSs make use and provide membership mechanisms –also known as group membership service [28, 108, 113]. The membership service keeps track of the active and connected nodes reporting changes on the system configuration (i.e. failure or join of a replica).

The information provided by membership services is useful in determining if the replicated system's progress condition is fulfilled. A progress condition is the rule that has to fulfill a replicated system to be enabled to work in order to provide guarantees upon failures and recoveries. This progress condition can be based on a majority of alive replicas –when a primary component membership [24] is used.

Therefore an important aspect in replicated systems is how they manage node disconnection occurrences –which degrades their performance, fault tolerance and high availability support– and node connections or reconnections in order to maintain their original support. In fact, an intuitive basic algorithm to run when a replica connects or reconnects to the replicated system would consist in the following steps:

1. First of all, it is necessary to check if its state is outdated in relation to the replication system state.

2. If the replica's state is outdated in regard to the replicated system state, a recovery process for updating it must be started. In this process an updated replica –or replicas– plays the *recoverer* role transferring the information –*recovery information*– needed to update the outdated replica which adopts the *recovering* role.

This recovery process of outdated nodes can be carried out in many ways, ranging from the simplest one (a backup transfer) to more complex alternatives. But ideally, this process must be performed without interfering the common work

of the replicated system. To do so, replicated systems use *recovery protocols* which deal with these situations –in a coordinated way with the consistency management performed by the replication protocol–, and the way in which the recovery protocols handle recovery processes depends on the recovery information strategy used and the adopted failure model.

With regard to the type of recovery information transferred to the outdated node, traditionally, two different techniques have been used: *version-based* and *log-based* strategies. The first one consists in transferring the last state reached by the replicated system to the outdated node while the second consists in transferring the set of replication messages lost by the outdated node in order to be processed in the *recovering* node. Usually, the first technique is more interesting for long term failures –where the outdated node has lost a lot of state changes– while the second one is more interesting for short term failures –where the outdated node has lost few update propagation messages–.

Literature has proposed many failure models for replicated systems [29, 69, 106, 115]. Among all them, this thesis focuses on the *fail-stop* [114] and *crash-recovery with partial amnesia* [29] failure models because they are widely used in replicated systems –and more specifically in replicated databases.

The first one makes replicated systems discard crashed replicas, substituting them by "new" ones. Thus, when connecting a new node to the replicated system, the recovery protocol must first transfer the whole state to the new node –recovering node– before becoming fully operational. Note that a failed node can still be recovered, but in that case it adopts a new identity and it does not need to maintain its previous state. This failure model is interesting due to its simplicity. When a node crashes it is forgotten by the replicated system which substitutes it by a new replica. Then its semantic is very simple: a node can be active and updated or active being updated. Moreover, the algorithm for determining the information set that must be transferred is very easy: whole state. But, it is impractical for replicated systems managing large states because it leads to long transfer periods implying in the replicated system the following consequences:

- Longer periods with low performance levels for systems based on active replication. This effect is obvious for replicated systems which do not allow outdated replicas to work. But, it is also present in systems that allow outdated replicas to work, because the rate of work concluded correctly is degraded. Note that replicated systems based on passive replication do not suffer this effect.
- Longer periods with decreased fault tolerance support. Only fully updated replicas can be used to guarantee the correct and consistent state evolution in the replicated system.
- Higher times of unavailability if the replicated system does not fulfil the progress condition (i.e. systems based on primary partitions).

A particular case of systems managing big amount of state is data bases. In order to avoid such drawbacks, latest replication/recovery proposals for database systems [79, 70] have adopted the *crash-recovery with partial amnesia* failure model. In this case crashed nodes are not discarded. The replicated system waits for their reconnection in order to start the recovery process. In this case the recovery protocol transfers to each recovering node only the information lost during its disconnection crash interval. Once the recovering node has updated all its lost information, it becomes a fully operational node. So, in this approach the recovery process does not need to transfer the whole state but only the subset lost or outdated by the recovering node, shortening the recovery process and diminishing its associated problems.

But when assuming this second failure model, a new set of problem appears. Most of them are already solved in the literature. View management, aborting started local transactions in the disconnected node –in transactional replicated systems–, and node reinclusions to the replicated system are problems that have been already analysed by the scientific community.

However a particular set of problems have not been considered completely: those related to the amnesia phenomenon [38]. In this case, the problem resides on the difficulty that this failure model introduces to establish the accurate subset of information to transfer when recovering reincluded nodes. It appears because in some protocols the last assumed state in the recovering node is not really the last one, since such recovering replica may have lost some information that other replicas propagated to it before crashing. This phenomenon can lead to undesired situations.

It must be noticed that the "end-to-end argument" guideline proposed in [112] for placing funcionalities in layered –and modular– distributed systems can be used for solving the amnesia problem. In fact, the amnesia problem is a clear example of the problems that tried to address the "end-to-end argument" being described in [112] as the deliverey guarantees problem. This thesis focuses on this problem in the context of replicated systems using group communications systems.

1.2 Methodology

The methodology used for dealing with each problem detected and studied in this thesis has been the same one.

As a first step, a bibliographical research is performed in order to set the background about the existing problem in the scientific community in this area. If this process of information compilation ends finding references –and proposed solutions– in the literature to this problem, they are considered in order to know the goodness and accurateness of the solutions presented in the thesis. However, if the problem has not been already addressed properly, the search is extended

to similar and related scientific fields for detecting similar problems and the provided solutions.

Later, once the information gathering has been concluded, the problem is analysed and formalized, considering previous references, for establishing a theoretical basis that will be used later for demonstrating the validity and correctness of the provided solutions.

The process of generating a solution for a specific problem starts studying different possibilities and variations. Among all the considered solutions a selection process is started. First, their validity is checked against the problem formalization, discarding immediately those that do not succeed in this process. Second, the remaining possibilities –if there are several– are compared in order to determine which one presents better behavior attending to several characteristics. Finally, the selected solution is described, compiling all the information generated in the design process.

The research result is validated presenting and confronting it with the scientific community. To do so, papers –where the solution and research results are included– are written and sent for publication in specialised conferences of this area. In order to ensure the quality of this process, papers are only sent to conferences which publish at Springer LNCS, Association of Computer Machinery (ACM) and IEEE. The first submission of a paper is also published as a technical report. Once the conference review comments are received, the paper is improved including the more important suggestions, either if the paper has been accepted or not. In the first case, the reviewed paper is submitted in its last version to the conference. In the second case, the paper –including the improvements– is sent to another conference.

In conferences, the comments and suggestions received in the questions time after the paper presentation are considered and included in the thesis research for improving the overall quality.

1.3 Motivation and Scope

This thesis is focused on how the assumption of the *crash recovery with partial amnesia* failure model introduces the necessity of considering special solutions in order to avoid undesired situations –mainly resulting in replicated state inconsistencies– in these replicated systems due to the *amnesia phenomenon*.

To start with, it is presented the *amnesia phenomenon*. To do so, first it is detailed an example for describing it and its possible effects. At the same time there are outlined the two levels at which this phenomena appears. Later, there are included the formalizations for this phenomena in transactional replicated systems and non-transactional replicated systems, presenting in both cases which properties must be fulfilled in order to manage it accurately.

A special case that is also considered in this thesis are possible arising replicated state inconsistencies when combining the *amnesia phenomenon* with the majority progress condition. To do so it is first outlined with a simple example (look at Section 6.2) the replicated consistency problem which arises when combining the *amnesia phenomenon* –non-correctly handled– with a specific replicated system composition allowed by the *majority partition* progress condition. This problem as it would be seen can lead different state evolutions among the members of the replicated system.

Later, this problem is formalized, establishing the replicated system conditions that would generate it, and the properties that must be fulfilled for overcoming it.

Notice that despite being a rare problem, it should be accurately managed for avoiding critical situations. Therefore, two different approaches are presented for overcoming it, being each one of them interesting for transactional replicated systems with different characteristics. On the one hand, it is presented a solution for critical systems where already performed and committed work at replicated system level can not be undone nor lost. On the other hand, it is proposed the use of a technique used in partitionable systems, reconciliation, whose main advantage is its zero overhead in normal work.

Related to the previous case it is presented in this thesis a new approach for the majority partition progress condition in replicated systems. It is included due to the importance of these rules in replicated systems. Their importance resides in the fact of being key aspect in order to guarantee the replicated state correctness in *replicated systems*. To this end, two basic approaches can be adopted: *primary partition systems* where only the *majority partition* is allowed to work and *partitionable systems* where multiple partitions may work concurrently. The latter implies the use of costly quorums [12] for merging partition states, which may not always be feasible without breaking correctness. Therefore, *primary partition* is widely accepted in order to guarantee correctness in the replicated state, as well as the capability, for any reconnected (thus potentially outdated) node, of being always in disposition to be updated to continue working.

This principle is based on the idea of a *progress condition*. Thus, considering that the distributed system knows the pre-configured set of possible nodes $D = \{r_1, ..., r_n\}$, the condition of *primary partition* is granted when the interconnected alive nodes conform a subset of D that satisfies the *progress condition*. Thus, the progress condition enables a partition $P \subseteq S$ (i.e. a subset of the distributed system) to go on working with guarantees.

Traditionally, replicated systems have adopted the concept of *majority partition*, meaning that $\frac{n}{2} + 1$ nodes of the system must be alive and fully up-to-date in order to enable the partition to work. More precisely, this statement implies that the notion of "up-to-date member" is equivalent to "member of the *primary view*". In the scope of recovery systems for database replicated systems based on broadcast primitives, the *Enriched View Synchrony* was presented in [9], decoupling the concept of "up-to-date member" from the "member of the

primary view" one. This distinction drives the authors to separate the working view concept from the majority partition one. The work also demonstrates how this new approach models more accurately the evolution state in node replicas, and how it fits better for applications where the recoveries are long and complex.

In addition, during the process run for recovering a newly reconnected node on a primary partition, several problems arise in order to guarantee the correctness of the system. Because of these problems, many recovery protocols introduce several restrictions on the set of nodes allowed to work during that process. The most advanced protocols disallow for working just the recovering node, whilst other approaches prohibit the activity in the entire partition during part of -or even all- the process run to recover nodes.

A direct implication of these two characteristics is that the aforementioned condition introduces a drawback in the availability of the system, since the nature of the update process of reconnected nodes makes it expensive, at the time that the $\frac{n}{2}+1$ updated condition requires an immediate update of the nodes in order to proceed in partitions with just $\frac{n}{2}+1$ composing nodes. Consequently, it results extremely convenient to find a mechanism for relaxing the progress condition, thus allowing to proceed partitions with just $\frac{n}{2}+1$ *alive nodes*, which will lead to a higher availability, also enabling background recovery processes.

This work formalizes a new and more relaxed progress condition for replicated systems. Its viability and correctness in recovery terms are also proven to be guaranteed by a proposed general recovery information strategy, ensuring that primary partitions will always be able to go on working. This proposal is presented in the scope of a middleware recovery protocol intended to provide fault tolerance for replicated systems based on linear interaction approach [124], although the formalized principles are also applicable to replication protocols based on constant interaction. The only assumption made by our model is that the information gathered during the failure of nodes can be structured as an activity log (i.e. log-based gathering). This recovery protocol supports different recovery strategies with the idea to minimize the effort and cost of the recovery process, without stopping the replicated system work for *primary partitions*. It is also intended to perform partial recoveries, when needed. Finally, as our design is performed as a middleware recovery system, it can be easily applied to different transactional scenarios, in addition to database replicated systems. In fact, this recovery system is highly oriented to replicated processes which manage large amounts of state, not being such interesting for replicating small objects, where usually transferring the whole state to outdated nodes will present better performance.

After presenting and formalizing the amnesia phenomenon and its associated problems, there are detailed how recovery protocols can face its undesired effects concerning this phenomenon with several replication aspects.

Taking as starting point the solution presented in [19] it is performed an analysis of how different replication configurations in replicated transactional systems can affect the amnesia phenomenon and its accurate support. This analysis is

enriched with the study of the overhead introduced when supporting correctly amnesia in different transactional replication configurations.

Later, there are briefly described some database recovery protocols –as example of recovery techniques for replicated transactional systems– proposed in the literature [79, 70, 82, 20, 3, 74, 19, 20] and surveyed in [57]. This survey is enriched here by emphasizing how they face the amnesia phenomenon, attending to the fulfillment of the properties described in the formalization which correct observance guarantees an accurate amnesia phenomenon management. This study also gives special importance to the database replication protocols they are designed for, because as it was depicted in [57], the recovery protocols are very dependent from the characteristics of the replication protocols used, and the information that these replication protocols store. In this study it is observed how a correct amnesia support depends on the combination of an adequate recovery information generation policy and an accurate way for notifying the last really committed changes in the node that must be recovered.

The aftermath of this study is a recovery protocol categorization grouped first by the used technique –version-based or log-based–, and secondly by the granularity used for managing the recovery information. In this categorization, it is analyzed the correct amnesia phenomenon management, and in the cases it is not guaranteed they are proposed changes in these techniques for overcoming it.

On the sequel it is considered the design of a general strategy for recovery protocols implemented for transactional replication protocols based on linear interaction, in contrast of using the constant interaction [124] approach. This is provided because linear interaction –consisting in broadcasting one message per operation-, in spite of its high performance cost, will be the only feasible alternative for object-oriented replicated systems with large data states to transfer, and with a transactional support, such as FT-CORBA with its complementary *Transaction Service*, where constant interaction will either lead to huge messages or be impractical in case of partial replication, since the state to be transferred should be collected from different source nodes. The problem, as it will be shown, is the management required by linear-recovery protocols which is more complex because it must manage multiple messages per transaction. In addition, for ensuring correctness under linear interaction, messages belonging to not-yet-committed (as well as for rolled-back) transactions, must be adequately treated.

The idea is to obtain a recovery protocol for linear interaction replication protocols which minimizes the effort and cost of the recovery process, without stopping the replicated system work for *primary partitions*. It is also intended to perform partial recoveries, when needed. Finally, as our design is performed as a middleware recovery system, it can be easily applied to different transactional scenarios, specially including database replicated systems.

When presenting the proposed solution it would be outlined the on-going transactions problem which arises due to the use of linear interaction in database

replication protocols –implying the broadcast of messages belonging to not yet-committed transactions–. Besides, this replication system will interleave messages belonging to different transactions, messages that will be applied to the database in their delivery total order. Finally, each transaction is committed when its commit is applied. In this context, if a node crashes, all associated changes to not yet committed transactions are lost whilst associated updates to committed transactions remain permanent. Afterwards, when the crashed node becomes active again, the recovery process updates it by reapplying (among others) the messages associated to not yet committed transactions at the crash time, while the committed transaction messages at the crash time are not reapplied. In this scenario, inconsistencies can arise if these reapplied messages were interleaved with committed transaction messages in the original work sequence because this original order is misunderstood in the recovered node. These inconsistencies appear if the mentioned transactions conflict, and the selected isolation level tolerates such conflicts. As it will be noticed this problem only occurs when an outdated node reconnects to a *working* replicated system, and it has not lost the *working condition* from the instant when the outdated node crashed.

Later, it is presented a generic recovery protocol for non-transactional replicated systems which manages accurately the amnesia phenomenon. This solution is based on the use of checkpointing and logging ideas widely used in distributed systems [51]. Therefore, when a crashed node reconnects first restores the checkpoint and applies the received messages before the crash –amnesia recovery–, and second receives the broadcast messages during its disconnection. This solution emphasizes which information must be maintained when applying the messages during the *amnesia recovery* process in order to: repeat the forgotten work –work already done before the crash but lost in it–, and do the work not performed before the crash. Moreover, the *amnesia recovery* solution has to avoid the repetition of the work already performed and which follows the exactly-once semantics [72] (i.e. state changes with permanent effects). Then it is included an amnesia overhead analysis for different non-transactional replication configurations in order to know the time cost associated to managing accurately the amnesia problem.

As it has been already said in [112], authors considered this problem when proposed their "end-to-end argument" guideline for modular distributed systems. The guideline said that functions placed in the low levels of a layered design are redundant or low value when compared with the effort of putting them there. So, the idea would be to put functions usually as top as possible. And, this the approach followed when proposing the generic solution for managing it correctly either in transactional and non-transactional systems. As a result, it is provided some minimal mechanisms able to assist the applications for solving the amnesia problem.

Finally the generic solution proposed for transactional systems is simulated using different storage systems. The obtained results are included, compared

and analyzed.

1.4 Outline

This thesis is structured in five different parts. The first Part, *Background*, will describe the thesis context. Chapter 1 is a brief Introduction to the thesis contents and motivations. In Chapter 2 it is presented the research projects where this thesis has been developed. Some concepts and definitions used in the thesis body are defined in Chapter 3, while Chapter 4 details the considered system model. The second Part, *Contributions*, contains the contributions of this thesis. The *Amnesia Phenomenon* is detailed and formalized in Chapter 5. Chapter 6 details and formalizes a source of possible inconsistencies when combining the *amnesia phenomenon* with the *majority partition* progress condition, while Chapter 7 presents a refinement in the *majority partition* progress condition, considering the problems formalized in previous chapters. Later, Chapter 8 studies how can be provided amnesia support –and the cost associated to– in transactional replicated systems. Next chapter, Chapter 9 surveys different recovery techniques for replicated transactional systems proposed in the literature analysing if they manage accurately the amnesia phenomenon, and also proposing a classification. A recovery proposal for transactional replicated systems based on linear interaction is presented in Chapter 10. Chapter 11 details a log-based recovery protocol for non-transactional replicated systems which manages accurately the amnesia phenomenon and considers several semantics when recovering. Finally, Chapter 12 contains an analysis of the overhead introduced by the proposed solution based on a simulation. The third part extends the related work with Chapter 13 while the fourth part contains the conclusions and future work in Chapter 14.

For concluding the fifth part *Annex*, includes additional and complementary information.

Part I

Background

Chapter 2

Research Context

This chapter describes the research group and research projects where the thesis work has been developed.

2.1 SiDi Group

The SiDi Group has been working for a long in several projects with the aim of designing and implementing middleware replication solutions whose goal is to provide highly available and fault-tolerant information systems. Among these projects the most important ones are: GlobData (FP5-IST-1999-20997), DeDiSys (FP6-2003-IST-2-004152), MADIS (MCYT TIC03-09420-C02) and CONDEP (TIN2006-14738-C02-01). Therefore, all this research has been oriented to designing and developing middleware frameworks for supporting replication and recovery protocols either for traditional –static– and dynamic environments. At the same time designing, implementing and evaluating different replication and recovery protocols in replication solutions.

This thesis has been developed in the context of the research projects MADIS and CONDEP. These projects are described on the sequel for giving the necessary context.

2.2 MADIS

This project follows the SiDi group research in the replicated systems field for providing highly available and fault tolerant applications. This research is focused on both process and data replication, proposing protocols and architectures for facilitating the development of highly available applications. More

specifically, this project has the aim of designing and implementing architectures capable to provide support for multiple replication models –active, passive, coordinator-cohort–, selecting each time in a dynamic way the model which provides better behavior and performance to the application.

Another important aspect considered in this research project are the recovery protocols. These protocols are important for building real fault tolerant applications, because their mission is to recover disconnected nodes, updating them before becoming fully operative ensuring then the original fault tolerance support provided by the replicated system.

Thus, the result of this project was a two layer architecture which supported the execution of different replication protocols –providing different consistency levels and different replication configurations–, providing a common interface for all them. The results of this project have been presented in different publications as [5, 3, 4, 42, 43, 75]

In this project the author has also co-authored in the following publications [4, 31, 32, 36, 40, 43, 75].

2.3 CONDEP

The global goal of this project is to make progress in supporting the construction of reliable systems, both in traditional (static) and dynamic environments. Different kinds of protocols are needed for support, depending on different layers of the middleware architecture being developed at the SiDi. In general, the more extended architectures that offer reliability support recur on a lower layer in non-reliable communications. It contains the membership services, used as the basis for a large number of distributed algorithms, in particular those in need of some kind of agreement among the components of the distributed system. Typical examples are group communication services. They should be located in a second layer of the architecture. On top of this, both the replication manager and the replication and recovering protocols, each located in the third layer, need the mentioned membership data. This project researches on: (1) the improvements described above for static environments; (2) the design, specification and implementation of a dependable architecture for dynamic environments; (3) new supports for group membership, distributed agreement, secure communications, authentication, and consistency and replication management in those dynamic environments. Finally, all these issues will be conveniently adapted for intermediate environments (e.g., decentralised distributed systems), producing generalised solutions that will be compared with their equivalent approaches in static environments.

The author has co-authored in the following papers derived from this project [11, 33, 34, 35, 37, 38, 39, 41, 58, 93, 111].

Chapter 3

Concepts and Definitions

This chapter is intended for detailing the basic concepts and definitions about replicated systems that would need the reader of this thesis. Obviously, this chapter is not intended for these readers who are experts in this research field.

This chapter is structured as follows. First of all there is the Section 3.1 which gives the definition that will be used in this thesis for different concepts. Sections 3.2, 3.3, 3.4 and 3.5 compound a block related to failure. Section 3.2 details the differences between fault, failures and errors. In Section 3.3 are referenced the most common failure models for replicated systems, while in Section 3.4 it is detailed when this thesis considers that a node has crashed. Later, some comments about failure detectors are included in Section 3.5. Finally, Section 3.6 outlines the two existing recovery strategies from an information point of view.

3.1 Glossary

Before starting to talk about the basic concepts and definitions that will be used in this thesis it is necessary to introduce a summary of the most used words in this research field, and their meaning in the thesis context.

Replicated System. A replicated system is a set of nodes interconnected through a network, where the information is wholly or partially replicated. Replicated systems can be either transactional or non-transactional.

Node. Each one of the members that compound the replicated system. Also known as "replica".

Node States. Nodes can have different states from a replication and recovery point of view. Traditionally, from a replication point of view, the nodes can

be *alive* –or working–, and *crashed* –also disconnected or in other words non working as replicated member–. From a recovery point of view, nodes can be *updated* –have the last state reached in the replicated system– or *outdated*.

Group Communication System. A Group Communication System [24] is a middleware that provides communication funcionality among interconnected nodes throw several group communication primitives.

Group Communication Primitive. Each one of the different communication mechanisms [24] provided by a group communication system. They differ in the communication guarantees they provide in reliability and delivery order when sending messages in regard to other sent messages.

Replication Protocol. Algorithm used in a replicated system in order to propagate updates among its members guaranteeing the desired level of data consistency.

Transactional Replicated System. In these replicated systems the work is performed into the boundaries of transactions. Sections 5.3, 6.3.1 and Chapters 8, 9, 10.

Transaction. The main characteristic of a transaction is that all its operations are performed atomically. If all its operations succeed, the transaction commits. Contrarily, if one operation fails, the transaction rollbacks, aborting its transaction operations that had succeeded. In fact, transactions ensure that their committed work guarantees the ACID properties.

Distributed Transaction. In a replicated system, instead of using simple transactions there are used distributed transactions. In this case, each transaction must be checked in all replicas, and only can be committed if it can be committed in all nodes. If the transaction fails in one node, it must be aborted in all nodes. This is known as *distributed commit*.

Constant Interaction. A transaction is done in a constant interaction way when the number of messages used for broadcasting it and decide its abort or commit is constant. In this technique the replication protocol can either broadcast operations or readsets and writesets, but in most cases it simply spreads readsets and writesets.

Linear Interaction. A transaction is done in a linear interaction way when for each included operation the system broadcasts a message. In this case, instead of being also possible to transfer either writesets/readsets or operations, usually are broadcast the operations. Chapter 10.

On-going Transaction. It is said about broadcast transactions whose commit or abort has been neither decided nor broadcast. They can be also named as *non-yet-committed*. Section 10.3.

Lost transactions. Set of broadcast transactions that have not been received by a node –because it is crashed or disconnected–. Also known as "missed transactions".

Non-Transactional Replicated System. In this replicated systems the work is not performed between the boundaries of transactions, therefore ACID properties are not ensured. These systems are also known as *process replication*. Sections 5.4, 6.5 and Chapter 11.

Recovery protocol. Algorithm used in a replicated system for updating outdated nodes. Sections 5.5, 8.2, 9.3, 10.2 and 11.3.

Recovery information. Information set needed by a recovery protocol for updating outdated nodes. Sections 3.6 and 5.6.

Log-Based recovery. This is the name of the recovery technique that uses as basic recovery information the log of broadcast messages in the replicated system. Sections 3.6.1, 5.6.1.

Version-Based recovery. This is the name of the recovery technique that uses as basic recovery information the state of the data maintained in the nodes of a replicated system. Sections 3.6.2 and 5.6.2.

Progress Condition . This is the condition that must fulfil the replicated system in order to go on working. The most common progress condition in replicated systems is the *majority partition* . Chapters 6 and 7.

Database. A database is a structured collection of records or data. These data is structured following a certain model. Nowadays, the most common model for databases is the relational model [25].

3.2 Faults, Failures and Errors

When talking about reliability, dependability or fault tolerance in information systems it is very important to know the differences between different words that if there are not known can lead to misinterpretations. Thus, it is important to know the exact use in the thesis context: faults, failures and errors. In [97], these terms were defined like their relationships.

Faults are defined as anomalous execution conditions. Their causes cover a great range: from mistakes in systems specifications or implementations to external disturbances. And faults are manifested in systems as errors, where the logical state differs from its intended value. Thus, an error is a discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition. Finally, failures are caused by errors, and it is said that a failure has occurred in a system when it is unable to perform its required functions within specified performance requirements.

After determining their meaning it is time to talk about failure models.

3.3 Failure Models

Replicated systems are used to provide fault tolerance and high availability. The background idea of a replicated system relies on replicating the service in different nodes that can fail independently. Therefore, if one of these nodes fails the service is still provided by non-failed nodes, ensuring then the fault tolerance and high availability.

In this context, it is said that a replicated system is a *k-fault* tolerant system if it supports k simultaneous faults without presenting an incorrect behavior.

But, this support is not straightforward to provide. On one hand, there is the inherent complexity to distributed systems. And on the other hand, the use of replicated systems introduces new types of failures that must be considered (i.e. those associated to the use of networks) when designing and implementing the system. In order to simplify the design and implementation of these replicated systems, literature has proposed and encouraged some assumptions about the fault types and the fault rates that have the components of a replicated system. So, these assumptions will influence the design and redundancy degree of the replicated system, determining then its fault tolerance level.

These failure assumptions have been structured in different failure models as detailed in [29, 69, 106, 115]. The failure modes proposed go from the *fail-stop*, the less disruptive one, to the *byzantine*, the one supporting most disruptive failures. In the first one, it is assumed that nodes only fail by halting, remaining in this state once they have halted. In the other extreme any kind of failure is assumed. Between these two extremes a wide variety of failure models exists.

In [115] most commonly failure models used in distributed systems were summarized and classified. Table 3.1 presents all these failure families.

In [29] the author presented a classification of server failures. An *omission* failure happens when a server does not respond to an input. If the server responds outside the assumed response time interval a *timing* failure has occurred, being *early* and *late* the possible timing failures. Another server failure type is the *response* one. In this case, either the returned value is incorrect –*value* failure– or the adopted action is not the desired one –*state transition* failure–. The last failure type is the *crash* failure that occurs if the server stops responding incoming inputs. But, the author assumes in this case that crashed replicas can either remain halted or be restarted. In the first case, the author assumes that crashed servers behave like in all failure models proposed in [115] (see table 3.1), denoting it as *halt-crash* failure. In the second case, [29] distinguishes several restart behaviors depending on the server state at restart time. When the server restarts in a predefined state, without depending on the inputs received before crashing the author denotes it as *amnesia-crash* failure. If some part of the restart state is established to a predefined initial state whilst the other part is equal to the state before crashing, the server has suffered a *partial-amnesia crash* failure. The last option, the *pause crash* failure implies that the server

Name	*Description*
Fail stop	Processors fail by halting, remaining in this state. Other processors can detect the failed processor [114].
Crash	Processors fail by halting, remaining in this state. But, in this case other processors can not detect the failure [53].
Crash+Link	Processors fail by halting, remaining in this state. Moreover, links can lose some messages, but do not delay, duplicate or corrupt messages [16].
Receive-Omission	Processors can halt –remaining in this state– or can receive only a subset of the messages sent to them [102].
Send-Omission	Processors can halt –remaining in this state– or transmitting only a subset of the messages they have to send [68].
General-Omission	This is a combination of the two previous failure models [102].
Byzantine Failures	In this failure model processors fail in an arbitrary way [114].

Table 3.1: Schneider Failure Models classification [115].

restarts with the same state it had before crashing.

As it can be seen, instead of modeling the failures that can occur in distributed systems –including therefore replicated systems ones– [115, 29] present both overlapping and complementary failure models. Thus an accurate study of the existing failure models, and an analysis about how they fit the system requirements must be done before selecting one or a combination.

So, when designing a replicated system a basic question that must be answered is which failure model to assume. The most conservative approach would be to adopt the *byzantine* failure model, in fact the worst case. But, as it has been demonstrated in [106] it needs the highest degree of replication for providing a *k-fault* tolerant system –$3k+1$ instead of $k+1$ for the *fail stop*–, it decreases the overall system dependability because there are more possible sources of failure.

It must be noticed that due to the complexity associated to replicated systems its design and development has been usually divided into the design and development of several small pieces –each one providing a basic functionality needed by a replicated system– that when combined compound a fully operational replicated system. Traditionally, at least two main pieces can be distinguished: *GCS* and the replication protocol itself.

The *GCS* is the component which provides communication primitives to the replicated system. Primitives that are used by the replication algorithm for performing its work. Normally, the designers and developers of these GCSs consider and solve some of the failures that can arise in the communication

work –at network level–, therefore the replication protocol can rely on the GCS for managing these failures, being only necessary to deal with those that the GCS can accurately manage. Thus, the *GCS* provides what is called *hierarchical masking* in [29].

Traditionally, in the literature the *byzantine* failure model has not been used due to its complexity and to the fact that it needs a high degree of redundancy which at the same time jeopardizes the system dependability, so few works as [118] have intended to support them. In fact, the most commonly used failure model for replicated systems has been the *fail stop*. The reason is its simplicity: nodes only can fail by halting, not being possible to resume them. Simplicity was very important as researchers were usually more concerned about providing good performance levels in their replication techniques, as far as performance is their "Achilles' heel" in regard to centralized solutions. So in the literature there are a lot of proposals adopting this failure model as [12, 16, 87].

The main problem of the *fail stop* failure model arises when a replicated system using it tries to maintain its original *k-fault* tolerance level once one or some nodes have crashed. Notice that maintaining the original *k-fault* tolerance level for replicated systems must be one of its main goals, if it is not considered their fault tolerance will degrade until they do not provide any level.

In this case, when the *fail stop* failure model has been assumed crashed replicas remain in this state. Then it is necessary to join a new replica for substituting the crashed node, being therefore necessary to transfer to this new replica the whole state of the system. This work way is useful if the state in the replicated system is not very large, but as soon as the state amount increases this updating process is more expensive: in computing and time terms. Thus, this option will not work well for replicated systems which manage large states, because will lead to very long transfer periods. This situation implies: longer periods with low performance levels for systems based on active replication, longer periods with decreased fault tolerance support and higher intervals of unavailability if the replicated system does not fulfill the progress condition (i.e. systems based on primary partitions) as commented in [32, 34].

Then for avoiding these problems when trying to maintain the original *k-fault* tolerance it will be assumed that nodes fail by crashing but they can work again once their failure period has ended. This assumption is usually named the *crash recovery with partial amnesia* failure model [29]. This new assumption introduces the possibility of recovering previously crashed nodes, trying to introduce as less complexity as possible like it will happen if adopting more complex failure models as the *byzantine* one. This is the reason why in the literature can be found a wide range of replication proposals based on this failure model [6, 3, 19, 20, 31, 36, 37, 38, 58, 70, 74, 79, 82].

In these proposals, the adoption of this failure model decreases the problems associated to the assumption of the *fail stop* failure model because only lost state (and not whole) must be transferred. The provided advantages are not free, since they are provided at the cost of introducing more complexity in the

replication and recovery algorithms.

Once the most widely used failure models for replicated systems and the reasons for using them have been detailed, it is time to determine how components, or in this case, nodes crash. This is important because the way they crash will affect the way they can be recovered.

3.4 Crashed Nodes

Attending to the definition given by [29] it is said that a node crash occurs when from certain instant on the node does not respond to any new input. Therefore, the author of [29] explains the node crash consequences but does not tell anything about the cause of such crash.

But a node crash can happen for many different reasons. Nodes belonging to a group which does not fulfill the progress condition –due to a network partition– are crashed in replicated system terms when this model is adopted. Also, nodes that switch off, or nodes that lose their energy source supply also can be considered as crashed because they can not go on working. And in catastrophic situations as fires or explosions that affect the integrity of the nodes, it is also said that node crashes have occurred. Finally, but not more infrequent, software misfunctioning can be assumed as a node crash. All these situations imply a node crash under the [29] terminology, but depending on the way the node has crashed it will have a different state at its restart and therefore different actions must be taken for updating it –when possible.

The way a node crashes is important when adopting the *crash recovery with partial amnesia* failure model because it will affect the state it will have at restart time.

For simplicity reasons in this thesis, it will be considered that a node crash occurs because the processor halts in a non expected way and then the node is powered off. This implies that it is impossible to take special stop actions because the node stops in an abrupt way. This way of crash implies that all the volatile state is lost in the node, and that all started and on-going operations are stopped abruptly without finishing them in a controlled way. But, all the state that has been stored persistently before the crash is maintained at restart time.

There are other crash variants: as network partition and programmed switch off (i.e. using a UPS –uninterrupted power supply–). In the first case, volatile state is not lost and on-going operations can be finished in a controlled way. In the second, the volatile state is lost, but on-going operations can be also ended in a controlled way. Therefore, the solutions that will be presented in this thesis for the assumed crashing way –the one described in the previous paragraph– will be also valid for these other crashing ways. This is due to the fact that the assumed crash way is a more restricter variant covering therefore these other

crashing modes.

However, the assumed way of nodes crashing does not cover other crash causes. This may happen when the crash occurrence is related to a problem in the physical integrity of the node. If there is a failed component –not the device of persistence storage– which halts the node activity may be simply switching off the node and substituting the failed component by a new one is enough. In this case, the assumed crash way –previously described– is valid for covering this crashing mode.

But, if the failed component is the device of persistence storage or the overall physical integrity of the node has been affected (i.e. a fire) the assumed crash occurrence is not useful. In both cases the disk must be substituted –in the second case including the whole computer– and the restarted node will be considered a new replica. Therefore, its recovery is better adapted to the *crash stop* failure model. Anyway, it must be noticed, that the *crash recovery with partial amnesia* failure model assumption also covers the *crash stop* failure model one. If only the disk has failed, other strategies can be adopted to overcome its failure. One possibility could be to use a RAID (Redundant Array of Independent Disks), thus if one of the disk fails one of the others can substitute it.

3.5 Failure Detectors

Failure detectors are a basic element for developing fault tolerant systems using replicated systems. Their mission is to detect membership changes in the replicated systems, detecting each time which nodes are correct and which are faulty. Thus, failure detectors are a key component when designing recovery protocols.

Failure detectors in replicated systems are based on consensus, all alive replica members must agree about correct or failed nodes. Therefore, they must solve the problem of reaching consensus either in synchronous, partially synchronous or asynchronous replicated systems [47].

Achieving consensus in synchronous systems is very easy because both message transmission and process execution are bounded in time terms. The problem is that these systems are not feasible.

On the other hand, asynchronous systems are very easy to implement, but these systems can not establish an upper time bound neither on message transmission delays nor on process execution steps. This assumption difficults the process of determining which nodes are crashed or alive in the replicated system, because it can not be distinguished in a reliable way among crashed nodes, slow nodes or slow connections and even more difficult to reach a consensus [53]. In fact, [54, 45] demonstrated the impossibility for solving in a deterministic way this problem if the system is subject even to a single failure. In order to avoid these problems, researchers have performed a big effort for overcoming them,

presenting different ways.

One way consisted in defining weaker problems and solve them [46, 8]. Another way for avoiding these problems has consisted in using *unreliable failure detectors* [23, 22, 65].

Another alternative consists in proposing different partial-synchrony models [45, 47, 23] and solve the problem on them. In [86] authors analyzed the implementation of *unreliable failure detectors* in partially asynchronous systems, demonstrating which were feasible and which were not.

Some replication techniques use failure detectors in order to release locks owned by failed nodes in order to avoid accessing problems to common resources. The idea would be to free a shared resource as soon as the system detects that the node which owns the lock on this resource has crashed. For this problem, in [17] the author proposes the use of temporary locks which expire after an established timeout if the owner of the lock does not renew them. This technique avoids the necessity of failure detectors, because if a node crashes it will not renew the lock and the shared resource it owned would be free once the timeout passes. This technique is known as advisory locks.

