UNIVERSIDAD PÚBLICA DE NAVARRA

Departamento de Matemática e Informática

DESIGN AND IMPLEMENTATION OF DATABASE REPLICATION PROTOCOLS IN THE MADIS ARCHITECTURE

Dissertation presented by:

José Enrique Armendáriz Íñigo

Supervised by:

Dr. José Ramón González de Mendívil Moreno

Dr. Francesc Daniel Muñoz i Escoí

A Marta, contigo he aprendido lo que que realmente significa la palabra "amor".

A mi hermano Juan Martín. Mi modelo de entrega, trabajo, coraje, amor y respeto.

Gracias por enseñarme tanto.

A Joserra y Txerra por todo vuestro trabajo, crítica, esfuerzo y amistad.

Mucho de este trabajo no habría sido posible sin vosotros.

A JR y Paco por su paciencia, dedicación y por mostrarme el camino.

Gracias de parte de vuestro padawan por introducirme en el mundo de la investigación.

"¡Que la fuerza os acompañe!"

A mis padres, Aurora y Martín. Nunca podré ser tan maravilloso como vosotros.

Os debo todo lo que soy.

"This is not the end. It is not even the beginning of the end.

But, it is, perhaps, the end of the beginning"

Wiston Churchill

Resumen

El objetivo de esta Tesis ha sido el diseño y la implementación de dos protocolos de replicación basados en O2PL para una nueva arquitectura middleware de replicación de bases de datos denominada MADIS. Esta arquitectura se caracteriza por ofrecer una interfaz JDBC a las aplicaciones de usuario y dar soporte, mediante la modificación del esquema de la base de datos, a un amplio rango de protocolos de replicación. Esto último permite aislar el control de concurrencia (gestionado por el sistema gestor de bases de datos) del control de replicación (gestionado por el protocolo de replicación utilizado en MADIS).

Los dos protocolos desarrollados, BRP y ERP, se formalizan empleando un sistema de transición de estados que facilita la comprobación de su corrección (1-Copy-Serializable). Adicionalmente, se formaliza un nuevo protocolo para MADIS, TORPE, basado en la entrega en orden total de las difusiones por parte de los sistemas de comunicación a grupo, con el objeto de comparar el comportamiento de este tipo de protocolos, ampliamente empleados en la literatura actual, con los dos protocolos anteriores.

La implementación de estos tres protocolos en MADIS verifica los supuestos teóricos planteados para el BRP y el ERP en sus pruebas de corrección y muestra que los protocolos de orden total son, en general, la mejor opción para la replicación de bases de datos. No obstante, se observa que el protocolo ERP es la mejor alternativa para entornos con cargas bajas dentro de nuestro entorno de pruebas.

Finalmente, se propone un protocolo de recuperación, diseñado también como un sistema de transición de estados, que es válido tanto para el BRP como el ERP. Este protocolo emplea la sincronía de vistas para gestionar la recuperación de nodos fallidos estableciendo las particiones de recuperación que se crean en el nodo recuperador y a recuperar, de manera que este último pueda comenzar a atender transacciones de usuario aunque no haya terminado de recuperarse. El objetivo es abortar el menor número de transacciones que estén siendo ejecutadas en el sistema.

Abstract

The main goal of this Thesis is the design and implementation of two O2PL based replication protocols adapted to a middleware database replication architecture called MADIS. This architecture provides a JDBC interface to user applications and supports, through a database schema modification, a wide range of replication protocols. This last feature permits to isolate the concurrency control (managed by the underlying database management system) from the replica control (done by the current replication protocol plugged in MADIS).

The two protocols developed, BRP and ERP, are formalized as state transition systems that ease their correctness proof (1-Copy-Serializable). Additionally, another replication protocol for MADIS, TORPE, is formalized based on the total order delivery of multicast messages featured by group communication systems, a widely used approach in the literature. Our purpose is to compare its behavior with the two previous protocols.

The implementation of these three protocols in MADIS verifies the theoretical assumption done for BRP and ERP in their correctness proofs and shows that total order protocols are, in general, the best option to achieve database replication. However, it might be noted that the ERP protocol is the best option in low workloaded environments under our experimental settings.

Finally, a recovery protocol, introduced as well as a state transition system, is proposed for both BRP and ERP. This recovery protocol uses the view synchrony in order to manage the recovery of faulty nodes by setting recovery partitions in the recoverer and recovering nodes; thus the latter may execute user transactions even though it has not been recovered yet. The aim of this proposal is to rollback as few as possible transactions in the system.

Agradecimientos

En primer lugar, quiero dar las gracias a mis directores. A J.R. porque creyó y cree en mí. Hemos pasado muchas cosas juntos desde Junio de 2001 hasta hoy. En todas ellas ha dejado una huella imborrable en mí. Me encanta cómo es capaz de asimilar y generar más y más ideas. A Paco por aceptar dirigirme y poder trabajar con él. Me gusta el empeño, seriedad y orden que pone en el trabajo. Aunque es más, si cabe, todo el saber científico que atesora. Trabajar con él es siempre un placer, sabe motivar para seguir adelante. Deseo con toda mi alma que se me haya "*pegado*" algo de ellos. Muchas, muchas gracias a los dos. Pido disculpas a sus familias por el tiempo que les ha quitado el nuevo "*hijo*".

Por otro lado, ya conocía a una persona maravillosa, pero he conocido a un mejor investigador, J.R. Garitagoitia. Son innumerables las horas que he pasado trabajando con él. Recuerdo cuando llegué con un algoritmo de 8 hojas a su despacho. Trabajar con él, es como la canción: "*para cuando tú vas, yo vengo ...*". Gracias al nuevo fichaje, J.R. Juárez, que se ha dedicado en cuerpo y alma a dar vida y forma a todo este trabajo. Fines de semana, noches y viajes incluidos. Le debo una y bien grande.

Además quiero agradecer a todos mis compañeros del Dpto. de Matemática e Informática por acogerme y darme la oportunidad de trabajar con ellos. Gracias también a la gente del ITI de Valencia por hacerme sentir como en casa y poder trabajar con ellos, especialmente a M.C. Bañuls, F. Castro, H. Decker, L. Irún, R. de Juan, E. Miedes e I. Ruiz. También quiero dar las gracias, de todo corazón, por la paciencia, las discusiones y comentarios realizados, a M. Patiño-Martínez y R. Jiménez-Peris.

En el apartado personal, quiero agradecer a mis padres todo el esfuerzo que han hecho conmigo. Siempre estáis ahí, aunque muchas veces no me dé cuenta y no sepa agradecerlo. Gracias Martín, eres mi compañero de viaje ideal. Gracias Marta, "*has aguantado carros y carretas*" y me has dado comprensión, cariño y confianza.

Table of Contents

1 Introduction **1**

1.1 About this Thesis . . . 8

1.1.1 Research Goals . . . 8

1.2 Thesis Organization . . . 9

2 System Model and Definitions **11**

2.1 General Architecture . . . 11

2.2 Group Communication System . . . 13

2.2.1 View Synchrony . . . 14

2.2.2 Communication Service . . . 15

2.2.3 Membership Service . . . 16

2.3 Database Management System . . . 16

2.4 Transactions . . . 19

2.5 State Transition Systems . . . 21

2.6 Discussion . . . 23

3 Middleware Replication Protocols **25**

3.1 Introduction . . . 26

3.2 Basic Replication Protocol Description . . . 27

3.3 BRP Correctness Proof . . . 31

3.4 Enhanced Replication Protocol . . . 42

3.4.1 Performance . . . 43

3.4.2 Decreasing Abortion Rate . . . 44

3.4.3 The ERP Automaton . . . 45

3.5 ERP Correctness Proof . . . 46

3.6 The TORPE Replication Protocol . . . 57

3.7 Discussion . . . 60

3.7.1 Comparison with Related Works . . . 63

4 MADIS: A Slim Middleware for Database Replication **67**

4.1 Introduction . . . 67

4.2 The MADIS Architecture . . . 69

4.3 Schema Modification . . . 70

4.3.1 Modified and Added Tables 71
4.3.2 Triggers 73
4.4 Consistency Manager 77
4.4.1 Connection Establishment 77
4.4.2 Common Query Execution 78
4.4.3 Commit/Rollback Requests 80
4.5 Protocol Interface 81
4.5.1 Connection Establishment 82
4.6 Experimental Results 83
4.6.1 Overhead Description 83
4.6.2 Overhead in the MADIS Architecture 84
4.6.3 Comparison of Overheads with Other Architectures 87
4.6.4 Experimental Results of the Replication Protocols Implementation in MADIS 88
4.7 Discussion 97
4.7.1 Comparison with Related Works 97

5 About Failures and the Recovery Process **101**
5.1 Replication Protocol Modifications 102
5.2 Recovery Protocol Outline 103
5.3 Recovery Protocol Description 107
5.3.1 Site Failure 110
5.3.2 Site Recovery 111
5.4 Discussion 115
5.4.1 Comparison with Related Works 117

6 Conclusions **121**
6.1 Summary 121
6.2 Future Lines 124

Bibliography **125**

List of Figures

2.1 Main components of the system . . . 12
2.2 Communication System . . . 13
2.3 Database Management System . . . 17
2.4 Transaction execution model . . . 21

3.1 Execution example of a local (left) and a remote transaction (right) . . . 27
3.2 State transition system for the Basic Replication Protocol (BRP). *pre* indicates precondition and *eff* effects respectively . . . 28
3.3 Valid transitions for a given $status_i(t)$ of a transaction $t \in \mathcal{T}$ . . . 29
3.4 *Happens-before* relationship for the BRP of a given transaction t between its execution at the master site and the rest of nodes . . . 36
3.5 CASE (I): $node(t) = node(t') = x$ . . . 41
3.6 CASE (II): $node(t) = x$ and $node(t') = y$ . . . 41
3.7 CASE (IV): $node(t) = i$ and $node(t') = j$ . . . 42
3.8 State transition system for the Enhanced Replication Protocol (ERP) automaton . . . 43
3.9 *Happens-before* relationship for a given transaction t between its execution at the master site and the rest of nodes . . . 50
3.10 CASE (I): $node(t) = node(t') = x$ . . . 56
3.11 CASE (II): $node(t) = x$ and $node(t') = y$ . . . 56
3.12 CASE (IV): $node(t) = i$ and $node(t') = j$ . . . 57
3.13 State transition system for TORPE, where updates are propagated using the total order group communication primitives provided by the GCS . . . 59
3.14 Valid transitions for a given $status_i(t)$ of a transaction $t \in \mathcal{T}$ . . . 60

4.1 The MADIS Architecture . . . 70
4.2 Query execution . . . 79
4.3 Update Execution . . . 80
4.4 Commit suceeded vs aborted . . . 81
4.5 Connection Establishment . . . 83
4.6 Database tables description of the experiment . . . 85
4.7 MADIS absolute overhead (in ms) . . . 86
4.8 Relative MADIS/JDBC overhead (in %) . . . 86
4.9 Relative COPLA/JDBC overhead . . . 87

4.10 Relative RJDBC/JDBC overhead . . . 88
4.11 BRP response time in a 5-server system varying the submission rate and the number of clients . . . 91
4.12 ERP response time in a 5-server system varying the submission rate and the number of clients . . . 91
4.13 TORPE response time in a 5-server system varying the submission rate and the number of clients . . . 92
4.14 Response time of the replication protocols in a 5-server system varying the submission rate . . . 93
4.15 Response time of the replication protocols for a submission rate of 10 TPS varying the number of servers . . . 94
4.16 Response time of the replication protocols in a 5-server system for a submission rate of 10 TPS varying the transaction length . . . 95
4.17 Conflict Rate: Abort rate for BRP and ERP in a 5-server system for a submission rate of 20 TPS . . . 96