As conclusion, it can be said that the scientific community has made a big effort for developing failure detectors in replicated systems.

In regard to failure detectors, this thesis as it is said in Chapter 4 considers partially synchronous systems and it assumes the use of a group communications systems which makes use of a membership monitor being constructed on top of a failure detector. Thus, how failure detectors work and are implemented is beyond of the scope of this thesis.

3.6 Recovery Information and Strategies

Another important aspect when designing recovery protocols is which recovery strategy is adopted. The selected recovery strategy is related to the type of information used in the recovery process. Usually, two main ways of recovery have been considered in the distributed systems literature: *log-based* or *version-based*. In the first one, the broadcast messages during the node disconnection are used as recovery information, while in the second a copy of the state –or only of the modified state during the node disconnection– is transferred to the recovering node. Obviously, each one can be implemented in different ways but all proposed recovery techniques in the literature belong to one of these family techniques or combination of both.

3.6.1 Log-based

The **log-based** policies use the broadcast messages as recovery information. Traditionally, the recovery systems based on this strategy started to generate

the recovery information –store persistently the replication broadcast messages– when the membership monitor detected disconnected nodes in the replicated system, as it is done in [70]. This information must be maintained until the outdated nodes have applied the missed messages. A message only can be deleted when all replicas that have lost it have received and applied it in their recovery.

This recovery families are mainly used for transactional replicated systems that have assumed the *crash recovery with partial amnesia* failure model. This is due to the fact that transactional systems persist their state and therefore, transferring and applying in recovering nodes only the messages they lost during their disconnection is enough. Moreover, in some works its use is recommended for short term failures, encouraging the use of a version-based technique for long term failures.

However, it is not used for replicated system which adopted the *fail stop* failure model because in these scenarios it is cheaper to adopt a version-based technique consisting on transferring to the new node the whole state.

This technique can also be used in process replication, but then combined with checkpointing policies. The use of checkpointing is necessary because in process replication all the state –or some part of the important state– is volatile, therefore when a node crashes –in the sense it has been explained in Section 3.4– this volatile state is lost. Then, for reconstructing this volatile state in the recovery two approaches can be adopted: applying all the messages from the beginning or using checkpoints. The first option implies to maintain a log of all broadcast messages from the beginning, that as it can be assumed is undesirable in most common cases. The second one consists in generating a state checkpoint and recover a crashed node from this checkpoint. In this case the messages log must be restarted each time a new checkpoint is performed.

3.6.2 Version-based

The **version-based** recovery approach consists in transferring to each outdated node either a whole copy of the state or the last state of updated data items during its disconnection period that caused its outdated state.

To transfer a whole copy of the state has been commonly used when the replicated system adopted the *fail stop* failure model. But, when the *crash recovery with partial amnesia* failure model is adopted the recovery information set can be constrained to the state modified during the node disconnection.

Chapter 4

System Model

In this chapter it is described the basic system model considered in this thesis. In fact, it must be said that in this thesis two different system models are taken under account: one for transactional replicated systems and another one for non-transactional replicated systems. The single difference between these two system model replicated systems is that in the first one all the replication work is performed between the boundaries of transactions, while in the second one it does not.

4.1 General Configuration

Our model considers a replicated system, which is compound by several replicas, whose architecture is shown in figure 4.1, and where each replica is located 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 state is fully replicated in each node, so each replica has a copy of the whole state.

The system uses a ROWAA approach therefore reads are performed in only one replica while updates are performed in all replicas. This implies that only updates must be broadcast by the replication protocol.

The replicated system uses a *GCS* which provides point-to-point and broadcast deliveries. The minimum guarantee provided is a FIFO and reliable communication.

It is also assumed the presence of a group membership service, who *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. The group membership service combined with the *GCS* provides *Virtual Synchrony*[14] guarantees, thus each time a membership change happens, it supplies consistent

information about the current set of reachable members. This information is given in the format of *views*. Sites are notified about a new view installation with *view change events*.

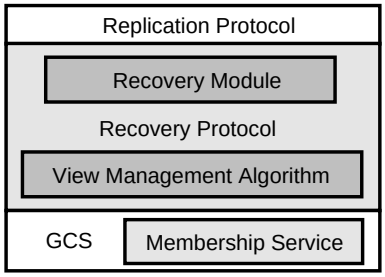


Figure 4.1: Node Architecture

The view notification mechanism is extended with node application state information providing the *enriched view synchrony* [9] approach. This makes simpler and easier the support of system cascading reconfigurations. These enriched views (*e-view*) not only inform about active nodes, but they also inform about the state of active nodes: outdated or up-to-date. The use of e-views refines the *primary partition* model into the *primary subview* model, therefore the system only can work when a *progress condition* is fulfilled[1] as it is detailed in [32]. At the same time the state consistency is ensured because only the primary *subview* is able to work in partition scenarios. Thus, this subview is the only one allowed to generate recovery information, which will be afterwards used for recovery. For similar reasons, a node can not serve client requests until it has not been fully updated.

4.1.1 Transactional Replicated System

When the previous system model is extended with the assumption that all the work is performed into the boundaries of transactions, it is obtained the transactional replicated system considered in this thesis.

It is assumed that these replicated systems broadcast a constant number of messages per transaction –constant interaction. Thus, the updates associated to each transaction are broadcast using only one single message, while the other message rounds, when used, are intended for reaching consensus. Notice, that when constant interaction is used the replication protocol uses writesets in order to propagate updates.

[1] This characteristic prevents the system from working in the starting phase until a primary subview is reached, and therefore, during this initial phase, the recovery protocol must not perform any work.

4.1.2 Non-Transactional Replicated System

If the replicated system work is not performed following transactional semantics it is obtained the behavior of a non-transactional replicated system. It is also known as process replication.

Part II

Contributions

Chapter 5

The Amnesia Phenomenon

This chapter describes the *amnesia phenomenon* in a generic way, detailing how it manifests. Once this phenomenon has been presented, it is formalized both for replicated transactional systems and non-transactional replicated systems.

5.1 Introduction

Recovery protocols are a key element when building fully operational replicated systems for ensuring their fault tolerance. These recovery protocols have as main goal to update replicas that became outdated by different reasons.

One of these possible reasons is that they crash, stop working, losing all their volatile state –state that has not been stored persistently–, being this the most complex case of node outdateness. Therefore, how the replicated system must react when the membership monitor notifies that a node has crashed? It depends. In fact, it depends on the failure model adopted.

Usually, the replicated systems adopted the *fail stop* failure model, due to its management simplicity. Crashed replicas are conceptually abandoned being substituted by a new one. In this case the recovery protocol simply has to transfer to the new joined replica the whole system state –or a consistent version–.

But in last proposals, replicated systems when managing large states have preferred to adopt the *crash recovery with partial amnesia* failure model. This is due to the fact, as it has been said, that it allows to design more efficient recovery protocols because in this case it is only necessary to transfer to the recovering node the information it has lost during its failure time.

But, when this failure model is assumed several circumstances can cause that the state reached after applying a recovery in an outdated node is not consistent with the state of other replicas. This circumstance is what is denoted the

amnesia phenomenon. Therefore, in this chapter, it is described the problem, determining how it manifests at different levels. Later, the amnesia phenomenon and its associated problems are formalized for establishing the properties that must fulfil a recovery process for overcoming it.

Once it has been formalized, a logical recovery scheme is presented which considers the step of recovering the state lost due to the amnesia phenomenon. Moreover, there are presented the design criteria that must be applied when designing a recovery protocol.

So, this chapter is structured as follows. Section 5.2 details the amnesia phenomenon. The formalization of the amnesia phenomenon for transactional replicated systems and process replication is performed in Section 5.3 and Section 5.4 respectively. In Section 5.5 is presented the logical recovery scheme, while Section 5.6 outlines the recovery information and strategies that can be used. The criteria that must be followed when designing a recovery protocol are detailed in 5.7. Finally, some related work is given in Section 5.8, and Section 5.9 concludes the chapter.

5.2 Phenomenon Description

What is *amnesia phenomenon*? When does it appear? How does it manifest and which are the problems related to? These are basic questions that must be answered before being able to propose "correct ways" for managing it.

An important aspect in the process design of replicated systems is to propose a recovery protocol because it will be the key stone for ensuring the original k-fault tolerance of this replicated system. In this scenario, the *amnesia phenomenon* appears when the designer of a recovery protocol for a replicated system assumes the *crash recovery with partial amnesia* failure model [29] in order to provide more efficient recovery processes.

This assumption implies that crashed nodes can be recovered –updated– once they become alive and reconnect to the system, if the replicated system has gone on working during their disconnection. The idea of this recovery process is that outdated replicas reach a state consistent with the replicated system state. Therefore, as these recovering replicas have a previous state it is mandatory to determine which is the exact information set that must be transferred to the recovering node, in order to ensure that the achieved state in the outdated node, after applying the recovery process, is consistent. Thus, the recovery protocol must know which is the real state reached by the outdated node when it reconnects, in order to transfer to it the exact needed information, thus avoiding the problems of losing some state changes or applying others twice or more. If it is not correctly determined, the state reached in the recovered node can be inconsistent with the replicated state –an undesired situation– that in worst cases can lead to catastrophic situations.

In this recovery scenario, the *amnesia phenomenon* implies that the real state upon the starting point for the recovering node is different to the last state it is assumed it has had before crashing. The idea is that the recovering node will have delivered some messages before crashing, therefore the system can assume that these messages have been correctly processed by this replica, assumption that as it was demonstrated in [122] is not right.

5.2.1 Amnesia Example

The example is based on the system model presented in Section 4.1.1 for transactional replicated systems. More particularly, the considered transactional replicated system uses constant interaction, broadcasting only one message per transaction, using total order communications and virtual synchrony –based on the same view delivery concept.

Therefore, consider a replicated system compound by three replicas $\{R_1, R_2, R_3\}$. The information is wholly replicated, and at the beginning all them have the same state, as it is show in Figure 5.1.

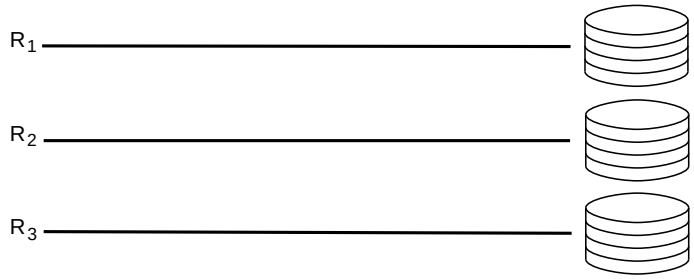


Figure 5.1: Starting system

Later at t_0, replica R_1, after processing locally the transaction T_n broadcasts its updates to all members –including itself– for processing T_n as a remote transaction. This step is shown in Figure 5.2.

During the processing of T_n each node modifies its local state in a temporary way as the Figure 5.3 depicts.

At this point, each node decides deterministically if T_n can be committed or not, so taking all them the same decision. But these steps of transaction processing and confirmation are done at different instants in each replica. Thus, it is possible that two replicas –R_1, R_2– commit T_n before R_3 does. This situation is presented in Fig. 5.4. As a consequence the state modified in a temporary way by T_n in R_1 and R_2 is persisted, while in R_3 remains as volatile state.

Then, as it is shown in Figure 5.5, replica R_3 crashes at t_1. As it crashes before committing T_n, the T_n changes –which are temporary– are lost.

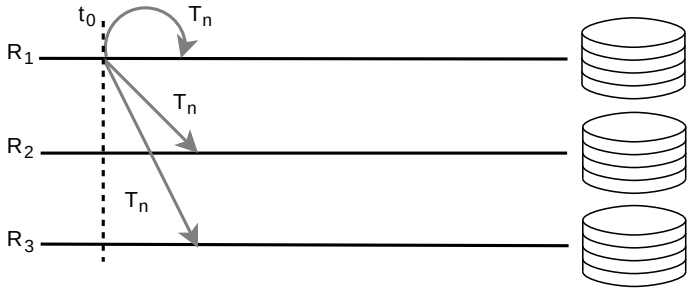


Figure 5.2: Transaction propagation

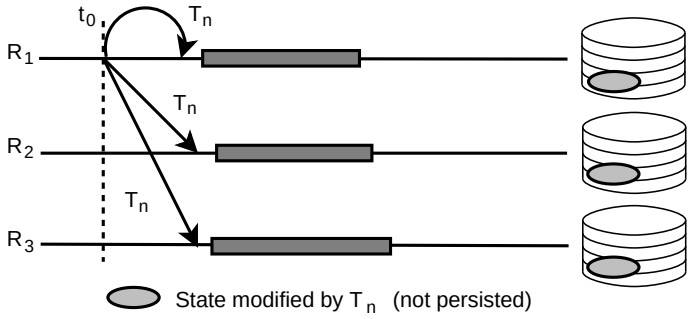


Figure 5.3: Transaction processing

Whereas R_3 is crashed as the other two replicas are alive the system goes on working because the majority progress condition is fulfilled. Thus, R_3 misses all the performed and committed changes during its disconnection –R_3 *missed* state–.

Some time later, at t_2, R_3 reconnects to the replicated system (Figure 5.6). This event is notified to the other two replicas starting then a recovery process for R_3. But, it must be noticed that if this recovery process only consists in transferring to R_3 the state it has missed during its disconnection, the state reached by R_3 when this recovery process finishes will be inconsistent with the other replicas state –an undesired situation. This is due to the fact that when R_3 has reconnected to the system, it has the same state it had before the process of T_n by the system, having no idea about T_n changes. That happens because R_3 in spite of having received and partially processed the T_n associated message it has not been able to process it completely, and therefore all its performed changes are lost due to the atomicity characteristic even though the GCS has successfully delivered the message to R_1, R_2 and R_3. This state lost is caused by the amnesia phenomenon, therefore the forgotten state constitutes a problem caused by the incorrect handling of recovery information.

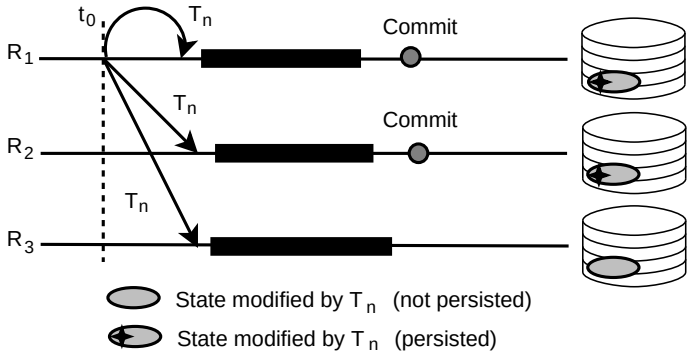


Figure 5.4: Transaction commit

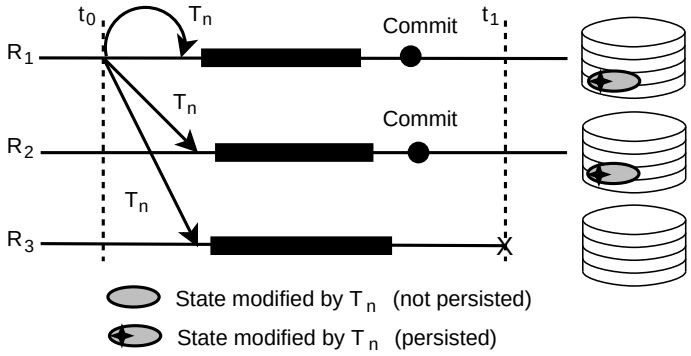


Figure 5.5: Crash of replica R_3

Obviously, in order to avoid this state inconsistency problem after applying recovery it will be necessary to include the forgotten state in the recovery process.

This example intended for transactional replicated systems based on constant interaction using a single message can be easily modified to show the amnesia phenomenon in other systems.

Lets gather now in a transactional replicated system based on constant interaction which uses more than one message per transaction. The simplest case would be to use one message for broadcasting the transaction updates and another one for spreading the final decision –either commit or abort–. Then assume that the commit of a transaction has been delivered in all replicas but at least one of them crashes before being able to process the commit, then it loses the transaction changes due to the crash. This example also fits very well for showing the manifestation of the amnesia phenomenon in transactional replicated systems based on linear interaction –where a message is sent typically for each operation in the transaction.

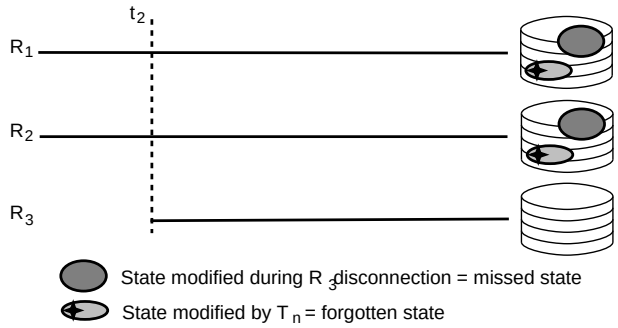


Figure 5.6: Reconnection of replica R_3

In regard to non-transactional systems, when a message has been delivered to all replicas, any replica who crashes before processing completely the message will suffer amnesia for the work of the message not processed.

It must be said that this *amnesia phenomenon* manifests differently in transactional replicated systems from non-transactional replicated systems.

5.2.2 Transactional Replicated Systems

When talking about transactional replicated systems it must be first noticed that the *consistent* state meaning will depend on the consistency level adopted in these replicated systems. For simplicity reasons in this thesis it will be assumed that the replicated system consistency level is the 1-copy-serializability –the strongest one–, therefore all replicas have the same state.

In this scenario, when a node crashes it is possible that some of its delivered messages have not been processed before crashing due to workload reasons. Moreover, it is possible that it has started to process the commit message for a transaction but it has not been able to commit it really, implying that all changes that have not been committed are lost. Therefore, if this problem is not considered at recovering time the recovery process can lead to state inconsistencies.

The crash also implies that the node can forget some of its delivered messages if they are not stored persistently. Thus, the node will not be able to apply in the recovery process both the already processed messages belonging to non committed transactions and the non processed messages.

Therefore, it can be concluded that the *amnesia phenomenon* manifests at two different levels:

- *Transport/Replication level.* At this level, it implies that the replica does not remember *which messages have been received.* Actually, the amne-

sia implies that delivered messages non-persistently stored are lost when the node crashes, generating a problem when the replicated system assumes that they have been correctly processed in all nodes, even in the crashed node –behavior that can not be always ensured–, because message delivery does not actually imply that it has been correctly processed as demonstrated in [122].

- *Replica level.* The amnesia is manifested here in the fact that the node "forgets" *which were the really applied messages.*

Thus, extra information has to be maintained to solve this problem, being afterwards used in the amnesia recovery process.

5.2.3 Non-Transactional Replication

Basically, it can be said that crashed nodes in non-transactional replicated systems at their reconnection time have lost the last actual state reached before the crash, being necessary to restore it before applying the replication messages missed during their disconnection in order to avoid diverging state evolutions.

In fact, crashed nodes at reconnection time have lost all their volatile state, while their permanent performed changes are maintained. Therefore, it will be necessary to reconstruct all their volatile state before the crash, as a first step in their recovery process.

Moreover, it is possible that some messages delivered before the crash event have not been applied in the crashed replica (e.g. due to workload). This situation makes necessary to distinguish between messages delivered and applied from those delivered but not applied before the crash. The latter should be retried.

The amnesia phenomenon also manifests at message delivery. The underlying idea is that once a message is delivered, the group communications system does not maintain it. At this point, if a node crashes before storing the message persistently, when the node reconnects the system is unable to reapply it partially –message already applied–, or to apply it.

Then, it can be concluded that the amnesia problem arises again at two different levels:

- *Transport/Replication level.* At this level, it implies that the replica does not remember *which messages have been received.*
- *Replica level.* The amnesia is manifested here in the fact that the node has lost all its volatile state and "forgets" *which were the really applied messages.*

Obviously, extra information has to be maintained to solve this problem, being afterwards used in the amnesia recovery process. The next section describes ways to avoid amnesia problems at recovery time for log-based recovery policies.

5.3 Transactional Amnesia Formalization

The next step consists in formalizing the amnesia problem for transactional replicated systems. It must be noticed that in a transactional scenario the important state is persistent, meaning that all the changes performed within the boundaries of a transaction which has been committed remain permanent, not being lost even in the presence of failures. It must be noticed, that this depends on the kind of failure. Disks might break down and also computers. So, in such cases their information is not only lost, but also unrecoverable.

Consider a replicated database system –the most common case of replicated transactional systems–, $N = \{n_1, n_2, ..., n_n\}$, compound by n replicas, being $n > 2$ (primary partition assumption [24]). It uses an eager update everywhere protocol based on a GCS which provides an atomic broadcast primitive for spreading messages and virtual synchrony. It also uses constant interaction, broadcasting each transaction in a single message.

In this system, each installed view –working view– is identified as $\mathcal{V}_x$, being x the view identifier. $T_x = \{T_{x,1}, T_{x,2}, ..., T_{x,m}\}$ are the transactions delivered (and not aborted in this view –aborted transactions are not considered because they are not relevant for recovering purposes–). As the system uses the atomic broadcast primitive [69] for spreading transactions, all alive nodes deliver the broadcast transactions in the same order, using this order at execution time. This order is being reflected by the second subindex.

$\forall\, n_y \in \mathcal{V}_x$ it is denoted as T^D_{x,n_y} the transactions subset of T_x really delivered to n_y and, respectively, T^C_{x,n_y} the transactions subset of T_x really committed in n_y; fulfilling $T^C_{x,n_y} \subseteq T^D_{x,n_y}$. Virtual synchrony [24] ensures that $T^D_{x,n_y} = T_x$. View transitions are represented as $\mathcal{V}_x \rightarrow \mathcal{V}_{x+1}$.

Then $\forall\, \mathcal{V}_i \rightarrow \mathcal{V}_{i+1}$ triggered by a node crash, it will be at least one node n_l : $n_l \in \mathcal{V}_i \setminus \mathcal{V}_{i+1}$.

Considering that $T_i = \{T_{i,1}, T_{i,2}, ..., T_{i,m}\}$ is the transactions set delivered and committed in the replicated system during $\mathcal{V}_i$, it can be assumed that $\forall\, n_k \in \mathcal{V}_i \cap \mathcal{V}_{i+1}$:

- $T_i = T^D_{i,n_k} = T^C_{i,n_k} = \{T_{i,1}, T_{i,2}, ..., T_{i,m}\}$ if they remain alive for a while.

While $\forall\, n_l \in \mathcal{V}_i \setminus \mathcal{V}_{i+1}$, due to [122] it might happen the following:

- $T_i = T^D_{i,n_l} \neq T^C_{i,n_l}$, where $T^C_{i,n_l} = \{T_{i,1}, T_{i,2}, ..., T_{i,m-s}\}$, being $0 \leq s \leq m$ in the general case, but $s \geq 1$ when $T^D_{i,n_l} \neq T^C_{i,n_l}$

It must be noticed that the assumption that $T^D_{i,n_l} = T^D_{i,n_k}$ is only valid if the used communication guarantees provide it. As it is the case with atomic broadcast

protocols. This is necessary because the Virtual Synchrony does not enforce any message delivery guarantee to crashed nodes.

In spite of assuming that $s \in \{0, .., m\}$ for simplicity reasons, it is also possible sometimes that $s > m$ due to workload reasons.

When n_l reconnects to the system, it triggers a new view $\mathcal{V}_{i+x}$, being $x > 1$. Later, the system must update it through the recovery process, transferring to it its lost transactions, which are:

- Transactions *forgotten* from its last seen view, $\mathcal{V}_i$: $T^F_{i,n_l} = T_{i,m-s+1}, ..., T_{i,m}$
- Transactions *missed* during its disconnection: $T^M_{n_l} = T_{i+1} \cup ... \cup T_{i+x-1}$

Then, for solving the amnesia phenomenon –*forgotten state*– when recovering n_l the two following properties must be provided:

- *Prop. FS1*: n_l must remember its last committed transaction, $T_{i,m-s}$;
- *Prop. FS2*: the replicated system must maintain and provide a way for obtaining the transactions subset T^F_{i,n_l} or their associated updates.

Once this *forgotten state* has been updated in the recovering replica, the recovery protocol can start with the recovery process itself, transferring *missed* data: $T^M_{n_l}$.

Therefore, any solution for these systems must fulfill these two properties for managing accurately the amnesia problem.

It exists another way for solving the amnesia phenomenon in n_l which consist in transferring to the recovering node the whole state. We denote it as property *Prop. CS1* being an alternative to the previous ones. Obviously, this solution –the one used when adopting the fail-stop failure model– is not interesting when talking about systems managing large states.

5.4 Non-Transactional Amnesia Formalization

The amnesia formalization for non-transactional replicated systems differs slightly from the formalization for transactional replicated systems. In this case instead of talking about *transactions* it is necessary to talk about *messages*.

Moreover, in non-transactional replication it is necessary to consider that the whole state –WS– of the system can be divided into volatile –VS– and permanent state –PS–. Then, it is fulfilled that $WS = VS \cup PS$ and $\emptyset = VS \cap PS$.

In fact, in these scenarios most of the replicated state is volatile state, while in a transactional system is all permanent. Then, it is necessary to handle specially the volatile state in the presence of crashes, because it is lost.

In this case it is also considered a replicated system, $N = \{n_1, n_2, ..., n_n\}$, compound by n replicas, being $n > 2$ (primary partition assumption [24]). It uses an eager update everywhere protocol based on a GCS which provides an atomic broadcast primitive for spreading messages and virtual synchrony.

In this system, each installed view –working view– is identified as in the previous section as $\mathcal{V}_x$, being x the view identifier. $M_x = \{M_{x,1}, M_{x,2}, ..., M_{x,m}\}$ are the messages delivered. As the system uses the atomic broadcast primitive [69] for spreading operations, all alive nodes deliver the broadcast messages in the same order, using this order at execution time. This order is being reflected by the second subindex.

$\forall\, n_y \in \mathcal{V}_x$ it is denoted as M^D_{x,n_y} the messages subset of M_x really delivered to n_y and, respectively, M^A_{x,n_y} the messages subset of M_x really applied in n_y; fulfilling $M^A_{x,n_y} \subseteq M^D_{x,n_y}$. Virtual synchrony [24] ensures that $M^D_{x,n_y} = M_x$. View changes are represented as $\mathcal{V}_x \rightarrow \mathcal{V}_{x+1}$.

In regard to the volatile state, VS, it is denoted as VS_{n_y} the volatile state of the replica n_y.

Then $\forall\, \mathcal{V}_i \rightarrow \mathcal{V}_{i+1}$ triggered by a node crash, it will be at least one node n_l : $n_l \in \mathcal{V}_i \setminus \mathcal{V}_{i+1}$.

Considering that $M_i = \{M_{i,1}, M_{i,2}, ..., M_{i,m}\}$ is the message set delivered and applied in the replicated system during $\mathcal{V}_i$, it can be assumed that $\forall\, n_k \in \mathcal{V}_i \cap \mathcal{V}_{i+1}$:

- $M_i = M^D_{i,n_k} = M^A_{i,n_k} = \{M_{i,1}, M_{i,2}, ..., M_{i,m}\}$ if they remain alive for a while.
- VS_{n_l} will be the volatile state resulting of applying the message set $\{M_{0,1}, ..., M_{i,m}\}$

While $\forall\, n_l \in \mathcal{V}_i \setminus \mathcal{V}_{i+1}$, due to [122] it might happen the following:

- $M_i = M^D_{i,n_l} \neq M^A_{i,n_l}$, where $M^A_{i,n_l} = \{M_{i,1}, M_{i,2}, ..., M_{i,m-s}\}$, being $0 \leq s \leq m$ in the general case, but $s \geq 1$ when $M^D_{i,n_l} \neq M^A_{i,n_l}$
- VS_{n_l} will be the volatile state resulting of applying the messages set $\{M_{0,1}, ..., M_{i,m-s}\}$

As for transactional formalization the assumption that the crashed replica delivers the same message set as not crashed nodes is not ensured by virtual synchrony. Therefore, it must be guaranteed by other communication guarantees as the one provided by atomic broadcast protocols.

In spite of assuming that $s \in \{0, .., m\}$ for simplicity reasons, it is also possible sometimes that $s > m$ due to workload reasons.

When n_l reconnects to the system, it triggers a new view $\mathcal{V}_{i+x}$, being $x > 1$. Later, the system must update it through the recovery process, being necessary

to recover first its VS_{n_l} before the crash and applying to it its lost messages, which are:

- Messages *forgotten* from its last seen view, $\mathcal{V}_i$: $M^F_{i,n_l} = M_{i,m-s+1}, ..., M_{i,m}$
- Messages *missed* during its disconnection: $M^M_{n_l} = M_{i+1} \cup ... \cup M_{i+x-1}$

Then, for solving the amnesia phenomenon –*forgotten state*– when recovering n_l the three following properties must be provided:

- *Prop. FS1*: n_l must remember its last applied message, $M_{i,m-s}$;
- *Prop. FS2*: the replicated system must maintain and provide a way for obtaining the messages subset M^F_{i,n_l} or their associated updates.
- *Prop. FS3*: n_l must remember its last volatile state VS_{n_l}

In regard to the properties that must be fulfilled for supporting the amnesia phenomenon in a transactional replicated system it appears a new property, *Property FS3*. The first two ones are equivalent to the ones proposed for the transactional replicated system. Anyway, it must be noticed that if the transactional replicated system has also volatile state the *Property FS3* must be fulfilled too.

Once this *forgotten state* has been updated in the recovering replica, the recovery protocol can start with the recovery process itself, transferring *missed* data: $M^M_{n_l}$.

Obviously, it is possible to transfer the whole state, *Prop. CS1* , avoiding the amnesia phenomenon, as it has been commented in the transaction formalization. But, the idea is to avoid to transfer the whole state.

Demonstrating that properties *FS1* and *FS2* are necessary is straightforward, but may be the necessity of *FS3* is no so direct. Its necessity can be demonstrated using contradiction. Assume that property *FS3* is not necessary for avoiding inconsistencies and that the last volatile state before crashing of a replica was $VS_{n_l} = S$. Therefore, when this replica recovers after crashing it will start to apply its forgotten messages M^F_{i,n_l} provided by properties *FS1* and *FS2* over an $\emptyset$ or initialized volatile state –$VS_{n_l} = \emptyset$ or $VS_{n_l} = S_0$– instead of applying it over the correct volatile state $VS_{n_l} = S$ –the one that it would have had if it had not crashed when applying these messages. So, this inconsistency in the volatile state can lead to new inconsistencies if some work performed by M^F_{i,n_l} or subsequent messages depends on the VS_{n_l}. So, property *FS3* is mandatory for avoiding possible inconsistencies in the recovery process due to the amnesia phenomenon.

A possible example of the previous problem when property *FS3* is not provided is the following one. Assume a cluster of application servers compound by three

replicas, $N = \{R_1, R_2, R_3\}$, where a service $Serv$ is deployed in all them being all allowed to serve client requests. The deployed service performs some work or another attending to b, the number of served sessions by the whole cluster at each instant, where $b \in VS$. So, if $b < b_{threshold}$ $Serv$ does A, else $Serv$ does B. Consider also that the recovery of a crashed replica is performed transferring the messages it has lost during its disconnection and not transferring the whole state.

Then at certain instant, t_0, replica R_2 crashes, losing its volatile state, $V_{R_2} = S$, having $M^F_{i,R_2} \neq \emptyset$ and being $M_{i,m-s}$ its last correctly applied message. Later, at instant t_1 replica R_2 reconnects starting its recovery process. Therefore, it will start to apply first the forgotten messages, $M_{i,m-s}$, subsequently its missed messages $M^M_{R_2}$ –transferred by other replicas– and then the messages propagated after its reconnection denoted as M_{i+x}.

At this point, assume that message M_a, a subsequent message of $M_{i,m-s}$, requests the service whose work depends on variable $b \in VS_{R_2}$. Notice that its value just before the crash was $b_{t_0} = b(M_{i,m-s}) > b_{threshold}$ while at recovery time $b_{t_1} = b_O < b_{threshold}$, being $b_{t_0} \neq b_{t_1}$ –because *FS3* is not provided. Then the results of M_a work would be different in R_2 compared to those obtained in R_1 and R_3 and that would have been also obtained in R_2 if it had not crashed.

Obviously, this would have not happen if the property *FS3* is provided. In this case R_2 will apply the $VS_{R_2} = S$ as first step in its recovery process, then when applying message M_a, $b_{t_1} = b(M_{i,m-s})$ obtaining the same result when applying M_a as if R_2 has not crashed.

5.5 Basic Recovery Schema

After presenting the amnesia phenomenon, its related problems and its formalization it is time to know how it must be included in the overall recovery process. Then if the amnesia phenomenon is considered the logical steps that must be performed in a recovery process are the ones presented in the figure 5.7.

Recovery process:
1 - *Amnesia Recovery Process*
2 - *Update Recovery Process*
3 - *Current Recovery Process*

Figure 5.7: Logical Recovery Process

The recovery process is initiated each time the replicated system detects the presence of outdated nodes. The outdateness cause will be node disconnections –network partitions– or node crashes.

- *Amnesia Recovery Process (ARP).* This is the first recovery step to be

performed when a node is being recovered. But, it is only necessary if the node becomes alive after crashing. The goal of this step is to recover what it is called *forgotten* state. The conditions that must be guaranteed for performing this step are the ones presented in *Properties FS1 and FS2* for transactional replicated systems, and *Properties FS1, FS2 and FS3* for non-transactional replicated systems

- *Update recovery Process (URP).* In this step, outdated nodes update their missed state. The outdated nodes recover during this phase the state changes lost during their disconnection.
- *Current Recovery Process (CRP).* This last step is done if the replicated system is working during the outdated node recovery. It is performed once the node has applied all its missed views, and its goal is to apply in the recovering node any message received during the previous stages (which could not be applied, since the node was not updated yet). Obviously, recovery processes performed in non working partitions do not need to apply this step.

As it has been said these are the logical steps that must be performed in a recovery process. Later, when implemented in real recovery processes some of these steps can be combined or mixed depending on the recovery techniques and the information maintained for this purpose.

5.6 Amnesia Recovery Information and Strategies

Once it has been explained and formalized the amnesia phenomenon either for transactional replicated systems or non-transactional replication and the logic recovery schema that includes the amnesia recovery has been presented, it will be detailed how the two main recovery strategies presented in 3.6 must be used for supporting the *amnesia phenomenon*.

Thus, node recovery must include both the recovery of "missed state", and the recovery of the "forgotten state". The first one refers to the data state that the node has not received because it was disconnected. The second one covers the state received but later lost (due to the amnesia) when the node crashed. Therefore, the recovery system must maintain information allowing it to handle these two out-of-date causes.

5.6.1 Log-Based Strategy

The **log-based** policies use the broadcast messages as recovery information. If the amnesia is not considered, the recovery information only refers to messages

missed by crashed nodes. Thus, in this case, the recovery information must be created when the membership monitor detects disconnected nodes in the replicated system. Then, the information must be maintained until the outdated nodes have applied the missed messages.

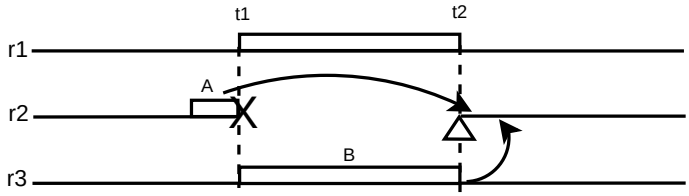


Figure 5.8: Log Recovery Information for Transactional systems.

However, the amnesia recovery information for *log-based* policies relates to those messages delivered and not being processed in a crashed replica.

Notice, that for transactional replicated systems using more than one message for broadcasting and processing transactions, the replicas have started to process some transactions but have not finished them because they need to receive the messages that conclude these transactions. Therefore, it appears the phenomenon on non-yet-committed or on-going transactions. When a node crashes all changes associated to these transactions are lost because they have not been committed, being necessary later to reprocess them. This phenomenon implies some problems and needs a special management as it is later explained in Section 10.3.

Figure 5.8 shows the information needed to recover an outdated node using the log-recovery strategy in replicated transactional systems. In this figure node $r2$ crashes at time $t1$ and reconnects at time $t2$. At this moment, the system must start the $r2$ recovery.

- Firstly, it needs the A recovery information block used to recover the amnesia –ARP stage–, it contains the messages received but not committed by $r2$ before its crash at $t1$ –it has to fulfil the *Prop. FS1* and *Prop. FS2*.
- Secondly, the system needs B block which contains the messages missed by $r2$ (either committed or not-yet-committed) during its failure time in order to perform the *URP*.
- Finally, $r2$ must perform the *CRP* starting to apply the messages it has delivered after $t2$. These messages must be enqueued for applying them after in the *CRP*.

Obviously, the A block can be managed and maintained by $r2$ whilst the B block must be generated and managed by a non-failed node.

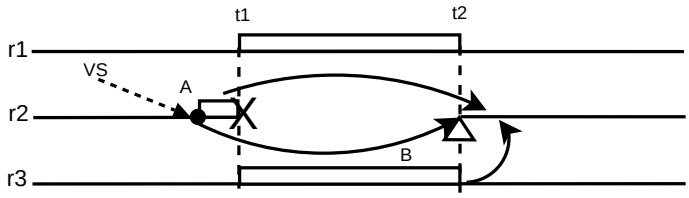


Figure 5.9: Log Recovery Information for Non-Transactional Systems.

It must be remarked that for non-transactional systems it is necessary, as it has been said previously in Section 5.4, to have a copy of volatile state.

In figure 5.9 it is shown the information needed to recover an outdated node using the log-recovery strategy in non-transactional replicated systems, presenting some differences. The events sequence in this figure is the same as for figure 5.8: node $r2$ crashes at time $t1$ and reconnects at time $t2$. At this moment, the system must start the $r2$ recovery.

- Firstly, it needs the *checkpoint* block –VS– which recovers the volatile state before crashing –fulfilling the *Prop. FS3*.
- Secondly, it applies the A recovery information block which contains in this case the messages delivered but not applied before $t1$ –fulfilling the *Prop. FS1* and *Prop. FS2*. Once the A block is processed the ARP finishes.
- Later, the system needs the B block which contains the messages missed by $r2$ during its failure time in order to perform the URP stage.
- Finally, $r2$ must perform the CRP starting to apply the messages it has delivered after $t2$. These messages must be enqueued for applying them after CRP.

Obviously, on one hand the VS must be managed and maintained by $r2$. On the other hand, A block can be managed and maintained by $r2$ whilst the B block must be generated and managed by a non-failed node.

5.6.2 Version-Based Strategy

The **version-based** recovery approach, mainly intended for transactional systems, consists of transferring to each outdated node the last state of updated data items during its disconnection period that caused its outdated state. The whole transfer is not considered in this thesis as long as the main purpose of adopting the crash recovery with partial amnesia failure model is to avoid this

recovery technique. Thus, a snapshot of the lost updated state must be transferred. As the transferred information only refers to committed changes it must be completed with recovery information related to the on-going transactions when the recovery version-based information snapshot was taken. This extra information, if needed, can be obtained from the applied log-based policies.

This recovery information policy can be supported in two different ways:

- By maintaining information about the last state of changed items, or
- Using the whole database information.

It must be noticed that the *version-based* amnesia recovery information is included in the data state, as well as in the transferred ongoing transactions. In this sense, it is assumed that messages belonging to on-going transactions are maintained in the system. The only thing that must be guaranteed is that the transferred state includes all the changes performed after the last change performed in the outdated node –remember *Prop. FS1.*

The recovery information necessary to recover the outdated nodes using version-based strategies is detailed graphically in figure 5.10. As in the example shown in figure 5.8, the $r2$ node fails at $t1$ and reconnects at $t2$ time. In this case, in order to recover $r2$, the recovery protocol first transfers the A recovery information block which is a snapshot of the committed lost state by $r2$. The block A contains the data state generated by:

- committed transactions before $t1$ that have been lost by $r2$ due to amnesia, and
- committed transactions between $t1$ and $t2$ (including committed transactions completely missed by $r2$ or ongoing transactions at $t1$ time committed before $t2$).

Afterwards, the system will transfer the B block. This recovery information block only contains ongoing transactions, regardless their start time were previous or posterior to $t1$.

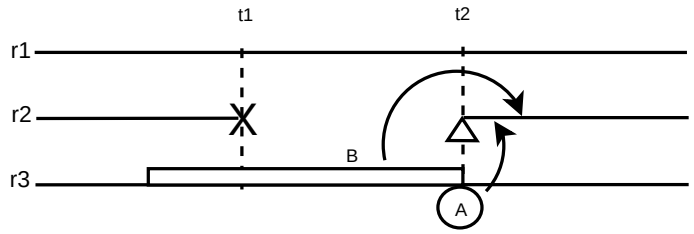


Figure 5.10: Version Recovery Information.

5.7 Recovery Protocols Design Criteria

Which criteria must be followed when designing a recovery protocol? The idea is to know which principles or guidelines must be considered for designing and developing them. Usually, these criteria are related to correctness and overhead characteristics.

5.7.1 Correctness Criteria

First of all, the main basic criterion should be to ensure its correctness in consistency terms. In other words this principle means that the recovery protocol does what it is supposed to do: to update an outdated replica, reaching in the recovering replica a consistent state regarding the state of updated replicas. If this criterion is not ensured the recovery protocol is useless. Obviously, as it has been demonstrated in previous sections, recovery protocols for replicated systems which assume the *crash recovery with partial amnesia* failure model need to fulfill the properties previously determined in order to overcome the *amnesia phenomenon*.

Another correctness criterion design –although it can be considered a finiteness criterion– must ensure that the recovery process will end successfully in a bounded time. The paper [103] has demonstrated that under certain workload circumstances some recovery protocols are not able to finish the recovery process in a recovering replica. Therefore, the replica remains always outdated without reaching the state of the other replicas. Only, being possible for the *recovering* replica to reach the state of other replicas if the workload decreases. Notice, that it is possible that the *recovering* node has all the information it needs to reach the consistent replicated state because it has received all its missed state –being not outdated in this sense–, but the speed in which it is able to process this information and new incoming requests is lower than the speed at which new requests come in, being in this sense its really applied state more outdated than before. Logically, this behavior is undesired because the replica instead of having all the information it needs for being fully recovered, it is stressed. This will imply sometimes high delays in the process of requests for which it acts as delegate server.

5.7.2 Overhead Criterion

The other main criterion to consider is the overhead introduced by recovery protocols. As it is known, recovery protocols are used to increase the fault tolerance and high availability –in terms of service availability– of replicated systems. But, to do so they must perform extra work that will introduce some cost that decreases the performance level and high availability –in terms of response time– of replicated systems –one of its original goals. Thus, the recovery

protocol must also be designed to work minimizing the overhead introduced in the replicated system overall performance.

In the optimal case, and hypothetical one, the recovery protocol will not introduce any overhead in the replicated system. But, this goal is impossible to reach. But when appears the overhead? Three main cases can be observed, depending on if there are not crashed replicas in the system, if there are, or if there are replicas being recovered.

A recovery protocol may introduce overhead when in the replicated system there are not crashed replicas. At first glance, it can be thought that in this case the recovery protocol neither has to update any outdated replica nor has to generate recovery information –information that will be used later by the recovery protocol–. So, no overhead should be introduced. But, as it has been seen if the *crash recovery with partial amnesia* failure model is assumed the recovery protocol needs to generate recovery information even if there are not crashed nodes in the system in order to overcome the amnesia problem. Therefore, a minimum cost is introduced, the cost of generating the necessary recovery information. Notice, that if the GCS used provides the successful delivery [122] it would not be necessary to generate the recovery information in this case. Later, in Chapter 9 it will be seen how some literature recovery proposals generate recovery information during this scenario, whilst others do not.

Obviously, when there are crashed replicas, it is necessary to generate recovery information for the time they restart in order to be updated –recovered–. Information used to recovery the *missed* state. Therefore in this case the recovery protocol will always introduce some overhead in the performance of the replication protocol it works for. This work scenario can be mixed with the previous one, because in both cases it is necessary to generate recovery information, when the *amnesia phenomenon* is possible. But, as soon as the recovery protocol uses two different techniques for recovering the *forgotten* and *missed* state, the overhead introduced will be slightly different.

The other case is when an outdated node is recovered, independently if there are crashed nodes or not –combining their overheads. Naturally, in this case the recovery protocol also introduces some cost. This is the cost of transferring the information lost by the outdated node during its disconnection. This cost is basically compound by the cost of collecting the information to be transferred in the *recoverer* node, transferring it to the *recovering* one, and applying the information. In some cases, version-based recovery techniques minimize the cost of recovery information generation transferring the main part of the cost to the step when the information is collected –executing very expensive SQL sentences, for instance. Moreover, some of these version-based techniques lock database tables during the recovery process decreasing the performance of the replication protocol.

5.7.3 Summary

In conclusion, it must be said that when a recovery protocol is designed it must ensure the two first criteria correctness: both in consistency and time-bound terms. Whilst, the basic goal for the overhead criterion is to minimize it in all the predefined cases.

At this point, it can be proposed some parameters for describing the recovery protocols both for their correctness time-bound criterion and for their introduced overhead. Therefore, these parameters can be used for comparing them and select which approach best suits the overall replication system needs.

For the *correctness time-bound* criterion it would be interesting to determine the workload threshold, in the way that for any workload value higher than this threshold the system can not ensure the correctness time-bound criterion. As this threshold can vary depending on the replication technique used, the number of replicas, and other parameters. It would be necessary to generate threshold graphics representing the workload threshold evolution for each replication technique depending on the number replicas. This would be very useful when selecting a recovery solution depending on the workload requirements of the replicated system.

In regard to the *overhead* criterion it would be also interesting to study and represent through graphics the overhead evolution –for the three different cases. For simplification reasons, it would be interesting to assume for the second and third case, that there is only one crashed replica and a replica being recovered respectively.

5.8 Related Work

In [29] the author proposed a categorization of the crashed nodes failure depending on the state they have when they restart –if the failure model supports it– for distributed systems.

As it has been said in the Introduction, replicated systems at the beginning commonly adopted the *fail-stop* failure model as [14], while last trends in transactional replicated systems [82, 20, 19, 6, 79] have adopted the *crash recovery with partial amnesia*. The second one is largely used because allows to design more efficient recovery processes. But, its main drawback relates to the fact that its associated problems have not been deeply considered by researchers as they were more focused on developing very fast replication and recovery protocols. Therefore, not all recovery proposals for systems assuming the *crash recovery with partial amnesia* handle its problems accurately.

Anyway, some papers as [122] have already pointed out the difference between message delivery and message processing. A very important difference when talking about *crash recovery with partial amnesia* failure model because implies

that all not delivered messages in a replica have been already correctly processed. For overcoming this problem they proposed the *successful delivery* which implied that a message has been both delivered and correctly processed. Therefore, a message is not discarded by a GCS until it has not been successful delivered in all replicas. Then, if it is assumed that the GCS provides the *successful delivery* behavior, the amnesia phenomenon is avoided.

It must be said that some authors also noticed the necessity of persisting messages at some point in the broadcast process when designing atomic broadcast algorithms based on consensus [84, 109] for the crash-recovery model.

In [84] the author forced the system to persist delivered messages in order to avoid possible consistency problems associated to the fact that nodes can disconnect. Authors of [109] also make mandatory in their proposal that for each consensus round each replica has to persist its proposed messages, being possible later to replay these consensus rounds as recovery process. As an optimization proposed to persist the queue of agreed messages. Although both solutions provided a basic mechanism for overcoming the amnesia problem neither of these two solutions specified the necessity of persisting the messages as an atomic step of the delivery process –at the basic approach of [109] it was not necessary–, leading also sometimes to undesired situations if problems appeared between message delivery and message persisting. But in spite of the development of these approaches, most GCS provide atomic broadcast implementations that are not based on them and do not consider the necessity of persisting messages. So, they can not manage correctly the amnesia phenomenon. Authors of [91] also introduced the necessity of checkpointing the state of each replicated process in two levels: application level and atomic broadcast algorithm level in order to provide more correct recovery processes.

Also in former proposals for using atomic broadcast in replicated databases as [100] the authors introduced the concept of irrevocability for the delivery process. It implied that a process must not forget the messages it has already delivered. Therefore, the necessity of remembering the delivered messages was already detected as it has been pointed out in this chapter.

5.9 Conclusions

In this chapter it has been described the amnesia phenomenon, and how it can derive to a consistency problem when the *crash recovery with partial amnesia* failure model is adopted.

Subsequently, it has been formalized both for transactional replicated systems and process replication. Proposing the properties that must be fulfilled in both cases for overcoming it. Later, it has been proposed a logic recovery schema that includes the amnesia recovery step.

Finally, the chapter includes a set of design criteria that must be followed when

constructing a recovery protocol for replicated systems.

Chapter 6

Amnesia and Majority Partitions

Replicated transactional systems only can go on working if they fulfill a progress condition. This condition fulfillment is adopted for ensuring the consistency of the replicated system. Replicated transactional systems usually assume the crash-recovery with partial amnesia failure model, and the majority partition progress condition. But, despite the large use of such combination most of these works do not handle accurately a very special phenomenon that can lead to diverging states in different replicas causing, when happening, critical situations. In this chapter it will be described and formalized the problem, and a solution is provided.

6.1 Introduction

This chapter presents a replicated consistency problem which arises when combining the *amnesia phenomenon* –non-correctly handled– with a specific replicated system composition allowed by the *majority partition* progress condition. A problem that can lead to different state evolutions among the members of the replicated system.

Later, the problem is formalized, establishing the replicated system conditions that would generate it, and the properties that must be fulfilled for overcoming it.

Despite being a rare problem, it should be accurately managed for avoiding critical situations. Therefore, two different approaches are presented for overcoming it, being each one of them interesting for transactional replicated systems with different characteristics. On one hand, a solution is presented for critical systems where already performed and committed work at replicated system level

can not be undone nor lost. On the other hand, it is proposed the use of a technique used in partitionable systems, reconciliation, whose main advantage is its zero overhead in normal work.

The rest of the chapter is structured as follows. It is first outlined in Section 6.2 with a simple example the arising problem. Later, it is performed a short formalization about progress conditions in Section 6.3. In Sections 6.4 and 6.5 it is formalized the amnesia issue with the progress condition both for transactional and non-transactional replicated systems, presenting some solutions in Section 6.6. Finally, related work is included in Section 6.7, and Section 6.8 concludes the paper.

6.2 A Sample Problem

As it has been said, combining the amnesia problem –which appears when the replicated system adopts the crash-recovery with partial amnesia failure model– with the replicated system progress condition –primary partition– can lead the replicated system to state inconsistencies. The problem is that the system is unable to guarantee the correct system data state progress. This inconsistency problem can be seen with the following example.

Consider that the information system of a hospital is compound by three replicas, $\alpha = \{R_1, R_2, R_3\}$, and all the hospital terminals work against it. All three replicas are fully updated –with the same state– and working at the instant t_0. Then, a doctor introduces a first patient diagnosis in the system through T_n –including the necessary analysis that need to be performed for refining it–, being delivered and committed in all replicas.

After performing and studying this analysis, the doctor introduces in the system that the patient is forbidden to eat some particular food –because its ingestion can derive in severe health patient consequences– through T_{n+1}. T_{n+1} is delivered to all replicas, but only committed in R_1. This situation is depicted in Figure 6.1.

This is due because R_2 and R_3 nodes crash before being able to commit T_{n+1}, as Figure 6.2 shows, moreover, R_2 and R_3 lost the T_{n+1} associated message because the replication protocol does not persist it. R_2 and R_3 crash implies that the hospital information system does not fulfil the primary partition progress condition, so it stops working.

Once the hospital IT staff has repaired R_2 and R_3, these replicas are reconnected to the system, but in this view change it also crashes R_1 as it is detailed in Figure 6.3.

Then the information system fulfils the progress condition, but it arises a consistency problem, R_2 and R_3 have not seen the T_{n+1} changes. So, as they fulfil the progress condition they can go on working, but if they work they will start from the state reached after committing T_n and not T_{n+1} –the last really committed

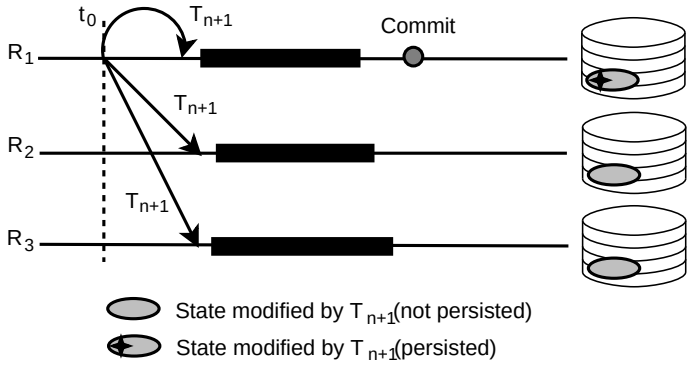


Figure 6.1: Transaction commit

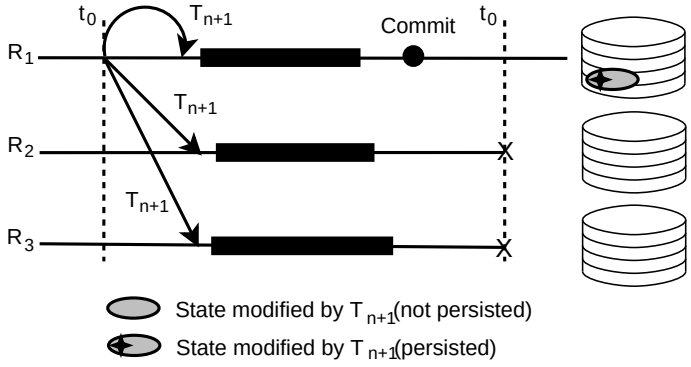


Figure 6.2: Crash of R_2 and R_3

transaction in the replicated transactional system– leading to a diverging state evolution to R_1 state –which is the correct one.

It must be said that this situation or another combination of events that leads to a similar situation is very improbable in a replicated system. And this probability diminishes as long as the number of replicas increases. But, it must be correctly managed in order to avoid undesired situations in the replicated consistent state. In the previous example, the inconsistency can imply that the patient eats something that he is forbidden, causing severe damages to his health.

As previous step to presenting possible solutions that can be applied for solving this problem, it is formalized.

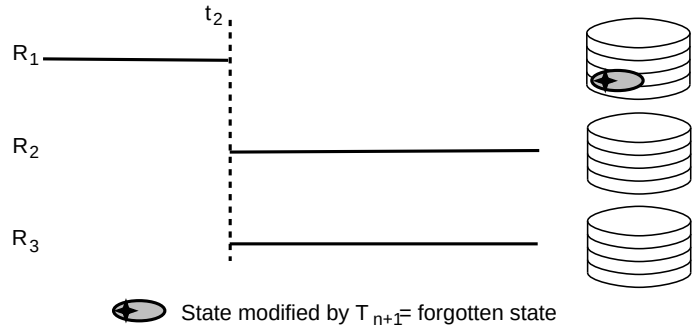


Figure 6.3: Reconnection of R_2 and R_3

6.3 Progress Condition

Progress condition is the condition that must be fulfilled by a replicated system to be enabled to work. Usually, replicated systems have adopted the primary partition condition [24]. So, in this case the replicated system is allowed to work if a majority of its replicas is alive.

6.3.1 Progress Condition Formalization

Considering a replicated transactional system, $N = \{n_1, n_2, ..., n_n\}$, compound by n replicas –with $n > 2$–, N_x represents that it has a working view, $\mathcal{V}_x$, while with N_x^* that it has not any working view, being $\mathcal{V}_x$ the last working view installed in the system. Minority partitions are represented by $\mathcal{V}_y^*$, where y is the last working view seen by the members of this partition.

Thus, it is in a working view, N_x, if it has a $\mathcal{V}_x$: $card(\mathcal{V}_x) >= \lfloor \frac{n}{2} \rfloor + 1$. Contrarily, it is said that it is in a non-working view N_x^*. Minority partitions, $\mathcal{V}_x^*$, always fulfil that $card(\mathcal{V}_x^*) < \lfloor \frac{n}{2} \rfloor + 1$.

For formalization reasons, two different view counters are used: one for total installed views –first subindex–, and another one for working installed views –second subindex. The first subindex is used for noticing that membership changes also occur in non-majority partitions, installing "views", although usually authors only use the view concept for partitions which fulfil the progress condition. So, this first counter is increased in any members group view change –but it has not any purpose in a real system–, while the second one is only increased when a new working view is installed –being the counter that must be used in a real system–. Possible view transitions are shown in table 6.1.

TRANSITION CASES
Node Addition
T1: $\mathcal{V}_{x,j} \cup \mathcal{V}^*_{k,l} \rightarrow \mathcal{V}_{x+1,j+1}$
T2: $\mathcal{V}^*_{x,j} \cup \mathcal{V}^*_{k,l} \rightarrow \mathcal{V}^*_{max(x,k)+1,max(j,l)}$
T3: $\mathcal{V}^*_{x,j} \cup \mathcal{V}^*_{k,l} \rightarrow \mathcal{V}_{max(x,k)+1,max(j,l)+1}$
Node Removal
T4: $\mathcal{V}_{x,j} \rightarrow \mathcal{V}_{x+1,j+1}$
T5: $\mathcal{V}_{x,j} \rightarrow \mathcal{V}^*_{x+1,j}$
T6: $\mathcal{V}^*_{x,j} \rightarrow \mathcal{V}^*_{x+1,j}$

Table 6.1: View Transitions.

6.4 Transactional Problem Formalization

Assume a replicated transactional system, $N = \{n_1, n_2, ..., n_n\}$, compound by n replicas, being $n > 2$.

$\forall T5$ transitions triggered by node crash/es it will be at least one $n_l : n_l \in \mathcal{V}_{x,j} \setminus \mathcal{V}^*_{x+1,j}$.

Considering that $T_j = \{T_{j,1}, T_{j,2}, ..., T_{j,m}\}$ is the transactions set delivered and committed in the replicated system during $\mathcal{V}_{x,j}$, it can be assumed that $\forall\, n_k \in \mathcal{V}_{x,j} \cap \mathcal{V}^*_{x+1,j}$:

$$T_x = T^D_{j,n_k} = T^C_{j,n_k} = \{T_{j,1}, T_{j,2}, ..., T_{j,m}\}$$

While $\forall\, n_l \in \mathcal{V}_{x,j} \setminus \mathcal{V}^*_{x+1,j}$, as it has been formalized in subsection 5.3, it might happen the following: $T_j = T^D_{j,n_l} \neq T^C_{j,n_l}$, where:

$T^C_{j,n_l} = \{T_{j,1}, T_{j,2}, ..., T_{j,m-s}\}$, being $0 \leq s \leq m$.

Due to the *amnesia phenomenon* minority partitions will be distinguished, $\mathcal{V}^*_{z,j}$ –which is used in a generic way–, from $\check{\mathcal{V}}^*_{z,j}$ and $\hat{\mathcal{V}}^*_{z,j}$. The first ones, are minority partitions whose last seen view is j, but they can not ensure that they do not have the amnesia phenomenon in relation to this view because all their n_l nodes that have seen the j view fulfil that $n_l : n_l \in \mathcal{V}_{x,j} \setminus \mathcal{V}^*_{x+1,j}$. While second ones, $\hat{\mathcal{V}}^*_{z,j}$, are minority partitions that have seen also the j view, and at least one of their nodes that has seen j fulfills $n_m \in \mathcal{V}_{x,j} \cap \mathcal{V}^*_{x+1,j}$.

Later, if in the first transition of type $T3$ to a new working view, $\mathcal{V}_{k,j+1}$ –recall that the last installed working view in the system was $\mathcal{V}_{x,j}$–, the new installed view fulfils the following:

$\mathcal{V}_{k,j+1} = A \cup B$ where:

- $A = \{n_l \in \mathcal{V}_{x,j} \setminus \mathcal{V}^*_{x+1,j} : n_l \notin \hat{\mathcal{V}}^*_{z,j} : x+1 < z < k\}$, are the nodes that were alive in the last working view, but crashed –so they were not alive in any $\mathcal{V}^*_{x+1,j}$– triggering the view change that lead $N_j \rightarrow N^*_j$, and did not belong to any minority view that can recover the whole j view.
- $B = \{n_k \notin \mathcal{V}_{x,j} \cap \hat{\mathcal{V}}^*_{z,j} : x < z < k\}$, are the nodes that did not belong to the last working view, and that have not recovered the whole j view in any minority partition.

Then, the new reached majority is enabled to go on working. But, in this situation a problem can arise if the s term for A nodes fulfils that $s > 0$. This is because this new installed majority will be unable to reach the last consistent replicated state –the one reached after applying $T_{j,m}$–, due to the fact that:

- $\forall\, n_l \in A$ it is fulfilled that $T^F_{j,n_l} \neq \emptyset$
- $\forall\, n_s \in B$ it is either fulfilled that $T^D_{j,n_k} = T^C_{j,n_k} = \emptyset$ or $T^F_{j,n_l} \neq \emptyset$

So, the arising consistency problem conditions are:

- *Condition 1*: $T3$ transition $\rightarrow \mathcal{V}_{k,j+1}$
- *Condition 2*: $\mathcal{V}_{k,j+1} = A \cup B$
- *Condition 3*: $\forall\, n_l \in A$ it is fulfilled that $T^F_{j,n_l} \neq \emptyset$

The properties that must fulfil the replicated system to avoid this possible situation are similar to the ones proposed for solving the general amnesia phenomenon in subsection 5.3: in fact the *Prop. FS1* is necessary as it is defined in 5.3, while the *Prop. FS2* must be slightly modified to overcome this problem:

- *Property FS2**: each node $n_l \in A$ must maintain and provide a way for obtaining its T^F_{j,n_l} transactions subset or associated updates, instead of trusting in "the replicated system".

The generic solution, described in Section 8.3 and presented in [38, 36], fulfils also both properties, as explained in the next section.

Once formalized the problem for transactional systems it would be formalized for non-transactional replicated systems.

6.5 Non-Transactional Problem Formalization

The problem formalization for non-transactional replicated systems follows the same steps as the formalization performed in the previous section.

It must be also assumed a non-transactional replicated system, $N = \{n_1, n_2, ..., n_n\}$, compound by n replicas, being $n > 2$.

$\forall T5$ transitions triggered by node crash/es it will be at least one $n_l : n_l \in \mathcal{V}_{x,j} \setminus \mathcal{V}^*_{x+1,j}$.

Considering that $M_j = \{M_{j,1}, M_{j,2}, ..., M_{j,m}\}$ is the messages set delivered and applied in the replicated system during $\mathcal{V}_{x,j}$, it can be assumed that $\forall\, n_k \in \mathcal{V}_{x,j} \cap \mathcal{V}^*_{x+1,j}$:

$$M_x = M^D_{j,n_k} = M^A_{j,n_k} = \{M_{j,1}, M_{j,2}, ..., M_{j,m}\}$$

While $\forall\, n_l \in \mathcal{V}_{x,j} \setminus \mathcal{V}^*_{x+1,j}$, as it has been formalized in subsection 5.4, it might happen the following: $M_j = M^D_{j,n_l} \neq M^A_{j,n_l}$, where:

$M^A_{j,n_l} = \{M_{j,1}, M_{j,2}, ..., M_{j,m-s}\}$, being $0 \leq s \leq m$.

Due to the *amnesia phenomenon* it will be distinguished minority partitions, $\mathcal{V}^*_{z,j}$ –which is used in a generic way–, between $\check{\mathcal{V}}^*_{z,j}$ and $\hat{\mathcal{V}}^*_{z,j}$. First ones, are minority partitions whose last seen view is j, but they can not ensure that they do not have the amnesia phenomenon in relation to this view because all their n_l nodes that have seen the j view fulfil that $n_l : n_l \in \mathcal{V}_{x,j} \setminus \mathcal{V}^*_{x+1,j}$. While second ones, $\hat{\mathcal{V}}^*_{z,j}$, are minority partitions that have seen also the j view, and at least one of their nodes that has seen j fulfills $n_m \in \mathcal{V}_{x,j} \cap \mathcal{V}^*_{x+1,j}$.

Later, if in the first transition of type $T3$ to a new working view, $\mathcal{V}_{k,j+1}$ –recall that the last installed working view in the system was $\mathcal{V}_{x,j}$–, the new installed view fulfils the following:

$\mathcal{V}_{k,j+1} = A \cup B$ where:

- $A = \{n_l \in \mathcal{V}_{x,j} \setminus \mathcal{V}^*_{x+1,j} : n_l \notin \hat{\mathcal{V}}^*_{z,j} : x + 1 < z < k\}$, are the nodes that were alive in the last working view, but crashed –so they were not alive in any $\mathcal{V}^*_{x+1,j}$– triggering the view change that lead $N_j \rightarrow N^*_j$, and did not belong to any minority view that can recover the whole j view.
- $B = \{n_k \notin \mathcal{V}_{x,j} \cap \hat{\mathcal{V}}^*_{z,j} : x < z < k\}$, are the nodes that did not belong to the last working view, and that have not recovered the whole j view in any minority partition.

Then, the new reached majority is enabled to go on working. But, in this situation a problem can arise if the s term for A nodes fulfils that $s > 0$. This is because this new installed majority will be unable to reach the last consistent replicated state –the one reached after applying $M_{j,m}$–, due to the fact that:

- $\forall\, n_l \in A$ it is fulfilled that $M^F_{j,n_l} \neq \emptyset$
- $\forall\, n_s \in B$ it is either fulfilled that $M^D_{j,n_k} = M^A_{j,n_k} = \emptyset$ or $M^F_{j,n_l} \neq \emptyset$

So, the arising consistency problem conditions for non-transactional replicated system are:

- *Condition 1*: $T3$ transition $\rightarrow \mathcal{V}_{k,j+1}$
- *Condition 2*: $\mathcal{V}_{k,j+1} = A \cup B$
- *Condition 3*: $\forall\, n_l \in A$ it is fulfilled that $M^F_{j,n_l} \neq \emptyset$

The properties that must fulfil the replicated system to avoid this possible situation are similar to the ones proposed for solving the general amnesia phenomenon in subsection 5.4: in fact the *Prop. FS1* and *Prop. FS3* are necessary as they are defined in 5.4, while the *Prop. FS2* must be slightly modified to overcome this problem:

- *Property FS2**: each node $n_l \in A$ must maintain and provide a way for obtaining its M^F_{j,n_l} messages subset or associated updates, instead of trusting in "the replicated system".

It can be noticed that the modification is very similar to the one proposed for transactional systems.

6.6 Solutions

In this section different approaches for solving this problem in replicated systems are provided.

6.6.1 Transactional Systems

Persisting Messages

This solution is in fact the generic approach presented in Section 8.3. So, the idea consists in storing persistently the delivered messages in each replica as an atomic step of the delivery message, being only possible to delete them once they have been correctly processed in the replica.

Working in this way it is always ensured that $\forall T5$ transition triggered by node crashes –reaching $\mathcal{V}^{*}_{x+1,j}$– all $n_l \in A$ has persisted its T^{F}_{j,n_l}. Thus, when they reconnect and start their recovery process they can apply them. So, if in the first transition of type $T3$ –reaching $\mathcal{V}_{k,j+1}$– it is fulfilled that $\mathcal{V}_{k,j+1} = A \cup B$, then the A nodes in spite of having the $T^{F}_{j,n_l} \neq \emptyset$, they have permanently stored the messages associated T^{F}_{j,n_l}. Hence, they are able to reach the last consistent state of the replicated system, avoiding diverging state evolutions, when the amnesia problem is combined with a $T3$ transition.

Obviously, persisting messages as soon as they are delivered implies an overhead during the replication work. A study of this overhead cost is presented in Section 8.4. An overhead that will penalize constantly the replication work in order to avoid problems for situations that will rarely occur.

Mobile approach

Mobile approach. Another possible solution is to do nothing and assume these situations can happen. In this case, the idea is that among the alive nodes that compound the new primary partition –instead of not having the last consistent state– decide a new last consistent replicated state, allowing the system to go on working from this point, the $T_{j,m-s}$ with highest $m - s$ value of A nodes.

Later, when a replica which really reached the last consistent state of the replicated system reconnects, it must undo the transactions not processed in the new consistent replicated state before being recovered.

This solution avoids the overhead of persisting messages and simply implies to undo –in very rare occasions– some transactions –usually very few–. This solution is similar in concept to some approaches used in reconciling processes for partitionable systems [7].

Selecting Alternatives

Which solution must be adopted? It depends on the replicated system characteristics. The first solution solves the problem ensuring that committed transactions are not lost, but implies a constant overhead during the normal work for solving a problem that will rarely happen. While the second solution avoids the problem without implying any overhead, but some transactions must be undone when this improbable scenario happens. So, it depends on the replicated system tolerance to undo some already committed transactions. If this tolerance is critical it is necessary to select the first approach, while if there are not important problems of undoing some committed transactions, the second one can be adopted.

6.6.2 Non-Transactional Systems

Persisting Messages

For non-transactional systems it can also be used the approach based on persisting messages for transactional systems. This is in fact the solution presented in Section 11.3. Therefore, as for transactional systems, this solution consists in persisting messages atomically in the delivery process –in each replica–, and deleting them once they have been correctly applied in the replica.

This ensures that $\forall T5$ view transition triggered by node crashes –where the system reaches $\mathcal{V}^{*}_{x+1,j}$– all $n_l \in A$ have persisted their M^{F}_{j,n_l}. Therefore, when they perform their recovery process they can apply their respective forgotten messages. In this case, if in the first transition of type $T3$ –reaching $\mathcal{V}_{k,j+1}$ a working view– the system fulfils that $\mathcal{V}_{k,j+1} = A \cup B$, then the A nodes have persisted their respective M^{F}_{j,n_l} messages. Hence, they reach in the recovery process the last consistent state of the replicated system, avoiding then diverging state evolutions, when the amnesia problem is combined with a $T3$ transition.

Logically, the process of persisting messages atomically in their delivery process implies an overhead in the replication work. A study of this overhead cost is presented in Section 11.5. This overhead is really important because it penalizes constantly the replication work for overcoming problems that will rarely occur.

6.7 Related Work

Different semantics have been defined for facilitating the implementation of recovery protocols. On the sequel we will include considerations about how different proposals face this problem.

In [51] the authors surveyed rollback–recovery protocols in message passing systems for non-transactional distributed systems. Among the considered protocols there were the *Logging* protocols based on checkpointing and logging. In this case each node can recover its last achieved state setting its last performed checkpoint and reapplying the messages logged –at delivery time– from this checkpoint. Checkpoint and logged messages were stored persistently. Then as we can see these protocols avoid the problem presented in this chapter when talking about replicated systems as long as a node has access to all its received messages which have been logged. If instead of talking about transactions we translate *Prop. 1* and *Prop. 2* to message terms, the checkpoint ensures *Prop. 1* and the logged messages ensure *Prop. 2*.

The enriched view synchrony [9] (a.k.a. EVS) proposed for simplifying the programming of applications which will have long and complex reconfigurations can not avoid by itself this problem. The EVS decouples the “up-to-date nodes” idea from “members of the primary view” notion –because the second ones can

be outdated. And defines a hierarchy for grouping nodes: subview, subview set and view –or enriched view. And the EVS semantics imply the following. A living node always belongs to exactly one subview, one subview always belongs to exactly one subview set, and a subview set always belongs to exactly one view. Moreover, only primary subviews are enabled to work.

This grouping hierarchy proposed by the EVS is used in recovery protocols as follows. Consider a majority of nodes alive and belonging to the same subview –which implies that they belong to the same subview set and view and that they can work. When a crashed node becomes alive it has its own subview, subview set and view. As soon as the GCS triggers the corresponding view change this node maintains its own subview and subview set separated from the others, but all nodes become member of the new created view. Later, in order to start the recovery the outdated node merges its subview set with the subview set of the previous nodes, but remains in its own subview. Once the recovery completes, the recovered node merges its subview with the other nodes subview. At this point, it becomes a member of the primary subview so it can work.

But, all this behavior does not avoid the problem considered in this chapter. We can see it with the previous example –explained in Section 6.2 and depicted in figures 6.1, 6.2 and 6.3. Figure 6.4 shows the evolution of this example using the EVS semantics.