5.1 In a failure free environment, the transaction master site directly commits. This is not a correct approach in the presence of failures. A site may commit a transaction that the rest of sites do not commit (left). The solution is to use the uniform reliable multicast and the delivery of the message to the master site (right) . . . 102
5.2 State transition system for the ERP so as to avoid data inconsistencies due to a site failure . . . 104
5.3 The rest of nodes store objects modified while a site is crashed. In this case, node j has failed and partitions (i.e. set of data items) P_1 and P_2 have been modified during view id'. When it rejoins the system, the recovery metadata is transferred to node j and the recovery protocol itself at node j can determine the partitions to be recovered . . . 105
5.4 Object state transfer between the *recoverer* node i and a *recovering* node j. User transactions may perform their operations with no restrictions unless they try to access the recovery partitions. Partitions are released once changes are applied . . . 106
5.5 Signature and states for the ERP recovery protocol . . . 107
5.6 Specific recovery state transition system for ERP . . . 108
5.7 Add-ons on the state transition system of ERP so as to support recovery features . . . 109
5.8 Valid transitions for a given $sites_i(j).state$ of a node $j \in N$ at node i . . . 110
5.9 Actions to be done by the recovery protocol when a remote node (left) or a transaction master site (right) fails . . . 111
5.10 Actions to be done by the recovery protocol when a *recovering* node (left) or the *recoverer* node (right) fails . . . 112
5.11 A new joining node may commit a transaction whose existence does not know (left). Its modification (right) allows joining nodes to execute that transaction, even though they have not recovered yet . . . 113

5.12 Description of the ERP recovery protocol. It recovers node k that has been *crashed* just for one view, id'. Data items $\{p, x, y\}$ have been updated in that installed view 115

List of Tables

4.1 Parameters of experiments 90
4.2 Conflict Rate: Data access distribution to achieve different conflict rates 96

Chapter 1

Introduction

This thesis is englobed inside the database replication research field. Database replication consists in multiple nodes with a Database Management System (DBMS on the sequel) storing multiple copies of some data items [BHG87]. Data access is done by means of transactions. A transaction represents a logical unit of read and write operations. The main reason for using replicated data is to increase system availability. By storing critical data at multiple sites, the system may continue working even though some sites have failed. Another goal is performance improvement. As there are many copies users access their closest site to perform operations. The drawback of this is that updates have to be propagated to all copies of the updated data item. Hence, read operations may run faster than write operations. Another drawback closely related to the latter is data consistency, since some sites may have outdated information. Therefore, there is a constant tradeoff between data consistency and performance.

There are two approaches to achieve database replication: via middleware [ACZ03, AGME05, CMZ04, EPMEIBBA04, JPPMKA02, LKPMJP05, MEIBG+01, PMJPKA00, PA04, RMA+02, RBSS02] or by means of database internals modification [AAES97, AT02, BHG87, HSAA03, KA00a, KA00b, KPA+03, WK05]. Middleware based solutions use a layer between clients and database replicas. This simplifies the development since the database internals remain inaccessible. Furthermore, middleware solutions may be maintained independently of the underlying DBMS. We may add several tables, triggers and stored procedures to facilitate database replication using SQL and procedural languages. However, this solution may cause a degradation of the system performance due to the overhead introduced to accomplish replication. The other alternative to achieve database replication is the modification of the database internals so as to add

communication facilities to interact with the other DBMS sites of the system. This approach has a strong dependency on the database engine for which the system is developed, and it must be reviewed each time the original DBMS software release is updated. On the other hand, its performance is generally better than the one achievable using a middleware based architecture. This approximation has different requirements and needs, and their comparison with middleware based solutions is not fair. Nevertheless, there are several works in the literature dealing with this technique. In Postgres-R and Dragon [Kem00, KA00b, KA00a, KPA+03], a DBMS core is modified in order to include distributed support to the database engine.

Both approaches share a main goal: how to coordinate replica control with concurrency control. This is done by means of a particular replication protocol that manages data consistency. The strongest correctness criteria for a replicated database is 1-Copy-Serializability (1CS) [BHG87]; a system managing a replicated database should behave like a DBMS managing a non replicated database insofar as users can tell. In a non replicated database, users expect the interleaved execution of their transactions to be equivalent to a serial execution of those transactions. Since replicated data should be transparent to them, they would like the interleaved execution of their transactions on a replicated database to be equivalent to a serial execution of those transactions on a non replicated database.

Database replication techniques have been classified according to [GHOS96]: who can perform updates (*primary copy* [Sto79] and *update everywhere* [KPA+03]) and the instant when a transaction update propagation takes place (*eager* [CL91] and *lazy* [PST+97, SAS+96]). In eager replication schemes, updates are propagated inside the context of the transaction. On the other hand, lazy replication schemes follow the next sequence: update a local copy, commit the transaction and propagate changes to the rest of available replicas. Data consistency is straightly forward by eager replication techniques although it requires extra messages. On the contrary, data copies may diverge on lazy schemes and, as there is no automatic way to reverse committed replica updates, a program or a person must reconcile conflicting transactions. Regarding to who performs the updates, the primary copy requires all updates to be performed on one copy and then propagated; whilst update everywhere allows to perform updates at any copy but makes coordination more complex [WPS+00]. Another parameter considered for replication protocols is the degree of communication among sites [WPS+00]: *constant interaction*, where a constant number of messages are exchanged between sites for a given transaction, and *linear interaction*, where a site

propagates each operation of a transaction to the rest of sites. The last parameter is how a transaction terminates [WPS+00]: *voting*, when an extra round of messages are required to coordinate replicas such as the 2-Phase-Commit (2PC) [BHG87] protocol or *non voting*, a site decides on its own whether a transaction commits or is rolled back.

According to this database replication classification and its study, depicted in [GHOS96], show that the first eager replication protocols, those based on the 2-Phase-Locking (2PL), such as the Distributed 2PL (D2PL) [BHG87] is the worst of all replication techniques. D2PL needs to propagate each operation performed by a transaction to all available sites, increasing the deadlock rate and message overhead. The deadlock rate rises as the third power of the number of nodes in the network, and the fifth power of the transaction size. Respectively, if we focus on lazy replication schemes we reach to the same conclusion, since as long as there exists a transaction waiting for another, in fact waits are more frequent that deadlocks [GHOS96], a reconciliation has to be done. The reconciliation rate rises as the third power of the number of nodes or the size of the transaction. In mobile environments disconnected nodes may not use an eager replication protocol, but if they use a lazy replication scheme the conflicting updates done during the disconnection period must require reconciliation too. Therefore, the reconciliation rate increases as the square of nodes and proportionally to the disconnected period.

Hence, we are interested in developing eager replication protocols (no reconciliation) with constant interaction (hence, distributed deadlock between sites, if occurs at all, it will be at the end of the transaction lifetime) in fixed networks. Following the Read One Write All Available (ROWAA) policy [GHOS96], the Optimistic 2PL (O2PL) [CL91] is introduced where all operations issued by a transaction are firstly performed at a single node and afterwards the update operations are grouped and propagated to the rest of sites. The O2PL has constant interaction but is deadlock prone. Hence, a deadlock prevention schema has to be defined so that distributed deadlock is avoided. Both, the D2PL and O2PL, are voting protocols and unilateral aborts may be prevented [Ped99]. O2PL has been implemented in several architectures as in MIRROR [XRHS99] where O2PL is enhanced with a novel-state-based real-time conflict. The O2PL algorithm has been extended to comprise object based-locking as in [HT98], because object supports more abstract operations than the low-level read and write operations. O2PL has been extended to the mobile environment as it is depicted in [JBE95]. The algorithm introduced there is called O2PL for Mobile Transactions (O2PL-MT) which allows a read unlock for an item to be executed at any

copy site of the item.

Concurrently to this, a research on Group Communication Systems (GCSs) [CKV01] was emerging. The delivery guarantees provided by a standard GCS, more precisely total order delivery [CKV01], permit to order transactions so that all sites apply updates performed in the database following the order set up by the GCS. Moreover, the GCS includes also a membership service that permits to detect faulty nodes [CKV01]. In [AT02, KA00a, KA00b, JPPMKA02, LKPMJP05, RMA+02, WK05] several database replication protocols based on GCSs are introduced. This solution scales well, i.e. has constant interaction, but total order implementation needs extra message rounds or sequencers [DSU04] that introduce additional latency to the system. Moreover, if transactions executed in the system have low conflict rates then this solution is expensive in terms of message latency. To improve this last drawback, the Optimistic Atomic Broadcast (OAB) is introduced [KPA+03]. As messages delivered in a LAN are almost spontaneously totally ordered [Ped99], the OAB optimistically delivers a message containing a transaction and, afterwards, if the total order decided by the GCS diverges from the optimistic approach the transaction will be finally rolled back. Otherwise, it will be committed.

On one hand, GCS based replication protocols solve distributed deadlock by means of the GCS total order delivery guarantee. Although this is not enough to prevent deadlocks in each site [LKPMJP05]. However, in a system where aborted transactions are resubmitted total order does not guarantee the fairness of transactions. Let us suppose that there exists a periodic transaction that continuously update the same set of elements. We also assume that there exists another transaction that modifies the same set of elements. The latter will be continuously aborted if the transaction period of the former is less than the duration of a conflicting transaction and its transaction restart time. On the other hand, O2PL has a deadlock prevention function that avoids distributed deadlock cycles. The deadlock prevention function imposes a global ordering for transactions. The ordering factor can be based on different transaction parameters such as the number of restarts, timestamp, etc. The first solution has nothing to do with transaction intrinsics while the latter does. However, O2PL may continually penalize certain pattern of transactions, e.g. if we order transactions according to the size of the write set then a transaction whose write set is smaller than the ones of the remaining transactions will be penalized continuously.

Data consistency depends on the guarantees the replication protocol may provide. As it has been pointed out earlier, the 1CS is the strongest correctness criteria. This may be achieved if

the underlying DBMS provides the appropriate transaction isolation level, in this case serializable as considered in [BBG+95]. Most commercial databases, such as Oracle [Ora97] and PostgreSQL [Pos05], provide Snapshot Isolation (SI) [BBG+95]. Using this isolation level read operations never block as they get a database snapshot of committed data when the transaction started. This form of transaction isolation does not guarantee serializable behavior, therefore 1CS is not achievable. However, applications using SI databases may be modified in order to produce only serializable executions [FLO+05]. There are several recent approaches for database replication with DBMSs providing SI, such as Postgres-R(SI) [WK05] that is the evolution of Postgres-R [Kem00] where the internals of PostgreSQL are modified to support SI. A more recent work features a middleware database replication protocol providing 1-Copy-SI (1CSI) [LKPMJP05] where transactions read the latest system snapshot. In [EPZ05] another database replication protocol providing Generalized SI (GSI) is introduced.

In this Thesis, we focus in the middleware approach to achieve database replication. Implementing replication protocols in a middleware architecture may lead to re-implement some features that may be obtained from the DBMS [AGME05]. However, replication protocols need certain metadata information in order to properly work. Replication techniques may also be classified considering the way the metadata collection is implemented [Pee02]:

- *Middleware-based.* This technique inserts proprietary code. As the application makes changes to the database, the middleware is able to capture the changes and store them in message queues or other data format internally used by the rest of the middleware. It does not require database schema changes. The change capture middleware typically does not depend heavily on the idiosyncrasies of the underlying database. However, it is intrusive, difficult and costly to implement.

  It is incompatible with third-party tools, interactive query/update, and often web front-ends. The middleware typically also cannot capture changes made through interactive SQL query/update tools, including those provided with the database by the database vendor, which is unacceptable to many database administrators who rely on interactive tools in production environments.

- *Trigger-based.* It captures the changes into separate queues or uses the log itself as the change queue for later replay at target sites. This way of implementing replication installs

insert, update, and delete triggers on all replicated tables to capture changed data. Its advantages are that it does not require changes to the application and they work with third-party and interactive query/update tools. However, it presents some drawbacks as it is not fully heterogeneous and cannot work with all popular databases. Some databases that do have logs do not document their transaction logs and/or APIs, which means that database-specific log sniffers and readers are not only more difficult to code, but are likely to be unsupported by the database vendor.

- *Shadow-Table-based*. It sets up a duplicate "shadow" database table (or external storage, such as a disk file) for every replicated table, containing all rows and columns of the primary table. The shadow-table technique is one of the standard techniques used for implementing DBMS (the alternative approach is by way of logs, either with a writeahead log or a private workspace [TS01]). Hence, its cost is assumed in a replicated architecture if the underlying DBMS uses this technique. It presents the advantages of the previous technique but it is unworkable if the database size is large because of its severe space and performance overhead.