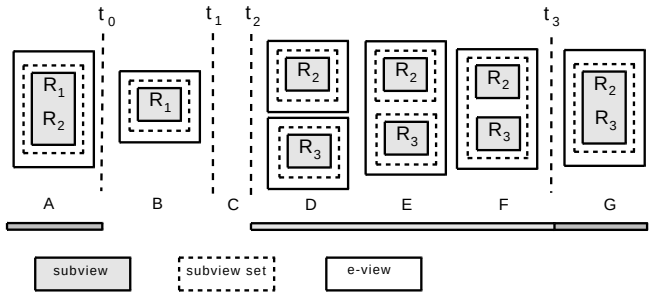


Figure 6.4: Progress Condition Problem

At the beginning when R_1 and R_2 are alive and working –assume view n– they belong to the same subview, subview set and view attending to EVS semantics –Figure 6.4 (A)–. When R_2 crashes the replicated system stops working because it does not satisfy the majority partition progress condition. Remember that R_2 crashes before being able to commit T_b. Thus:

- $T_n = \{..., T_a, T_b\}$
- $T^D_{n,R_1} = T^C_{n,R_1} = T_n$
- $T^D_{n,R_2} = T_n$, but $T^C_{n,R_2} = \{..., T_a\} \neq T_n$
- so $T^F_{n,R_2} = T_b$

At this moment, the replica R_1 remains alive –Figure 6.4 (B)– without working from a replication point of view until it fails.

Later, R_2 and R_3 become alive. Obviously, both replicas do not become alive at same instant, but it does not mind. At this moment each one has its own subview, subview set and view –Figure 6.4 (D)–. When the GCS realizes about it, it triggers a new view, becoming both replicas member of the same view but maintaining each one their own subview and subview set –Figure 6.4 (E)–. In order to start the R_3 recovery its subview set is merged with R_2 one –Figure 6.4 (F)–. This recovery ends when R_3 reaches the same state as R_2 –the state reached after applying T_a– and at this moment it joins to the same subview of R_2 –Figure 6.4 (G)–. Then, the replicated system has reached a primary subview being enabled to work according to the EVS specification. The problem is that this working view has as last state the one reached after applying T_a, and not the correct one that is the one reached after committing T_b, because the node who has to ensure the consistency among the view n and the new one, R_2 is unable to remember or obtain T_b –it has not committed this transaction before crashing and has lost the message when crashing–. Therefore, a state inconsistency will arise if they start working.

The problem of EVS is that neither avoids the amnesia phenomenon nor manages it accurately. In fact, it does not satisfy any property *Prop. 1* and *Prop. 2* as long as it does not say anything about how to manage the recovery information. Thus, it can not avoid the problem presented here. Only if EVS is combined with persisting the messages at each node as soon as they are delivered will be enabled to avoid this problem.

In regard to Virtual Synchrony it must be said that by itself can not avoid this problem. This derives from the fact that it only considers the message delivery and not the message processing, so the virtual synchrony alone is not enough for avoiding the presented problem.

It must be noticed that in [122], authors analyzed the basic phenomenon which underlies behind the amnesia problem. They proposed in such paper the concept of *successful delivery* that when correctly implemented, it overcomes both the amnesia generic problem and the amnesia issue with the progress condition presented in this chapter.

These problems are solved when successful delivery is combined with virtual synchrony, and more specifically the one based on the same view delivery semantic. In this case, if a node fails when processing the message, the system will enforce the delivery of this message in all nodes in the next view. The idea is that if one of the nodes of a view can not successful delivery a message the other nodes belonging to such view neither can, therefore the nodes remaining alive in the next view –if a crash has happened– will try to successfully deliver it in this new view. Therefore, the basic amnesia phenomenon is avoided, forcing to successfully deliver in the subsequent view those which can not be successfully delivered in the previous one. In this case, this delivery semantic always ensure that $T^F_{j,n_l} = \emptyset$, so the problem presented in this chapter is avoided.

Assuming the example of Section 6.2, as R_2 is not able to process correctly T_b due to its crash, the successful delivery semantic will force to quit T_b from T_n^D and T_n^C. That will imply removing T_b from: T_{n,R_1}^C, T_{n,R_1}^D, and T_{n,R_2}^D. And try to successfully deliver T_b in the next view if possible. Therefore, in this case would be fulfilled:

- $T_n = \{..., T_a\}$
- $T_{n,R_1}^D = T_{n,R_1}^C = T_n$
- $T_{n,R_2}^D = T_{n,R_2}^C = T_n$
- so $T_{n,R_2}^F = \emptyset$

In [109], authors proposed a new variant of atomic broadcast algorithm –based on consensus– for the crash-recovery failure model. They propose as basic recovery mechanism to persist messages at *propose* time in the consensus work, persisting other important consensus information when it concluded at replica level in order to avoid problems at recovery time. Thus, later at recovery time the recovering replica simply must to replay consensus rounds reconstructing the queue of agreed messages but to do so it needs all other participants alive, situation that in the presented problem is not fulfilled. So, this solution as it is basically defined does not avoid the problem. Moreover, authors also proposed an optimized recovery solution based on persisting the agreed messages queue –combining it with some checkpointing policy. But, as they do not force to do it atomically with the delivery process therefore the problem persisted, due to the possibility that the replica crashed before persisting the delivered messages.

The solution proposed in [91] does not avoid this problem due to the fact that it relies in other replicas to replay consensus rounds.

Anyway, in [109, 122] authors do not formalize the amnesia phenomenon and do not study the associated problem when combined with the majority progress condition.

6.8 Conclusions

This chapter shows how combining the amnesia phenomenon, which arises when replicated systems assume the *crash recovery with partial amnesia* failure model, with a particular scenario allowed for the most commonly used progress condition –majority partition– in replicated systems can lead to diverging replicated state evolutions. Later, it has been formalized, proposing the properties that must be ensured in order to overcome these undesired situations. Subsequently, it has been proposed two different approaches for solving this problem, being interesting each one for different replication scenarios.

This phenomenon in spite of being very rare can cause catastrophic consequences in consistency concerned replicated systems, so in these systems it must be accurately managed.

Chapter 7

"$\frac{n}{2}+1$ *alive nodes*" Progress Condition

From the knowledge obtained in the previous chapter it can be derived a new majority progress condition. Traditionally, the condition established to allow a replicated system to go on working from a recovery point of view is based on the existence of $\frac{n}{2}+1$ fully up-to-date nodes. In this chapter, it is formalized the problem, in order to prove a mechanism for relaxing this condition, at the time it is also kept the guarantee, for any possible primary partition, of providing a correct evolution of the replicated system. There are also discussed the drawbacks and advantages related to this relaxed progress condition.

7.1 Introduction

Apart from maintaining the original *k-fault* tolerant level of replicated systems, another important aspect in these *systems* is to guarantee the replicated state correctness. To this end, two basic approaches can be adopted: *primary partition systems* where only the *majority partition* is allowed to work and *partitionable systems* where multiple partitions may work concurrently. The latter implies the use of costly quorums [12] for merging partition states, which may not always be feasible without breaking correctness. Therefore, *primary partition* is widely accepted in order to guarantee correctness in the replicated state, as well as the capability, for any reconnected (thus potentially outdated) node, of being always in disposition to be updated to continue working.

This principle is based on the idea of a *progress condition*. Thus, considering that the distributed system knows the pre-configured set of possible nodes $D = \{r_1, ..., r_n\}$, the condition of *primary partition* is granted when the interconnected alive nodes conform a subset of D that satisfies the *progress condition*.

Thus, the progress condition enables a partition $P \subseteq S$ (i.e. a subset of the distributed system) to go on working with guarantees.

Traditionally, replicated systems have adopted the concept of *majority partition*, meaning that $\frac{n}{2}+1$ nodes of the system must be alive and fully up-to-date in order to enable the partition to work. More precisely, this statement implies that the notion of "up-to-date member" is equivalent to "member of the *primary view*". In the scope of recovery systems for database replicated systems based on broadcast primitives, the *Enriched View Synchrony* was presented in [9], decoupling the concept of "up-to-date member" from the "member of the primary view" one. This distinction drives the authors to separate the working view concept from the majority partition one. The work also demonstrates how this new approach models more accurately the evolution state in node replicas, and how it fits better for applications where the recoveries are long and complex.

In addition, during the process run for recovering a newly reconnected node on a primary partition, several problems arise in order to guarantee the correctness of the system. Because of these problems, many recovery protocols introduce several restrictions on the set of nodes allowed to work during that process. The most advanced protocols disallow for working just the recovering node, whilst other approaches prohibit the activity in the entire partition during part of -or even all- the process run to recover nodes.

A direct implication of these two characteristics is that the aforementioned condition introduces a drawback in the availability of the system, since the nature of the update process of reconnected nodes makes it expensive, at the time that the $\frac{n}{2}+1$ *updated* condition requires an immediate update of the nodes in order to proceed in partitions with just $\frac{n}{2}+1$ composing nodes. Consequently, it results extremely convenient to find a mechanism for relaxing the progress condition, thus allowing to proceed partitions with just $\frac{n}{2}+1$ *alive nodes*, which will lead to a higher availability, also enabling background recovery processes.

This chapter formalizes a new and more relaxed progress condition for replicated systems. Its viability and correctness in recovery terms are also proved to be guaranteed by a proposed general recovery information strategy, ensuring that primary partitions will always be able to go on working. This proposal is presented in the scope of a middleware recovery protocol intended to provide fault tolerance for replicated systems based on linear interaction approach [124], although the formalized principles are also applicable to replication protocols based on constant interaction. The only assumption made by our model is that the information gathered during the failure of nodes can be structured as an activity log (i.e. log-based gathering). This recovery protocol supports different recovery strategies with the idea to minimize the effort and cost of the recovery process, without stopping the replicated system work for *primary partitions*. It is also intended to perform partial recoveries, when needed. Finally, as our design is performed as a middleware recovery system, it can be easily applied to different transactional scenarios, in addition to database replicated systems. In fact, this recovery system is highly oriented to replicated processes

which manage large amounts of state, not being such interesting for replicating small objects, where usually transferring the whole state to outdated nodes will present better performance.

This chapter is structured as follows: the information model is described in Section 7.2. In Section 7.3 the most restrictive progress condition and the relaxed proposal are presented and formalized. It also includes a comparison, as well as the implications of the discussion for building an adequate recovery protocol. Subsequently, the related work is included in section 7.4. Finally, Section 7.5 summarizes the chapter.

7.2 Recovery Information Model

In this section, it is presented a model that allows us to represent in a simple way the information maintained at each node. It will be used afterwards in this work to demonstrate the $\frac{n}{2}+1$ *alive nodes* progress condition correctness.

This model considers for each node member of the replicated system its committed information as well as the recovery information maintained in this node. As the replicated system is intended to work using atomic broadcast primitives, this recovery information is abstracted as an activity log. This recovery log information contains messages belonging to on-going[1] and committed transactions which obviously were not received by a failed node or not yet applied by the node. Aborted or rolledback transactions are immediately deleted from the log.

Since the GCS notifies nodes about each view (understood as a single change in the group composition), this log-recovery relates to each one of the missed views. The view where the recovering node is being reconnected is also included in a recovery log, since the activity of the current view cannot be applied to the node until it is fully up to date. Thus, the recovery of a node is split up into two parts: the log-recovery of previous views and the log-recovery of the current one.

The model to represent the information available in a node is summarized in $S = (B, R, A)$, where B represents the data state committed, R is the log-recovery of previous views, and A is the log-recovery of the current view.

Afterwards, the view concept is introduced in the model, thus the model can be used to represent the data state contained in a node in view terms, relating it to the replicated system *view history*. Therefore, the information available in the view j by a node that has not failed is:

$$S^j = (B^{j-1}, R^{j-1}, A^j)$$

[1]Notice that on-going transactions only exist if the replication protocol uses linear interaction.

B^{j-1}, is the committed state when the view v_j started, R^{j-1} is the log recovery generated until the view $j-1$ including itself, and A^j is the log recovery generated during the view j. At the beginning of v_j, A^j is empty and when the system installs the view v_{j+1} it becomes part of R^j.

This model derives on the following characteristics:

- B and R do not exist at S^0,
- $B^{j+1} = B^j + (R^{j-1} + A^j)$ committed messages during view v_j,
- $R^j = R^{j-1} + A^j$, if this expression is iterated, at the end $R^j = A^0 + ... + A^j$

It must be remarked that the log-recovery contents, modelled as R and A, vary as long as transactions are committed and aborted or failed nodes reconnect to the system and are further recovered.

7.3 Progress Conditions

In this section, there are discussed either the most extended, more restrictive progress condition, and the relaxed one proposed in this chapter. It is also proved the correctness of the relaxed condition, as well as the recovery information needed to guarantee its correctness. Finally, both conditions are compared.

7.3.1 $\frac{n}{2}+1$ Updated Nodes

This condition is the commonly used one for recovery purposes. It guarantees that a majority partition will always contain at least one full up-to-date node. This node can be used then to recover the outdated nodes present in this new view. Thus, it will always be possible for a majority partition to fully update each one of the outdated nodes. Then, once this view reaches $\frac{n}{2}+1$ updated nodes (i.e. the progress condition) it can start to work.

Its correctness demonstration is trivial. Assume that a new majority partition does not have any fully up-to-date node. If this scenario is reached, it implies that the last installed working view[2] contained less than $\frac{n}{2}+1$ updated nodes. This situation, obviously is prohibited under this progress condition. Consequently, the $\frac{n}{2}+1$ updated nodes guarantee the capability to progress of any majority view.

The main advantage of this progress condition relies in the fact that the recovery information is always guaranteed to exist in a majority partition, since it is ensured by construction that at least one node was present and fully updated in the last working view.

[2]Recall that only majority partitions install new views.

Although this condition provides correctness and liveness guarantees, it presents important drawbacks. The most relevant arises on minimal majority partitions, with just $\frac{n}{2}+1$ alive nodes and where just one of them is fully up-to-date. Those partitions will be disallowed to work until every node ($\frac{n}{2}+1$) is updated. Since the recovery process of such nodes is costly, the inactivity time becomes inconveniently long.

7.3.2 $\frac{n}{2}+1$ Alive Nodes

To mitigate the inconveniences introduced by the mentioned progress condition it becomes necessary, as it will be shown in this section, to combine the condition with a recovery log storing policy which persistently stores each broadcast message as soon as it is delivered by the GCS.

The adoption of this progress condition guarantees that in each installed and working view there are always at least $\frac{n}{2}+1$ alive replicas. Thus, it is ensured that all these replicas will generate the A term corresponding to the installed view, even if they are not up-to-date. So, the A term means a "seen view", even if such view has not been applied in the node. Therefore, the system has, for each *installed* view, at least $\frac{n}{2}+1$ nodes that have generated the associated view recovery information, if needed[3].

Consequently, this system behavior guarantees that, in any possible *installed* view (over a majority partition) the recovery system can lead its replica members to the last reached data state, even if no one of them is fully up-to-date[4]. This is possible because this condition ensures that, the cooperation of all alive nodes enables the recovery system to reconstruct the entire activity of the replicated system, and thus, to recover any outdated nodes. In fact, some of the nodes of a recovering view (at least one) have been alive in each one of the installed views. In terms of the information model, they have the sequence of log-recoveries, $\{A^c, ..., A^i\}$ (although the sequence may be spread over different nodes) that allow the recovery system to update them.

To demonstrate this assertion, assume that the last data state reached in the replicated system is S^n, which can be expressed as $S^n = \{B^{n-1}, R^{n-1}, A^n\}$. Also assume that, in each installed view, it has always existed at least a failed node. Then, S^n can be reconstructed applying the messages maintained in $\{R^{n-1}, A^n\}$. And this information can be expressed in A terms as it is shown in figure 7.1. Therefore, as the progress condition guarantees that each A term is always seen by at least $\frac{n}{2}+1$ alive nodes, it is easy to see that, spread over the alive nodes in any possible majority partition, there are the A terms for any missed view, and they are able to reconstruct the last data state.

This condition differs from the "$\frac{n}{2}+1$ Updated Nodes" one in the sense that

[3]It is not needed if there is no failed node.

[4]By definition, at least one of these nodes was in the last installed view but it is possible that it did not apply this view because it was outdated.

Taking as starting point:
$S^n = \{R^{n-1} + A^n\}$
where $R^{n-1} = \{R^{n-2} + A^{n-1}\}$
Then, if R^{n-1} is substituted
$S^n = \{R^{n-2} + A^{n-1} + A^n\}$
Finally, if this step is performed until the first view
the S^n can be expressed as
$S^n = \{A^0 + ... + A^{n-1} + A^n\}$

Figure 7.1: Reconstruction State.

it allows to install majority views in a majority partition (and consequently, to work) as soon as it exists in the group one fully up-to-date node[5]. Therefore, the system does not need to wait for a recovery process that leads the system to a $\frac{n}{2}+1$ updated nodes situation. Obviously, if when a majority partition is reached and it does not contain any fully up-to-date node, it must wait the completion of the recovery process of at least one node in order to install the new view, and consequently, start working.

With this condition the recovery process requires more information, but it usually permits the system to work sooner when it reaches a majority partition from a minority one. Thus, every node contained in a working partition will store (if there are failed nodes) the activity seen in such partition at any time. This activity, modelled as a log, will be potentially used by the recovery process on node reconnection.

7.3.3 Comparison

The biggest difference among these two progress conditions appears in system transitions when a majority partition is reached. Therefore, the "$\frac{n}{2}+1$ Updated Nodes" one will not let the system work until $\frac{n}{2}+1$ replica nodes are fully updated, whilst the second one lets the system work as soon as one of its members is fully up-to-date. So, the latter is a less restrictive condition, reducing the unavailability time of a replicated system in these transitions.

However, the second one implies a more complex recovery process, that also needs a costly storing policy which forces the system to persist any broadcast message. However, such storing policy is highly convenient in truly fault-tolerant systems in order to bypass the amnesia problem.

The adoption of the progress condition presented in this work will be useful for systems that need to provide their service as soon as possible and are frequently subject to the *minority-to-majority transition*.

[5]Recall that only up-to-date nodes can start transactions.

7.4 Related Work

The use of majority partition progress condition derives from the quorum voting replication strategies [59] used in replicated systems. In fact, this is the simplest version of quorum voting strategies.

At the beggining these quorum voting strategies where combined with atomic commit protocols in order to ensure consistency and progress. The most basic commit protocol, the two phase commit [61] –2PC–, had the problem of blocking if a replica –holding some locks– crashed. In order to avoid this blocking problem they were presented the three phase commit protocols [116] –3PC. 3PC protocols in spite of not blocking presented the unilateral abort problem. This problem leaded the 3PC to abort transactions even if a single replica server crashes. For this reason, authors of [66] presented a 3PC variant which commits transactions as long as a majority of replicas are up, avoiding the unilateral abort problem.

It must be said that this last protocol [66] can work with a majority of replicas up and not a majority of updated replicas because it includes internally in its commit work the version number that for every data item being accessed has each replica server participating. Compared to the proposed condition in this chapter it presents several drawbacks. Its main problem is that it must be combined with a 3PC protocol, with the cost that it implies instead of using atomic broadcast. Also, it is unable to manage the amnesia phenomenon correctly being possible to arise the different problems related to it if the system adopts the crash recovery with partial amnesia failure model. And finally, it forces to access also a majority of replica servers for read operations while in the proposed condition in this chapter accessing only one is enough.

7.5 Conclusions

Inspired by the decoupling of “primary view” from “primary partition”, this chapter proposes and demonstrates a new progress condition combined with a specific storing recovery information policy. The advantages and drawbacks of both progress conditions are also compared.

This work details in first place an information model to represent the information maintained in a replica node, in the context of replicated transactional systems which use a log-based recovery strategy as [19]. The model represents on one hand the committed state, and on the other hand the log-recovery information maintained at each replica node. Moreover, there are detailed the three basic rules fulfilled by this recovery information model.

In second stage, it has firstly shown, using the proposed information model, the correctness of the traditional and widely assumed progress condition for replicated systems, the $\frac{n}{2}+1$ *updated nodes*. Then, a new progress condition, the $\frac{n}{2}+1$ *alive nodes*, has been presented and its correctness has been also

demonstrated. Moreover, it has been shown that its correctness is subordinated to the persistent storage of each broadcast message when failed nodes exist.

Finally, both progress conditions have been compared, and it is shown how, for the transitions from a minority partition to a majority one, the condition proposed here is more convenient than the traditional one. The first one allows to work a majority partition as soon as one of its members is up-to-date. Contrarily, the traditional one only allows the majority partition to work when $\frac{n}{2}+1$ are up-to-date, which is often subject to costly recovery processes. The drawback of our proposed condition is the necessity of persistently storing any broadcast message when failed nodes exist. It also has been shown how this storage becomes necessary in any case in realistic systems, where the crash-recovery with partial amnesia failure model is adopted.

Therefore, the adoption of one or another approach will depend on the necessity of high availability even for the transitions from a non-majority partition to a majority one.

Chapter 8

Transactional Amnesia Support

After having described and formalized the amnesia phenomenon problem either for transactional replicated systems or for non-transactional replicated systems in the previous chapters, it is time to provide ways for managing it accurately and therefore overcoming the problems associated when it arises.

8.1 Introduction

In this chapter, it will be provided a generic way for handling accurately the amnesia phenomenon in transactional replicated systems. The idea is to provide a recovery technique that fulfills the properties presented in Section 5.3 and that can be combined with any existing recovery technique without disturbing its basic work way.

As it has been said transactional replicated systems can work either in a constant interaction or in a linear interaction way. This thesis assumes that replication protocols broadcast writesets and readsets when working in a constant interaction way, while they spread operations under the linear interaction approach. Evidently, there are more possibilities but these are not considered in this thesis due to space constraints.

Depending on the recovery policy used, log-based or version-based, the information that must be maintained to solve the amnesia problem differs. Thus, extra information is needed to be maintained to this particular end, being afterwards used in the amnesia recovery process.

This chapter is structured as follows: Section 8.2 outlines the wrapping recovery protocol. The next section, Section 8.3 provides ways to avoid amnesia

problems at recovery time using a log-based recovery policy. Later, in Section 8.4 this chapter studies how different replication configurations behave when the *amnesia phenomenon* arises, taking as starting point the basic generic solution presented for a basic replication configuration. On the sequel, Section 8.5 analyzes the overhead introduced when applying this solution in different replication configurations. Section 8.6 includes some related work, and Section 8.7 concludes the chapter.

8.2 Recovery Protocol

In the basic considered replicated systems each transaction is only processed in one replica –denoted as *delegate server* by [123]–, the replica that serves the client request, and only its associated updates are broadcast among all alive replicas. In this scenario, log-based recovery protocols use as recovery information the broadcast replication messages missed by crashed nodes. So, the solutions provided for log-based protocols must maintain these messages as long as they are not applied by all replicas, as it has been commented in Section 5.6.1.

The recovery protocol has the logic stages presented in Section 5.5. Figure 8.1 concretes this recovery process for transactional systems based on constant interaction.

Recovery Process:
- 1 - Amnesia Recovery Process:
  - if the *outdated node* is the (re)connected node it performs the *ARP*
- 2 - Update Recovery Process:
  - the *outdated node* for each non-applied view v_i:
    - if the *outdated node* does not have the log-recovery view information for v_i:
      - it demands the log-recovery information for v_i to the recoverer node (v_i)
    - the *outdated node* updates view v_i
    - notify the alive nodes that v_i has been recovered in this node
- 3 - Current Recovery Process:
  - if the replicated system was working during the node recovery (all lost views have been already applied):
    - the outdated node reapplies the messages delivered after reconnecting

Figure 8.1: Recovery Algorithm.

The first step, the ARP, is used by (re)connected nodes to retrieve the right state they had at their crash time. In this process, the outdated node must

apply the messages received and not yet committed before its failure. A deeper discussion of this recovery step is done in Section 8.3.

The *URP* stage is performed by every outdated node. Each outdated node starts the recovery of its missed views by selecting its *recoverer node* for such views (in a primary partition, a single node can be used for every view[1]). Since views are sequentially numbered, the first view to be recovered is the next one to its last fully applied view, and finishes when it reaches the last applied view in the *recoverer node*[2]. As this recovery is performed view by view, it is possible to request to the *recoverer node* just the log-recovery information for the lost views. Once the outdated node has this information (i.e. missed messages), it applies them to recover this view. The application of these missed messages must be performed in the same order they were delivered in the replication system total order. When the outdated node has finished the recovery of this view it must notify all alive nodes that it has recovered this view, therefore they may discard the available recovery information for such view.

Every time a membership change occurs it must be checked how it affects to each recovery process started. Obviously, if the outdated node crashes the recovery process ends. If the *recoverer node* has failed the outdated node must look for a new one for going on with the recovery process.

The third step, *CRP*, is performed by outdated nodes recovered as long as the system is working (i.e. if the (re)connection was into a primary partition). In this scenario, the system generated activity during the recovery process, but the recovering node delayed the application of such activity (it just persisted it in a queue as part of a "seen view"). Thus, once all the non-previously-applied views are applied in the recovering node, it must conclude its recovery by applying all these delayed messages in their delivery order. If a new working view is installed during the recovery of a node, a new queue is created for the new view[3].

After outlining the recovery protocol for transactional systems based on constant interaction it is time to determine how to handle the amnesia phenomenon.

8.3 Amnesia Recovery Support

The amnesia recovery in an outdated node has to be done before recovering the missed messages lost during its failure period, as it is explained in Section 5.5 and Section 8.2 in order to guarantee the state consistency.

In Section 5.2 it has been detailed that the amnesia can be manifested at two different levels: transport and replication. Therefore, the log-based recovery

[1] This approach allows *partial recoveries* if the outdated node is in a system which does not fulfil the progress condition.

[2] If the system is in a working view the previous view to the current one is considered the last one applied in the *recoverer node*.

[3] Notice that in this case the outdated node has already the information of the previous view.

protocols must manage the information necessary for performing the recovery in an adequate way for both levels.

8.3.1 Transport Level

The amnesia at the transport level implies that received messages non-persistently stored are lost when the node crashes. If this occurs, the amnesia recovery could not be performed using a log-based approach. So the system must ensure these messages are available for recovery purposes by storing them in a persistent way.

The simplest, easiest option is that each node manages its own amnesia recovery information persistently. This storage of received messages must be kept until the respective owner transaction is either committed or aborted[4]. Moreover, if the messages must be maintained after its transaction commit, they must be marked in some way to remember that they have been already applied.

In addition, the permanent storage of a received message must be performed atomically with the group communications system message delivery. Under this principle, if the node is not able to persistently store the message, the GCS does not consider this message as delivered (thus ensuring that the delivery is coupled with the persistent storage). With these conditions *Property FS2* is always fulfilled.

Another approach, that it can be considered, will consist in storing persistently messages in their replica senders, but this option presents several drawbacks. One of them implies that senders must maintain for each sent message a list of replicas that have acknowledged the commit of this message, being only possible to delete the message when all replicas have acknowledged it. Other important drawback is that the recovery process can only be performed when all senders are alive –the only one scenario that can ensure that *Property FS2* is totally fulfilled–, a condition that can delay a lot of time the recovery process, situation highly discouraged. In spite of having several problems which discard its use in many transactional replicated systems, there are some replicated systems which their work way fits well with this solution. This is the case of hot passive on-processing [34] or voting replication systems [124].

Therefore, the original way presented for solving the amnesia at transport level is selected as the best option for overcoming it.

It must be noticed that the support at transport level would not be necessary if the replication system makes use of what is called *successful delivery* presented in [122]. This is due to the fact, that this semantic avoids the misconception that a delivered message is a really applied message.

[4] i.e. discarding them on transaction aborts, or maintaining them for committed transactions *only* when failed nodes exist.

8.3.2 Replica Level

The amnesia problem relates at the replication level to the fact 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. Thus, the information about the success of such commit must be also stored because it is needed by the recovery amnesia process in order to know which are the messages that must be applied (as discussed previously). So, at the replicated system level, the problem is to know if a "commit" was successfully applied before the failure or not. At this level the ideas is to fulfill the *Property FS1*.

The mechanism for generating this information consists in maintaining some information about the last committed transaction for each open connection. Thus, when a transaction commit is performed in the replica, the system must write this information in a single atomic step, as part of the transaction itself. Thus, on commit success, the system contains the identity of this last committed transaction. Afterwards, when the connection is closed with all its transactions terminated, the entry corresponding to this connection is erased.

Then this generated information is useful in the recovery process to check if messages marked as not committed have been really committed. When the node becomes alive again and starts its amnesia recovery it will check if there are messages marked as not committed, but its owner transaction is marked as committed in the replica.

A similar problem arises regarding the state associated to not committed messages (messages belonging to not yet committed transactions), since it is lost at the crash instant, since the replication system is also a transactional system. Therefore, these messages applied by the replication system but not committed must be again reapplied.

There are two possible scenarios where messages maintained as non-committed belong to a transaction whose owner connection does not have an entry in the table of committed transactions:

- *The node did not start to apply this connection and its transactions before the node crash.* Then, these messages must be applied in the amnesia recovery process.
- *The connection is closed before committing some of its related transactions (implying that these transactions will be aborted) but the node crashes before the system erases those messages.* Thus, all the messages will be reapplied in the amnesia recovery process but they will be aborted again as it has happened in the normal work way.
- *The transaction was really committed, the system has not marked yet its messages as committed, and it has already deleted its table connection*

entry. This scenario must be avoided, because it will lead the system to apply twice a transaction if a recovery process is performed. So, when the "remove connection" message is applied, the removal of the corresponding entry in the system must be done after the protocol considers committed the transaction.

As a result, the amnesia recovery stage will just consist in reapplying the messages marked in the log recovery as "not applied" or "not committed", first checking against the replica if they were not really committed.

It must be noticed that database management systems used for building replicated databases –one of the most common transactional replicated systems– provide ways for recovering internally their state.

8.4 Replicated Systems Characteristics and Amnesia

Once in the previous section has been described how the amnesia problem can be solved in a transactional replicated system using a log-based technique, this section will describe how the replication characteristics affect the way in which the system must manage the amnesia problem. Table 8.1 shows the characteristics being considered:

Characteristics Set
Active, Update Everywhere or Passive Replication
Eager or Lazy Replication
Voting or non-Voting Technique
Constant or Linear Interaction
Communication Guarantees

Table 8.1: Characteristics Set.

The following subsections detail the effects on the amnesia support.

8.4.1 Active, Update Everywhere or Passive Replication

The adoption of an active [123] (AR), update everywhere [124] (UER) or passive [123] (PR) replication policy will not have any effect in how the replicated system must manage the amnesia problem. It is due to the fact that the amnesia phenomenon does not depend on how many replicas can serve client requests, but on how the changes are spread and stored at each replica.

In figures 8.2 and 8.3 it can be seen two different variants of passive replication, *hot passive asynchronous* and *hot passive on processing* respectively, that will be used later in the overhead analysis in Section 8.5.

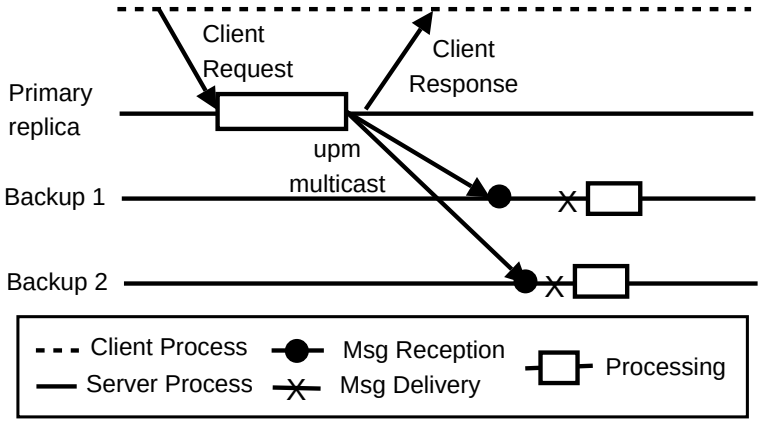


Figure 8.2: PassiveReplication (Hot Passive Asynchronous)

8.4.2 Eager or Lazy Replication

The selection between eager (*ER*) or lazy (*LR*) replication [62] has a great effect in the amnesia phenomenon and its management.

In fact there are some configurations in the eager approach that are managing indirectly the amnesia phenomenon. This is the case of eager passive replication systems, also known as hot passive replication in [120], based on on-processing synchronization [34]. These systems block the client answer until the primary receives the process acknowledgement from all secondary replicas. Therefore, the replicated system work way controls which slaves have committed which transactions, providing the necessary information for managing the amnesia problem at the replication level. The other eager passive replication techniques, on-delivery and on-reception synchrony [34] will not provide this intrinsic amnesia support, being necessary in these cases to adopt our amnesia support proposal.

Another case is eager update everywhere replication protocols as [81] which commit a transaction once all active replicas have locally committed it. This work way also provides the needed information to control the amnesia phenomenon at the replication level.

But not all eager solutions provide this intrinsic amnesia replication level support, only those whose transaction replication work includes the processing in all alive replicas into the transaction boundaries. Eager solutions that do not provide this behavior must adopt a system similar to our proposal either for replication and transport level.

This is the case of active replication protocols, which always work in an eager way. When these active replication protocols are based on total order broadcast they rely on a fully deterministic transactions execution as it is commented in

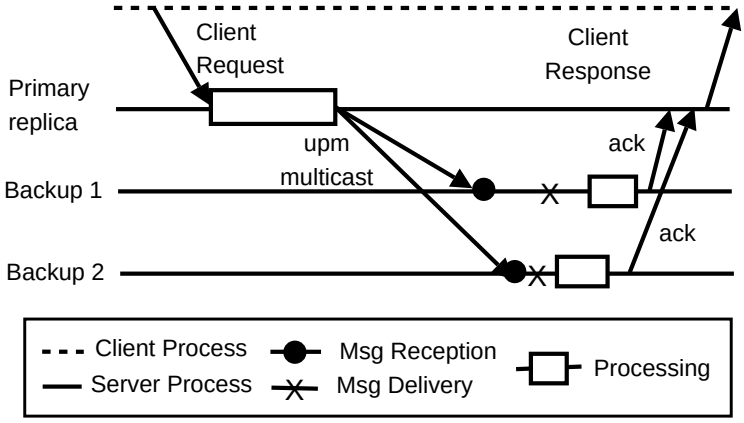


Figure 8.3: Passive Replication (Hot Passive on Processing)

[123]. Therefore, they do not provide a support for amnesia, making necessary to add an extra support.

The lazy replication approaches need always to adopt our amnesia support proposals or similar ones in order to support our assumed failure model. In these solutions, their work way does not provide the information needed for providing the amnesia support.

Anyway, it must be noticed that from a recovery point of view using a lazy replication technique will have a disruptive behavior in regard to the replicated consistency state because it will not be guaranteed at any time that all the information needed to reach the last consistent state in the system is available. Therefore, it is not ensured the success of these recovery processes. In fact, a priory some lazy replication protocols have had to implement some eager replication level to provide fault tolerance as it does [76].

Only lazy hot passive replicated systems, based on asynchronous and reliable communication, which propagate changes before answering to the client can provide amnesia support using our amnesia support proposal, and ensure the replicated consistency correctness. But in order to provide this support, the client must remember which requests have not been already answered.

8.4.3 Voting or non-Voting technique

The use of a replicated system based on a voting termination technique (VT) [124] can provide to the recovery protocol information if a node has not committed a transaction before its failure, because the system needs to know the decisions adopted in all nodes. Thus, its use provides information that is necessary for overcoming the amnesia problem implying a partial implementation of

our proposal. Whereas if a non-voting technique (NVT) is adopted the replicated system will have to adopt the whole system proposed in this work.

8.4.4 Constant or Linear Interaction

The use of constant (CI) or linear (LI) interaction [124] in the replication protocols will not have great effect on the amnesia level support.

At this point it only must be remarked that the adoption of a linear replication approach will difficult the transaction messages management because the messages belonging to a transaction can be spread among several views, and they can also be interleaved with the messages owned by other transactions, forcing to persist messages belonging to non yet committed transactions with our approach. Obviously this problem only appears when the system adopts a log-based recovery strategy as it is commented in [36].

8.4.5 Communication Guarantees

Until so far, it has been considered that all replicated systems use a total order broadcast (TOB) for propagating changes[5], with virtual synchrony [14, 24]. Therefore it is ensured that all replicas have delivered the same sequence of messages before a membership change occurs.

But what does it happen when these communication guarantees are relaxed? Will it affect the amnesia problem and the way it must be managed? In the following paragraphs it is evaluated.

Communication primitives which provide reliable communication are considered but which allow different delivery orders in different replicas, as it occurs when causal or FIFO (R-$FIFO$) delivery orders are used. The problem of these communication configurations is that they do not guarantee, in most cases, a consistent replicated state, allowing diverging state evolutions in different replicas, situation that can not be accepted. Only the intrinsic characteristics of some replicated systems allow to use more relaxed delivery orders without damaging the replicated consistent state. This is the case of PR (also known in the literature as primary copy systems) where R-$FIFO$ is enough to guarantee the replicated state consistency, because in these systems there is only one sender. These relaxed orders can also be applied in AR or UER if they use *commutative updates* [62]. In these cases, where relaxing the total order delivery does not affect badly the replicated consistent state correctness our amnesia support proposal can be applied.