- *Control-Table-based*. It creates a control table for every replicated table. This control table does not duplicate any data fields from the replicated table. The control table contains the primary key field of the table and two fields information such as the time, the transaction identifier and site of the last update of the record. It presents the advantages of the prior techniques and does not duplicate data. It permits fine-grained selection of updates performed after a given time and it is database independent and works on all databases.

In a middleware architecture, users access to the system using a standard database interface (such as JDBC) [CMZ04, EPMEIBBA04, JPPMKA02, LKPMJP05]. Hence, applications do not have to change because of the middleware architecture usage. There exist middlewares with linear interaction [CMZ04, EPMEIBBA04], these approaches are very similar and have scalability problems, since a transaction is not allowed to proceed, after an update, until all nodes have performed the previous update operation. On the contrary, in [PMJPKA00, JPPMKA02, PMJPKA05, LKPMJP05] replication is managed by ad-hoc replication protocols which belong to the constant interaction family. However, user transactions may only perform stored procedures in [JPPMKA02] while in [LKPMJP05] they are free to perform any SQL statement although the

correctness criteria is rather different, 1CSI.

There are other middleware approaches like COPLA [IMDBA03] where a full object oriented replication architecture providing object state persistence in Relational DBMSs with different replication protocols [AGME05, MEIBG+01, RMA+02] is introduced. Users may choose among all of them according to the application profile. The replication protocols implemented in COPLA share some inconveniences. The first one is that they have to re-implement several features that are included in any DBMS (e.g. lock or version management) while in [LKPMJP05] it is unnecessary. Another inconvenience is the persistence that is provided by a RDBMS that introduces a mismatch between the application schema definition and its storage. Although allowing multiple consistency protocols to be plugged into, the system provides a proprietary API for the applications to gain access to distributed databases, reducing the generality of the solution.

Finally, we focus on how failures and recoveries are handled by a replicated database system. These tasks are managed by taking advantage of the virtual synchrony [CKV01] provided by GCSs. The GCS reports to the system about failures in form of view change messages excluding unreachable sites. This happens in such a way that each site can individually decide on the outcome of pending transactions avoiding any extra communication. In the same way, we consider that sites will join again the system after a failure, we will have to cope with the recovery process. This, however, is not as straightforward as the exclusion of nodes. Two key issues must be handled [Kem00]: the current state of the database has to be installed efficiently at the joining node, interfering as little as possible with concurrent processing of transactions in the rest of the system; and, a synchronization point must be determined from which on the new node can start processing transactions itself.

In [KBB01] several different alternatives, using the Enriched View Synchrony concept, are shown for information transmission to the node recovery: embedding the recovery process inside the GCS; the whole database; versioning; and, lazy data transfer. All these solutions greatly depend on the DBMS considered, since these are not standard features.

Another recovery solution proposed in the literature is [JPPMA02] which is a recovery protocol for the protocol proposed in [PMJPKA00]. They use totally ordered logs for transferring missed updates associated to a given partition. One node is selected as the recoverer of a given partition which transfers missed information and current updates. Once this process is finished, the node is considered to be online.

1.1 About this Thesis

The aim of this thesis is to develop an eager replication protocol whose formal correctness proof (1CS [BHG87]) does not depend on the total order delivery imposed by the Group Communication System (GCS) [CKV01]. The development of this protocol is enclosed inside the Middleware Altamente DISponible (MADIS) research project (TIC2003-09420-C02), a national research project jointly developed by Grupo de Sistemas Distribuidos from Universidad Pública de Navarra (Pamplona, Spain) and Instituto Tecnológico de Informática from Universidad Politécnica de Valencia (Valencia, Spain).

MADIS is a middleware architecture that provides database replication: (*i*) Users may see a JDBC interface to interact with the middleware, i.e. they do not modify their original applications. (*ii*) Data consistency is managed by replication protocols. Another key point of this project is the design of a generic protocol interface that permits to implement a wide range of replication protocols and switching among them. As all replication protocols do not have the same metadata needs, such as the version number or the timestamp of the last modification or the owner of an object among others, we have to modify the database schema in order to support a wide range of replication protocols. This will be thoroughly explained in this Thesis.

The replication protocols presented in this Thesis maintain replica consistency while the concurrency control is left to the underlying DBMS. This implies that we do not need to reimplement any specific database feature at the middleware level, such as lock tables. It only propagates the updates performed by transactions. Deadlock situations among sites are prevented by defining a dynamic deadlock prevention schema based on priorities.

1.1.1 Research Goals

The main research goals of this Thesis are the next:

1. **The MADIS Architecture**. As it has been previously pointed out, we have collaborated in the design, development and implementation of the MADIS architecture. MADIS is a database replication middleware architecture that tries to minimize the main drawbacks of the middleware approach. Hence, it tries to offer a very similar interface to the one already provided by the underlying DBMS. Replication protocols will take advantage of the concurrency control offered by the DBMS so as to focus themselves exclusively in replica

control. Moreover, the database schema may be modified to simplify the metadata collection for replication protocols.

2. **Middleware Database Replication Protocols**. A classic eager update everywhere replication protocol, such as O2PL, is taken as the cornerstone for developing new replication protocols adapted to MADIS. It will be necessary to study their communication needs (e.g. message ordering required or not), distributed deadlock prevention, site failure treatment and atomic commitment protocol. We are very interested in comparing these protocols with those based on the total order delivery guarantee [JPPMKA02, Kem00, KA00b, KPA$^+$03, PMJPKA00, PMJPKA05].

3. **Correctness Proof**. The strongest correctness criteria for replicated databases is 1CS. It will be formally shown that the replication protocols developed are correct. Hence, a state transition system [Sha93] will be used for the formalization of these protocols, followed by a formal reasoning of the correctness properties.

4. **Implementation and Comparison of Replication Protocols**. From results obtained in the previous two items, the designed replication protocols will be implemented in MADIS in order to compare their performance. The key points to be compared are: transaction response time, abortion rate and the influence of delivery guarantees.

5. **Recovery Protocol Study and Design**. As MADIS is a highly available system, we must ensure this property by the specification of a recovery protocol. Hence, a new recovery protocol must be defined so that it interferes as little as possible with previously available nodes during the recovery process. Moreover, user transactions may be executed at the recovering node although it has not been recovered yet.

1.2 Thesis Organization

This Thesis is organized as follows. Chapter 2 introduces the system and the formalization models used throughout the thesis for the definition of the protocols proposed. Chapter 3 presents two O2PL based replication protocols formalized as state transition systems along with their respective correctness proofs. It also introduces the formalization of a total order replication protocol as a state transition system. Chapter 4 is devoted to explain the MADIS architecture, its design and

implementation. It also shows some experimental results such as the overhead introduced by MADIS and the comparison of the implementation of the O2PL based protocols in MADIS as well as their comparison with a total order replication protocol implementation. The recovery protocol for the O2PL based protocols is depicted as a state transition system in Chapter 5. Finally, Chapter 6 summarizes the major results of this Thesis, along with some outlines of future research directions.

Chapter 2

System Model and Definitions

This Chapter introduces the formal model and the tools used by the middleware architecture so as to support the replication and recovery protocols which this Thesis is based on. The abstraction of the architecture is divided in four parts in order to present its main components and the interfaces they offer to interact between them. We start with the communication model description which mainly consists in the view-oriented Group Communication System (GCS) [CKV01]. As we are developing a replication protocol for a middleware architecture providing a JDBC interface, we have to consider the Database Management System (DBMS) model. Finally, we introduce the transaction model that points out our replication protocol basics.

We describe the behavior of our protocols using a formal state transition system [Sha93]. Thus, protocols are described based on a set of actions, that are enabled if the state variables satisfy certain conditions. Respectively, the body of an action modifies the state of variables so that other actions may be enabled or disabled. This permits us to define liveness and safety properties by way of an assertion language.

2.1 General Architecture

The system model, see Figure 2.1, considered in this Thesis is an abstraction of the MADIS middleware architecture [IBDdJM+05]. This middleware has been implemented in Java and provides a JDBC interface to its client applications. The main parts of this architecture are described as an abstraction in this Chapter. Details of its implementation will be introduced in Chapter 4, where the interfaces presented there may easily be ported to its JDBC, DBMS or GCS own actions.

The architecture is composed by $\mathcal{N}$ sites (or nodes) which communicate among them by message exchange, $m \in \mathcal{M}$, where $\mathcal{M}$ is the set of possible messages that may be generated in our system, using the communication primitives provided by a GCS [Bar04, CKV01, HT94]. We assume a partially synchronous system; different sites may run at different rates, and message delay is unknown but under certain bounded limit. Otherwise, with an asynchronous distributed system with failures no consensus can be reached [FLP85] and group membership cannot be solved, excepting the addition of a failure detector [CT96].

In regard to failures, we assume a *partial amnesia crash* [Cri91]. We consider this kind of failures as we want to deal with node recovery after its failure. It occurs when, at restart, some part of the state is the same as before the crash, while the rest of the state is reset to a predefined initial state. In our model, all committed transactions prior to a node failure are maintained when it joins again the system. On the other hand, active transactions and state variables are missed once the node crashes. Hence, the recovery protocol must transfer the missed updates of faulty nodes and update the state variables associated to the protocol.

We assume a fully replicated system. Each site has a DBMS containing a copy of the entire database and executes transactions on its data copies. In the following, $\mathcal{T}$ denotes the set of all possible transactions and $\mathcal{OP}$ denotes the set of all possible operations that may be submitted to the database. A transaction, $t \in \mathcal{T}$, submits operations (SQL statements), $op \subseteq \mathcal{OP}$, for its execution over its local DBMS via the middleware module. The replication protocol coordinates the execution of transactions among different sites to ensure 1CS [BHG87].

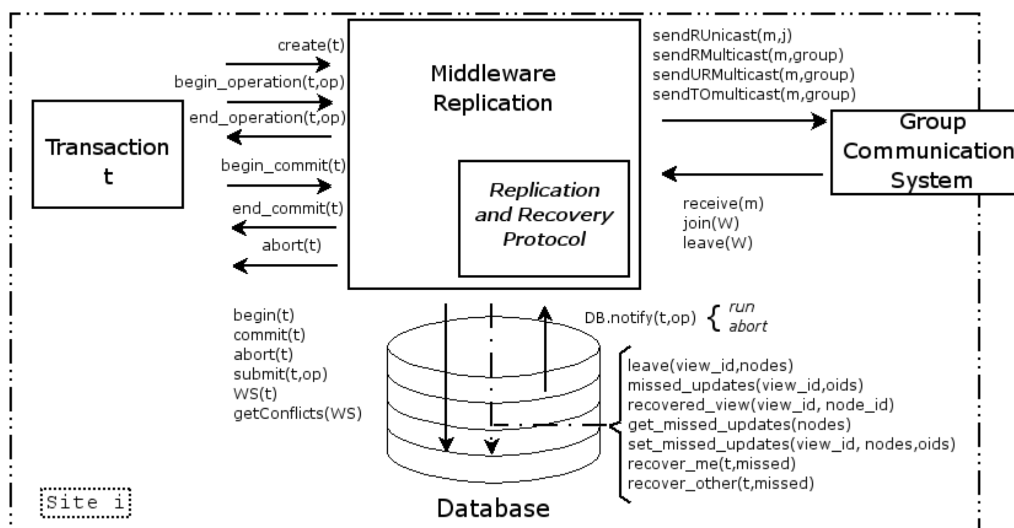


Figure 2.1: Main components of the system

2.2 Group Communication System

The GCS interacts only with the replication and recovery protocol module. It has been split into two parts: the Communication Service (CS) and the Membership Service (MS). Their respective interfaces are introduced in Figure 2.2.

The communication among sites is mainly based on reliable or total order multicast as provided by standard GCSs [HT94, CKV01], where multicast messages are grouped by views and delivered to the list of available nodes in that view. GCS also provides a membership service so as to detect failures and recoveries [CT91]. A *view change* is generated each time a node crashes or recovers after a failure [CKV01]. More formally, a view, denoted $\mathcal{V}$, is defined as a pair $(\mathcal{V}.id, \mathcal{V}.availableNodes)$ where $\mathcal{V}.id \in \mathbb{Z}$ and $\mathcal{V}.availableNodes \subseteq \mathcal{N}$ is the set of nodes in such a view. $\mathcal{VID} = \{\langle id, availableNodes\rangle\colon id \in \mathbb{Z}, availableNodes \subseteq \mathcal{N}\}$ is the set of all possible installed views in the system.

Reliable multicast does not restrict the order in which messages are delivered. Besides, its cost is low in terms of physical messages per multicast. This low cost is one of the reasons to select it for our replication protocol proposal [DSU04, KPA+03]. For some messages the protocol also uses the traditional reliable unicast. On the other hand, total order multicast ensures that messages are delivered in the same order at all nodes that deliver them [CKV01].

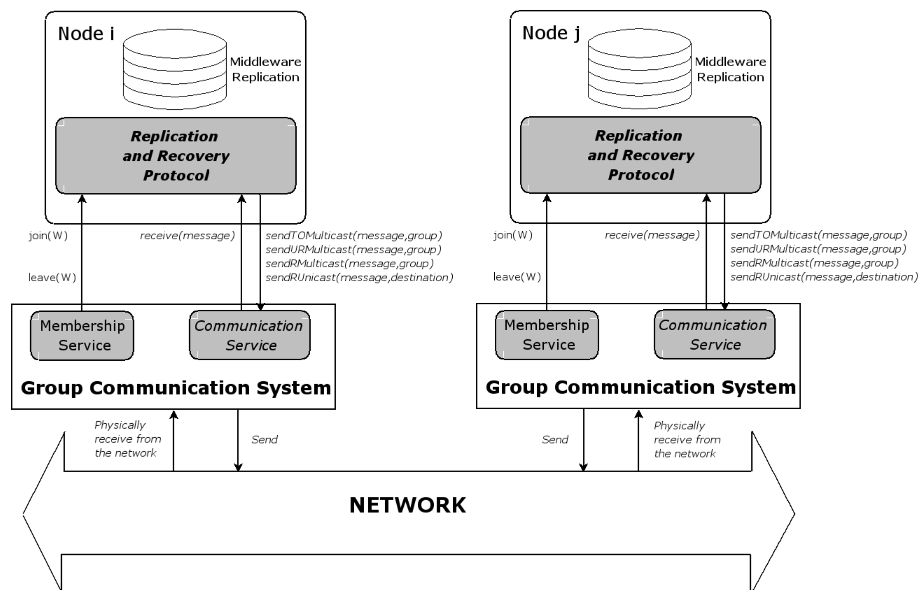


Figure 2.2: Communication System

2.2.1 View Synchrony

We have made the assumption that the GCS is view-oriented. This means that certain events (such as: message sending; message delivery; node failures; and, node recoveries) occur at all available sites in the context of views. It acts as a synchronization point for distributed systems and it is denoted as *view synchrony* [BBD96]. The real utility of view synchrony in a GCS is not in its individual components, reliable multicast and membership service, but in their integration. Informally, view synchrony can be specified through the following properties on message reliable deliveries:

- *Agreement*. All processes that survive from one view to the same next view deliver the same set of messages.
- *Uniqueness*. A message is delivered in at most one view.
- *Integrity*. A message is delivered at most once by any process and only if some process actually multicasts it.

We cope with transient or software (e.g. source and reception buffers management) communication failures. Our communication system reliability will not always be held. Access to certain nodes may be missed due to possible hardware communication failures. At least in environments where links are not replicated. Therefore, network partitions may appear, in such a case we will adopt the *primary partition* model [RSB93].

In case of failures, the GCS must also guarantee a uniform reliable multicast [HT94], stating that if a site (faulty or not) delivers a message in the context of a view, then all non-faulty nodes will eventually deliver this message in the given view [CKV01]. This feature is desirable in order to guarantee atomic commitment in replicated databases. We need the following properties to guarantee uniform delivery:

- *Uniform Agreement*. If a site (whether correct or faulty) delivers a message m, then all correct sites eventually deliver m.
- *Uniform Integrity*. For any message m, every site (whether correct or faulty) delivers m at most once, and only if m was previously multicast by the origin site of m.

The latter property is also interesting for the recovery process of a node, since delivered messages may be grouped by installed views. Hence, all updates performed by transactions as long

as there are crashed nodes will be much easier to store this way. Afterwards, when a failed node joins the system again, we split the recovery process on the views missed by the recovering node.

The total order multicast must additionally include the following property [CT96, DSU04, HT94] in order to be guaranteed:

- *Uniform Total Order*. If sites i and j both total-order deliver messages m and m', then i total-order delivers m before m', if and only if j total-order delivers m before m'.

2.2.2 Communication Service

Each site has an input buffer of messages, denoted $channel_i \subseteq \mathcal{M}$ for each $i \in \mathcal{N}$, where messages delivered by other nodes are stored. Therefore, sending a message implies filling the associated buffer of all its possible destinations. We do not assume that message reception follows a FIFO policy. Replication protocols may choose to receive one of the messages contained in $channel_i$ does not need any special message ordering. The CS provides the following actions to the replication protocol (see Figure 2.2):

- **sendRMulticast**($m, group$). It multicasts a message from a node, $m \in \mathcal{M}$, to a set of nodes ($group \subseteq \mathcal{N}$). This action will put the given message in the associated buffer of each node. More formally: $\forall j\colon\ j \in group : channel_j \leftarrow channel_j \cup \{m\}$.

- **sendRUnicast**($m, destination$). This action sends a message, $m \in \mathcal{M}$, to a given destination, where $destination \in \mathcal{N}$. In other words: $channel_{destination} \leftarrow channel_{destination} \cup \{m\}$.

- **receive**(m). A message, $m \in \mathcal{M}$, sent to a given node is stored in its associated buffer, according to its special delivery guarantees. The protocol must explicitly invoke this action in order to receive the message. Hence, this action removes the message from the buffer: $channel_j \leftarrow channel_j \setminus \{m\}$.

- **sendURMulticast**($m, group$). It uniformly multicasts a message, $m \in \mathcal{M}$, to a set of nodes ($group \subseteq \mathcal{N}$) in the context of a given view, $\mathcal{V}$.

- **sendTOMulticast**($m, group$). It total-order multicasts a message, $m \in \mathcal{M}$, to a set of nodes ($group \subseteq \mathcal{N}$) in the context of a given view, $\mathcal{V}$. More formally: $\forall j\colon\ j \in group : channel_j \leftarrow channel_j \cup_{TO} \{m\}$.

2.2.3 Membership Service

The MS has to maintain the set of the currently available nodes. This list may change whenever a new node joins or leaves the group. When this list changes, the membership service reports the change to the members by installing a new view. The MS installs the same view at all mutually connected members. Therefore, a consensus on the new installed view among all members must be done. This new view will increase its identifer and contain the list of current available nodes.

The sequence of view change events generated by the MS are installed in the same order on all available nodes. Views have a partial order relationship defined by its identifier ($\mathcal{V}.id$). Nodes i and j belonging to two different views, $\mathcal{V}$ and $\mathcal{V}'$ with $\mathcal{V}.id < \mathcal{V}'.id$, will both install $\mathcal{V}'$ after $\mathcal{V}$. We assume a primary partition model [RSB93] since we are interested in maintaining a globally consistent replicated database. Therefore, we only allow a group of nodes to update the database. The chosen nodes are the ones forming the majority of members in the group. The remaining nodes, those belonging to a minority partition, will behave as failed nodes.

We have defined the following actions that notify the protocol a view change event whenever a node (or several) joins or leaves the group of nodes as shown in Figure 2.2:

- **join**($\mathcal{V}$). This action passes as parameter the new installed view in the GCS due to the recovery, or addition, of a given node. The $\mathcal{V}$ parameter is a tuple containing the fields: $\mathcal{V}.id$ that corresponds to the new view identifier; and, $\mathcal{V}.availableNodes$ containing the list of available nodes in this new installed view.

- **leave**($\mathcal{V}$). This action is invoked by the GCS each time a node (or several) crashes. As the previous action, the parameter contains the identifier of the new view and the list of current available nodes.

2.3 Database Management System

Each site includes a DBMS that stores a physical copy of the replicated database. The database contains data items uniquely identified by an object identifier, $oid \in \mathcal{OID}$ with $\mathcal{OID}$ being the set of all possible object identifiers. We assume that the DBMS ensures ACID properties of transactions [BHG87] and satisfies the SQL serializable transaction isolation level as depicted in [BBG+95]. As we are in a middleware architecture providing a JDBC interface, replication

protocols only have the standard way to access the DBMS, i.e. by way of SQL statements in the context of a transaction.

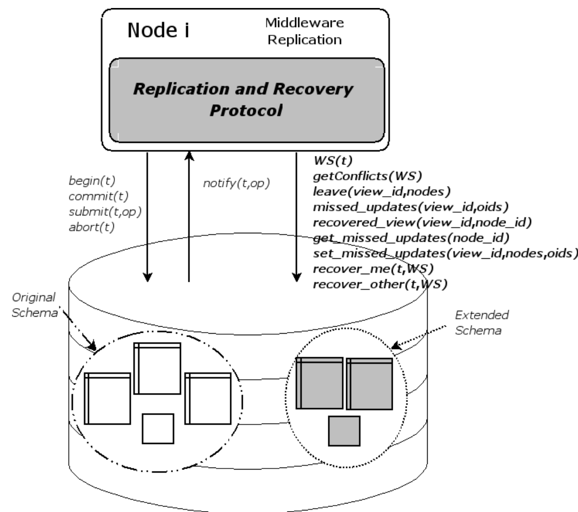


Figure 2.3: Database Management System

The DBMS, as depicted in Figure 2.3, gives to the middleware common operations in order to support standard database operations. It is important to note that these operations may be straightly ported to their equivalent JDBC methods. We will give a rough outline of all of them:

- **begin**(t). This action starts a transaction $t \in \mathcal{T}$ in the underlying database.
- **submit**(t, op). It submits an operation, $op \subseteq \mathcal{OP}$, to the database in the context of transaction $t \in \mathcal{T}$. When the operation is submitted to the DBMS, the transaction may not issue any other operation until the DBMS responds back again reporting the successful completion of the operation, as we will see afterwards.
- **commit**(t). It commits the transaction in the database.
- **abort**(t). Changes done in the context of this transaction are rolled back.

Each time an operation is submitted to the DBMS by a given transaction, the DBMS informs about the completion of the given operation by way of the next action:

- **notify**(t, op). It informs about the success of an operation. It returns two possible values: *run* when the submitted operation has been successfully completed; or, *abort* due to DBMS internals, e.g., deadlock resolution or enforcing serialization.

We also assume that after the successful completion of a submitted operation by a transaction, it can be committed at any time. In other words, a transaction may be unilaterally aborted by the DBMS only while it is performing a submitted operation.

We modify the original database schema with additional metadata tables that store information needed to ease replication and recovery tasks to the protocols. Right now, we will only focus in the recovery metadata table called `MISSED`, the rest will be explained in Chapter 4. It contains three fields: `VIEW_ID`, `NODES` and `OIDS`. The first field contains the view identifier which acts as an index to select the crashed nodes (or not recovered yet), `NODES`, and data items updated in that view, `OIDS`.

Several functions, which are not provided by DBMSs, have been defined in order to facilitate data replication to the replication and recovery protocols [IBDdJM+05]. They can be built by the implementation of the appropriate database triggers, stored procedures and functions. The first two procedures deal with transactions issued by user applications, the remainder are related to the recovery process:

- **WS**(t). It retrieves the set of objects written by t and the respective SQL statements that have modified the database, $WS(t) = \{\langle oids, ops\rangle : oids \subseteq \mathcal{OID}, ops \subseteq \mathcal{OP}\}$.