Another possible communications relaxation should be the use of non-reliable communications, but under these conditions it is impossible to build any kind of fault tolerant system, because there is no guarantee about which messages

[5] In passive replicated systems a FIFO order is enough.

have been received by which replicas. Thus, these systems must use a protocol which provides reliability. Therefore, the study of non-reliable communication primitives must not be considered.

8.5 Overhead

After detailing how the different replication characteristics can affect the amnesia support in a transactional replicated system, the overhead introduced for providing amnesia support is analyzed, detailing first the basic processing time (BPT) and later the processing time supporting amnesia ($PTSA$). Both processing times will be expressed for each propagated transaction in terms of spread time (st), persisting time (pt) and processing time (Pt).

The st depends on: the communication guarantees provided, amount of messages to spread for transaction and the message size. In regard to the communications, two possibilities are considered: *TOB* for *UER* and *R-FIFO* for *PR*. For *TOB*, the *fixed sequencer* is assumed in the broadcast-broadcast (BB) variant [44], which uses two reliable broadcasts for message propagation. On the other hand, the *R-FIFO* as it is considered for *PR* can be implemented using only one reliable broadcast for message propagation because there is only one sender. Then, if α is the cost of a reliable broadcast, the *TOB* has a cost of 2α for spreading a message while the *R-FIFO* cost is α.

For *CI* replication protocols, it is considered that only one message is broadcast per transaction, while *LI* protocols spread as messages as update operations contains the transaction, n. In this context, message size (S) is important because some *GCS* have a maximum message size bound to spread, S_M. Thus, messages greater than S_M must be spread sending $\lceil \frac{S}{S_M} \rceil$ messages. That can occur in *CI* protocols because they transfer the resulting transaction writeset, while messages in *LI* protocols are always smaller because they only transfer one update operation.

The message persisting time is expressed as $\beta + \gamma(\lceil S/k \rceil)$, where β is the upper bound time for write disk accesses, γ is the storing time of a block size message k, and S is the message size. It is considered that only one write disk access is needed per message. Then, the pt in *CI* protocols where only one message is broadcast is $\beta + \gamma(\lceil S/k \rceil)$, while in *LI* protocols is $n(\beta + \gamma)$, being n the number of update operations and assuming that their message size is always smaller than k.

For Pt, the π value is considered which depends on the replication system load and on the consistency provided by the replication protocol. Moreover, in *CI* protocols it will depend on the message size to apply (writeset) while in *LI* protocols will depend on the number of update operations and their complexity. Thus π is very difficult to model, anyway it is considered that any transaction always fulfills the rule $Pt > pt$.

Configuration	*BPT*	*PTSA*
C1 - UER, ER, CI, NVT and TOB	$(\lceil \frac{S}{S_M} \rceil + 1)\alpha$	$(\lceil \frac{S}{S_M} \rceil + 1)\alpha + \beta + \gamma \lceil \frac{S}{k} \rceil$
C2 - UER, ER, LI, NVT and TOB	$2n\alpha$	$n(2\alpha + \beta + \gamma)$
C3 - UER, ER, CI, VT and TOB	$(\lceil \frac{S}{S_M} \rceil + 2)\alpha + \pi$	$(\lceil \frac{S}{S_M} \rceil + 2)\alpha + \pi$
C4 - PR (hot passive on-processing), ER, CI, and R-FIFO	$(\lceil \frac{S}{S_M} \rceil + 1)\alpha + \pi$	$(\lceil \frac{S}{S_M} \rceil + 1)\alpha + \pi$
C5 - PR (hot passive asynchronous), ER, CI, and R-FIFO	$\lceil \frac{S}{S_M} \rceil \alpha$	$\lceil \frac{S}{S_M} \rceil \alpha + \beta + \gamma \lceil \frac{S}{k} \rceil$

Table 8.2: Processing Times.

Several observations must be made for table 8.2. The first one is that replication configurations which perform the transaction processing as core part of their propagation process, as C3 and C4, discard pt due to the rule $Pt > pt$. Therefore, π hides the $\beta + \gamma(\lceil S/k \rceil)$ value. Another consideration is that all configurations perform the message persisting process as an atomic step in the delivery process.

From table 8.2 can be concluded that *VT* configurations will not have any overhead for supporting amnesia, but from the beginning they have a BPT higher than the others, while the others present an overhead which depends on the persisting time, pt. Overhead which increases with the message size, S, in *CI* and the number of messages, n, in *LI*.

Another issue is that the overhead introduced for *R-FIFO* configurations is higher than for *TOB* configurations in percentage terms.

At this point it must be remarked the difficulty to compare *CI* configurations with the *LI* configuration. This is because the first ones depend on the size of the writeset to propagate, persist and/or apply, while the second depends on the number of update operations to apply and their complexity, depending then on the transactions characteristics. In a generic way, it can be said that if a transaction contains a lot of update operations but changes few data items the *CI* configuration will introduce less overhead. Contrarily, if the transaction has few update operations but changes a lot of data items the *LI* configuration will introduce less overhead. Thus, only when in our transactional system one of these transaction types predominate can be adopted accurately the configuration that best fits our necessities.

Anyway it must be pointed out that providing amnesia support can only alter the original order cost for the *LI* in regard to the others, depending on the transaction characteristics.

8.6 Related work

A wide range of proposals for solving the recovery problem [12] have been presented in the literature. Traditionally, among these recovery protocols, the ones oriented for replicated processes have adopted the fail-stop failure model, as [14]. In these cases, they transfer the whole data state to new or "reconnected" nodes, a good approach for systems with few data state. Whereas for systems which manage large data amounts, as replicated databases, it has been largely recommended the crash recovery as it is proposed in [82, 20, 19, 6, 79] in order to minimize the recovery information to transfer. Few protocols as [87] have adopted the fail-stop failure model for large data state scenarios.

Among these last recovery protocols, some of them as [82, 20, 6] are version-based, while others as [82, 19] are log-based. First ones, which only transfer changed data, are typically useful for long-term outages whilst the latter, which send lost messages, present better performance for recovering short-term failures. Therefore, combining a version-based technique with a log-based one to construct a recovery framework has been proposed in several works as [82, 19] to improve the recovery features, choosing the recovery strategy that presents a lower cost each time an outdated node is detected.

Thus all these recovery protocols that have adopted the crash-recovery with partial amnesia failure model have to solve the amnesia problem. In fact, [19] proposes a solution for the amnesia phenomenon in log-based strategies based on logging received messages. This protocol is focused on replicated systems with the following characteristics: update everywhere, eager and using total order delivery.

In [79] it is presented a different way to deal with the amnesia problem in similar scenarios. In this paper the authors mix checkpointing with message transfer, in other words, they combine as recovery information to transfer a snapshot of the data state and the needed messages from the snapshot instant point. The combination of these two techniques allows the protocol to overcome the amnesia problem. The problem of this solution depends on the way in which the checkpoint process is performed, because if the whole data state is transferred the benefits of adopting the crash-recovery failure model are lost.

The [122] authors noticed that message delivery does not imply message processing, therefore they presented the *successful delivery* concept. They said that a message has been successfully delivered to a node when this node has acknowledged the message processing to the GCS. The background idea is that GCS can deliver multiple times a message to a node but it can only be delivered successfully once. To do so, they proposed the *end-to-end* atomic broadcast primitive which ensures that each message is successfully delivered once. This delivery mechanism provides an easy way to support amnesia if the acknowledgement is returned once the application has processed the message and has persisted its updates.

8.7 Conclusions

In this chapter, it is detailed first how transactional replicated systems based on total order broadcast communication primitives can manage the *amnesia phenomenon*, avoiding its related problems for log-based recovery strategies. Later, it is analyzed the amnesia support provided by other replicated systems configurations depending on their characteristics.

Finally, it is performed an analysis of the overhead introduced for providing amnesia support in different replicated system configurations using a log-based recovery strategy. The obtained results show that it is difficult to compare the overhead introduced between linear and constant interaction replication protocols. However, the study also shows that the differences in processing times in constant interaction replication protocols are not altered when amnesia is properly managed.

Chapter 9

Amnesia Support Review

This chapter reviews some recovery protocol proposals for the most usual transactional replicated systems –replicated databases–, analyzing their support for both the basic amnesia problem, detailed in 5.2, and the problem when this amnesia phenomenon combines with the majority progress condition 6.2. Moreover, it is presented a recovery protocol categorization determining for each category its support for both phenomena and proposing modifications when needed for supporting them accurately.

9.1 Introduction

Before starting with the review itself it must be noticed that it is based on the formalizations and Properties presented in Section 5.3 and Section 6.4.

This chapter is structured as follows: in Section 9.2 it will be pointed out some aspects of the GCS used by these replicated systems. Section 9.3 reviews the considered recovery protocols. Later, Section 9.4 presents a recovery categorization and finally Section 9.5 concludes the chapter.

9.2 Group Communication System Issues

Before surveying the recovery protocols and the replication protocols they are designed for, some issues dealing with replication aspects that would be used later in the study will be remarked.

As it has been said in the system model the replicated systems use a GCS which provides some communication primitives used by replication protocols to perform their work. It also provides a *membership monitor* which informs group

members about membership events –node connections, disconnections, network partitions, etc.–

Additionally, some GCSs provide *virtual synchrony* [24, 13] (or *view-synchronous multicast* according to [67]) since it ensures that all replicas have delivered the same sequence of messages before any replica fails or any replica is added. According to [24], the most relaxed property related to multicast delivery that provides virtual synchrony is *same view delivery*; i.e., that all destinations of each multicast deliver each message when they belong to the same *view* (a view change arises when one process fails or rejoins the group). Virtual synchrony provides a replicated work way which facilitates the recovery protocols implementation.

9.3 Considered Recovery Protocols

In this section the recovery protocols considered in this study are briefly described, highlighting only the details that are important from the amnesia support point of view. When detailed, for each recovery protocol some remarks are included for our study about the replication protocols to which they are associated. For other details readers are encouraged to look at the original papers. Note that in most cases the replication and recovery protocols were originally described in different papers (or even no replication protocol was described). As a result, a solution for the amnesia problems was not the target of such papers.

For determining if these recovery protocols provide accurate amnesia support it will be studied if they fulfill either the two properties *FS* or the property *CS* presented in Section 5.3. Moreover, it will be also considered the accurate management of the progress condition problem due to the amnesia phenomenon, presented in Chapter 6, taking under account the fulfillment of the properties presented in 6.3.1.

9.3.1 Protocols by Kemme, Bartoli and Babaoğlu

Multiple recovery protocols for replicated databases are presented in [82]. All of them are proposed for database replication protocols sharing the following characteristics: update everywhere protocol –ROWAA approach [62]–, based on total order broadcast –without a terminating phase– propagating a message per transaction –constant interaction [121]– and virtual synchrony. Moreover, replicated data objects are tagged with version numbers. The provided correctness criterion is *one-copy-serializability*.

These recovery protocols fulfill the following issues:

1. *Single Site Recovery*. The recovering node first brings its own database into a consistent state. To do so the underlying database maintains a log

of performed writes during the normal processing, storing the initial and resulting values for each changed data object. Then, once it reconnects it checks this log in order to store in the database the changes of committed transactions that were not already applied in the database.

2. *Data Transfer*. An operating site –recoverer node– must provide the current database state to recovering nodes. Different techniques can be used, from transferring the full database to transferring only the set of updated objects.

3. *Determination of a Synchronization Point*. If transaction processing is allowed during the recovery process it must be ensured that the recovering node will reflect the updates performed by these transactions. This synchronization process can be done in different ways but it depends strongly on the data transfer technique.

On the sequel the recovery protocols proposed in [82] are described, focusing on the data transfer and synchronization points.

Database State Transfer Checking Version Numbers

In this protocol global transaction identifiers are used, marking each data object with the identifier of the last transaction that updated it –allowing later the system to determine the information set to transfer in recovery processes–. The recovering node informs to the recoverer node about its *cover* transaction, (i.e. the transaction with the highest global identifier that successfully committed, $T_{i,m-s}$). Thus, the recoverer can determine the updates lost by the recovering node –information that must be transferred–, including the updates associated to $T^F_{n_l}$. The recovering node can easily determine its *cover* transaction by reviewing its single site recovery log.

The amnesia phenomenon is avoided with this replication protocol because the recovering node tells to the recoverer node which is its last real committed transaction, $T_{i,m-s}$. Thus, the recovery process transfers all data objects that were modified by transactions delivered between the last real committed transaction in the recovering node and the first transaction propagated after it become alive. These lost updates are transferred using a DT. In properties terms:

- *Prop. FS1*: It is fulfilled because the recovering node remembers its last really committed transaction.

- *Prop. FS2*: Data modified by T^F_{i,n_l} are marked with the associated transaction identifier, so they will be later transferred in the recovery process. Notice that it is possible that this data is included when recovering $T^M_{n_l}$ and not by T^F_{i,n_l}, because it has been modified also during the node disconnection.

In regard to the progress condition problem this recovery protocol does not avoid it. This is due to the fact that the outdated nodes affected by the amnesia phenomenon rely on the replicated system to obtain the information lost due to the amnesia phenomenon. In property terms it must be said that the *Prop. FS2'* as it is defined in 6.3.1 is not fulfilled.

This recovery protocol presents the drawback of scanning the entire database when checking for the database subset to transfer, then the following proposal was designed to overcome it.

Restricting the Set of Objects to Check

In order to avoid the full scan on the entire database, and with this the overhead and long locking time that it may cause, the use of a so-called "reconstruction table" is proposed. A record in this table consists of an object identifier and a global identifier informing about the last transaction that updated the object. Each update is recorded in the reconstruction table, unless all sites have successfully performed the update.

In contrast to 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 object level locks on the respective data items.

This proposed optimization does not handle correctly the amnesia phenomenon. This is because the reconstruction table only is generated when there are failed nodes. Then, only those objects modified in a view with failed nodes are included in this table. It implies that if a failed node has not been able to process correctly a message delivered before crashing –in a view with no failed nodes–, when it reconnects it will not apply the associated updates. And the reconstruction table will not store these updates because they have been performed in a view without failed replicas, being unable the recovery system to transfer the correct information set. Expressing it in properties terms:

- *Prop. FS1*: It is fulfilled because the recovering node remembers its last really committed transaction.
- *Prop. FS2*: It is not always fulfilled because data modified by T^F_{i,n_l} is only stored in the reconstruction table and marked with the associated transaction identifier if there are failed nodes. Therefore, the system will not have the T^F_{i,n_l} changes in the reconstruction table when a replica crashes in a replicated system where there were not any crashed node.

Therefore, this recovery protocol must be modified in order to support the amnesia phenomenon. One possibility consists in generating the recovery information in the reconstruction table either with or without the presence of failed nodes,

ensuring always the *Prop. FS2*. Other possibility consists in using our proposed solution in [38] ensuring then always *Prop. FS2*.

As the basic amnesia problem is not correctly managed by the original definition of this recovery protocol, it is also unable to support correctly the progress condition problem. And, in regard to the two possible modifications, only adopting our proposed solution in [38] would ensure the fulfilment of *Prop. FS2'* as it is defined in 6.3.1.

Filtering the Log

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, e.g., the use of multi-version concurrency control, as in `PostgreSQL`, or `Oracle`. 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.

This recovery technique must be combined with one of the previous ones, that will be used for generating the recovery information and determining the subset to transfer, for working. Therefore, its amnesia support depends on the support provided by the combined technique. Hence, if the last one –*Restricting the Set of Objects to Check*– is selected in its original description the amnesia phenomenon will not be managed accurately. The same happens when studying its correct support to the problem arising when the amnesia phenomenon is combined with the progress condition as it is specified in Chapter 6.

Lazy Data Transfer

Up to this point, all mentioned solutions use view changes as synchronization points. Then any recovering node enqueues all new broadcast transactions for applying them once it has recovered its lost views. This approach, despite being simple, has several drawbacks (detailed in [82]) leading the authors to decouple the synchronization point from the view change.

In this new approach any recovering site discards the messages delivered –instead of enqueuing them–. After the view change –triggered by its reconnection–, the recoverer site starts the transfer. When the transfer is about to complete, the recoverer and the recovering sites agree a delimiter transaction –one of the transactions broadcast in the new view– as synchronization point. Then, the recoverer site transfers all changes performed by transactions with lower identifier than the delimiter transaction one. While, the recovering site starts enqueuing transaction messages with greater identifier than the delimiter transaction one, for applying them once the data transfer is completed.

This recovery proposal differs from the others in the synchronization point, but depends on the previous ones for obtaining the recovery information. Then, it

will support the amnesia phenomenon and the problem when this phenomenon is combined with the progress condition depending on the recovery information generation policy being used.

9.3.2 Protocols by Holliday

The recovery protocols proposed by J. Holliday in [70], were designed for the replication protocols *Broadcast Writes*, *Delayed Broadcast* and *Single Broadcast* described in [1]. According to the classification in [124], these are eager update everywhere and non-voting protocols. Concurrency control is performed by the DBMS with Strict 2PL. These replication protocols make also use of a GCS which provides atomic broadcast, virtual synchrony and a membership monitor. They provide the *one-copy serializability* correctness criterion.

These replication protocols differ in the number of messages used for propagating transactions. *Single Broadcast* only spreads a message per transaction, *Delayed Broadcast* propagates two messages per transaction –writeset and commit messages–, while *Broadcast Writes* sends a message per write transaction operation –linear interaction–.

On the sequel it will be summarized the recovery approaches presented in [70].

Single Broadcast Recovery

This recovery approach is designed for replication protocols which broadcast a single message per transaction as [88]. Another author criterion design is to avoid to transfer the whole database in the recovery process if possible, and the selected mechanism for doing so consists in reapplying in outdated nodes the messages that they have lost.

Therefore, this recovery protocol relies on a GCS which provides a log of delivered messages. If the GCS does not provide this log, the recovery protocol must designate some replicas as *loggers*. These *loggers* will have a log where they will store persistently the delivered messages, either those notifying view changes and those broadcasting update transactions –keeping only those associated to committed transactions, deleting aborted ones–.

Then, when a node reconnects –a view change is triggered– it requests a *logger* to be brought up-to-date, informing about its last view –last view in which the recovering node was alive–. Thus, the *logger* transfers to the outdated node the messages broadcast during the views it was crashed –maintaining their original order–. Sometimes the *logger* will not have, due to log storing policies, all the messages necessary in the recovery, thus it will transfer the whole database. It must be remarked that as long as a node is being recovered the replicated system can not work, starting only when the recovery process has been completed.

This protocol does not support the amnesia phenomenon accurately, because when a crashed node reconnects, the system starts to transfer the messages

broadcast in the views it was failed. Then, this information does not include messages delivered in the crashed node before crashing but not processed correctly. In properties terms:

- *Prop. FS1*: It is not fulfilled because the recovering node only remembers its last seen view, i.e., it does not maintain nor propagate the $T_{i,m-s}$ identifier.
- *Prop. FS2*: This property is not fulfilled because messages –recovery information– are stored by view. But it can be easily overcome using messages as basic recovery information unit.

This fact, also implies that this protocol does not avoid the progress condition problem, because neither *Prop. FS1* nor *Prop. FS2'* are fulfilled.

One possibility for handling accurately the amnesia phenomenon in this recovery protocol would be to use message or transaction identifiers –which are equivalent in this replication protocol–. Then the recovery protocol can be modified forcing each replica to mark which is its last really committed transaction. Therefore, when a replica reconnects after a crash, it can inform about its last committed transaction to the *Logger* –instead of using the identifier of its last seen view– for the recovery. Then both properties as they are defined in Section 5.3 are ensured. But, the *Prop. FS2'* defined in 6.3.1 is not fulfilled, so the problem arising when the amnesia phenomenon is combined with the progress condition is not avoided.

Another possibility would consist in giving the *Loggers* role to all the replicated system members. Therefore when a node reconnects, it will perform a local recovery step consisting in checking if some of the persisted messages in its local log have not been correctly processed, applying them in this step. In this case the properties *Prop. FS1* and *Prop. FS2* defined in Section 5.3 are also ensured, like the properties defined in 6.3.1. Therefore, both the original amnesia problem and progress condition problem are overcome.

In both cases, notice that it is necessary that *Loggers* store persistently the broadcast messages even when there are no failed nodes. In the second approach, the messages that have been seen by all nodes –because there are not any failed nodes– can be removed from the log –of a replica– as soon as they are correctly processed in this replica. While, in the first approach, these messages can only be removed view per view. And the messages broadcast in a view where there were not failed nodes, only can be removed if in the subsequent view there are not any failed nodes –which is a non sense– or when the nodes whose crash triggered this view change are recovered and the system ensures that they have correctly processed all the messages broadcast when there were not failed nodes.

The second solution provides better recovery support as all replicated members are *Loggers*, and provides a simpler way for managing messages that have been seen by all nodes. Moreover, it also avoids the problem associated to the amnesia

phenomenon when combined with the progress condition. Therefore, its use must be encouraged.

Delayed Broadcast Recovery

The *Delayed Broadcast* replication protocol decouples the writeset broadcast from the commit broadcast for any transaction –weak voting technique [123]–. 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. Two possible solutions for the problems caused by the writeset-commit decoupling are presented:

1. *Log Update Method*. In this approach, at each view change, *loggers* must examine their logs or the database state for determining if there exist on progress transactions in the nodes without failure. If there are, the *logger* must mark these transactions in order to copy their writeset message in the log associated to the view when their commit was broadcast. So, when a previously failed node rejoins to the group, the *logger* begins transferring writesets of in progress transactions when the node failed, following with messages of transactions originated and committed while the node 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 nodes without failure.

2. *Augmented Broadcast Method*. This second method gives additional process for managing on-going transactions and requires a change in the lock policy for recovering nodes during the global recovery. The new replication protocol forces to include the writeset in the broadcast commit message for these transactions that have delivered the writeset in a previous view. The nodes that have already seen the first broadcast writeset message ignore the writes included in the commit message, and *loggers* store the augmented commit message. The existence of augmented messages obliges global recovery to change its lock policy as it is described in [70].

In this case, as the policy for determining the start point recovery is the same one as before –the identifier of the last view in which the crashed node was alive–, an accurate amnesia phenomenon support is not provided. Explained in properties terms:

- *Prop. FS1*: As before, it is not fulfilled because the recovering node only remembers its last seen view.

- *Prop. FS2*: This property is not fulfilled because messages –recovery information– are stored by view.

Consequently, it must be said that this recovery protocol can not handle accurately the progress condition problem.

Therefore, the modifications proposed in the previous recovery protocol are also valid for this one. Anyway, it must be pointed out that in this case handling delivered messages correctly is more difficult because the system broadcasts two messages per transaction –*writeset* and *commit* or *rollback*– with its associated complexities.

If the second solution –all nodes are *Loggers*– is selected, when the recovering node performs the additional local recovery step –checking if some of the persisted messages in its local log have not been correctly processed for applying them– it must discard the messages belonging to transactions whose commit message is not also stored in the log. This is because these messages would be later applied in the *Global Recovery* process.

Broadcast Writes Recovery

The *Augmented Broadcast* global recovery method presented for the *Delayed Broadcast* replication protocol could be used also for the *Broadcast Writes* one –which broadcasts a message for each write operation, in other words linear interaction–. Then all writes must be attached to the commit message to be broadcast for on-going transactions, as it does the *Augmented Broadcast*. But in this recovery protocol *loggers* must take special care for removing the logged messages of aborted transactions due to deadlocks, in order to not reapply them in recovering nodes.

As the two other recovery protocols proposed by Holliday it does not handle correctly the amnesia phenomenon problem, because the underlying mechanism for determining the recovery information set to transfer is the same one –to send the identifier of the last view seen by the crashed node–. In properties terms:

- *Prop. FS1*: As in two previous ones, it is not fulfilled because the recovering node only remembers its last seen view.
- *Prop. FS2*: This property is not fulfilled because messages –recovery information– are stored by view.

Therefore, this protocol does not avoids the progress condition problem.

Anyway, the proposed solutions for the previous recovery protocol will also work for this one.

9.3.3 Parallel Recovery by Jiménez, Patiño and Alonso

In [79] the authors presented a recovery protocol whose main goal was to avoid stopping the replicated system work when performing recovery processes.

The replication protocol for which it was designed used a GCS that provided strong virtual synchrony, reliable multicast and a membership monitor. The replicated database was divided into disjoint partitions, andthe system forced transactions to access only single partitions. Each partition had a master site –which processed the transactions accessing this partition– and the rest of replicas worked as backups –which only applied updates–, therefore it is a passive replication protocol per partition. And transactions are broadcast using only one message –constant interaction–. The transactional system supports Strict 2PL, providing *one-copy serializability*.

Each node has a log –one per partition– which contains the committed updates in the same order they were applied. Updates are only logged once their commit is confirmed. When a crashed node reconnects to the system it informs about the LSN –*log sequence number*, a global number– of its last committed transaction on each partition. Then the selected recoverer for each partition will collect and transfer from its log the set of messages needed to recover this partition in the outdated node. In order to limit the recovery duration –interesting for long failure times– some form of checkpointing is assumed. Therefore, if it is necessary, the recovering site will first receive a recent checkpoint of the database and later can start applying messages from this checkpoint.

The combination of these two techniques, or the use of the *LSN* of the last committed transaction in the node being recovered allows the protocol to overcome the amnesia problem. The problem of this solution depends on the way in which the checkpoint process is performed, because if the whole data state is transferred the benefits of adopting the crash-recovery with partial amnesia failure model are lost. Expressed in properties terms:

- *Prop. FS1*: It is fulfilled with the use of *LSN*.
- *Prop. FS2*: This property is fulfilled because nodes store committed updates and combines this with a checkpointing technique when necessary.

This recovery protocol in spite of supporting the basic amnesia phenomenon does not avoid the progress condition problem because it does not fulfil the *Prop. FS2'*. This is due to the fact that T^F_{i,n_l} changes are not stored and maintained in the crashed replica.

9.3.4 The COLUP Recovery Protocol

A configurable eager/lazy replication protocol with a lazy recovery protocol is proposed in [74]. The replication protocol can be categorized as an update

everywhere approach with voting technique, using constant interaction. This protocol defined and provided its own correctness guarantees: *transaction* and *checkout* consistency. These correctness guarantees are somewhat equivalent in some circumstances to *snapshot isolation* and *read committed* respectively.

In this replication protocol each data object is owned by the replica where it was created. For any object, a set of nodes will maintain synchronous copies, while other replicas constitute the set of asynchronous copies. In these last nodes object updates will be eventually received, once they have been committed in synchronous replicas. The owner is responsible of managing object accesses and coordinating the propagation of their last versions.

Conflict transactions are solved in the processing node in an optimistic way, using object versions. To do so, for each accessed object –for those the node does not have a synchronous copy– it calculates the probability of having an outdated version. If the obtained value is higher than an established threshold the node assumes that its object version is obsolete, obtaining from the owner node the last version. Later, in the commit phase it checks for possible conflicts. Aborting the transaction if it has read obsolete values that were updated by other concurrently committed transactions.

In a node crash the ownership of its objects is assumed by an alive and synchronized replica. Then, alive nodes inform the new owner about *previous grants* conceded to these objects by the previous owner. Thus, the new owner can process the requests as if it was the original owner node of the object.

When a node recovers from a failure, it sends a message to the node that managed its owned objects in order to synchronize the activity in both nodes. In this process, the recovering node updates in a version-based way the state either of its owned objects and the objects for which it is a synchronized replica. During this process, the 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.

This recovery protocol provides accurate amnesia support because as soon as a node reconnects it starts to obtain the last state of its owned and synchronized objects in a version-based way. The objects that are maintained in this replica asynchronously are updated using the basic mechanism provided by the replication protocol. This protocol fulfils the *Prop. CS1*, because all the state is transferred: synchronized objects are transferred immediately in the recovering process, and non-synchronized ones are updated using the replication mechanism. Therefore, the other properties are not needed. It must be noticed that in spite of adopting the crash-recovery with partial amnesia failure model this recovery protocol transfers the whole state instead of sending only the missed information during the disconnection period.

But, this recovery protocol does not avoid the progress condition problem due to the fact that *Prop. FS2'* is not fulfilled, as long as the information this

property refers to is neither stored nor maintained in the replica having the amnesia problem.

9.3.5 CLOB: Short-Term Failure Recovery

CLOB (Configurable LOgging for Broadcast protocols) described in [19] is defined as a framework for reliable broadcast protocols that are used as a basis for database replication. Its aim is to log messages in the broadcast protocol core, providing with this automatic recovery for short-term failures, but discarding the log and using a version-based recovery protocol (e.g. [20]) for long-term outages.

In order to do so the recovery protocol has two logs: one for missed messages, another for received messages. In the first one, each node stores any message it delivers when there are failed nodes, maintaining them as long as there is any failed node that has not received them. In the second one, each node stores any received message, removing it as soon as it is correctly processed. So, when a crashed node reconnects –and the system uses the log recovery–, it first checks the log of received messages in order to process its last received messages that were not correctly processed before crashing. Later, it asks for its missed messages, and applies them.

Notice that if the outage period exceeds a given threshold, the reliable broadcast service will notify the replication protocol about that, and the logs will not be used.

The CLOB recovery protocol manages accurately the amnesia phenomenon because it considers a persistent log where each replica stores its delivered messages as soon as they are received. And these messages are only deleted once they are correctly processed. Then, when a crashed node reconnects, only needs to check this log and reapply the messages it contains. Talking about properties:

- *Prop. FS1*: It is fulfilled in an indirect way. All messages maintained in the queue represent delivered transactions non correctly processed, so instead of knowing its last really committed transaction it has the T^F_{i,n_l}.
- *Prop. FS2*: As it has been said above, each node stores persistently its own T^F_{i,n_l}.

This protocol fulfils both the *Prop. FS1* and *Prop. FS2'* –each replica stores persistently its delivered messages–, therefore it avoids the progress condition problem.

9.3.6 Protocol by Armendáriz

In [3] three replication protocols are considered –*BRP*, *ERP* and *TORPE* –, and a recovery protocol that can be applied on *ERP* and *TORPE* is proposed.

These two replication protocols are categorized for being eager update everywhere and sending a constant number of messages per transaction. They make use of a GCS which provides reliable broadcast, a membership monitor and virtual synchrony. The correctness guarantees provided by these protocols were *one-copy serializability*, provided thanks to the use of underlying DBMS which ensured serializability.

The main idea for the recovery protocol proposed in [3] is to store in a database table –in all alive replicas– the identifiers of objects modified when there are failed nodes, grouping them per views. Then, when a failed node reconnects, it informs about the last view in which it was alive. Later, a recoverer node transfers to the recovering node the identifiers of modified objects during its disconnection, and later transfers their values.

The recovery protocol proposed by Armendáriz for the replication protocols ERP and TORPE can not manage accurately the amnesia problem. In this case, the problem resides in the fact that this recovery protocol assumes that any delivered message is correctly processed, but this assumption, as demonstrated in [122], is not correct. So, all generated recovery information does not contain all the information that would be needed for supporting amnesia. Expressing all this in properties terms:

- *Prop. FS1*: It is not fulfilled because the recovering node only remembers its last seen view.
- *Prop. FS2*: This property is not fulfilled because the recovery information is grouped by view. And either it presents the problem of being generated only when there are failed nodes.

As *Prop. FS2* is not fulfilled, *Prop. FS2'* is neither fulfilled, then the progress condition problem is not avoided.

In [58] it is provided amnesia support to this recovery protocol. The adopted solution is the same one as proposed in Section 8.3, to log persistently the delivered messages. This reviewed approach supports the amnesia phenomenon and avoids the progress condition problem.

9.4 Amnesia Support Recovery Observations

It has been seen in the study how a correct amnesia support depends on the combination of an adequate recovery information generation policy and an accurate way for notifying the last really committed changes in the node that must be recovered.

On the sequel, it will be presented some observations obtained from the performed study. These observations are grouped first by the used technique –

version-based or log-based–, and secondly by the granularity used for managing the recovery information.

This categorization will not consider the recovery solution consisting in transferring the whole state, because the original goal of adopting the crash-recovery with partial amnesia failure model in replicated systems is to avoid its use.

9.4.1 Version-based Techniques

Version-based recovery protocols can overcome this problem in different ways, depending on the basic way used for performing the recovery processes.

Transaction identifier

The first one will consist in storing for each object the identifier of the last transaction that modified it. But, this must be done even if there are not failed nodes as it does the *Database State Transfer Checking Version Numbers* presented in [82], because if it is not done the amnesia support is not provided as it happens with *Restricting the Set of Objects to Check* presented also in [82]. Thus, in this case the recovering node only has to inform the recoverer node about the identifier of its last committed transaction. Therefore, properties *Prop. FS1* and *Prop. FS2* are ensured. But, this solution does not ensure *Prop. FS2'* because the information for solving the amnesia problem is not maintained in the crashed node, then it does not avoid the progress condition problem.

An alternative for this strategy will be to combine it with our amnesia generic solution approach described in Section 5.2. In this case it would not be necessary to generate this information even when there are not failed nodes. And, then this approach does not need the transaction granularity being enough with the view identifier granularity. It is due to the fact that in this case each replica maintains its own T^F_{i,n_l}, being only necessary to inform the recoverer node about the last seen view in the recovering node. Then, this solution overcomes the progress condition problem.

View identifier

Another possibility is to store for each object the identifier of the last view in which it was modified. The problem of this solution is that the recovery protocols that follow this approach start the recovery process from the first view lost by the recovering node, being impossible then to solve the amnesia problem, associated to the *forgotten state* –T^F_{i,n_l}– because even if *Prop. FS1* is ensured, *Prop. FS2* is not ensured. The non fulfilment of *Prop. FS2* implies that *Prop. FS2'* is not fulfilled either. It happens in *Protocol by Armendáriz* [3]. This can be solved as follows:

- One option for overcoming this would consist in including in the transfer recovery process the changes performed in the last view seen by the recovering node. So, this solution forces the system to generate recovery information even when there are not failed nodes. But, this approach presents some drawbacks. On one hand, it forces to transfer all the performed changes in a view –most of which will have been already seen by the recovering node– for solving the amnesia problem that will affect usually a very small subset of changes done in such view. On the other hand, it is possible that in very special cases transferring only the changes done in the last view seen by the recovering node is not enough for solving the amnesia problem (e.g. a sequence of very short views in time terms). Moreover, this solution does not overcome the progress condition problem because *Prop. FS2'* is not fulfilled.

- Discarding the previous option, another strategy will consist in combining this strategy with our generic approach –using in each replica a persistent log of delivered messages– as it is done in [58], fulfilling then the properties *Prop. FS1* and *Prop. FS2*. In this case, it is not necessary for the version-based strategy to generate information when there are not failed nodes, because it is already maintained in the queue. When this technique is adopted the recovery protocol overcomes the progress condition problem.

9.4.2 Log-based Techniques

In these techniques, recovery protocols use as recovery information the broadcast messages during the replication work. Therefore, the only way for solving the amnesia problem is to maintain in the system the messages that can be affected by the amnesia problem.

Transaction identifier

In this technique, stored messages –all replicas store messages– are not grouped by views, then when a crashed node reconnects it informs about the message corresponding to its last committed transaction. Then, the recoverer node sends to the recovering node the set of messages it has not correctly processed and it has lost. Notice, that this policy will overcome the amnesia phenomenon in all cases, only if logs store messages even when there are not failed nodes. If this behavior is not provided the *Prop. FS2* is not ensured when a replicated system transits from a view where all replicas were alive to another where there are failed nodes.

An important aspect of this technique is when messages or updates are stored in the log. If messages are persisted as soon as they are delivered, crashed nodes will have at recovering time the messages they have delivered but not processed correctly –those associated to T^F_{i,n_l}–. Then, they do not have to ask updated

replicas for these messages, only for those they have not seen. These protocols also avoid the progress condition problem. On the contrary, if messages –or updates– only are logged when they are really committed, crashed nodes will not have the messages necessary for overcoming the amnesia problem at recovering time. So, in this case the information for solving the amnesia phenomenon must be looked for in the recoverer replica. Therefore, this second strategy does not support the progress condition problem.

This is the case of the *Parallel Recovery by Jiménez, Patiño and Alonso* [79] protocol. This protocol also combines this technique with checkpointing for log shortening reasons. It must be noticed that this protocol stores updates once they are committed –non when they are delivered–, so crashed replicas must ask updated replicas for messages delivered but not correctly processed. This behavior, as it has been said in the previous section, implies that the progress condition problem is not avoided.