- **getConflicts**(WS). The set of conflictive transactions between a write set and current active transactions (an active transaction in this context is a transaction that has neither committed nor aborted) at a given site is provided by this function. More formally, $getConflicts(WS(t)) = \{t' \in \mathcal{T} : (WS(t').oids \cup RS(t').oids) \cap WS(t).oids \neq \emptyset\}$.

- **leave**($view_id, nodes$). This action only fills the view identifier and the set of nodes unavailable as a result of a $leave(\mathcal{V})$ action invoked by the GCS [CKV01].

- **missed_updates**($view_id, oids$). Each time a transaction commits, it inserts the object identifiers corresponding to the updates performed by the transaction into the `MISSED` metadata table in the respective *view_id* row.

- **recovered_view**($view_id, node_id$). It removes *node_id* from the record *view_id* contained in the `MISSED` table. This action is invoked after a recovering node, *node_id*, has recovered the *view_id* missed updates. If the `NODES` field becomes empty this registry is deleted from the `MISSED` table.

- **get_missed_updates**(*nodes*). It returns the rows that contain any $j \in nodes$ in the NODES field. This is used to define the objects missed by failed nodes once they rejoin the system. See Chapter 5 for details.

- **set_missed_updates**(*view_id*, *nodes*, *oids*). It adds (or creates) some metadata to the respective row of the MISSED table. This is used by recovering nodes to update its metadata information.

- **recover_me**($t, missed$). This is a special recovery function executed in the context of a recovering transaction, $t \in \mathcal{T}$. As its own name states, it is executed on a recovering node in order to abort any possible transaction (if any) currently accessing objects contained in *missed* ($missed \subseteq \mathcal{OID}$) and preventing that local user transactions access these data items (e.g. executing an "UPDATE" statement). Hence, this function defines a partition in the database. As this is a database function, the recovery protocol must wait for the *notify* action too. Meanwhile, it can receive the state of outdated registers from the recoverer node that will be applied in the context of this recovery transaction, once it receives the *notify* of this function. This recovery partition is released once the missed updates are applied in the node.

- **recover_other**($t, missed$). The other recovery function is like the previous one, but it is executed on the recoverer node, and in the context of a recovering transaction $t \in \mathcal{T}$, once it knows which registers are outdated. This function rolls back transactions currently updating objects contained in *missed* ($missed \subseteq \mathcal{OID}$) only at the time it sets up the database partition; afterwards, they will get blocked. It prevents local user transactions from updating these registers while they are being transferred to the recovering node (e.g. executing a "SELECT FOR UPDATE" statement). This function acts as a submitted operation, therefore we have to wait for its completion via the *notify* action. This partition remains in the recoverer node until the data items belonging to it have been transferred to the recovering node.

2.4 Transactions

Client applications access the system through their closest site to perform transactions by way of actions introduced in Figure 2.1. As it was pointed out, this is an abstraction, in fact applications

employ the same JDBC interface as the underlying DBMS. Each transaction identifier includes the information about the site where it was first created ($t.site$), called its *transaction master site*. It allows the protocol to know if it is a local or a remote transaction. Each transaction has a unique priority value ($t.priority$) based on transaction information.

A transaction $t \in \mathcal{T}$ created at site $i \in \mathcal{N}$ ($t.site = i$) is locally executed and follows a sequence of actions initiated by *create*(t) and continued by multiple *begin_operation*(t, op), *end_operation*(t, op) pairs actions in a normal behavior. The *begin_commit*(t) action states that the user wishes to commit the transaction. The *end_commit*(t) notifies about the successful completion of the transaction on the replicated databases, as it is depicted in Figure 2.4. However, an *abort*(t) action may be generated by the local DBMS or by a replication protocol decision. For simplicity, we do not consider an application abort. On the following, we give an outline of the main features of the actions performed by a user transaction:

- **create**(t). It starts the transaction on the system. This is only executed on the transaction master site ($t.site$).
- **begin_operation**(t, op). This is the way a user transaction executes SQL statements to the replicated database in the context of a transaction. In our replication protocol, it may not invoke this action again until it gets notified by the next action.
- **end_operation**(t, op). This is invoked by the middleware instance each time it has been successfully processed the SQL statement by the DBMS. In our concrete case, when the database has completed the operation.
- **begin_commit**(t). This action points out that the user application wishes to commit the transaction. The middleware will notify this fact to the replication protocol, that makes the replication protocol start to manage the commit of t at the rest of replicas.
- **end_commit**(t). This is the middleware response action to the previous one. It means that all updates of the transaction have been applied at all available replicas.

These actions represent the interface with the middleware. The invocation of an action is notified to the replication protocol, which takes the right steps so as to guarantee data consistency. As it has been pointed out, this replication protocol does not interact with the rest of replicas until the application wants to commit its transaction.

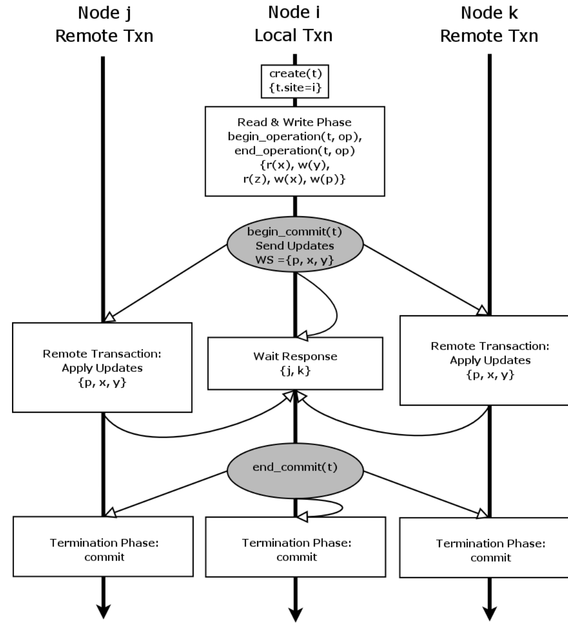


Figure 2.4: Transaction execution model

2.5 State Transition Systems

Several protocols will be introduced in this thesis. They will be formally introduced using a state transition system as presented in [Sha93]. In the following we briefly explain this system model. A state transition system M is defined by:

- $\mathbf{Signature}_M$. A set of actions.
- $\mathbf{States}_M$. A set of state variables and their domains, including an initial condition for each variable.
- $\mathbf{Transitions}_M$. For each action $\pi \in \mathbf{Signature}_M$:
  - $pre_M(\pi)$. It is the precondition of π in M. It is a predicate in $\mathbf{States}_M$ that enables its execution.
  - $eff_M(\pi)$. The effects of the action π in M. It is a sequential program that atomically modifies $\mathbf{States}_M$.
- A finite description of fairness requirements.

In the following we omit M for simplicity. We assume that the initial state is nonempty. For each action π, its associated precondition, $pre(\pi)$, and effects, $eff(\pi)$, define a set of *state*

transitions. More formally, $\{(p, \pi, q) \colon p, q$ are system states; p satisfies $pre(\pi)$; and q is the result of executing $eff(\pi)$ in $p\}$.

An *execution* is a sequence of the form: $\alpha = s_0 \pi_1 s_1 \ldots \pi_z s_z \ldots$ where the s_z's are system states, the π_z's are actions, s_0 is the initial state, and every (s_z, π_z, s_{z+1}) is a transition of π_z. An execution can be infinite or finite. By definition, a finite execution ends in a state. Note that for any execution α, every finite prefix of α ending in a state is also an execution. Let **Executions** denote the set of executions for a system. **Executions** is enough for stating safety properties but not its liveness properties (because it includes executions where system liveness requirements are not satisfied).

We next define the executions of the system that satisfy liveness requirements. Let Π be a subset of **Signature**. The precondition of Π, denoted $pre(\Pi)$, is defined by $[\exists \pi \in \Pi \colon\ pre(\pi)]$. Thus, Π is enabled (disabled) in a state if and only if some (no) action of Π is enabled in the state. Let $\alpha = s_0 \pi_1 s_1 \ldots \pi_z s_z \ldots$ be an infinite execution. We say that Π is enabled (disabled) infinitely often in α if Π is enabled (disabled) at an infinite number of s_z's belonging to Π. We say that Π occurs infinitely often in α if an infinite number of π_z's belong to Π.

An execution α satisfies *weak fairness for* Π [LT87] if and only if one of the following occurs:

1. α is finite and Π is disabled in the last state of α.

2. α is infinite and either Π occurs infinitely often or is disabled infinitely often in α.

Informally, this means that if Π is enabled continuously, then it eventually occurs.

An execution α is *fair* if and only if it satisfies every fairness requirement of the system. Let **FairExecutions** denote the set of fair executions of the system. **FairExecutions** is sufficient for defining the liveness properties of the system, as well as its safety properties.

We allow actions to have parameters. This is a convenient way of defining a collection of actions. For example, consider an action $\pi(i)$ with precondition $pre(\pi(i)) \equiv\ x = 0$ and effects $eff(\pi(i)) \equiv\ x \leftarrow i$, where x is an integer and where the parameter i ranges over $(1, 2, \ldots, 50)$. Action $\pi(i)$ actually specifies a collection of 50 actions, $\pi(1), \pi(2), \ldots, \pi(50)$.

We use the term *state formula* to refer to a predicate in the state variables of the system. A state formula evaluates to true or false for each state of the system. We consider assertions of the form *Invariant*(P) and P *leads-to* Q, where P and Q are state formulas.

Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be an (finite or infinite) execution of the system. The execution α satisfies *Invariant*(P) if and only if every state s_z in α satisfies P. The execution α satisfies P *leads-to* Q if and only if for every s_z in α that satisfies P there is an s_k in α, $k \geq z$, that satisfies Q.

The system satisfies *Invariant*(P) if and only if every execution of the system satisfies *Invariant*(P). Respectively, the system satisfies P *leads-to* Q if and only if every fair execution of the system satisfies P *leads-to* Q. We allow assertions that are made up of invariant assertions or leads-to assertions joined by logical connectives and containing parameters.

2.6 Discussion

In this Chapter we have introduced an abstraction of the MADIS middleware architecture which will be shown in detail in Chapter 4. This architecture provides a standard database interface (JDBC) to client applications, hence they do not need to be modified. In the same way, the middleware instance sees a JDBC interface with the underlying DBMS. The DBMS has to be modified so as to include some functions that facilitate database replication via middleware, although we do not have modified the database internals in order to generate these functions. It obviously penalizes system performance but it maintains its validity for different DBMS vendors. Each middleware instance has a replication protocol instance embedded in it. The replication protocol is responsible for the communication with the rest of nodes using a GCS.

The abstraction followed eases the definition of replication and recovery protocols. As this Thesis focuses on the replication and recovery protocols, it let us study them in depth without worrying at implementation details. We have described the three main components of our system as interfaces with actions that each component may invoke on the other and viceversa. This allows us to formulate the replication and recovery protocols as state transition systems, as it will be shown in Chapters 3 and 5.

We have described our failure system model. This will help us to clarify how our recovery algorithm proposal works. Its description will be formulated in Chapter 5. With all the additional features we have included in the underlying DBMS, it will not be a difficult task to accomplish database recovery without penalizing system availability.

Finally, we have introduced the formal notation we are going to use in the description of our protocols in Chapters 3 and 5. They are going to be defined as state transition systems. We have

stated all the background around this way of introducing protocols so that the definition of safety and liveness properties will be straightforward with this notation.

Chapter 3

Middleware Replication Protocols

This Chapter presents the Basic Replication Protocol (BRP) for the MADIS [IBDdJM+05] middleware architecture which is introduced in Chapter 4. The protocol is based on the O2PL protocol proposed by Carey et al. in [CL91], without needing lock management or previous transaction knowledge at the middleware layer. This fact avoids to reimplement on the replication protocol component features that can be obtained in a simple way from the local DBMS. The replication protocol is formalized and proved using a formal state transition system [Sha93]. The interfaces shown in Chapter 2 serve us to define the actions performed by the BRP in the different components of the system.

The 1CS [BHG87] property for database replication is obtained from the combination of assumed serializability [BBG+95] on the local DBMSs and the message ordering imposed by the replication protocol. As a result of the correctness proof, we may add several enhancements and variations for BRP. These modifications lead to the definition of the Enhanced Replication Protocol (ERP). This new protocol reduces response times and transaction abortion rates by removing the Two Phase Commit (2PC) rule [BHG87] and the use of queues. Failures and recovery issues for these protocols are thoroughly explained in Chapter 5. Finally, we are very interested in comparing BRP and ERP with total order based replication protocols [AT02, EPZ05, JPPMKA02, KA00b, KPA+03, WK05, LKPMJP05]. Hence, we introduce another replication protocol, named Total Order Replication Protocol with Enhancements (TORPE), using the same formalism as the previous ones that will be also implemented in MADIS for its comparison with the implementations of BRP and ERP.

3.1 Introduction

BRP is an eager update everywhere replication protocol adapted to the MADIS [IBDdJM+05] middleware architecture. It follows the idea of the atomic commitment protocol, more precisely the 2PC protocol, and is an adaptation of the O2PL protocol proposed by Carey et al. [CL91]. We need no lock management at the middleware level since we rely on the serializable behavior, as stated in [BBG+95], of the underlying DBMS. Besides, it uses basic features present in common DBMSs to generate the set of metadata needed to maintain each data item and conflict detection among transactions, as pointed out in Chapter 2. This allows the underlying database to perform the tasks needed to support the replication protocol and simplifies the BRP implementation. However, the BRP must have a dynamic deadlock prevention schema in order to prevent distributed deadlocks appearance, which is based on the transaction priority and its state in the system.

The BRP presents some drawbacks. It is a 2PC protocol so it must wait for the application of updates at all available sites before committing a transaction. The deadlock prevention schema may unnecessarily abort two conflicting transactions due to the lack of order in the message delivery at different sites. Nevertheless, the BRP gives the basis to obtain a new protocol that avoids such limitations. The study of the BRP and its subsequent formal correctness proof show us the key points that serve to improve its behavior while maintaining its correctness. This new protocol is the ERP which provides additional enhancements and modifications than those featured in the BRP. It optimizes its response time and reduces the abortion rate of the BRP. However, there exists a group of replication protocols [AT02, EPZ05, JPPMKA02, KA00b, KPA+03, WK05, LKPMJP05] that uses the total order delivery of multicast messages [CKV01] to order transactions in the system. Hence, distributed deadlock is avoided. We introduce TORPE as a simplification of the previous protocols that uses the total order multicast to propagate the updates done by a transaction to the rest of sites.

The contributions of this Chapter are the following: (*a*) We present an example of a lock-based replication protocol that delegates such a lock management to the underlying DBMS, simplifying the development of the replication protocol in the middleware layer. (*b*) The BRP is able to manage unilateral aborts generated by the DBMS since it is a 2PC protocol. Only a few current replication protocols are able to manage such kind of aborts [Ped99]. (*c*) It provides a formal correctness proof of a variation of the O2PL protocol [CL91]. We have not found a proof of this kind for the original O2PL protocol nor any of its variations. (*d*) We have avoided distributed deadlock

cycles among sites by means of a deadlock prevention schema based on dynamic priorities that come imposed by the atomicity of transactions. (*e*) A new replication protocol ERP, along with its correctness proof, that increases its performance provided that unilateral aborts do not occur and that the priority rule is modified; ERP does not have to wait for applying the updates at all available sites before committing the transaction. (*f*) The ERP reduces its abortion rate by the use of a queue and deciding a transaction abortion in its master site; hence if two conflicting transactions are delivered in different order at different sites then at least one of then never gets rolled back. (*g*) A formalization of a total order based replication protocol TORPE which is an adaptation of the previous protocols using the total order multicast provided by a GCS to send the updates of a transaction to the rest of available sites.

The rest of the Chapter is organized as follows: A formal description of the BRP in a failure free environment is given in Section 3.2. The BRP correctness proof is shown in Section 3.3. Section 3.4 introduces the ERP. The correctness proof of the ERP is depicted in Section 3.5. The formal description of TORPE as a state transition system is given in Section 3.6. Finally, some discussions about the protocols and related works are introduced in Section 3.7.

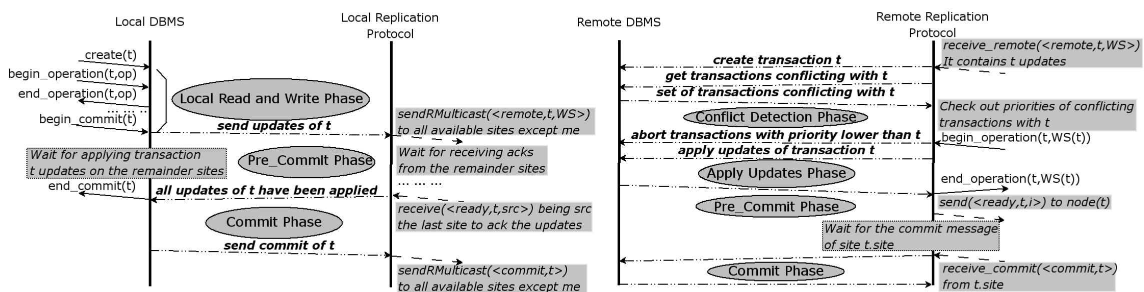


Figure 3.1: Execution example of a local (left) and a remote transaction (right)

3.2 Basic Replication Protocol Description

Informally, each time a client application issues a transaction (*local transaction*), all its operations are locally performed over its master site. The remaining sites enter in the context of this transaction when the application requests for the commitment of the transaction. All update operations are grouped and sent to the rest of available sites, without any further assumption about message ordering, following a ROWAA approach [BHG87]. If the given transaction is a read only one, as its associated writeset is empty, it directly commits. We do not consider its behavior for simplic-

Signature:
$\{\forall\, i \in \mathcal{N}, t \in \mathcal{T}, m \in \mathcal{M}, op \subseteq \mathcal{OP}$: **create**$_i(t)$, **begin_operation**$_i(t, op)$, **end_operation**$_i(t, op)$, **begin_commit**$_i(t)$, **end_commit**$_i(t)$, **local_abort**$_i(t)$, **receive_remote**$_i(t, m)$, **receive_ready**$_i(t, m)$, **receive_commit**$_i(t, m)$, **receive_abort**$_i(t, m)$, **receive_rem_abort**$_i(t, m)$, **discard**$_i(t, m)\}$.

States:
$\forall\, i \in \mathcal{N}, \forall\, t \in \mathcal{T}$: $status_i(t) \in \{$idle, start, active, blocked, pre_commit, aborted, committed$\}$,
initially $(node(t) = i \Rightarrow status_i(t) =$ start$) \wedge (node(t) \neq i \Rightarrow status_i(t) =$ idle$)$.
$\forall\, i \in \mathcal{N}, \forall\, t \in \mathcal{T}$: $participants_i(t) \subseteq \mathcal{N}$, initially $participants_i(t) = \emptyset$.
$\forall\, i \in \mathcal{N}$: $channel_i \subseteq \{m : m \in \mathcal{M}\}$, initially $channel_i = \emptyset$.
$\forall\, i \in \mathcal{N}$: $\mathcal{V}_i \in \{\langle id, availableNodes\rangle : id \in \mathbb{Z}, availableNodes \subseteq \mathcal{N}\}$, initially $\mathcal{V}_i = \langle 0, \mathcal{N}\rangle$.

Transitions:

create$_i(t)$ // $t.site = i$ //
pre $\equiv status_i(t) =$ start.
eff $\equiv DB_i.begin(t)$;
  $status_i(t) \leftarrow$ active.

begin_operation$_i(t, op)$ // $t.site = i$ //
pre $\equiv status_i(t) =$ active.
eff $\equiv DB_i.submit(t, op)$;
  $status_i(t) \leftarrow$ blocked.

end_operation$_i(t, op)$
pre $\equiv status_i(t) =$ blocked $\wedge$ $DB_i.notify(t, op) =$ run.
eff $\equiv status_i(t) \leftarrow$ active;
  if $t.site \neq i$ **then**
    $sendRUnicast(\langle ready, t, i\rangle, t.site)$;
    $status_i(t) \leftarrow$ pre_commit.

begin_commit$_i(t)$ // $t.site = i$ //
pre $\equiv status_i(t) =$ active.
eff $\equiv status_i(t) \leftarrow$ pre_commit;
  $participants_i(t) \leftarrow \mathcal{V}_i.availableNodes \setminus \{i\}$;
  $sendRMulticast(\langle remote, t, DB_i.WS(t)\rangle, participants_i(t))$.

end_commit$_i(t)$ // $t \in \mathcal{T} \wedge t.site = i$ //
pre $\equiv status_i(t) =$ pre_commit $\wedge$ $participants_i(t) = \emptyset$.
eff $\equiv DB_i.commit(t)$;
  $status_i(t) \leftarrow$ committed;
  $sendRMulticast(\langle commit, t\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$.

receive_ready$_i(t, m)$ // $t \in \mathcal{T} \wedge t.site = i$ //
pre $\equiv status_i(t) =$ pre_commit $\wedge$ $participants_i(t) \neq \emptyset \wedge$
  $m = \langle ready, t, source\rangle \in channel_i$.
eff $\equiv receive_i(m)$;
  $participants_i(t) \leftarrow participants_i(t) \setminus \{source\}$.

local_abort$_i(t)$
pre $\equiv status_i(t) =$ blocked $\wedge$ $DB_i.notify(t, op) =$ abort.
eff $\equiv status_i(t) \leftarrow$ aborted; $DB_i.abort(t)$;
  if $t.site \neq i$ **then**
    $sendRUnicast(\langle rem_abort, t\rangle, t.site)$.

receive_commit$_i(t, m)$ // $t \in \mathcal{T} \wedge t.site \neq i$ //
pre $\equiv status_i(t) =$ pre_commit $\wedge$ $m = \langle commit, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$;
  $DB_i.commit(t)$;
  $status_i(t) \leftarrow$ committed.

receive_remote$_i(t, m)$ // $t \in \mathcal{T} \wedge t.site \neq i$ //
pre $\equiv status_i(t) =$ idle $\wedge$ $m = \langle remote, t, WS\rangle \in channel_i$.
eff $\equiv receive_i(m)$;
  $conflictSet \leftarrow DB_i.getConflicts(WS)$;
  if $\exists\, t' \in conflictSet$: $\neg higher_priority(t, t')$ **then**
    $status_i(t) \leftarrow$ aborted;
    $sendRUnicast(\langle rem_abort, t\rangle, t.site)$
  else
    // *The delivered remote transaction has*
    // *the highest priority or no conflicts*
    $\forall\, t' \in conflictSet$:
      $DB_i.abort(t')$;
      if $status_i(t') =$ pre_commit $\wedge$ $t'.site = i$ **then**
        $sendRMulticast(\langle abort, t'\rangle,$
          $\mathcal{V}_i.availableNodes \setminus \{i\})$;
      $status_i(t') \leftarrow$ aborted;
    $DB_i.begin(t)$;
    $DB_i.submit(t, WS.ops)$;
    $status_i(t) \leftarrow$ blocked.

receive_abort$_i(t, m)$ // $t \in \mathcal{T} \wedge t.site \neq i$ //
pre $\equiv status_i(t) \notin \{$aborted, *committed*$\} \wedge$
  $m = \langle abort, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$;
  $DB_i.abort(t)$;
  $status_i(t) \leftarrow$ aborted.

receive_rem_abort$_i(t, m)$ // $t.site = i$ //
pre $\equiv status_i(t) \notin \{$aborted, *committed*$\} \wedge$
  $m = \langle rem_abort, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$; $DB_i.abort(t)$; $status_i(t) \leftarrow$ aborted;
  $sendRMulticast(\langle abort, t\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$.

discard$_i(t, m)$ // $t \in \mathcal{T}$ //
pre $\equiv status_i(t) =$ aborted $\wedge\, m \in channel_i$.
eff $\equiv receive_i(m)$.