View identifier

In this strategy broadcast messages are stored when they are delivered –in the same order delivery– being grouped by views –when there are crashed nodes–. Then, when a crashed node reconnects it informs to the system about its last seen view. At this point, the system starts to send to the recovering node the messages broadcast during the view it was crashed. Therefore, the amnesia problem is not solved as it occurs in all recovery protocols proposed in [70], because it will not contain messages seen by the crashed node but non correctly applied, in other words the recovery process does not transfer the messages corresponding to the transactions set T^F_{i,n_l}. In fact, neither *Prop. FS1* nor *Prop. FS2* are ensured. The same happens with *Prop. FS2'*, which is not ensured, then the progress condition problem is not avoided. For solving this problem, two different approaches can be adopted:

- A first proposal for avoiding the amnesia problem in this technique can consist in transferring in the recovery process the messages broadcast during the last view where the crashed node was alive. Then, this solution needs to store broadcast messages even if there are not failed nodes. But, it can be optimized if the recovering node informs about the identifier of the message associated to its last correctly processed transaction. Moreover, it must be noticed that if all nodes store broadcast messages the own crashed node will contain the messages it has received and not correctly applied, obtaining then the second approach.

- The second one consists in applying our proposed generic solution, that in fact is the solution already applied in [122, 19]. In [122], authors proposed the "*successful delivery*" approach. A successfully delivered message implies that it has been correctly processed. Therefore, they proposed that

the used GCS has to deliver the same message to a replica until it is successfully delivered in this replica. In [19], each node stores persistently all its delivered messages, being only removed when they are correctly processed. Obviously, if there are failed nodes, correctly processed messages are not removed but maintained in another log for recovering failed nodes during this view. This proposal solves both the amnesia phenomenon and the progress condition problem.

9.5 Conclusions

In this survey it has been analyzed how some recovery solutions for replicated databases, which have adopted the crash-recovery with partial amnesia failure model –in order to avoid to transfer the whole database–, manage the introduced amnesia phenomenon problem and the progress condition problem presented in Chapter 6.

The amnesia phenomenon problem appears because some works assume that all delivered messages are correctly processed, fact that as it is demonstrated in [122] is not true. Then, in most cases their provided recovery solutions do not handle correctly this problem. Among the studied papers only the recovery protocols proposed in [79, 74, 19] and two of [82] manage accurately this problem. But, even not all these recovery proposals are able to avoid the progress condition problem.

Moreover, for those studied recovery protocols which do not provide accurate amnesia support it has been proposed solutions for overcoming this situation.

Later, a categorization of the analyzed recovery techniques has been performed commenting if they provide accurate amnesia support –solving both problems or not, and how they can be improved to support them when they do not in its original definition.

Chapter 10

Amnesia in Linear Interaction Systems

In Chapter 9 it has been seen that most recovery protocols have been designed with replication protocols based on *constant interaction* –broadcasting a constant number of messages per transaction– while very few have been implemented for replication protocols using *linear interaction* –where a message is broadcast for each transaction operation–. This is due to the fact that researchers consider and also have demonstrated that *constant interaction* provides better performance behavior in replication techniques, therefore they have considered the *linear interaction* useless. However, *linear interaction* is at first glance the most natural approach for building a replicated system.

This chapter presents a general strategy for recovery protocols based on *linear interaction*, analyzing the problems that arise when considering the amnesia phenomenon. At the same time, it is also studied its provided support for snapshot isolation apart from 1-copy-serializability.

10.1 Introduction

This chapter presents a general strategy for recovery protocols based on linear interaction, in contrast of using the constant interaction [124] approach. Linear interaction, in spite of its high performance cost, will be the only feasible alternative for object-oriented replicated systems with large data states to transfer, and with a transactional support, such as FT-CORBA with its complementary *Transaction Service*, where constant interaction will either lead to huge messages or be impractical in case of partial replication, since the state to be transferred should be collected from different source nodes. But, as it will be shown, the management required by linear-recovery protocols is more complex because it

must manage multiple messages per transaction. In addition, for ensuring correctness under linear interaction, messages belonging to not-yet-committed (as well as for rolled-back) transactions, must be adequately treated.

In parallel, the proposed recovery strategy adopts the *crash-recovery with partial-amnesia* failure model because it supports the recovery of outdated nodes. Recovery which becomes a key point for building fully-functional fault tolerant systems in replicated systems with large data states. The traditional adopted failure model, *crash* or *fail-stop*, is not adopted because does not support outdated nodes recovery presenting only good behavior for replicated systems with few data state.

The idea is to obtain a recovery protocol for linear interaction replication protocols which minimizes the effort and cost of the recovery process, without stopping the replicated system work for *primary partitions*. It is also intended to perform partial recoveries, when needed. Finally, as its design is performed as a middleware recovery system, it can be easily applied to different transactional scenarios, specially including database replicated systems. The obtained results can be used to perform a generic revision of recovery protocols for constant interaction replicated systems.

This chapter is structured as follows. In Section 10.2 it is outlined the recovery protocol for transactional replicated systems based on linear interaction. Section 10.3 presents the consistency problem due to on-going transactions, created by the interleaved phenomenon. Some related work is given in Section 10.4, and finally 10.5 concludes the chapter.

10.2 Recovery Protocol and Amnesia Support

Since the replication protocol considered uses linear interaction for implementing its functionality, the most natural way for performing the recovery will follow a log-based strategy to recover outdated nodes as the one proposed in Chapter 8 for constant interaction systems.

The recovery protocol that must be used in a transactional replicated system based on linear interaction has the same stages as the recovery protocol presented in Section 8.2 and uses the same information as the one proposed in Section 5.6.1.

In regard to the amnesia support in the recovery process it must be taken the same approach as the one presented in Section 8.3. Obviously, some differences arise between these two approaches.

At replica level as linear interaction systems transactions work is broadcast in multiple messages the non committed state at each replica would be higher than for those systems using constant interaction. State that would be lost in a crash occurrence. Therefore, in this scenario it is more important to know which are the “really committed” transactions, in order to know which messages must be

reapplied in the recovery process. Then the information about the success of such commit must be also stored because it is needed by the recovery amnesia process in order to know which are the messages that must be applied in the ARP. So, at the replicated system level, the problem is to know if a "commit" message was successfully applied before the failure or not.

But, when performing the recovery process a new problem can arise due to the use of linear interaction. This problem is detailed in the following section.

10.3 On-Going Transactions and Consistency

The use of linear interaction in replication protocols implies the broadcast of messages belonging to not-yet-committed transactions –only the updates in a ROWAA approach. Thus, these messages belonging to different transactions are interleaved and applied to the replica in their delivery total order. Finally, each transaction is committed when its commit is applied. In this context, if a node crashes, all associated changes to not-yet-committed transactions are lost whilst associated updates to committed transactions remain permanent.

Afterwards, when the crashed node becomes again active, the recovery process updates it, reapplying among others the messages associated to not-yet-committed transactions at the crash time, while the committed transaction messages at the crash time are not reapplied (since they were already persisted in the replica). In this scenario, some inconsistencies could arise if these reapplied messages were interleaved with committed transaction messages in the original work sequence, because this original order is misunderstood in the recovered node. The inconsistencies appear if these transactions conflict and the selected isolation level tolerates these conflicts.

It must be remarked that this problem only occurs when an outdated node reconnects to a *working* replicated system, and it has not lost the *working condition* from the instant when the outdated node crashed. Moreover, notice that the interleaving phenomenon does not happen when the replication protocol uses constant interaction. This is due to the fact that all the updates associated to this transaction are broadcast using a single message –even when more than one message per transaction is used– instead of broadcasting the updates using multiple messages as the linear interaction does.

The following example shows this problem in a more intuitive way. Let us assume a replicated system of three nodes, $\alpha = r_1, r_2, r_3$. At the beginning, the three nodes are up-to-date and working. During a replicated system work period, the sequence of events shown in figure 10.1 happen.

In the linetime shown in figure 10.1 appears the following events:

- $T1$, transaction started in r_1 and compound by messages $m1, m3, m5$
- $T2$, transaction started in r_2 and compound by messages $m2, m4, m6$

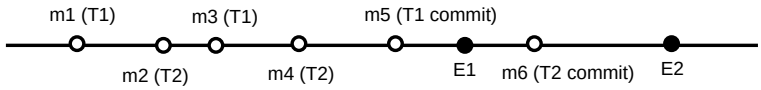


Figure 10.1: Linetime events

- $E1$, node r_3 crash
- $E2$, node r_3 recovery process start

As it can be seen, in the original sequence order, the messages of $T1$ and $T2$ are interleaved. $T1$ commit is performed before the crash of node r_3, while the $T2$ commit is done during the r_3 failure time. Therefore the final messages sequence seen in r_1 and r_2 is:

$$m1, m2, m3, m4, m5, m6$$

while the final message applied sequence in r_3 once it has been recovered is:

$$m1, m3, m5, m2, m4, m6$$

This message order misunderstood in r_3 is originated by the recovery protocol. In fact the node r_3 before $E1$ applies the same sequence message order as r_1 and r_2 that is:

$$m1, m2, m3, m4, m5$$

but, when it fails, it loses non committed changes, in this case the changes performed by $T2$. Thus, when it reconnects to the system its data state is:

$$m1, m3, m5$$

At this moment, the recovery process applies the not yet applied updates in r_3, which in this case are the $T2$ messages, causing the message order difference. This different message order in $T2$ could lead to a different data state with regard to the state in r_1 and r_2 if $T1$ and $T2$ conflict and the selected isolation level tolerates it. In this example a conflict could arise if $m2$ and $m3$ perform the following sentences respectively:

$m2 \rightarrow$ "$UPDATE\ employees\ SET\ salary = salary{*}1.05\ WHERE\ points > 10$"

$m3 \rightarrow$ "$UPDATE\ employees\ SET\ points = points{+}1\ WHERE\ points == 10$"

With these sentences, it is possible that in r_3 some employees increase their salary while in r_1 and r_2 their salaries are not increased. Thus, the recovery protocol can generate different data state evolutions in recovered nodes with regard to not recovered nodes. This problem appears because the recovery protocol does not store the transactions original context. For instance, the original table 10.1 will evolve to 10.2 in r_1, r_2 and to 10.3 in r_3 after applying T_1 and

EmployeeId	Salary	Points
001	18000	9
002	18000	10
003	21000	10
004	21000	11

Table 10.1: Employees.

EmployeeId	Salary	Points
001	18000	9
002	18000	11
003	21000	11
004	22050	11

Table 10.2: Employees in r_1, r_2.

T_2.

In order to avoid this problem, two solutions can be applied. The first and most natural one consists in selecting an isolation level that aborts this kind of conflicts, which in fact implies to apply the serializable isolation level at the replication protocol. Thus, this approach avoids the problem presented above allowing to use the two proposed recovery strategies: the log-based and the version-based.

Another option would be to relax the required consistency guarantees, which means to tolerate this kind of conflicts, but in order to avoid the above presented problem this approach implies to perform the recovery process under a special condition. Such special condition requires that the recovery process must be done when the recovery messages to apply do not conflict with committed transactions during its life. This means that the recovery messages to apply were not interleaved with conflicting committed messages. As controlling the fulfilment of this condition is difficult, it must be selected an easiest to control condition. This new condition would be to select as *Base Recovery Point* (BRP) a data state in the replicated system lifetime where there does not exist on-going transactions. Obviously this base recovery point must be posterior to the moment when the outdated node crashed. Then, the outdated node recovery is performed in two steps. In the first one, the outdated node recovers the data state up to the selected BRP, using the version-based approach. It must be remarked that in this step it can not be used the log-based recovery strategy, since the problem of different state evolution would not be avoided. In the second step, it would be applied the messages that have been delivered after the selected BRP (if they exist) using the log-based approach. This solution could be implemented in two different ways: reactive and proactive.

EmployeeId	Salary	Points
001	18000	9
002	18900	11
003	22050	11
004	22050	11

Table 10.3: Employees in r_3.

In the **reactive** one (figure 10.2), the base recovery point is selected once the crashed node reconnects to the working replicated system. The system has two options at this moment: to wait until the working system reaches the previous defined condition in a natural way or to force it either temporarily prohibiting the start of new transaction or delaying progressively the start of new transactions. The first one does not ensure to reach this condition at any known interval time. Thus, it is discouraged in replicated systems with a high work load, whilst the second one implies to stop in an aggressive way the replicated system work that is also an undesired situation.

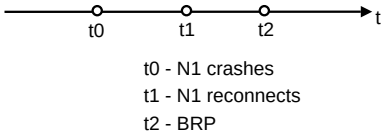


Figure 10.2: Reactive BRP

The **proactive** way (figure 10.3) consists in using BRPs previous to the crashed node reconnection. The idea with this option is to keep ready the recovery information before the failed node reconnects to the system. Therefore, as soon as a node fails (or as long as there exist failed nodes), the system starts to check when the replicated system fulfils the BRP conditions. Each time these conditions are reached, the recovery system will store the version-based recovery information of this point, erasing the previous one if existed. After that, when a failed node reconnects to the system its recovery is performed using for the first step the last BRP information generated. For the second step, it is used the messages broadcast after this point.

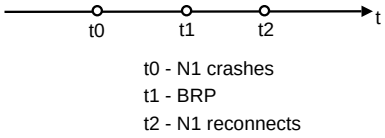


Figure 10.3: Proactive BRP

Obviously, if the replicated system reaches this condition easily with a high

frequency, the system can decide not to generate this information at each time the base recovery point condition is fulfilled.

Between the two proposed solutions to solve this problem of potential data node state divergence, it is preferable to select in first instance the establishment of the serializable isolation level. This election is based on the fact that this approach does not need to wait for any non-replication system work instant as the BRP (as the other one does). Obviously, working in a linear-interaction replication system combined with the serializable isolation level will present a cost in terms of performance and efficiency that must be considered.

It must be noticed that these two last proposals are based on the same concept that is used for the recovery solution proposed for non-transactional replicated systems in Chapter 11.

Moreover, the only possible solution to use a log-based recovery approach without using a version-based one forces the system to adopt the serializable isolation level. Another aspect that must be considered is which recovery information policy must be applied, and how it is affected by the linear recovery interaction. The following section is devoted to the discussion of this aspect.

10.4 Related work

In the area of recovery protocols for replicated distributed systems two basic approaches are used: version based and log based. The first one consists in transferring to outdated nodes those data items changed during their failure period, whilst the second one consists in transferring the messages missed by outdated nodes.

A wide range of proposals about this classic problem[12] have been presented for a long time in the last years either version-based [82], [20] and log-based [82], [19], [79]. First ones are typically useful for long-term outages whilst the latter ones present better performance for recovering short-term failures. Therefore, combining a version-based technique with a log-based one to construct a recovery framework has been proposed in several works as [82], [19] to improve the recovery features, choosing the recovery strategy that presents a lower cost each time an outdated node is detected.

The most widely assumed correctness criterion for replicated systems is *1-copy-serializability*, which consequently leads to recovery protocols intended to work with such systems, often using log-based approaches [79], [82], [70]. However, the use of other isolation levels has not been traditionally treated in recovery protocols, probably based on the assumption that replication protocols are intended to provide *1-copy-serializability*. In fact, this is the isolation level that best fits the consistency guarantee in a general distributed system. But, when the replicated state requires high transfer rates, its use implies a high performance cost. Also, for transactional systems, where isolation must be enforced

by using specific concurrency control mechanisms, this problem is even worst. These two drawbacks are specially problematic in replicated databases, where the enforcement of *1-copy-serializability* usually leads to extremely inefficient systems. Therefore, relaxed isolation guarantees are used there to alleviate the performance degradation associated to the highest isolation level. One of the most widely adopted relaxed levels is *Snapshot Isolation*, having the interesting property of allowing read-only transactions to proceed without being blocked or delayed by any other transaction. In this way, recent publications [88, 50, 81, 80, 30], have proposed some replication protocols providing *Snapshot Isolation* [10]. Moreover, the most extended DBMS (Oracle [55], PostgreSQL [105]...) provide snapshot isolation as the basic isolation guarantee.

On the other hand, recovery protocols are also typically designed to work for replicated protocols based on *constant interaction*[82]. Others ,simply outline how these protocols can work using linear interaction. In fact, a few works have designed recovery protocols[70] which work over linear-interaction-based systems. In [70], different log-based recovery protocols are presented including proposals either for constant and linear interaction, but always focused on serializable systems.

10.5 Conclusions

In this chapter, it has been detailed a middleware-based general log-based recovery strategy intended to provide fault tolerance support for *linear interaction*-based replication systems. This obtained system lets to perform on-line recoveries, fulfilling one important condition for building a high available system. Most important, this chapter studies which effects has the use of *linear interaction* on the recovery work, specially emphasizing the global data state consistency and the recovery information management.

Moreover, the chapter also analyses and designs an amnesia recovery process as part of the whole recovery strategy, supporting a more realistic failure scenario. This amnesia phenomenon has been discussed at the two different levels in which it could appear, and a basic strategy to bound the amnesia problem has been also detailed.

In addition, the proposed strategy supports re-inclusions in minority partitions, performing partial or full recoveries, helping the system to accelerate outdated nodes recovery.

Another important aspect demonstrated in this chapter, in Section 10.3, is that using a linear-interaction replication protocol forces the system to use serializable isolation level to avoid consistency problems after any log-based recovery process. In fact, the use of any other isolation level could let different replicas to reach different data states after applying the same transactions set.

In an indirect way, this chapter also has highlighted that the existence and

management of on-going transactions (due to linear interaction) from a recovery point of view presents several difficulties (replicated consistency, amnesia delimitation boundaries), whose solution reduces the whole system performance and scalability. Therefore this chapter reinforces the traditional arguments (traffic net overhead) that discourage the use of the linear interaction approach on replication systems.

A sequel of this work will be a generic revision of existing recovery protocols based on constant interaction taking under account the results obtained in this work for recovery systems working in linear interaction replicated systems.

Chapter 11

Non-Transactional Amnesia Support

This chapter presents a generic recovery protocol for non-transactional replicated systems which manages accurately the amnesia phenomenon. This solution is based on the use of checkpointing and logging ideas widely used in distributed systems [51]. Moreover, the provided solution avoids the repetition of the work already performed and which follows the exactly-once semantics [72] (i.e. state changes with permanent effects). Later, it is included an amnesia overhead analysis for different process replication configurations in order to know the time cost associated to managing accurately the amnesia problem.

11.1 Introduction

Non-transactional –process– replication based on message-passing systems, focus of this chapter, has largely assumed to substitute crashed replicas by new ones transferring to them the whole state as recovery solution [69], based on the *fail stop* failure model [114], due to its easy management and their accurate behavior for systems with few state (typical case in process replication).

This chapter, instead, recommends the use of the *crash-recovery with partial amnesia* failure model [29] for process replicated systems with large state, where transferring the whole state as it is done in the first approach will imply a high cost. This solution is based on the use of checkpointing and logging ideas widely used in distributed systems [51]. Therefore, when a crashed node reconnects first restores the checkpoint and applies the received messages before the crash –amnesia recovery–, and second receives the broadcast messages during its disconnection.

As it has been demonstrated and formalized in Section 5.4 when this failure

model is adopted, recovery processes must consider the *amnesia phenomenon* in order to avoid its associated problems. Moreover, the *amnesia recovery* must follow the [72] (i.e. state changes with permanent effects). This chapter proposes a solution for all these issues.

The chapter is structured as follows. Section 11.2 adapts the theoretical properties presented in Section 5.4 for real non-transactional replicated systems. In Section 11.3 it is detailed the basic recovery schema, whereas in Section 11.4 it is explained how to deal with the *amnesia phenomenon* with a log-based recovery strategy for these systems. Section 11.5 details the overhead introduced for providing amnesia support in different process replication configurations. Finally, some related work is given in Section 11.6, and Section 11.7 concludes the chapter.

11.2 Recovery Information

The recovery information for non-transactional replicated systems has been already outlined in Section 5.6.1 with the figure 5.9. But, as long as it is expressed in Section 5.6.1 this recovery information is not achievable because it needs to checkpoint the volatile state after processing each delivered message.

Obviously, this theoretical checkpoint policy must be relaxed performing checkpoints in a more spaced way, as it is shown in figure 11.1.

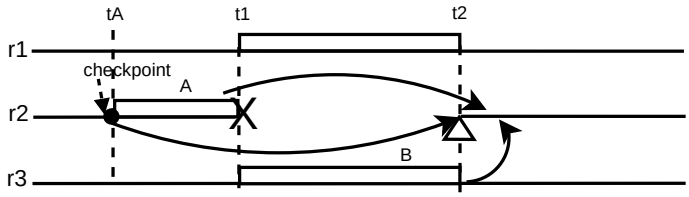


Figure 11.1: Real Log Recovery Information for Non-Transactional Systems.

Then, the original recovery process as it is detailed in Section 5.6.1 must be redefined in its two first steps as follows:

- Firstly, it needs the *checkpoint* block which recovers the volatile state at instant tA.
- Secondly, it applies the A recovery information block which contains in this case the messages processed after tA. Therefore, some of the A block messages will be reprocessed –those already processed between tA and $t1$– while other ones will be applied for the first time. Then, for these messages that must be reprocessed the system must avoid to repeat their permanent changes in order to avoid possible inconsistencies.

This behavior implies a redefinition of the properties presented in Section 5.4. The redefined properties are:

- *Prop. FS1'*: n_l must remember the last message processed before tA. In fact, that implies that the original $M_{i,m-s}$ is moved back to $M_{i,m-x}$ –fulfilling that $x > s$–, being $M_{i,m-x}$ the last message processed before tA.
- *Prop. FS2'*: the replicated system must maintain and provide a way for obtaining both the messages processed between tA and $t1$ in n_l and the messages that have been simply delivered in n_l before $t1$.
- *Prop. FS3'*: n_l must remember the volatile state VS_{n_l} at instant tA, after applying $M_{i,m-x}$.

Notice that these redefined properties can also be used for avoiding the problem formalized in Section 6.5, being only necessary to redefine *Prop. FS2'* as follows:

- *Property FS2'**: each node $n_l \in A$ must maintain and provide a way for obtaining its messages processed between tA and $t1$ in n_l and the messages that has been simply delivered in n_l before $t1$, instead of trusting in "the replicated system".

So, as the generic solution presented in this chapter fulfils the properties *Prop. FS1'*, *Prop. FS3'* and *Prop. FS2'**, this problem is also avoided.

In the next section it will extended the amnesia recovery focusing in the *Amnesia Recovery Process* –ARP–.

11.3 Recovery Protocol

The departing point for overcoming the amnesia problem is the basic recovery schema presented in figure 5.7. On the sequel it is presented how to perform the ARP in a non-transactional system considering different phenomena that will affect in the obtained result.

11.4 Amnesia Support

Once the general context has been presented it is time to describe how this amnesia recovery process is performed, and detailing the necessary information for the two levels considered in Section 5.2: *transport* and *replica*.

11.4.1 Transport/Replication Level

Amnesia implies at this level that delivered messages are lost at crash time. Therefore the idea is to store persistently these messages in each replica as soon as they are delivered –in an atomic way in the delivery process– as it is proposed in [38]. Moreover, this work way is similar to the adopted in logging recovery strategies for distributed systems [51].

Solving the amnesia problem at this level is a necessary but not sufficient condition for solving the amnesia at replica level. Thus, once it has been overcome it can be solved the amnesia at replica level.

11.4.2 Replica Level

For being able to manage the amnesia problem at replica level the system must know which work –meaning state changes– must be performed during the *amnesia recovery process*.

This *amnesia recovery process* combines the application of the last checkpoint and the received messages –persistently stored for avoiding the transport level amnesia– from this checkpoint [52]. Thus, each replica performs checkpoints periodically, storing permanently its in-memory state at this time. When a new checkpoint is created, the replica discards the previous checkpoint and the messages that lead the system to the current checkpoint, starting to log new incoming messages [52]. Therefore, the previous algorithm is refined to figure 11.2.

Recovery process:
1 - *Amnesia Recovery Process*
Restoring the Checkpoint
Reapplying messages
2 - *Update Recovery Process*
3 - *Current Recovery Process*

Figure 11.2: Intermediate Recovery Process

But there are some problems associated to the second phase of the amnesia recovery process, when applying the stored messages after the applied checkpoint, because some of this work must not be redone (e.g. persistent changes), because it has no sense. Therefore, it is necessary to know for each applied message its really performed changes, and more specifically changes that must not be repeated. And, for messages delivered but not applied before the crash it is necessary to know which work must not be performed during the recovery process.

Then, the next question that must be answered is which work must not be reprocessed –or processed– in the amnesia recovery. To do so, first some aspects that

must be considered for determining which state changes must not be reapplied are presented. The aspects considered are:

- in-memory or external changes,
- permanent or non-permanent changes,
- exactly-once semantics changes [72] or not,
- real time changes or not, meaning that these changes can only be performed –or only has sense to perform them– into the boundaries of an established window time interval (e.g. changes depending on an input data which changes its value outside the window time interval).

State changes can be classified attending to these issues, and depending on them it will be necessary to process them or not. Subsequently, some considerations about these issues are performed, their combinations, and how they influence in the necessity of reapplying changes or not.

- *In-memory* state changes are *volatile*, therefore all their changes are lost at crash time. And it has not sense to consider the *exactly-once* and *real-time* semantics for these changes, because they do not imply any action in external devices. Then, they can be reapplied –or applied–, without leading to undesired situations.
- *External* changes can be distinguished among those implying *permanent* changes and those implying *volatile* ones. In this case, considering *exactly-once* and *real time* semantics will imply different behaviours in each category.
  - Permanent effect changes, are always associated to the *exactly-once* or *real-time* semantics. In the first case, the system must ensure that they must be applied exactly once, so if a change has not been performed before the crash it must be applied later. Contrarily, if it has been already applied it must not be reprocessed, because its change is not lost. In the second case, the system must ensure that the change is processed only once within the established time boundaries. Then, if change has been processed before the crash it is not necessary to reapply it in the recovery process –remind that are *permanent* effects–, while if it has not been applied it must be only processed if the time window has not ended.
  - Volatile effect changes performed before the crash are lost, therefore in the recovery process they must always be reprocessed –those already performed– or processed –those that were not processed–. Only, those *volatile* changes following the *real time* semantics must be not performed if their time window has ended.

Therefore, summarizing the previous considerations the work that must be not reapplied:

- for *already* applied work:
  - *External* permanent changes, either following the *exactly-once* or *real time* semantics.
  - *External* volatile changes associated to the *real time* semantics if the time window has closed.
- for *non already* applied work:
  - *External* changes associated to the *real time* semantics once their time window has closed, either *permanent* or *volatile*.

Then, from a recovery point of view, how must manage each replica its already performed work attending to the previous characterization? The idea is to log each external access which must not be repeated, in this case only external accesses implying permanent changes. Notice that *real time* semantics are managed in a different way. Therefore, the log process for each external permanent access is done in two steps. First, it logs when the message/signal is sent, and second it closes the log when receives the process message/signal acknowledgement from the external device. If the used wire is reliable and the external device is enabled to store incoming orders/signals the second step, the acknowledgement requirement can be avoided, only being necessary the first step of the log process.

Afterwards, this log information can be used in order to avoid repeating already performed work in the *amnesia recovery process*, checking in the log if the task was already performed before the crash as it is shown in the algorithm presented in figure 11.3.

Recovery process:
1 - *Amnesia Recovery Process*
Restoring the Checkpoint
Reapplying messages
For each message check if it has been applied:
Yes - check its related log when reapplying
No - apply completely
2 - *Update Recovery Process*
3 - *Current Recovery Process*

Figure 11.3: Complete Recovery Process

It must be noticed that a problem will arise if the process does not receive the acknowledgement from the external device accessed, implying that it can not know if the external device has been able to perform the commanded order or

even if it has received the message/signal. In this case, a similar problem, to the *two generals problem* arises [85].

Other considerations that must be taken under account in the *amnesia recovery process* are the following ones:

- Another arising problem in this scenario relates to the fact that usual process replication does not work atomically as transactional systems do [63], supporting that at crash time in a replica some work associated to a message has been applied while other not. This problem has been already explored in some literature as [89], and some solutions have been provided. But, in our case, the previously provided solution for knowing which work must be reapplied in the *amnesia recovery process* overcomes this problem when a crash occurs. This stored work log information prevents the system for reprocessing previously performed permanent changes during the recovery process. Therefore, our previous algorithm can be modified to the algorithm presented in figure 11.4.

  Recovery process:
  1 - *Amnesia Recovery Process*
  Restoring the Checkpoint
  Reapplying messages
  For each message check its work log
  2 - *Update Recovery Process*
  3 - *Current Recovery Process*

  Figure 11.4: Final Recovery Process

  Obviously, it must be remarked that this message work log proposed policy can only be used for atomicity purposes in a crash context. In order to use it as a generic *undo* mechanism it would be necessary to log also performed volatile changes.

- It also must be noticed that for exactly-once operations it is necessary to distinguish between exactly once operations at replicated system level or replica level. In first case, in spite of a replica crash, if the replicated system has not stopped working one of the alive replicas will have performed the operation. In the second case, this work must be done in all replicas, therefore in crashed replica the recovery process must ensure its fulfillment.

- Some of these previous considerations, the ones related to permanent changes that are not lost at replica time, can be discarded if the performed checkpoint in the replica stores either the in-memory state –volatile– and its associated external state. Then, when restoring the checkpoint after the crash part of the permanent changes will be also undone, then when reapplying the messages it will be only necessary to consider the changes associated to real time semantics.

Once it is detailed how the amnesia problem can be avoided at the two levels it manifests, and have described the arising problems when performing the recovery process, in the following section the overhead introduced for supporting amnesia is analyzed.

11.5 Amnesia Overhead

In this section the overhead introduced due to supporting amnesia in process replication is studied. We will only consider the overhead during the normal work, in order to analyze how it influences in its performance. Overhead during recovery processes is not considered. We neither consider as overhead the regular operations on any replicated system; e.g., failure detectors, since the mechanisms needed by these low-level services are always in use and their overhead is not directly related to the amnesia problem.

The overhead must be studied at different levels: transport –replication– and replica level. At replication level the overhead introduced is related to the process of storing persistently the delivered messages. While at the replica level it will depend on several aspects. If the checkpointing solution is selected the overhead will be associated to the cost of generating the checkpoint, but it can be discarded if it is performed in a separated thread, with less priority than the replication work. The other aspect relates to the logging of external accesses that follow the exactly-once semantics as it has been commented before. Thus, the overhead at replica level will depend on the mean percentage of external accesses done in replica operations. Therefore, the overhead associated to the amnesia support at replication and replica levels is considered.

11.5.1 Replication Level

The overhead study at this level is performed considering the four main process replication configurations proposed in [121]: active replication (AR), semi-active replication (SAR), semi-passive replication (SPR) and passive replication (PR).

In AR all replicas receive the client request messages and process them, whereas in PR only a replica, the primary, receives the client request, processes the request and propagates the updates associated to this request to the other replicas –backups–. SAR allows to work in a non-deterministic way. All replicas receive the client request and process them like AR, but it works in a different way for non-deterministic operations. In this case, one of the replicas –*leader*– spreads its result of applying this request among the others –*followers*–. The last considered configuration SPR works very similar to PR in "good runs". But it presents two differences. The first one is that all replicas receive client requests but only one, the primary, processes the request. The second one is that when *backups* receive the spread update from the *primary*, first they acknowledge the

reception, and once the primary has received all acknowledgements, spreads another message among all replicas in order to really apply the previously spread update.

From a communications point of view: *AR* and *SAR* use total order broadcast (*TOB*), while *PR* and *SPR* utilize reliable FIFO (*R-FIFO*). In all cases the use of *sending view delivery* is emphasized.

The overhead study will be done detailing first the basic processing time (*BPT*) and after the processing time supporting amnesia (*PTSA*). Processing times will be expressed for each propagated operation in terms of spread time (*st*), persisting time (*pt*) and processing time (*Pt*).

The *st* depends on the communication guarantees provided. And it also depends on the message size for configurations which spread the updates associated to an operation.

As it has been said *TOB* is considered for *active* and *R-FIFO* for *passive*. For *TOB* the *fixed sequencer* is assumed in the broadcast-broadcast (*BB*) variant [44] implementation, which uses two reliable broadcasts for message propagation. On the other hand, the *R-FIFO* as it is considered for *PR* can be implemented using only one reliable broadcast for message propagation because there is only one sender. Then, if α is the maximum cost of a reliable broadcast, the *TOB* has a cost of 2α for spreading a message while the *R-FIFO* cost is α.

The client message request size (S) and the update propagation message (S_U) have their importance because some *GCS* have a maximum message size bound to spread, S_M. Thus, messages greater than S_M must be spread sending $\lceil \frac{S}{S_M} \rceil$ and $\lceil \frac{S_U}{S_M} \rceil$ messages.

The persisting time, *pt*, is expressed as $\beta + \gamma \lceil (S/k) \rceil$, where β is the upper bound time for write disk accesses, γ is the storing time of a block size message k, and S is the message size. It is considered that only one write disk access is needed for each message.

For *Pt*, the π and π_U value are considered, processing times for client request and update application respectively, which depend on the replication system load. Instead of being very difficult to model, anyway it is considered that any operation always fulfills the rule $Pt > pt$.

It must be noticed, that for *AR*, *SAR* and *SPR* the replication work starts as soon as the client performs its request because all replicas receive this request, while for *PR* it starts once the *primary* propagates the update. Moreover, for passive configurations we consider two different synchrony levels: hot passive on-processing (*HPP*) and hot passive asynchronous (*HPA*) as presented in [34].

From table 11.2 different observations can be extracted.

The first one is that *SAR*, *SPR (HPP)* and *PR (HPP)* do not present any kind of overhead because their processing times π and π_U hide the corresponding persisting times $\beta + \gamma \lceil \frac{S}{k} \rceil$ and $\beta + \gamma \lceil \frac{S_U}{k} \rceil$. The *SPR (HPA)* configuration only

Parameters	*Description*
α	maximum reliable broadcast cost
β	upper bound time for write disk accesses
λ	point to point communication cost
γ	storing time of a block size message k
S	client message request message size
S_U	update propagation message size
S_M	maximum message size
π	processing time for client request
π_U	processing time for update application

Table 11.1: Parameters.

presents the overhead corresponding to the persisting storage of the update propagation message with size S_U, while the overhead of the first broadcast message with size S is hidden by π. Other configurations present the overhead associated to store persistently the messages propagated for performing the replication work.

Notice that *SAR*, *SPR (HPP)* and *SPR (HPA)* perform their replication work spreading two messages: on one hand the client request message and on the other hand the update propagation message. In *SPR* configurations, as in active configurations, it is considered that the broadcast performed by the client is a part of the replication work, because the client spreads its request to all replicas. And this broadcast uses *TOB*. While in the *PR* this broadcast is not considered as replication work, because the client request is only received by the *primary*.

Another consideration is that *SAR* working in a deterministic way presents the same times as *AR*, while when it works in a non-deterministic way, the presented one in the table, presents the same cost as *SPR* (*HPP*). But *SAR* can be refined in order to work with the same cost as *SPR* (*HPA*), if it is allowed to answer to the client once it has broadcast the update propagation message to other replicas without waiting that they process it.

A problem that appears when comparing these configurations is the difficulty to compare S and S_U, assuming that S is the size of the client request message whereas S_U is the size of the update propagation message associated to this client request. Normally, S, the message size of the client request, will be smaller than S_M and k, while S_U will be greater or smaller depending on the updates associated to the process operation to perform. If a client request performs a lot of changes its corresponding update propagation message will be greater than S_M and k. Thus, in order to present a more accurate study it will be necessary to extend this work including an analysis of the workload profiles of different client requests.

Configuration		*Time*
AR	BPT	$(\lceil \frac{S}{S_M} \rceil + 1)\alpha$
	PTSA	$(\lceil \frac{S}{S_M} \rceil + 1)\alpha + \beta + \gamma \lceil \frac{S}{k} \rceil$
SAR	BPT	$(\lceil \frac{S}{S_M} \rceil + \lceil \frac{S_U}{S_M} \rceil + 2)\alpha + \pi + \pi_U$
	PTSA	$(\lceil \frac{S}{S_M} \rceil + \lceil \frac{S_U}{S_M} \rceil + 2)\alpha + \pi + \pi_U$
SPR (HPP)	BPT	$(\lceil \frac{S}{S_M} \rceil + \lceil \frac{S_U}{S_M} \rceil + 2)\alpha + \pi + \pi_U$
	PTSA	$(\lceil \frac{S}{S_M} \rceil + \lceil \frac{S_U}{S_M} \rceil + 2)\alpha + \pi + \pi_U$
SPR (HPA)	BPT	$(\lceil \frac{S}{S_M} \rceil + \lceil \frac{S_U}{S_M} \rceil + 1)\alpha + \pi$
	PTSA	$(\lceil \frac{S}{S_M} \rceil + \lceil \frac{S_U}{S_M} \rceil + 1)\alpha + \pi + \beta + \gamma \lceil \frac{S_U}{k} \rceil$
PR (HPP)	BPT	$(\lceil \frac{S_U}{S_M} \rceil + 1)\alpha + \pi_U$
	PTSA	$(\lceil \frac{S_U}{S_M} \rceil + 1)\alpha + \pi_U$
PR (HPA)	BPT	$\lceil \frac{S_U}{S_M} \rceil \alpha$
	PTSA	$\lceil \frac{S_U}{S_M} \rceil \alpha + \beta + \gamma \lceil \frac{S_U}{k} \rceil$

Table 11.2: Processing Times.