$\diamond$ **function** $higher_priority(t, t') \equiv t.site = j \neq i$
  $\wedge\, (a \vee b \vee c)$
(a) $t'.site = i \wedge status_i(t') \in \{$active, blocked$\}$
(b) $t'.site = i \wedge status_i(t') =$ pre_commit $\wedge$
  $t.priority > t'.priority$
(c) $t'.site = k \wedge k \neq j \wedge k \neq i \wedge status_i(t') =$ blocked $\wedge$
  $t.priority > t'.priority$

Figure 3.2: State transition system for the Basic Replication Protocol (BRP). *pre* indicates precondition and *eff* effects respectively

ity. This replication protocol is different from the eager update everywhere 2PL protocol model assumed by [GHOS96]. Instead of sending multiple messages for each operation issued by the

transaction, only three messages are needed per transaction: one containing the *remote* update, another one for the *ready* message, and, finally, a *commit* message.

All updates are applied in the context of another local transaction (*remote transaction*) on the given local database where the message is delivered. This node will send back to the transaction master site a message saying it is ready to commit the given transaction. When the reception of *ready* messages is finished, that is, all nodes have answered to the transaction master site, it sends a message saying that the transaction is committed. Figure 3.1 shows an execution, depicting actions and message exchange, of a local transaction (left) and its respective remote execution (right) when everything goes fine.

Our replication protocol relies for conflict detection on the mechanism implemented in the underlying DBMS which guarantees an ANSI SQL serializable isolation level [BBG+95]. This assumption frees us from implementing locks at the middleware level. However, this latter assumption is not enough to prevent distributed deadlock [BHG87]. We have avoided this problem using a deadlock prevention schema based on priorities.

In the following, we present this replication protocol as a formal state transition system using the formal model of [Sha93]. In Figure 3.2, a formal description of the signature, states and steps of the replication protocol for a site i is introduced. An action can be executed only if its precondition is enabled. The effects modify the state of the system as stated by the sequence of instructions included in the action effects. Actions are atomically executed. It is assumed weak fairness for the execution of each action.

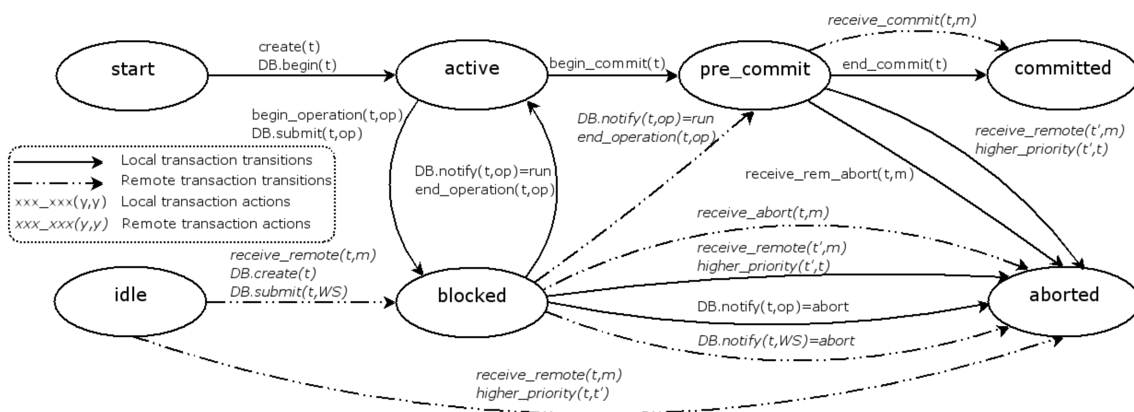


Figure 3.3: Valid transitions for a given $status_i(t)$ of a transaction $t \in \mathcal{T}$

We will start with the states defined for this replication protocol. Each site has its own state

variables (i.e., they are not shared among other nodes). The $status_i(t)$ variable indicates the execution state of a given transaction; its valid transitions are shown in Figure 3.3. The $participants_i(t)$ variable keeps track of the sites that have not yet sent the *ready* message to transaction t whose master site is i. $\mathcal{V}_i$ is the system current view, which in this protocol description context, with a failure-free assumption, is $\langle 0, \mathcal{N} \rangle$.

Each action is subscripted by the site at which it is executed. The set of actions includes: $create_i(t)$, $begin_operation_i(t, op)$, $end_operation_i(t, op)$, $begin_commit_i(t)$ and $end_commit_i(t)$. These actions are the ones executed by the application in the context of a local transaction. The $end_operation_i(t, op)$ is an exception to this. It is shared with a remote transaction that sends the *ready* message to its transaction master site when the operation has been completed. The $begin_commit_i(t)$ action sends the write-set and update statements of a transaction t to every site and starts the replica control for this transaction. This set of actions is entirely self-explanatory from inspection of Figure 3.2.

The key action of our replication protocol is the $receive_remote_i(t, \langle remote, t, WS \rangle)$ one. Once the *remote* message at node i is received, this action finds out the set of transactions that conflicts with the received write set (WS) in the local database. The remote updates, for that WS, will only be applied if there is no conflicting transaction at node i having a higher priority than the received one. The $higher_priority(t, t')$ defines a dynamic priority deadlock prevention function, since the transaction global priority depends on the state of the transaction ($status_i(t)$) and its own unique priority ($t.priority$). As a final remark, a delivered remote transaction has never a higher priority than other conflictive remote transaction at node i in the *pre_commit* state; this fact is needed to guarantee the transaction execution atomicity.

If there exists a conflicting transaction at i with higher priority, the remote message is ignored and a *remote abort* message to the transaction master site is sent. In this protocol version we do not allow transactions to wait among different sites, therefore deadlock situations are trivially avoided. Finally, if the remote transaction is the one with the highest priority among all at i, then every conflictive transaction is aborted and the transaction updates are submitted for their execution to the underlying DBMS. Aborted local transactions in *pre_commit* state with lower priority will multicast an *abort* message to the rest of sites. The finalization of the remote transaction ($end_operation_i(t, op)$), upon successful completion of $DB_i.submit(t, WS.ops)$, is in charge of sending the *ready* message to the transaction master site. Once all *ready* messages are col-

lected from all available sites, the transaction master site commits ($end_commit_i(t)$) and multicasts a *commit* message to all available nodes. The reception of this message commits the transaction at the remainder sites ($receive_commit_i(t, \langle commit, t\rangle)$).

When the remote updates fail while being applied in the DBMS (unilateral aborts), the $local_abort_i(t)$ is responsible for sending the *remote abort* message to the transaction master site. Once the updates have been finally applied, the transaction waits for the commit message from its master site. One can note that the remote transaction is in the *pre_commit* state and that it is committable from the DBMS point of view.

3.3 BRP Correctness Proof

This Section contains the proofs (atomicity and 1CS) of the BRP, introduced in Figure 3.2, in a failure free environment.

Let us start showing that BRP is deadlock free, assuming that deadlocks involving exclusively local transactions at a given site are directly resolved by the underlying local DBMS executing the $local_abort_i(t)$ action. The BRP does not permit a transaction to wait for another transaction at a different site. Any wait-for relation among transactions at different sites are prevented when $receive_remote_i(t, \langle remote, t, WS\rangle)$ is executed. By inspection, its effects $\forall t' \in DB_i.getConflicts(WS.oids)$: $(DB_i.abort(t') \wedge status_i(t') = \text{aborted})$ if $\forall t'$, $higher_priority(t, t')$ are true. On the contrary, $status_i(t) = \text{aborted}$, and the received remote transaction is not executed. Therefore, wait-for relations among transactions at different sites are excluded and distributed deadlock situations never occur in the system.

The BRP must guarantee the atomicity of a transaction; that is, the transaction is either committed at all available sites or is aborted at all sites. If a transaction t is in *pre_commit* state then it is committable from the local DBMS point of view. Therefore, if a local transaction commits at its master site ($node(t) = i$) (i.e. it executes the $end_commit_i(t)$ action), it multicasts a *commit* message to each remote transaction it has previously created. Such remote transactions are also in the *pre_commit* state. Priority rules ensure that remote transactions in the *pre_commit* state are never aborted by a local transaction or a remote one. Thus, by the reliable communication property, the *commit* message will be eventually delivered. Every remote transaction of t will be committed via the execution of the $receive_commit_j(t, \langle commit, t\rangle)$ action with $j \neq i$. On the contrary, if a

transaction t aborts, every remote transaction previously created for t will be aborted. Unilateral aborts are considered [Ped99], they may appear during the execution of a remote transaction by a DBMS decision. We formalize such behavior in the following Properties and Lemmas.

In this Section we use the notation and definitions introduced in Chapter 2. Each action has a precondition (*pre* in Figure 3.2), a predicate over the state variables. Its effects (*eff* in Figure 3.2) are a sequence program that is atomically executed and that modifies state variables. An execution α, is a sequence of the form $s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$. As stated in Chapter 2, it is sufficient to consider the set of all possible executions in order to state safety properties. However, liveness properties require the notion of fair execution. We assume that each BRP action requires weak fairness. Informally, a fair execution will satisfy weak fairness for π; if π is continuously enabled, then it will be eventually executed.

The following Property formalizes the *status* transition shown in Figure 3.3. It indicates that some *status* transitions are unreachable, i.e., if $s_k.status_j(t)$ = pre_commit and $s_{k'}.status_j(t)$ = committed with $k' > k$. There is no action in α such that $s_{k''}.status_j(t)$ = aborted with $k' > k'' > k$.

Property 1. *Let $\alpha = s_o\pi_1 s_1 \ldots \pi_z s_z \ldots$ be an arbitrary execution of the BRP automaton and $t \in \mathcal{T}$. Let $\beta = s_0.status_j(t)\, s_1.status_j(t) \ldots s_{z'}.status_j(t)$ be the sequence of status transitions of t at site $j \in \mathcal{N}$, obtained from α by removing the consecutive repetitions of the same $status_j(t)$ value and maintaining the same order apparition in α. The following Property holds:*

1. *If $node(t) = j$ then β is a finite prefix of the regular expression:*

   - *start · active · (blocked · active)* · pre_commit · committed*
   - *start · active · (blocked · active)* · pre_commit · aborted*
   - *start · (active · blocked)$^+$ · aborted*

2. *If $node(t) \neq j$ then β is a finite prefix of the regular expression:*

   - *idle*
   - *idle · blocked · pre_commit · committed*
   - *idle · blocked · pre_commit · aborted*
   - *idle · blocked · aborted*
   - *idle · aborted*

The Property is simply proved by induction over the length of α following the preconditions and effects of the BRP actions introduced in Figure 3.2. A *status* transition for a transaction t in Property 1 is associated with an operation on the *DB* module where the transaction was created, i.e. *pre_commit* to *committed* involves the *DB.commit*(t) operation. These aspects are straightforward from the BRP state transition system inspection in Figure 3.2 and the *status* transition shown in Figure 3.3.

The following Property introduces the BRP invariant properties. It states that if a transaction is *committed* at a different site from its master site, it is due to the fact that it has been already *committed* at the master site. If a transaction is *committed* at its master site it is because all remote transactions executing at the remainder available sites were previously in the *pre_commit* state. A remote transaction in the *pre_commit* state may only change its state due to a *commit* or an *abort* message coming from its master site. Finally, if a transaction is *committed* at its master site then all available sites will be either *committed* or in the *pre_commit* state. This Property is needed to prove Lemma 1.

Property 2. *Let* $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ *be an arbitrary execution of the BRP automaton and* $t \in \mathcal{T}$, *with* $node(t) = i$.

1. *If* $\exists\, j \in \mathcal{N} \setminus \{i\}\colon s_z.status_j(t) = committed$ *then* $s_z.status_i(t) = committed$.
2. *If* $s_z.status_i(t) = committed$ *then* $\forall\, j \in \mathcal{N}\colon \exists\, z' < z\colon\ s'_z.status_j(t) = pre_commit$.
3. *If* $\exists\, z' < z\colon s_{z'}.status_j(t) = s_z.status_j(t) = pre_commit$ *for any* $j \in \mathcal{N} \setminus \{i\}$ *then* $\forall\, z''\colon z' < z'' \leq z\colon \pi_{z''} \notin \{receive_commit_j(t, \langle commit, t\rangle), receive_abort_j(t, \langle abort, t\rangle)\}$.
4. *If* $s_z.status_i(t) = committed$ *then* $\forall\, j \in \mathcal{N} \setminus \{i\}\colon s_z.status_j(t) \in \{pre_commit, committed\}$.