Anyway, it can be observed that introducing amnesia support does not alter among different configurations their original performance cost order. Moreover, considering that nowadays computers increase faster their computation power than increase the networks their bandwidth, it can be stated that the overhead introduced for supporting amnesia does not affect significantly the performance of these replicated systems.

11.5.2 Replica Level

The overhead introduced at replica level derives from the fact that each replica must log the external accesses which imply permanent effects. As it has been said, the log for each external access is performed in two steps, one before sending the signal/message, another after receiving the respective acknowledgement.

In order to calculate the overhead associated to an operation it must be known the number of external accesses which imply the permanent changes. Assuming n as this number, the log associated to this operation must have n entries, and the associated overhead for supporting amnesia for each of its external accesses is shown in table 11.3. The times are expressed in the same terms used for presenting the overhead at replication level, compiled in table 11.1. But in this case the λ term for communications cost is used, because external devices are assumed to be connected to nodes through point to point channels.

It must be noticed, that to quantify this overhead is very difficult because it will depend on the processing time in the external device π, value that will present a great variance among different devices (e.g. sensors or printers). Moreover, it also will depend on the message size –which can present great differences among external accesses for different devices–, and on the top boundary of the used

Configuration	*Time*
BPT	$\lambda\lceil\frac{S}{S_M}\rceil$
PTSA	$2(\beta+\gamma)+\lambda(\lceil\frac{S}{S_M}\rceil+1)+\pi$

Table 11.3: Processing Times for each External Access.

communications channel –different for each accessed device–. Thus it will be very interesting to categorize the external accesses attending to the accessed external devices.

As it has been said previously in 11.4.2, if communications are reliable and the external device can store received messages, the process only must log the sent message/signal event shorting very much the introduced overhead.

11.6 Related Work

Literature has largely treated the recovery problem in distributed systems, either transactional systems [12, 82, 79] or process replicated systems [15, 51]. But in spite of this fact, the recovery protocols presented for process replication have largely assumed the fail-stop failure model, where the proposed solution for solving the recovery problem is to transfer the whole state to a new replica.

As it has been commented in this paper, this proposal presents good behavior in replicated systems which manage few state, where to transfer the whole state does not imply a high impact in the system performance. But, when talking about systems with large amounts of data state, to transfer the whole state will imply a cost that can not be tolerated. Therefore, it is necessary to adopt another strategy for providing fault tolerance. This strategy implies to recover previous crashed replicas transferring only the information that they have lost, assuming the crash recovery with partial amnesia failure model, and being therefore necessary to deal with the amnesia phenomenon. Assumption, that has been already proposed for replicated databases [19, 38].

In the field of replicated database management, Wiesmann [122] already proposed the "successful delivery" concept in order to partially deal with the amnesia problem. It proposes that the GCS needs to maintain the delivered messages until the receiving process has acknowledged the successful application of the message effects. Such a solution can also be used for process replication as outlined in the current paper, but must be complemented with additional mechanisms (e.g., missed updates recovery).

Obviously, to transfer the whole state approach will avoid the amnesia problem, but as it has been said the idea is trying to avoid this approach when managing large state amounts. Moreover, this approach also increases its complexity if it needs to transfer the replica external devices state –being necessary to collect

and transfer it–, while our proposal avoids this complexity.

Other recovery strategy studied in the distributed systems literature –not focused on replicated systems– is the *checkpoint-based* rollback recovery [51], also known as *Rollback-Recovery* protocols. In this case, this strategy forces each node to store periodically a checkpoint of its state, then if a crash occurs when the replica becomes alive applies the last checkpoint performed obtaining a consistent state. But, this strategy does not support the amnesia phenomenon because it can exist a work gap between the last checkpoint performed in the replica and the work really performed in it after the checkpoint. If this solution is adopted in a replicated system the amnesia support in a crashed replica can be provided forcing a not crashed replica to transfer to the crashed one the changes performed after its last really applied change.

But, in [51], it is also surveyed a recovery strategy which combines *checkpoint-based* policies with *message logging*. This other approach provides support for the amnesia problem in a natural way. This is due to the fact that *log-based* recovery strategies, also widely studied in the literature, as it is the combination of *checkpoint-based* policies with *message logging* are the ones that can provide amnesia support considering the changes pointed out in this work.

On the other hand, literature has not studied extensively the problems associated to redoing the work in recovery processes for process replicated systems because they usually have preferred to transfer the whole state. Few papers, as [96], have considered it. But, instead of this fact, some research has been done in the area of generic distributed systems [71]. Contrarily, in regard to the exactly-once semantics a lot of work has been done [72].

The amnesia recovery support presented in this work follows the *end-to-end argument* ideas presented in [112]. In fact, in order to provide "perfect reliability" from an amnesia point of view at the top replication level, the work is divided into different levels. Assigning different work to the level that can do it more efficiently and effectively. Therefore, replication layer stores permanently the delivered messages while each application replica stores external accesses.

It must be said that a similar study related to the amnesia support and its associated overhead in transactional replicated systems is presented in [38].

11.7 Conclusions

This chapter presents a solution for managing accurately the amnesia phenomenon in recovery processes when replicated systems adopt the *crash-recovery with partial amnesia* failure model.

The proposed approach combines several new issues with already existent partial solutions in different contexts –as checkpointing and message logging for distributed systems, exactly-once semantics in replicated environments and end-to-end arguments for general system design– in order to construct a general way

for solving the amnesia problems when the *crash-recovery with partial amnesia* is assumed as a starting design point in process replication. As far as the author knows this proposal is the first attempt for solving the amnesia problem which considers all the previous aspects as a whole, combining and putting together their individual solutions for providing a generic approach which considers all these details. This work approach points out for avoiding redundancies and eliminating possible blackholes when considering all these aspects separately. The resulting strategy establishes a combination of checkpointing and message logging mechanisms for providing the recovery with amnesia support. Moreover, the message logging is performed at two different levels –replication and replica– following the general idea of "end-to-end arguments" and considering the *exactly-once* semantics and the *persistence* of the performed changes. Thanks to this work way the recovery process can be performed efficiently and avoiding consistency problems.

Finally, the overhead introduced by our proposed solutions for supporting amnesia has also been analyzed. At replication level, the study has considered the four main configuration types of process replication established in [121]. And at replica level it has been noticed the great overhead variability associated to this amnesia support.

Chapter 12

Amnesia Solution Analysis

In this chapter it is presented an analysis about how it behaves the solutions proposed in Chapters 5, 6 based on a simulation of a replicated database based on a certification replication protocol.

12.1 Introduction

The solution proposed in this thesis for overcoming the two problems related to the amnesia phenomenon introduces some overhead in the work of the used replication protocol. But, this overhead is not always constant; it will vary depending on different characteristics of the replicated system. Therefore, the goal of this chapter is to present how the overhead behaves when different characteristics of the replicated system vary: workload, message size, number of replicas, message processing time in replicas, etc.

This chapter is structured as follows. First, in Section 12.2 are presented the overheads introduced by the proposed solutions. Later, Section 12.3 introduces the simulation that has been performed for analysing the behaviour of the overhead. This behaviour is subsequently explained in Section 12.4. Related work is detailed in Section 12.5 while Section 12.6 concludes the chapter.

12.2 Amnesia Solution Overheads

The basic solution for solving the two problems related to the amnesia phenomenon described in the Chapters 5, 6 consists in persisting the broadcast messages atomically with the delivery process.

Obviously, forcing the system to persist the messages in the delivery process implies to introduce an overhead in the overall performance. At minimum this

overhead would be equal to the cost of persisting in physical storage the messages. This cost will therefore depend on the size of the message to store and in the transfer write rate of the used device. Basically, it can be said that the faster the storage engine the better the system will behave.

But, in some situations this overhead will be higher than the cost of persisting the message. This happens when the persisting process becomes a bottleneck, in other words, when the rate of incoming messages to persist is higher than the speed at which the storage engine persists them.

The messages that must be persisted in the system for overcoming the amnesia problems are the update transactions –assuming a message per transaction– that must be broadcast –update transactions that have not been aborted locally. Therefore, the rate of incoming messages depends on the transactions per second workload that can process the replicated system, the % of read transactions of this workload and the local abort rate of update transactions. High workloads, low % of read transactions and low rates of update transactions locally aborted can convert the persisting process in a bottleneck.

It must be noticed, that from a persisting point of view high workloads and low rates of local aborts are bad news but from a replication point of view are good ones. So, the ideal storing engine must be able to deal with high workloads and really low local aborts of update transactions.

Forcing the system to persist messages atomically in the delivery process implies that the replicas can not deliver the message until they know that all alive replicas have persisted the message. Thus, it is necessary that replicas exchange messages in order to notify themselves that they have persisted the message before delivering them. Obviously, this extra messages round implies another overhead in the system. But, it must be noticed that this message is really small because is simply a control message. Moreover, as some GCS use internally an ack for confirming the reception of the message, this ack can be delayed in order to inform also that the message has been persisted.

12.3 Simulation

For observing how the proposed solution behaves it has been simulated a transactional replication protocol based on certification for a wholly replicated database. It works in an update everywhere approach so all replicas can serve client requests; read transactions are processed only locally while update transactions are broadcast to all nodes –ROWAA approach– using a single message per transaction –constant interaction. As it has been said a read transaction is only processed locally so at commit time if there are no conflicts the node serving the transaction commits it and answers to the client. While an update transaction is first processed locally in the node that is serving the request and at commit time –if it has not been aborted locally– is broadcast using total order to all

nodes. In this case it is broadcast both the writeset –WS– and the readset –RS– in order to provide serializability.

The total atomic broadcast is implemented using a sequencer with two reliable broadcasts. In the first broadcast the sender spreads the message to all nodes, the second broadcast –a small control message– is used by the sequencer to notify the delivery order. A reliable point to point communication is used by the nodes in order to notify that they have persisted the message to the other nodes. Note, however, that such additional round –to the two ones used by the basic atomic broadcast considered– only uses small control messages; i.e., they do not carry the request or update-propagation contents of the original message, so their size is small and such message round can be completed faster than the contents-propagation one in the regular case (Considering, e.g., that in database replication protocols the broadcast messages propagate transaction writesets and their size may be as big as several hundred KB). The simulation has used network values appropriate for a 1 Gbps LAN.

In table 12.1 are listed the values assumed for different parameters in the simulation. The value of some parameters has been varied in order to analyse how behaves the solution.

Parameter	*Value*
Database size	100000 items
Transaction processing time in serving replica	50 ms
Transaction processing time in other replicas	20 ms
Net average delay	0.15 ms
Workload	30, 100, 300 and 500 TPS
Number of nodes	3, 5, 7, 9, 11 and 21
Total order broadcast message size	100, 200, 300 and 500 KB
% of read transactions	0, 10 and 20

Table 12.1: Simulation values.

Moreover, as the solution consists in persisting the messages broadcast in total order by the replication protocol two different secondary storage systems have been considered. On one hand a hard disk drive of 7200 r.p.m. (a.k.a. HDD) as basic storage system commonly found in low- and middle-range personal computers. On the other hand a solid state disk based on flash memory. There are disks of this kind able to store 16 GB and with a transfer rate of 90 MB/s for less than 400 USD (December 2007 prices). Table 12.2 summarizes the main performance-related figures of both disks. In the simulation, we consider that there is a disk entirely dedicated to GCS log management, apart from the one being used by the DBMS.

The tested configurations in the simulation are the result of combining the work-

Hard Disk Drive	
Parameter	*Value*
Positioning disk average time	5.5 ms
Rotation disk average time	4.16 ms
Write transfer rate	40 MB/s
Flash Memory	
Parameter	*Value*
Write transfer rate	90 MB/s

Table 12.2: Storing system values.

load, number of nodes, the message size and the rate of read transactions. The experiment measure the transaction completion time and consisted in simulating each configuration with each considered storing engine. An additional without persisting messages has been performed for each configuration. This last one is used as the base level for comparison purposes.

Each test consisted in completing 40000 transactions in the whole system. In the simulation it has been forced that there are not local aborts –so all update transactions must be broadcast– because this is the worst scenario from a persisting point of view. Once the simulation has completed all these transactions it is calculated the average of committing transactions.

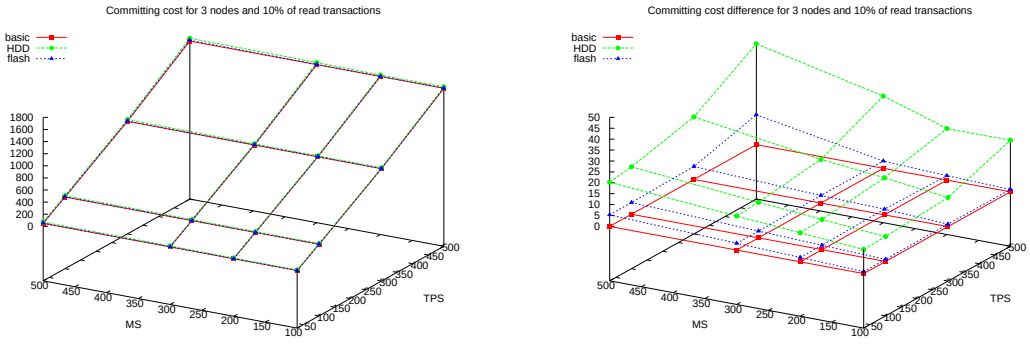
12.4 Results

For explaining the simulation results, different graphics have been prepared. They present the results without persisting –basic–, persisting in HDD –HDD– and persisting in flash memory –flash–. The percentage of read transactions used in these figures is 10 %, corresponding each one to 3, 9 and 21 replicas respectively.

Figures 12.1, 12.3 and 12.5 show the average completion time and persistence overhead in absolute values with two different graphics. In these figures, (a) graphics show the total cost for basic, HDD and flash storing policies. But, as it is really difficult to see differences in them (b) graphics have been attached. Those show the difference of HDD an flash in regard to the basic one. In both graphics, MS stands for Message Size (in KB) whilst TPS gives the workload in transactions per second. The vertical axis gives times expressed in milliseconds.

Figures 12.2, 12.4 and 12.6 depict the overhead in % introduced by the proposed solution for 3, 9 and 21 replicas respectively. As in previous figures, they show the results for the *basic* approach –without persisting–, for the *HDD* and *flash* storing engines. In this case the basic graphic only can be used as a reference point as it happened for (b) graphics in figures 12.1, 12.3 and 12.5.

Attending to these graphics on the sequel it will be explained how the proposed



(a) Committing time (in ms)

(b) Persistence overhead (in ms)

Figure 12.1: Results for 3 replicas and 10% read-only Txs.

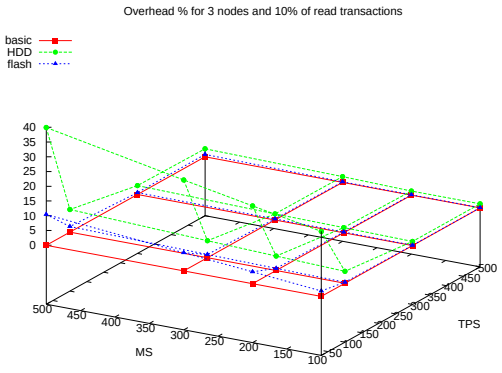


Figure 12.2: Overhead cost - 3 replicas and 10 % of read Tx.

solution behaves attending to several characteristics in the replicated system simulated.

12.4.1 Workload

For transaction workloads it can be observed from figures in a general way that for higher workloads the proposed solution presents: higher overheads in absolute terms at least for HDD –(b) graphics of figures 12.1, 12.3 and 12.5–, but lower overheads in percentage terms –figures 12.2, 12.4 and 12.6 either when using HDD or flash.

First of all, someone can state that the persisting overhead in absolute terms must not increase with the workload if the size of the messages to store does

not increase, contradicting then the trend observed for HDD in (b) graphics. This statement would be right if the write transfer speed of the storing engine was always high enough to not become a bottleneck when the workload rises. And (b) graphics show that; HDD becomes sooner a bottleneck –increasing then its overhead in absolute terms without increasing the message size– as it is the slowest storing engine tested, while flash memory –which has higher write transfers rates– does not show this trend at least for the tested workload ranges. As it can be inferred this trend manifests better for high message size values –worse cases– as shown HDD tendencies in (b) graphics.

It is also necessary to explain the apparently –at first glance– contradictory behaviour of the overhead in absolute –increase– and percentage –decrease– terms when the workload rises. The explanation for this phenomenon relates to the fact that the overall replicated system becomes a bottleneck for a certain workload, increasing the response time when the workload grows from this threshold level. From this assert, it can be stated that the tested replication protocol behaves in a bottleneck way because its time processing cost increases with workload when messages are not persisted as it can be seen in (a) graphics for the basic approach. Therefore, the persisting overhead decreases in percentage –even when it increases as it happens for HDD solution– when the workload rises because the basic cost of processing transactions grows more.

At this point, it can happen that the replicated system bottleneck hides or avoids to manifest in all its magnitude the persisting overhead. In fact, observing the (b) graphics of figures 12.1, 12.3 and 12.5 and figures 12.2, 12.4 and 12.6 it can be seen how when the number of replicas increases, the overhead increases both in absolute and relative terms –for the same workload and message size– because the effects of the replicated system bottleneck are lower.

Finally, it must be also specified that the persisting overhead depends on the workload of messages to persist which in spite of being related to, is different from the workload of incoming requests to the replicated system. The former one depends on the incoming workload, the rate of update transactions of this workload and the rate of transactions aborted locally.

12.4.2 Message size

For this parameter there can not be observed in figures any unsurprising result; when the message size grows the persisting overhead grows both for HDD and flash storing engines. And, obviously, it manifests in a sharper way for the HDD storing engine than for the flash one, and already expected result as the second one has a higher write transfer.

Moreover, from figures it also can be deduced that the message size has an important effect in the probabilities that the persisting solution becomes a bottleneck. And it affects in the following way, when higher the message size the more probabilities has the persisting solution to become a bottleneck.

12.4.3 Number of replicas

The number of replicas affect to the persisting overhead in an indirect way. As it has been said previously, when the system has more replicas it can process higher workloads without becoming a bottleneck, then the persisting engine must manage higher workloads without the barrier provided by the replicated system when it acts as a bottleneck so its introduced overhead manifests more in percentage terms.

In regard to the bottleneck phenomenon, the persisting solution has more probabilities of becoming a bottleneck when the system has more replicas as it can manage usually higher workloads, forcing the storage engine to persist a higher rate of incoming messages.

12.4.4 Storing engines

From the figures it can be said that any storing engine introduces some overhead in the replication work, being lower when higher its write transfer rate is. So, the flash memory introduces lower overheads than the HDD solution for any tested replication configuration both in absolute and percentage terms. In fact, it can be seen how the flash storing engine never becomes a bottleneck as it happens with the HDD solution as graphics (b) from figures 12.1, 12.3 and 12.5 demonstrate for any of the simulated workloads.

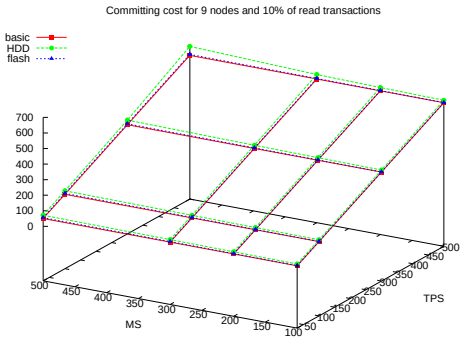
It must be said, that having a fast storing engine is not only interesting because it introduces lower overhead but also for decreasing the probabilities of becoming a bottleneck as it is seen in (b) graphics.

12.4.5 Other parameters

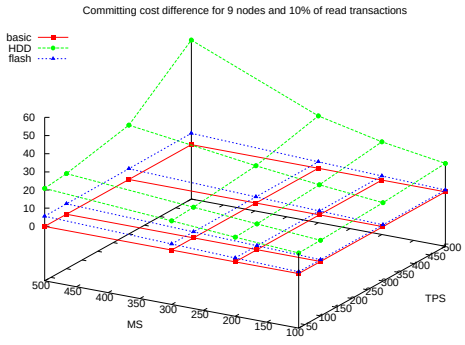
In the simulation other parameters have been considered: % of read transactions and % of local aborts. But, it can be said that the observed evolution was the expected one. The overhead decreased either when the % of read transactions increased or when the % of locally aborted transactions increased, because it implied less messages to persist.

12.4.6 Summary

From the results obtained in the simulation different conclusions can be stated. The first and obvious one is that any persisting solution introduces some overhead in the system. It is also important to say that this introduced overhead depends on the combination of several static and dynamic characteristics of the replicated system. This overhead in absolute terms increases with the workload, the message size, and % of write transactions, while in % the worst cases are those in which the system is able to process in a fast way the transactions.



(a) Committing time (in ms)



(b) Persistence overhead (in ms)

Figure 12.3: Results for 9 replicas and 10% read-only Txs.

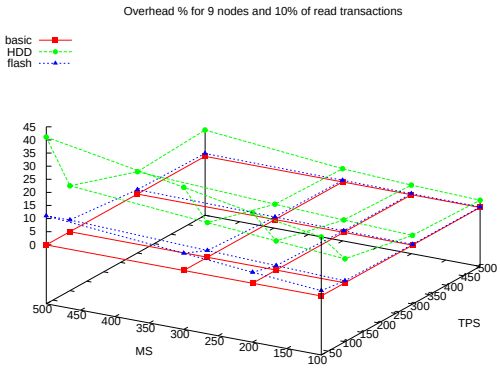
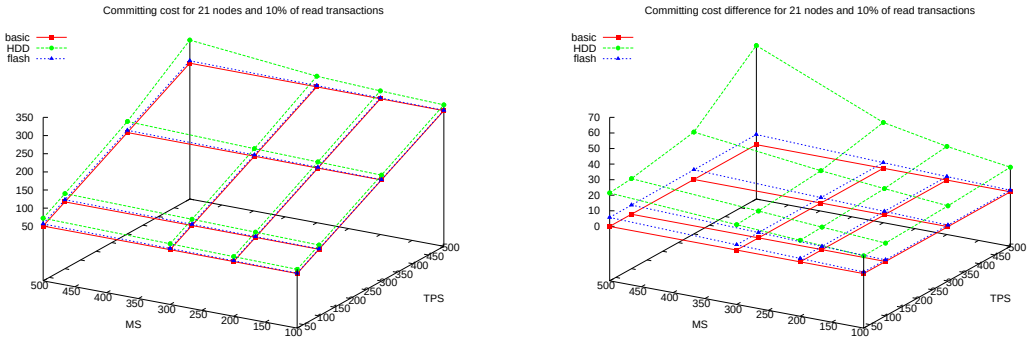


Figure 12.4: Overhead cost in % - 9 replicas and 10 % of read Tx.

Naturally, the overhead would be lower when faster is the used storage engine. And that the worst overhead cases appear when the persisting process becomes a bottleneck because in this case the overhead introduced by the persisting process is higher than the a priory expected cost of persisting –considering the message size and the write transfer speed. Evidently, as it has been said the phenomenon of becoming a bottleneck is more probable for slow storing engines. At this point it also must be remarked how the flash memory does not only become a bottleneck but even it maintains the introduced overhead in a low level range for usual workloads.

Moreover, it has been seen how the worst conditions from the persisting point of view are a system processing high workloads, with great message sizes, low % of read transactions and slow storing engines. In regard to the message



(a) Committing time (in ms)

(b) Persistence overhead (in ms)

Figure 12.5: Results for 21 replicas and 10% read-only Txs.

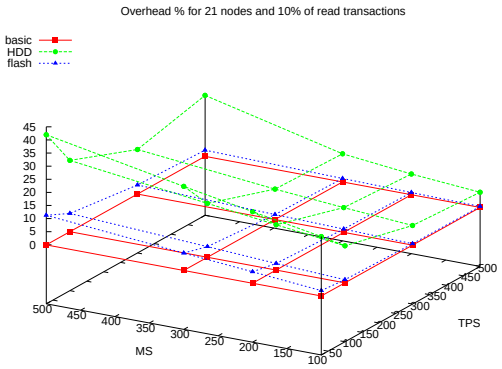


Figure 12.6: Overhead cost in % - 21 replicas and 10 % of read Tx.

size, it must be advanced that the replicated protocols which propagate the writeset and readset would behave worse from the storing point of view than those transferring the operations to perform. Moreover, in the particular case of certification it would behave better if instead of using serializability it would have used snapshot isolation, because in this last case the readset must not be transferred by the replication protocol.

Another obtained conclusion from the simulation is that sometimes when the replicated system gets overloaded it can hide or decrease the phenomenon of saturation in the persisting process. This conclusion can be converted in a guideline to follow when designing a replicated system and this rule will state that the capacity of processing transactions by the replicated system and the capacity of persisting messages by this replicated system must evolve parallely.

The idea is that having a replicated system that can manage high workloads is not worthy if the storing process acts as a bottleneck because the latter will decrease the overall performance. And the rule also works in the other way, so investing money in fast storing engines is not profitable if the replicated system can not deal with high workloads. In this last case, it only would be interesting if the size of message transactions is quite large. Other parameter to consider is the average time cost of processing transactions in nodes because when higher it is the replicated system saturates sooner, hiding therefore the overhead introduced by the persisting engines.

It must be remarked, that all these results have been obtained with a 0% aborts, so all the incoming update requests to the replicated system must be broadcast among the replicas. If the abort rate in the serving replica is higher the number of transactions to broadcast decreases, diminishing then the demand to the storing process, reducing therefore the probabilities of reaching the saturation point for the persisting process.

12.5 Related Work

There are many replicated database works in the literature that give some numeric results about how they behave. The problem is that most of them have not considered the amnesia phenomenon and its possible associated consistency problems that arise when working under specific conditions. So, it is really difficult to compare the results obtained with this simulation with these other works.

Among all these works it is interesting to point out the following two ones. The first one because gives the results associated for the same type of replication protocol. While the second one because the way it implements the total order broadcast can avoid in an straightforward way the problems associated to the amnesia phenomenon.

In [88] authors show the results of a replicated system based on a certification replication protocol. It uses also an update everywhere approach broadcasting only update transactions in a constant interaction way –single message per transaction– through a total order communication primitive provided by a GCS. The problem is that these results can not be compared with the ones obtained here, because in their case they provided Snapshot Isolation isolation level instead the Serializable one used in the simulation. Moreover, they do not specify the GCS they have used and the guarantees it provides from an amnesia point of view.

Authors of Sprint Middleware [18] provide some results of their solution which main characteristic is to take advantage of being an in memory database –IMDB. This solution uses as atomic broadcast protocol the Paxos [84] protocol. This Paxos protocol forces the system to persist messages in order to establish the

total order, therefore it can avoid in a straightforward way the basic amnesia phenomenon. But, these results can not be compared with the simulation results because their system is a partial replicated system.

Finally, this simulation can be seen as a continuation of the work started in [41], measuring the overhead introduced when forcing the system to persist atomically the total ordered broadcast messages in the delivery process. Moreover, this solution fulfills completely the properties defined by [93].

12.6 Conclusions

As it has been seen in this chapter the overhead introduced by the proposed solution varies with the number of nodes, workload, % of local aborts, % of read transactions, average time for processing transactions and last but not least message size. Thus, if this solution must be applied a proper study of the most common values of these variables in the replicated system must be performed in order to determine the minimum write transfer speed in order to avoid the saturation of the storing process.

Anyway, for minimizing the overhead cost the best option is to use the fastest storing engines. As it has been seen solid memory is a good option, but for systems managing big messages and high workloads there are other solutions as *ioDrive* [56] which provides 600 MB/s write transfer rates.

Part III

Related Work

Chapter 13

Related Work

In this chapter it is included some generic work related to recovery protocols or mechanisms.

13.1 Replicated Systems Recovery and Failure Models

The development of first computer networks in the sixties and seventies of XXth was the starting point of a new branch research area in the computer science world: distributed systems. These distributed systems have been used for very different purposes being one of those to increase the systems dependability through replication, what has been largely denoted as replicated systems.

Replicated systems increased the levels of dependability providing to the system fault tolerance. Therefore, a huge work was performed as demonstrated in [106, 115, 29] for analysing the possible failures that could arise in distributed systems. In these works authors categorized the failures and defined failure models in order to simplify their management. The idea was that distributed system designers in the design process selected one of the proposed failure models, assuming that their system only has to handle these kind of failures and considering that other failure occurrences in the system were negligible.

A great variety of failure models have been presented in the literature from the most severe one, *byzantine*, to the simplest one, *fail-stop*.

Replicated systems have traditionally assumed the *fail-stop* failure model as [12]. Its main advantage is simplicity because nodes only fail by halting remaining forever in this state. Therefore, failed nodes are discarded. This implies that if the original *k-fault* tolerance level wants to be maintained so each discarded replica must be substituted by a new one, being necessary to transfer to the

added replica the whole state. This way of maintaining its fault tolerance is its main drawback, when talking about replicated systems managing large states. In this scenario, the recovery processes are very expensive. In order to provide more efficient and costless recovery algorithms researchers have adopted the *crash recovery with partial amnesia* failure model as in [82, 20, 19, 6, 79, 70] for replicated systems managing large states. In very few cases have been adopted the fail-stop failure model as in [87].

13.2 Recovery Survey

A survey of some recovery protocols for database replication techniques is presented in [57]. In this paper authors classified recovery protocols attending to several characteristics. These characteristics included the characteristics of the replication protocols the recovery protocols work for: server architecture, server interaction, transaction termination and update propagation, and the own characteristics of the recovery protocols: transfer model, concurrency control during recovery and recovery-work distribution.

This paper also notices the tight relation between a replication protocol and the recovery protocol designed to work with it. This relation lies in the fact that the recovery information generation depends on the possibilities provided by the replication protocol. But, this survey does not consider the *amnesia phenomenon* and its correct management.

But, among all these proposals that have assumed the *crash recovery with partial amnesia* failure model it has been detected that they do not manage correctly what we call amnesia phenomenon. This is due to the fact that research in this area is usually more concerned about building replication algorithms which provide good performance levels, than ensuring the correctness of their recovery algorithms which are seen as an accessory in their replicated systems. Some of these authors made some assumptions that implied an incorrect handling of the amnesia phenomenon. And this misbehavior can lead to undesired situations.

In a generic way, it can be concluded that literature of replicated systems has been mostly oriented for designing replication protocols, while the recovery protocols have been mainly considered as a second step in spite of the very interesting works theorizing and formalizing failures for these systems as [106, 115, 29].

Anyway, literature has studied and analysed several aspects both of replication protocols tightly related to recovery protocols and recovery protocols themselves.

13.3 Successful Delivery

First of all, it is necessary to point out the importance of the [122] paper, as it has been commented previously. In transactional replicated systems the amnesia phenomenon arises due to the fact that some recovery protocols have been designed assuming that delivered messages are also messages correctly processed. But this assumption is not correct as demonstrated in [122]. This is the reason for highlighting this paper, because this assumption –that [122] shows to be false– combined with the *crash recovery with partial amnesia* failure model can lead to inconsistent states after recovery –if what it has been called in this thesis *amnesia phenomenon* is not correctly handled.

But, in [122] authors proposed the *successful delivery*, which ensured that a message is maintained by the GCS until it has not been processed by all node receivers. Therefore, a message only can be discarded by the GCS when all receiver nodes have acknowledged the message processing to the GCS. The background idea is that GCS can deliver multiple times a message to a node but it can only be delivered successfully once. To do so, they proposed the *end-to-end* atomic broadcast primitive which ensures that each message is successfully delivered once. This delivery mechanism provides an easy way to support amnesia if the acknowledgement is returned once the application has processed the message and has persisted its updates, being then an alternative to the general recovery protocols presented in papers [38] and [37].

13.4 Atomic Broadcast Based On Consensus

In regard to these works it must be pointed out the literature concerning algorithms for providing atomic broadcast based on consensus for the crash-recovery model [84, 109, 91]. In these works authors specify the necessity of perform some kind of persistence in order to avoid problems at recovery time. All these solutions also have in common that they not need to use the view concept as a synchronization point, in fact they use the consensus round concept as synchronization mechanism.

In regard to Paxos protocol [84] which can be used to implement an atomic broadcast based on consensus the author proposes as synchronization point the last decision –delivered message– written –meaning applied– in a *learner*. This approach therefore overcomes the basic amnesia problem presented in Chapter 5. Moreover, as it forces the *acceptors* that participate in the quorum for a consensus instance to persist their vote –message to order– as previous step to the conclusion of this consensus instance –which will imply the delivery of the message– it can also avoid the problem presented in Chapter 6 in a straightforward way. So, if a learner crashes, losing some delivered messages, later when it reconnects it only must ask to the system to run again the consensus instances subsequent to the last one to which belong the last message it has been applied

in its application, relearning then the messages that the system has delivered after. But, this forces the acceptors to hold the decisions they have adopted for a while till all learners does not acknowledge the correct processing of the message. Anyway, the proposed solution presented in this paper also can be applied to the learners in order to avoid the necessity of replaying consensus instances.

Different systems have been developed using the basic ideas proposed in [84] as Sprint [18] and Chubby [17]. Sprint is a system for providing partial replication using in memory database for increasing the performance of the replicated system. While Chubby lock service is used by Google for loosely-coupled distributed systems, providing what they name *advisory locks*.

Another interesting work is [90] where they proposed a failure detection mechanism for the crash-recovery model and different ways for solving consensus in these systems. In this paper, they demonstrated that if the number of correct processes was lower than the number of bad processes only accessing stable storage could solve the consensus problem, contrarily if the number of correct processes was higher than bad processes, consensus can be solved without persisting information. If access to stable storage was necessary their proposed protocol forced each process to access twice per consensus round: one at proposing time –when the process proposes its decision– and at deciding time –when the process receives the value decided for this round. It must be noticed that the authors considered that the actions of persisting can not be performed atomically with the send or reception of messages.

Authors of [109] make mandatory in their proposal that for any consensus round each replica has to persist its proposed messages –the step before agreeing the order for this round. Later, the basic approach allowed in the recovery process of a crashed node to replay all the consensus rounds avoiding rebuilding therefore the agreed messages queue. Thus as first variant allowed to overcome the basic amnesia problem presented in Chapter 5, –it does not use view synchronization– and also affords the amnesia problem presented in Chapter 6 because the whole queue of delivered messages is rebuilt in the recovering node. For solving this last problem, it is only necessary that the recovering replica remembers which was its last applied message, applying then the messages that are subsequent to this one in the rebuilt agreed messages queue in the replaying phase.

They proposed an optimization that consisted in persisting periodically the last consensus round executed and the agreed messages queue until this consensus round –it included– to avoid the necessity of replaying all the consensus rounds from the very beginning. But, this process of persisting periodically must be performed atomically, because if the consensus round stored persistently does not agree with the agreed messages queue persisted some inconsistencies could arise and the amnesia problems can arise again. Another improvement proposed consisted to checkpoint the application state instead of maintaining the log of delivered messages, substituting the later one by the application checkpointed state that correspond to the the actual set of delivered messages.

In [91] authors considered that the existing specifications of atomic broadcast were not completely satisfactory in the crash-recovery model. To overcome this problem they build their atomic broadcast specification using three primitives: *abcast*, *adeliver* and *commit* –new one–. This specification also divided the process state into two different states: the application state and the state at the atomic broadcast protocol. And a checkpointing policy was used to avoid inconsistencies between the application state and the atomic broadcast algorithm state when a crashed replica reconnects. This checkpointing policy consisted in persisting both states coordinately with certain periodicity. The application was the one which triggered the checkpointing process: first it checkpointed the application state and later instantiated the commit primitive which checkpointed the state at the atomic broadcast algorithm. So, when a replica reconnects after a crash it restarts from the last checkpointed state (both application and atomic broadcast protocol) without inconsistencies between them.

At this point, they proposed two variants: an uniform –more-consistent– and a non-uniform –more efficient– one. The first one was forced to access to stable storage at the beginning of each round consensus, while the other only at commit time. Therefore, the first one –uniform– could replay the consensus rounds not persisted in the checkpointed state avoiding the amnesia phenomenon, while the second one only can be used for these applications that can afford losing uncommitted parts –in this case the amnesia is not completely avoided. Anyway, it must be pointed out that neither of these two proposals could manage the problem of combining amnesia with the majority progress condition presented in this thesis. This is due to the fact that the replay phase –in the uniform solution– needs that all originally proposed values in the consensus round to be replayed must be available –condition that is not fulfilled.

The authors of [91] concluded the paper combining their two atomic broadcast specifications –uniform and non-uniform– for crash-recovery failure model with consensus with access to stable storage and without access.

13.5 Recovery Optimization Techniques

As one of the criteria when designing recovery protocols is to reduce the amount of information to transfer in the recovery process –in fact this is the basic reason for adopting the *crash recovery with partial amnesia*– in order to obtain more efficient recovery processes, some papers in the recovery literature have focused to this end.

In [82] authors proposed a set of recovery protocols –version-based– for transactional replicated systems, that were denoted *reconfiguration algorithms*. They started from the simplest one, transferring the entire database, and went on introducing some optimizations for decreasing the information amount to transfer or reducing the time for the recovery process.

The first optimization, *Checking Version Numbers* consisted in marking each

data item with the *transaction identifier* of the last transaction that modified it. Later, in the recovery process, the node being recovered notified which was its last committed transaction –*cover transaction*–. Then the recovery process simply had to transfer to the outdated node the data items modified by transactions whose transaction identifier was higher than the cover transaction identifier of the recovering node. In this way, the amount of information to transfer was restricted to the modified information.

The second optimization consisted in maintaining in a single table the information about which was the last transaction that modified each data item. The basic idea was to minimize the time for scanning in the database which data have been modified by transactions whose identifier is higher than the cover transaction identifier of the node being recovered. So, in the previous algorithm the scan was on the entire database, whilst in the second case was in only one table.

Another optimization was based on using the log that each database maintains for self recovery purposes, if this log maintains an entire physical after-image of each modified object. The main advantage of this solution consisted in that it does not block the process of new incoming transactions. In this case, the recovery algorithm simply scans backwards the log of an updated replica, sending to the recovering node the values of the data items modified by transactions whose identifier was higher than the recovering node cover transaction identifier.

Other recovery protocols tried to support both version-based and log-based recovery techniques, in order to select each time the most efficient one. This is the case of [19]. In this paper, the authors proposed to use a log-based technique for short term failures –few lost changes– and a version-based technique for long term failures –many lost changes–. The background idea is very interesting and follows the same principle of those papers as [75, 21] which propose metaprotocols for supporting multiple replication protocols selecting each time the one that best fits the changing replicated system environment.

The main problem of this solution resides on the fact to determine the threshold used as limit value for selecting one technique or another. Therefore, it is important to perform an exhaustive and experimental study of this threshold.

Moreover, this strategy can be refined if different recovery protocols are provided for each technique –version-based and log-based–, therefore the selected recovery protocol can fit better the performance necessities.

Other strategies try to minimize the set of information to transfer in the recovery process compacting it. Authors of [58, 103] apply this strategy in version-based and log-based techniques respectively.

In [58] the recovery information associated to forgotten state is generated and maintained in a database table, inserting a row for each new installed view –where there are failed nodes–. The table has three columns: one for the view identifier, another for the identifiers of the nodes that have lost the changes performed in this view, and other where the identifiers of the items modified

during this view are stored –this column forbids repeated identifiers, being once is enough–. Then, when a data item is modified and there are failed nodes, its identifier is stored in the corresponding view entry in the table. So, it is possible that the same data identifier appears in different rows of this table, because it has been modified in different views where there were failed nodes. Later, in the recovery process –performed view per view– the recoverer node sends for each view the current value of all data whose identifier is contained in the row view.

Then, the compacting technique in this recovery protocol works as follows. Each time a new data item identifier is going to be inserted in the row of the current view a double checking process is performed. First the system checks if the identifier already exists in the row, if it already exists is discarded, ending the checking process. But, if this is not the case, the system checks if this identifier is present in a previous row view. If this is the case, and the set of failed nodes in this previous view is included in the current set of failed nodes, the identifier is deleted from the previous view row and inserted in the current one. Obviously, this compacting technique decreases the information to transfer. Some simulation results are presented in [58], being noticed that it depends basically on the rate of data items modified in a repetitive way in different views.

Authors of [103] applied a compacting technique for a recovery protocol designed for a certification-based replication protocol. In this case, the recovery protocol transfer to recovering nodes the transaction writesets they have lost due to its disconnection. Then, in the original protocol the same data item could be transferred in many different writesets, but being only necessary to transfer its last value. Therefore, the authors applied a compacting technique that deletes from writesets obsolete values. In the experiments performed for this paper they also found that depending on the workload of the replication system only the compacted version was able to complete successfully the recovery process.

13.6 Recovery in Replicated Commercial Systems

Another important related work that must be considered in the thesis is how commercial systems manage recovery processes, and detect similar problems to the amnesia phenomenon.

Most extended databases [95, 92, 55, 105, 73] when working in a centralized way provide either backups or dump strategies for performing recovery processes. But, it is also necessary to know the recovery strategies they provide when working in a replicated way. For instance, MySQL [95] provides different replication configurations which use a master binary log as source of information replication. For each slave the master keeps track of the last position in the binary log that has been replicated, updating it after the slave correct processing.

If the master crashes it can happen that the master has not flushed the binary log position for a slave, having then a wrong position for this slave. In this case, the replication is stopped. In order to avoid this problem, the [95] proposes to set the master binary log synchronization parameter to the value which implies the highest rate of binary log flush, decreasing then the probabilities of not having flushed the binary log at crash time but without ensuring it at all. Thus, the system can not ensure that the amnesia phenomenon is correctly managed. A similar situation arises if a slave crashes or shutdowns uncleanly.

Moreover, it also would be necessary to study how behave enterprise information systems as [27, 64], or version systems working in a replicated way [26]. The [27] provides ways for replicating servers, but encourage to do it when there is less activity on the network. Therefore, this system uses a sort of lazy replication. When the replication process is scheduled, the replicator constructs a list of documents –those who must be replicated– in the source database that have changed since the last successful replication. The time this process is started is recorded in a replication history so that succeeding replications do not process changes that have been replicated in previous replication steps. Then the process starts. If the process succeeds the replication history is updated in the involved replicas. If it does not, the replication history is not updated, then in the next replication process will start from the same point time. As it can be seen, the [27] avoids the amnesia phenomenon problem, but it must be noticed that the replication it provides is too restricted.

Tivoli clustering [64] is a small example of the multiple clustering solutions provided by IBM. In [64] authors explain different ways of providing high availability with Tivoli Software Solutions. They consider two ways of providing high availability: hot standby and takeover. The first one consists in having an idle backup that starts to work if the primary fails, while in the takeover solution all servers are active and if one of them crashes one of the clustered servers will take the original workload assigned to the crashed node. But, they do not replicate services therefore if a node crashes the node that assumes the work of the crashed node it simply restarts from the last consistent state reached by the failed one. Therefore, in this case inconsistencies due to the amnesia phenomenon can not arise.

In [83] authors comment the clustered configuration supported by Parallel Sysplex, which is a shared cache model. They define the different components that compound a Parallel Sysplex. Among these components from a recovery point of view can be pointed out the Sysplex Failure Management and the Automatic Restart Management. The problem is that the work way provided information is really generic so it is really difficult to figure out if they can occur state inconsistency problems due to possible amnesia phenomenon. Anyway, as they use a shared cache model that is a refined version of shared disks, at the end they store the information in the same physical disks so different evolutions as can happen in the system model considered in this thesis can not occur.

Another field that must be considered of commercial systems are clusters of

application servers. Therefore, it would be interesting to know how they manage recovery processes. Among these application server clusters can be highlighted JBoss [77], or WebSphere [110].

JBoss [77] supports different ways of clustering. Attending to its manual the cluster scenario is a set of servers –whose state can be replicated or not– which work with a single shared database –all servers are client of the same database server.

If a load balancing policy wants to be used the state of the JBoss server instances in the cluster will be independent –not replicated–. The problem in this basic configuration is that if a server crashes all its client sessions are lost. To support failover the JBoss documentation [77] suggests to use replication of state servers. In this case if a server crashes, the clients of this server can be redirected to another server in the cluster which has replicated sessions. This replication among the servers of a cluster implies some extra communication work which is performed using the GCS JGroups [78].

But, JBoss manual [77] does not say anything about how the recovery of a crashed server of a cluster is performed. It simply says that JGroups provides a MERGE directive which joins the servers of a cluster if they previously got split –i.e. network partition– once the communications can be reestablished. Obviously, this merge process will also be triggered when a crashed server reconnects. However this merge process does not consider state transfer leaving this state management to the application deployed in the cluster. Anyway, as all the cluster servers share the same database server no different evolutions can happen at least at the durable state level. Notice, moreover if the database instance stops working the overall system also stops.

It must be said, that in the JBoss [77] documentation they do not consider a similar scenario to the one proposed in this thesis, that would be one where there is not server state replication and each server in the cluster has its own copy of the same durable information.

WebSphere [110] documentation also talks about clustering for providing high availability. In this case they define 6 different levels of Websphere system availability and differentiate between process and data availability. Upon the 4th level –the ones that provide higher availability– they also replicate the underlying database and other persisting storage devices in the system. Therefore, it is under these configurations where possible inconsistency problems associated to the amnesia phenomenon can appear. The problem is that from the information provided in this manual of high availability it can not be deduced whether the amnesia problem arises or not, or if in the case of happening it is correctly managed. This is because this manual presents many different cluster configurations with different software and architecture deployments but does not precise important details about communications and how recovery processes are performed.

The Weblogic application server [117] provides both replication in the middle tier

and replication at the database level. For this last one they define MultiPools –that can be used either for high availability or load-balancing policies– each one accessing a different DBMS instance. Obviously, as databases in this last case are replicated problems of state inconsistency can arise if recovery processes are not performed accurately. Anyway, from the information provided in the documentation it can not be figured out if that can happen or not.

Nevertheless, many commercial application servers provide replication –clustering– of server state as [77] and [98], but rely on shared persisting storage resources. So, in these configurations different evolutions of persistent state can not occur due to amnesia phenomena in the recovery process.

An interesting paper related to fault tolerance in middleware servers is [119]. Its authors comment that two common techniques for providing high availability in middleware servers are: replication and log-based recovery.

In regard to replication solution they explain that it implies to duplicate the infrastructure and introduces a relative overhead due to the communications that must be performed between replicated servers but avoid outages completely. They propose in this paper a log-based recovery for saving the middleware state –session and shared variables– when a crash occurs. They argue that it is a relatively cheap technique. As the servers can work in a collaborative way, when a server crashes and recovers, later other –non failed– servers of the same service domain must check if their state is consistent with the state reached after the recovery in the crashed server. The idea is to provide inter server consistency avoiding orphan messages. This can imply sometimes a roll back process in a non crashed server for ensuring the inter server consistency.

Later they perform several experimental results where compare their solution with other solutions including: persisting sessions in a local DBMS or storing session states in the main memory of a different computer which are commercial approaches for session state recovery.

It must be noticed that their proposed solution considers most of the things told in this thesis, but applied to the recovery of state servers. On one hand, when they use optimistic logging –between the servers inside a domain service– sometimes after a recovery process some sessions of non-crashed servers can become orphans –in other words are inconsistent– in regard to the state reached in the recovered node. Therefore, these orphan sessions must be rolled back to avoid such inconsistencies. On the other hand, when they use pessimistic logging –communications outside the service boundaries– orphans can not be created because messages are flushed before generating an event that can become orphan. So, after a recovery process can not appear inconsistencies among servers in different service domains. It must be precised, that they do not tell anything about the necessity of persisting messages atomically in the delivery process when using a pessimistic logging approach.

Authors of [101] present a consistent and scalable cache for J2EE application servers. They claim that current infrastructure for providing high availability

has been oriented to replicate in many cases only a single tier becoming a bottleneck the non-replicated tier. Therefore, they provide a replicated cache that considers both middle-tier and back-end tier and which guarantees snapshot isolation level. To do so, they consider that each replica contains an instance of the application server and an instance of the database. In the paper they do not explain anything about how the recovery process in a crashed replica is performed because it is beyond the scope of their paper. Anyway, it must be remarked that all comments performed in this thesis related to inconsistency problems that arise due to amnesia in recovery process must be considered when designing a recovery solution for this replication solution.

Part IV

Conclusions and Future Work

Chapter 14

Conclusions and Future Work

14.1 Conclusions

As it has been said at the Introduction in Chapter 1, literature proposals for replicated systems have been more concerned about providing replication protocols that combine high consistency restrictions –1-copy-serializable– with high performance services than ensuring the correctness of their associated recovery solutions which have been largely seen as a secondary element in their replication solutions. It can be said, that before focusing in recovery processes researchers must first obtain good replication solutions.

This is because among all the proposed failure models for distributed systems, replicated systems adopted at the beginning the fail-stop due to its simplicity. Later, they started to adopt the crash-recovery with partial amnesia failure model as a way of providing more efficient recovery processes when talking about replicated systems with large states.

As it has been demonstrated in this thesis the phenomenon described by [122] can lead to state inconsistencies after performing recovery processes when the crash-recovery with partial amnesia failure model is adopted. This thesis has formalized this problem, and another one that arises when the amnesia combines with a specific transition from a non-working system to a working one when the majority partition progress condition is used. Then solutions have been provided for solving these problems in different replication systems configurations. Moreover, a new version of the majority progress condition is proposed.

And finally in a review of proposed recovery protocols for transactional replicated systems it has been observed how most of these proposals are unable to handle this phenomenon accurately at recovery time. And how its great ma-

jority does not avoid the problem when the amnesia combines with the specific transition from a non-working system to a working one.

14.2 Future Work

Which are the future work and research lines that opens this thesis? First of all, this thesis will open a revision process of some recovery proposals for replicated systems –specially in transactional replicated systems– in order to avoid the commented problems associated to the amnesia phenomenon. Moreover, this revision will also consider the problem when the amnesia phenomenon combines with the special replicated system transition, seen in Chapter 6, when the majority progress condition is adopted in order to avoid replicated state inconsistencies.

Obviously, new proposed recovery proposals for replicated systems which assume the *crash recovery with partial amnesia* failure model must consider the amnesia phenomenon and ensure that the properties presented in Chapters 5 and 6 of this thesis are fulfilled in order to avoid its associated problems. Or at least check that the problems associated to this phenomenon can not appear due to the basic work way provided by the replication protocol.

Another line of work would be to do a performance comparison about all recovery techniques that manage accurately the amnesia phenomenon in order to determine under which circumstances each one presents a better behaviour. This will help designers to select the more efficient recovery techniques in regard to different system characteristics.

Moreover, another interesting work will consist in comparing the performance behaviour provided by different atomic broadcast implementations which ensure that delivered messages are persisted atomically in this process. This will help to determine which of these implementations provide the more efficient atomic broadcast –with persisting– primitive for different communication configurations and workloads.

In regard to the process replication the future work will consist in designing and implementing a recovery protocol that on one hand generates all the information needed for reapplying the messages and on the other hand takes under account the different work semantics. Later, it will be necessary to perform a comparison of the obtained results with an approach using the *fail stop* failure model.

Bibliography

[1] Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, and Ioana Stanoi. Exploiting atomic broadcast in replicated databases. *LNCS*, 1300:496–503, 1997.

[2] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *ICDE*, pages 230–241. IEEE Computer Society, 2005.

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

[4] José Enrique Armendáriz-Iñigo, Hendrik Decker, Francesc Daniel Muñoz-Escoí, Luis Irún-Briz, and Rubén de Juan-Marín. A middleware architecture for supporting adaptable replication of enterprise application data. In *Trends in Enterprise Application Architecture, VLDB Workshop, TEAA 2005, Trondheim, Norway, August 28, 2005, Revised Selected Papers*, pages 29–43, 2005.

[5] José Enrique Armendáriz-Iñigo, José Ramón Garitagoitia-Padrones, José Ramón González de Mendívil, and Francesc Daniel Muñoz-Escoí. A basic replication protocol for the MADIS Middleware architecture. Technical report, Technical Report ITI-ITE-05/01, Instituto Tecnológico de Informática, 2005.

[6] José Enrique Armendáriz-Iñigo, Francesc Daniel Muñoz-Escoí, Hendrik Decker, José Ramón Juárez-Rodríguez, and José Ramón 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.

[7] Mikael Asplund, Simin Nadjm-Tehrani, Stefan Beyer, and Pablo Galdámez. Measuring Availability in Optimistic Partition-tolerant Systems with Data Constraints. In *International Conference on Dependable Systems and Networks (DSN)*, June 2007.

[8] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, Daphne Koller, David Peleg, and Rüdiger Reischuk. Achievable cases in an asynchronous environment (extended abstract). In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA*, pages 337–346, 1987.

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

[10] 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.

[11] Josep Maria Bernabé-Gisbert, José Enrique Armendáriz-Iñigo, Rubén de Juan-Marín, and Francesc D. Muñoz-Escoí. Providing Read Committed Isolation Level in Non-Blocking ROWA Database Replication Protocols. In *XV Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 07), Torremolinos, Spain*, pages 159–171, June 2007.

[12] Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading, MA, EE.UU., 1987.

[13] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *11th ACM Symposium on Operating Systems Principles*, pages 123–138, New York, NY, USA, 1987. ACM Press.

[14] Kenneth P. Birman and Robbert Van Renesse. *Reliable Distributed Computing with the ISIS Toolkit.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1993.

[15] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under unix. *ACM Trans. Comput. Syst.*, 7(1):1–24, 1989.

[16] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Primary-backup protocols: Lower bounds and optimal implementations. In *Proceedings of the Third IFIP Working Conference on Dependable Computing for Critical Applications*, pages 187–198, Mondello, Italy, 1992.

[17] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[18] Lásaro Camargos, Fernando Pedone, and Marcin Wieloch. Sprint: a middleware for high-performance transaction processing. *SIGOPS Oper. Syst. Rev.*, 41(3):385–398, 2007.

[19] Francisco Castro, Javier Esparza, María Idoia Ruiz, Luis Irún, Hendrik Decker, and Francesc Daniel Muñoz. CLOB: Communication Support for Efficient Replicated Database Recovery. In *13th Euromicro PDP*, pages 314–321, Lugano, Sw, 2005. IEEE Computer Society.

[20] Francisco Castro, Luis Irún, Félix García, and Francesc Daniel Muñoz. FOBr: A version-based recovery protocol for replicated databases. In *13th Euromicro PDP*, pages 306–313, Lugano, Sw, 2005. IEEE Computer Society.

[21] Francisco Castro-Company and Francesc Daniel Muñoz-Escoí. An exchanging algorithm for database replication protocols. Technical report, Technical Report ITI-ITE-07/02, Instituto Tecnológico de Informática, Valencia, Spain, 2007.

[22] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. *J. ACM*, 43(4):685–722, 1996.

[23] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[24] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):427–469, December 2001.

[25] Edgard F. Codd. Derivability, redundancy and consistency of relations stored in large data banks. *IBM Research Report, San Jose, California*, RJ599, 1969.

[26] Ben Collin-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. Version control with subversion, 2007. Accessible in URL:*http://svnbook.redbean.com/*.

[27] IBM Corporation. Lotus Domino Administrator 6.5.1 Help, 2003. Accessible in URL: *http://www-12.lotus.com/ldd/doc/domino_notes/6.5.1/help65_admin.nsf/Main*.

[28] Flaviu Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.

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

[30] Khuzaima Daudjee and Kenneth Salem. Lazy Database Replication with Snapshot Isolation. In *32nd International Conference on Very Large Data Bases, ACM, Seoul, Korea, September 12-15*, pages 715–726, 2006.

[31] Ruben de Juan-Marín. Linear-CLOB Recovery Protocol. Technical report, ITI-ITE-06/02, Instituto Tecnológico de Informática, Valencia, july 2006.

[32] Rubén de Juan-Marín. (n/2+1) Alive Nodes Progress Condition. In *Sixth European Dependable Computing Conference, EDCC-6, Student Forum*, pages 3–8, 2006.

[33] Rubén de Juan-Marín, Luis Irún Briz, and Francesc D. Muñoz-Escoí. Ensuring Progress in Amnesiac Replicated Systems. In *3rd International Conference on Availability, Reliability and Security, March 2008, Barcelona, Spain*, March 2008.

[34] Rubén de Juan-Marín, Hendrik Decker, and Francesc D. Muñoz-Escoí. Revisiting hot passive replication. In *2nd International Conference on Availability, Reliability and Security*. IEEE, 2007.

[35] Rubén de Juan-Marín, Luis Héctor García-Muñoz, Jose Enrique Armnedáriz-Íñigo, and Francesc D. Muñoz-Escoí. Reviewing Amnesia Support in Database Recovery Protocols. In *9th International Symposium on Distributed Objects, Middleware and Applications, Vilamoura, Portugal*, pages 717–734. Springer, November 2007. Accepted for publication.

[36] Rubén de Juan-Marín, Luis Irún-Briz, and Francesc D. Muñoz-Escoí. Recovery strategies for linear replication. In *4th International Symposium on Parallel and Distributed Processing and Applications, Sorrento, Italy, Lecture Notes in Computer Science, vol. 4330*, pages 710–723, 2006.

[37] Rubén de Juan-Marín, Luis Irún-Briz, and Francesc D. Muñoz-Escoí. Process Replication with Log-Based Amnesia Support. In *6th International Symposium on Parallel and Distributed Computing (ISPDC 2007), Hagenberg, Austria*, pages 367–374, July 2007.

[38] Rubén de Juan-Marín, Luis Irún-Briz, and Francesc D. Muñoz-Escoí. Supporting amnesia in log-based recovery protocols. In *Euro-American Conference On Telematics and Information Systems (EATIS 2007), Faro, Portugal*, 2007.

[39] Rubén de Juan-Marín, Luis Irún-Briz, and Francesc D. Muñoz-Escoí. A cost analysis of solving the amnesia problems. Technical report, Technical Report ITI-ITE-08/08, Instituto Tecnológico de Informática, April 2008.

[40] Rubén de Juan-Marín, Luis Irún-Briz, and Francesc Daniel Muñoz-Escoí. Numbing transactions to avoid stopping the system activity. In *Sixth European Dependable Computing Conference, EDCC-6*, October 2006.

[41] Rubén de Juan-Marín, María Idoia Ruiz-Fuertes, Jerónimo Pla-Civera, Luis Héctor García-Muñoz, and Francesc D. Muñoz-Escoí. On Optimizing Certification-Based Database Recovery Supporting Amnesia. In *XV Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 07), Torremolinos, Spain*, pages 145–157, June 2007.

[42] Hendrik Decker, Luis Irún-Briz, Francisco Castro-Company, Félix García-Neiva, and Francesc D. Muñoz-Escoí. Extending wide-area replication support with mobility and improved recovery. In *Advanced Distributed Systems: 5th International School and Symposium, ISSADS 2005, Guadalajara, Mexico, January 24-28, 2005, Revised Selected Papers*, pages 10–20, 2005.

[43] Hendrik Decker, Luis Irún-Briz, Rubén de Juan-Marín, José Enrique Armendáriz-Iñigo, and Francesc D. Muñoz-Escoí. Wide-area replication support for global data repositories. In *16th International Workshop on Database and Expert Systems Applications (DEXA 2005), 22-26 August 2005, Copenhagen, Denmark*, pages 1117–1121, 2005.

[44] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[45] Danny Dolev, Cynthia Dwork, and Larry J. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, 1987.

[46] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33(3):499–516, 1986.

[47] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[48] COmputing Research Education. Final 2007 australian ranking of ict conferences, 2007. Accessible in URL: *http://www.core.edu.au/rankings/Conference%20Ranking%20Main.html*.

[49] Sameh Elnikety, Steven Dropsho, and Willy Zwaenepoel. Tashkent+: Memory-aware load balancing and update filtering in replicated databases. In *Proc. EuroSys 2007*, pages 399–412, March 2007.

[50] Sameh Elnikety, Fernando Pedone, and Willy Zwaenepoel. Database replication using generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems*, pages 73–84, Orlando, FL, USA, October 2005.

[51] Elmootazbellah 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.

[52] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. On the use and implementation of message logging. In *FTCS*, pages 298–307, 1994.

[53] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory*, pages 127–140, London, UK, 1983. Springer-Verlag.

[54] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[55] Steve Fogel and Paul Lane. Oracle database administrator's guide, 10g release 2 (10.2) b14231-02, 2006. Accessible in URL: *http://www.oracle.com/pls/db102/db102.homepage*.

[56] Fusionio. iodrive, 2007. Accessible in URL: *http://www.fusionio.com*.

[57] Luis H. García-Muñoz, José Enrique Armendáriz-Iñigo, Hendrik Decker, and Francesc D. Muñoz-Escoí. Recovery protocols for replicated databases - a survey. In *Workshop FINA-07, in the AINA-07 Conference*, pages 220–227. IEEE-CS Press, 2007.

[58] Luis H. García-Muñoz, Rubén de Juan-Marín, José Enrique Armendáriz, and Francesc Daniel Muñoz-Escoí. Improving Recovery in Weak-Voting Data Replication. In *7th International Symposium on Advanced Parallel Processing Technologies, Guangzhou, China*, pages 131–140, November 2007.

[59] David K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM.

[60] Google. Google scholar, 2008. Accessible in URL: *http://scholar.google.com/*.

[61] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[62] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM Press.

[63] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[64] Vasfi Gucer, Satoko Egawa, David Oswald, Geoff Pusey, John Webb, and Anthony Yen. High Availability Scenarios with IBM Tivoli Workload Scheduler and IBM Tivoli Framework, 2004. Accessible in URL: *http://www.redbooks.ibm.com/abstracts/sg246632.html*.

[65] Rachid Guerraoui, Michel Hurfin, Achour Mostéfaoui, Rui Carlos Oliveira, Michel Raynal, and André Schiper. Consensus in Asynchronous Distributed Systems: A Concise Guided tour. In *Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems*, pages 33–47, London, UK, 1999. Springer-Verlag.

[66] Rachid Guerraoui, Rui Oliveira, and André Schiper. Atomic updates of replicated data. In *EDCC-2: Proceedings of the Second European Dependable Computing Conference on Dependable Computing*, pages 365–382, London, UK, 1996. Springer-Verlag.

[67] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, 1997.

[68] Vassos Hadzilacos. *Issues of fault tolerance in concurrent computations (databases, reliability, transactions, agreement protocols, distributed computing)*. PhD thesis, 1985.

[69] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. ACM Press, 2nd edition, 1993. ISBN 0-201-62427-3.

[70] JoAnne Holliday. Replicated database recovery using multicast communication. In *NCA*, pages 104–107. IEEE-CS Press, 2001.

[71] Yennun Huang and Chandra M. R. Kintala. Software implemented fault tolerance technologies and experience. In *FTCS*, pages 2–9, 1993.

[72] Yongqiang Huang and Hector Garcia-Molina. Exactly-once semantics in a replicated messaging system. In *ICDE*, pages 3–12. IEEE Computer Society, 2001.

[73] Informix Corporation. Informix red brick decision server administrator's guide, version 6.0, 1999.

[74] Luis Irún, Francisco Castro, Félix García, Antonio Calero, and Francisco Daniel Muñoz. Lazy recovery in a hybrid database replication protocol. In *XII Jornadas de Concurrencia y Sistemas Distribuidos*, pages 295–307, 2004.

[75] Luis Irún, Hendrik Decker, Rubén de Juan, Francisco Castro, José Enrique Armendáriz, and Francesc 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.

[76] Luis Irún-Briz. *Implementable Models for Replicated and Fault-Tolerant Geographically Distributed DataBases. Consistency Management for GlobData*. PhD thesis, Polytechnic University of Valencia, 2003.

[77] JBoss. Jboss clustering. Accessible in URL: http://docs.jboss.org/jbossas/jboss4guide/r4/html/cluster.chapt.html, 2007.

[78] JGroups. Reliable multicasting with the jgroups toolkit. Accessible in URL: http://www.jgroups.org/javagroupsnew/docs/manual/html/index.html, 2007.

[79] 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. IEEE-CS Press, 2002.

[80] Bettina Kemme. *Database Replication for Clusters of Workstations*. PhD thesis, Swiss Federal Institute of Technology Zürich, Switzerland, August 2000. No. 13864.

[81] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. Database Syst.*, 25(3):333–379, 2000.

[82] Bettina Kemme, Alberto Bartoli, and Özalp Babaoğlu. Online reconfiguration in replicated databases based on group communication. In *DSN*, pages 117–130. IEEE C.S., 2001.

[83] Frank Kyne, Alan Murphy, and Kristoffer Stav. Clustering solutions overview: Parallel sysplex and other platforms, 2007. Accessible in URL: *http://whitepapers.silicon.com/0,39024759,60301022p,00.htm*.

[84] Leslie Lamport. The part-time parliament. *ACM Transanctions on Computer Systems*, 16(2):133–169, 1998.

[85] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[86] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. On the implementation of unreliable failure detectors in partially synchronous systems. *IEEE Trans. Computers*, 53(7):815–828, 2004.

[87] Edmond Lau and Samuel Madden. An integrated approach to recovery and high availability in an updatable, distributed data warehouse. In *VLDB*, pages 703–714, 2006.

[88] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In Fatma Ozcan, editor, *SIGMOD*, pages 419–430. ACM, 2005.

[89] David B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Language Design for Reliable Software*, pages 128–137, 1977.

[90] Sam Tueg Marcos Kawazoe Aguilera, Wei Chen. Failure detection and consensus in the crash recovery model. In *DISC*, pages 231–245, 1998.

[91] Sergio Mena and André Schiper. A new look at atomic broadcast in the asynchronous crash-recovery model. In *SRDS*, pages 202–214. IEEE-CS Press, 2005.

[92] Microsoft Corporation. Sql server programming reference, 2005. Accessible in URL: *http://msdn2.microsoft.com/en-us/library/ms203801.aspx*.

[93] Francesc D. Muñoz-Escoí, Rubén de Juan-Marín, José Enrique Armendáriz-Iñigo, and José Ramón González de Mendívil. Persistent Logical Synchrony. In *7th International Symposium on Network Computing and Applications, July 2008, Cambridge, MA, USA*, January 2008.

[94] 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.

[95] MySQL AB. Mysql 5.1 reference manual, 2006. Accessible in URL: *http://dev.mysql.com/doc/*.

[96] Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith. Strongly consistent replication and recovery of fault-tolerant CORBA applications. *Comput. Syst. Sci. Eng.*, 17(2):103–114, 2002.

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

[98] Oracle. Oracle9ias containers for j2ee. ejbs developers guide, rel. 2 (9.0.4), 2003.

[99] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. Middle-r: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4):375–423, 2005.

[100] Fernando Pedone, Rachid Guerraoui, and André Schiper. Exploiting atomic broadcast in replicated databases. In *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, pages 513–520, London, UK, 1998. Springer-Verlag.

[101] Francisco Pérez-Sorrosal, Marta Patiño-Martínez, Ricardo Jiménez-Peris, and Bettina Kemme. Consistent and scalable cache replication for multi-tier j2ee applications. In *Middleware 2007, ACM/IFIP/USENIX 8th International Middleware Conference, Newport Beach, CA, USA, November 26-30, 2007, Proceedings*, pages 328–347, 2007.

[102] Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Trans. Softw. Eng.*, 12(3):477–482, 1986.

[103] Jerónimo Pla-Civera, María Idoia Ruiz-Fuertes, Luis Héctor García-Muñoz, and Francesc D. Muñoz-Escoí. Optimizing Certification-Based Database Recovery. In *6th International Symposium on Parallel and Distributed Computing (ISPDC 2007), Hagenberg, Austria*, July 2007.

[104] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In Hans-Arno Jacobsen, editor, *Middleware*, volume 3231 of *Lecture Notes in Computer Science*, pages 155–174. Springer, 2004.

[105] PostgreSQL Global Development Group. Postgresql 8.1.4 documentation, 2007. Accessible in URL: *http://www.postgresql.org/docs/manuals/*.

[106] David Powell. Failure mode assumptions and assumption coverage. In *FTCS*, pages 386–395, 1992.

[107] Computer Science Conference Ranking. Conference rankings, 2008. Accessible in URL: *http://www.cs-conference-ranking.org/conferencerankings/alltopics.html*.

[108] Aleta M. Ricciardi and Kenneth P. Birman. Using process groups to implement failure detection in asynchronous environments. In *PODC '91: Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 341–353, New York, NY, USA, 1991. ACM.

[109] Luís Rodrigues and Michel Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication. *IEEE Trans. Knowl. Data Eng.*, 15(5):1206–1217, 2003.

[110] Birgit Roehm, Balazs Csepregi-Horvath, Pingze Gao, Thomas Hikade, Miroslav Holecy, Tom Hyland, Namie Satoh, Rohit Rana, and Hao Wang. Ibm websphere v5.1 performance, scalability, and high availability websphere handbook series, 2004. Accessible in URL: *www.redbooks.ibm.com/abstracts/sg246198.html*.

[111] María I. Ruiz-Fuertes, Rubén de Juan-Marín, Jerónimo Pla-Civera, Francisco Castro-Company, and Francesc D. Mu noz Escoí. A Metaprotocol Outline for Database Replication Adaptability. In *2nd International Workshop on Reliability in Decentralized Distributed Systems, Vilamoura, Algarve, Portugal*, pages 1052–1061. Springer LNCS, November 2007.

[112] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[113] André Schiper. Dynamic group communication. *Distributed Computing*, 18(5):359–374, 2006.

[114] Fred B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, 1984.

[115] Fred B. Schneider. What good are models and what models are good? In S. Mullender, editor, *Distributed systems*, pages 17–26. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2nd edition, 1993. ISBN 0-201-62427-3.

[116] Dale Skeen. Nonblocking commit protocols. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142, New York, NY, USA, 1981. ACM.

[117] BEA Systems. Bea weblogic server and weblogic express 8.1 documentation, 2006. Accessible in URL: *http://edocs.beasys.com/wls/docs81/index.html*.

[118] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 59–72, New York, NY, USA, 2007. ACM.

[119] Rui Wang, Betty Salzberg, and David Lomet. Log-based recovery for middleware servers. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 425–436, New York, NY, USA, 2007. ACM.

[120] Andy J. Wellings and Alan Burns. Programming replicated systems in ada 95. *The Computer Journal, 39(5)*, pages 361–373, 1996.

[121] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In *ICDCS '00: Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, page 464, Washington, DC, USA, 2000. IEEE Computer Society.

[122] Matthias Wiesmann and André Schiper. Beyond 1-Safety and 2-Safety for replicated databases: Group-Safety. In *9th International Conference on Extending Database Technology*, pages 165–182, 2004.

[123] 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.

[124] Matthias Wiesmann, André Schiper, Fernando Pedone, Bettina Kemme, and Gustavo Alonso. Database replication techniques: A three parameter classification. In *SRDS*, pages 206–215, 2000.

Part V

Annex

Appendix A

Thesis Publications

This appendix lists the papers published in the context of this thesis.

A.1 Publications List

The table A.1 shows the publication list. The information provided by the table columns for each paper is the following one:

RefNum The reference number of the publication in the thesis bibliography.

JourName Journal name where the paper has been published.

ConfName Conference name where the paper has been accepted. As some conferences publish their accepted papers in journals, filling this one does not exclude in some cases filling the previous one.

CORE Conference class in the COmputing Research Education [48] classification of 2007.

EIC EIC index conference value (source CSCR [107]). The value goes from 0.5 to 1 being 1 the highest possible value.

Cit Paper citations (source Google Scholar [60]).

RefNum	*JourName*	*ConfName*	*CORE*	*EIC*	*Cit*
[4]	LNCS	TEAA 2005	–	–	3
[11]	–	JCSD 2007	–	–	1
[31]	–	–	–	–	0
[32]	–	EDCC 2006	–	–	0
[33]	–	ARES 2008	B	–	0
[34]	–	ARES 2007	B	–	1
[35]	LNCS	OTM 2007	A	0.6	1
[36]	LNCS	ISPA 2006	–	0.55	11
[37]	–	ISPDC 2007	–	–	0
[38]	–	EATIS 2007	–	–	7
[39]	–	–	–	–	–
[40]	–	EDCC 2006	–	–	0
[41]	–	JCSD 2007	–	–	1
[43]	–	DEXA Workshops 2005	–	–	0
[58]	LNCS	APPT 2007	–	–	1
[75]	LNCS	Europar 2005	A	0.68	33
[93]	–	NCA 2008	C	–	0
[111]	LNCS	OTM 2007	A	0.6	0

Table A.1: Publication List.

Index