Proof.

1. By induction over the length of α. The Property holds for the initial state $s_0\colon s_0.status_j(t) =$ idle. By hypothesis, assume the Property holds at s_{z-1}, the induction is proved for each (s_{z-1}, π_z, s_z) transition of the BRP automaton. If $s_{z-1}.status_j(t)$ = committed, the *status* does not change. By Property 1.2, the only enabled action is $\pi_z = discard_i(t, m)$, with $m \in \mathcal{M}$, which does not modify $s_{z-1}.status_i(t)$. If $s_{z-1}.status_j(t) \neq$ committed, only $\pi_z = receive_commit_j(t, \langle commit, t\rangle)$ makes $s_z.status_j(t)$ = committed. By its precondition, we have that $\langle commit, t\rangle \in s_{z-1}.channel_j$. The only action that sent such a message is $\pi_{z'} =$

$end_commit_i(t)$ with $z' < z$. By its effects $s_{z'+1}.status_i(t)$ = committed and by Property 1.1, the $status_i(t)$ never changes. Hence, the Property holds.

2. Let z be the first state such that $s_z.status_i(t)$ = committed. We assume that for some $j \in \mathcal{N}, \nexists z' < z$: $s_{z'}.status_j(t)$ = pre_commit. Let $\pi_z = end_commit_i(t)$, be the action making $s_z.status_i(t)$ = committed from $s_{z-1}.status_i(t)$ = pre_commit. By its effects, $s_z.participants_i(t) = \emptyset$. By Property 1.1, $\exists z'' < z$: $\pi_{z''} = begin_commit_i(t)$, hence $s_{z''}.participants_i(t) = \mathcal{N} \setminus \{i\}$. Thus, $j \in s_{z''}.participants_i(t) \wedge j \notin s_z.participants_i(t)$. The only action that remove j is $\pi_{z'''} = receive_ready_i(t, \langle ready,\ t, j\rangle)$ where $z'' < z''' < z$. The $\pi_{z'} = end_operation_j(t, WS.ops)$ action is the only one that generates such a message. By its effects $s_{z'}.status_j(t)$ = pre_commit and $z' < z'''$. Thus, by contradiction, the Property holds.

3. The effects of the $receive_commit_j(t, \langle commit, t\rangle)$ and $receive_abort_j(t, \langle abort, t\rangle)$ actions make $status_j(t)$ to be *committed* or *aborted* respectively. In order to prove the Property we need to show that if $s_{z'}.status_j(t)$ = pre_commit and $(s_{z'}, \pi_{z'+1}, s_{z'+1})$ is a transition then $\pi_{z'+1} \in \{receive_commit_j(t, \langle commit, t\rangle), receive_abort_j(t, \langle abort, t\rangle)\}$ will be the unique actions that change the pre_commit value of $status_j(t)$. By inspection of the actions at j we have the following candidates:

   - $\pi_{z'+1} = local_abort_j(t)$. This action, modeling a deadlock resolution or a unilateral abort by the DB_j module, is disabled at $s_{z'}$ because $s_{z'}.status_j(t)$ = pre_commit and is committable by DB_j module at any time.

   - $\pi_{z'+1} = receive_rem_abort_j(t, \langle rem_abort, t\rangle)$. This action is not enabled because t is remote at j. The $\langle rem_abort, t\rangle$ message is only received at the transaction master site of t, $node(t) = i$.

   - $\pi_{z'+1} = receive_remote_j(t', \langle remote, t', WS'\rangle)$, with $t' \in \mathcal{T} \wedge node(t') \neq j$. If $t \in DB_i.getConflicts(WS'.ops)$ then t will be aborted in case of $higher_priority(t',\ t)$ is true. However, $higher_priority(t', t)$ is false due to the fact that $s_{z'}.status_j(t)$ = pre_commit, being $node(t) \neq j$. Therefore, $s_{z'+1}.status_j(t)$ = pre_commit. A new remote transaction at j may not abort a remote transaction in the *pre_commit* state at j.

4. If $s_z.status_i(t)$ = committed, by Property 1.1 we have that $\forall s_{z'} \in \alpha$: $s_{z'}.status_i(t) \neq$ aborted $\wedge$ $\langle abort, t\rangle \notin s_{z'}.channel_j$ for all $j \in \mathcal{N}$. Thus, the $receive_abort_j(t, \langle abort, t\rangle)$ action is disabled at any state of α. By Property 2.2, we have that $\exists z' < z$: $s_{z'}.status_j(t)$ = pre_commit for all $j \in \mathcal{N}$. By Property 2.3, either $s_z.status_j(t)$ = pre_commit or $\exists z' < z'' < z$: $\pi_{z''} = receive_commit_j(t)$. In the latter case, $s_{z''}.status_j(t)$ = committed and by Property 1.2 its associated *status* never changes. Therefore, $s_z.status_j(t) \in$ {pre_commit, committed}. □

The following Lemma, a liveness property, states the atomicity of committed transactions.

Lemma 1. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the BRP automaton and $t \in \mathcal{T}$ with $node(t) = i$. If $\exists j \in \mathcal{N}$: $s_z.status_j(t)$ = committed then $\exists z' > z$: $s_{z'}.status_j(t)$ = committed for all $j \in \mathcal{N}$.*

Proof. If $j \neq i$ by Property 2.1 (or $j = i$) $s_z.status_i(t)$ = committed. By Property 2.4, $\forall j \in \mathcal{N} \setminus \{i\}$: $s_z.status_j(t) \in$ {pre_commit, committed}. Without loss of generality, assume that s_z is the first state where $s_z.status_i(t)$ = committed and $s_z.status_j(t)$ = pre_commit. By the effects of $\pi_z = end_commit_i(t)$, we have that $\langle commit, t\rangle \in s_z.channel_j$. By Property 2.4 invariance either $s_z.status_j(t)$ = committed or $s_z.status_j(t)$ = pre_commit and $\langle commit, t\rangle \in s_z.channel_j$. In the latter case the $receive_commit(t, \langle commit, t\rangle)$ action is enabled. By weak fairness assumption, it will be eventually delivered, thus $\exists z' > z$: $\pi_{z'} = receive_commit_j(t, m)$. By its effects, $s_{z'}.status_j(t)$ = committed. □

We may formally verify that if a transaction is aborted then it will be aborted at all nodes in a similar way. This is stated in the following Lemma.

Lemma 2. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the BRP automaton and $t \in \mathcal{T}$ with $node(t) = i$. If $s_z.status_i(t)$ = aborted then $\exists z' \geq z$: $s_{z'}.status_j(t)$ = idle for all $j \in \mathcal{N} \setminus \{i\}$ $\vee$ $s_{z'}.status_j(t)$ = aborted for all $j \in \mathcal{N}$.*

Proof. We have the following cases:

(I) If $s_z.status_k(t)$ = aborted, $k = i$, and it has reached this state due to a $\pi_z = local_abort_k(t)$ action, then $s_{z-1}.status_i(t)$ = blocked $\neq$ pre_commit (Property 1.1). Thus, $\forall j \in \mathcal{N} \setminus \{i\}$: $s_z.status_j(t)$ = idle. It is simple to show that this situation is permanent as the transaction has not sent yet its updates to the rest of sites.

(II) If $s_z.status_k(t)$ = aborted, $k = i$, and it has reached this state due to a π_z = *receive_rem_-abort*$_k(t, \langle rem_abort, t\rangle)$ action, then by its effects an $\langle abort, t\rangle$ message is multicast to all nodes except for i. It enables $\forall\, j \in \mathcal{N} \setminus \{i\}$: *receive_abort*$_j(t, \langle abort, t\rangle)$ or $status_j(t)$ = aborted ($status_i(t) \neq$ committed by Lemma 1). Eventually $\exists\, z' > z$: $\pi_{z'}$ = *receive_abort*$_j(t, \langle abort, t\rangle)$ for any $j \in \mathcal{N} \setminus \{i\}$.

(III) If $s_z.status_k(t)$ = aborted, $k \neq i$, and it has reached this state due to π_z = *local_abort*$_k(t)$ action, then it sends a $\langle rem_abort, t\rangle$ message to the transaction master site ($node(t) = i$). The *receive_rem_abort*$_i(t, \langle rem_abort, t\rangle)$ action is enabled or $status_i(t)$ = aborted ($status_j(t) \neq$ committed by Lemma 1). The execution of $\pi_{z'}$ = *receive_rem_abort*$_i(t, m)$, $z' > z$, yields to case (II).

(IV) If $s_z.status_k(t)$ = aborted, $k \neq i$, and it has reached this state due to π_z = *receive_re-mote*$_k(t, \langle remote, t, WS\rangle)$ action, then it sends a $\langle rem_abort, t\rangle$ message to the transaction master site ($node(t) = i$). The rest is similar to case (III).

(V) If $s_z.status_k(t)$ = aborted, $k = i$, and it has reached this state due to a π_z = *receive_re-mote*$_k(t', \langle remote, t', WS'\rangle)$ action, then an $\langle abort, t\rangle$ message is multicast to all nodes excluding i. The rest of the process is the one depicted in (II).

(VI) If $s_z.status_k(t)$ = aborted, $k \neq i$, and it has reached this state due to a π_z = *receive_abort*$_k(t, \langle abort, t\rangle)$ action, then for case (II) and (V) concludes. The $\langle abort, t\rangle$ message is received by the rest of nodes. □

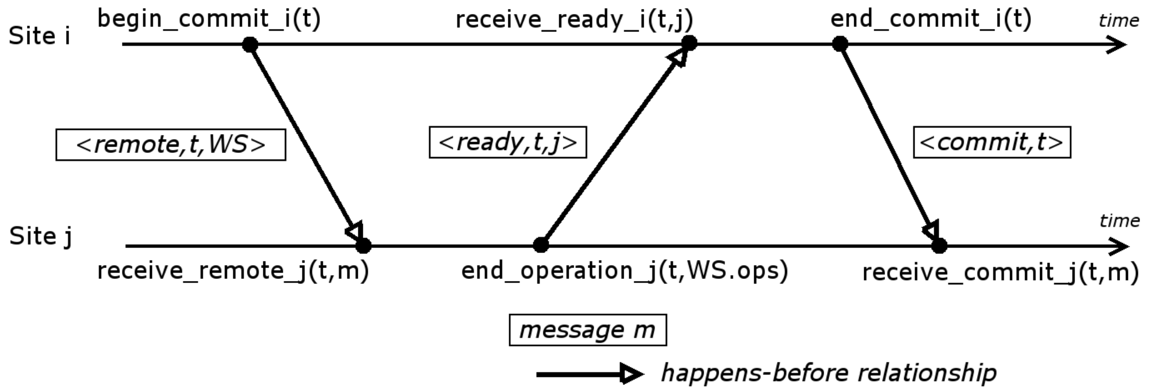


Figure 3.4: *Happens-before* relationship for the BRP of a given transaction t between its execution at the master site and the rest of nodes

Before continuing with the correctness proof we have to add a definition dealing with causality (*happens-before* relations [Lam78]) between actions. Some set of actions may only be viewed as causally related to another action in any execution α. We denote this fact by $\pi_i \prec_\alpha \pi_j$. For example, as it can be seen in Figure 3.4, with $node(t) = i \neq j$, $end_operation_j(t, WS.ops) \prec_\alpha receive_ready_i(t, \langle ready, t, j\rangle)$. This is clearly seen by the effects of the $end_operation_j(t, WS.ops)$ action as it sends a $\langle ready, t, j\rangle$ message to i. This message will be eventually received by the transaction master site that enables the $receive_ready_i(t, \langle ready, t, j\rangle)$ action, since $status_i(t) =$ pre_commit, and, by weak fairness of actions, it will be eventually executed. The following Lemma indicates that a transaction is *committed* if it has received every *ready* message from its remote transaction ones. Moreover, these remote transactions have been created as a consequence of the $receive_remote_j(t, \langle remote, t, WS\rangle)$ action execution.

Lemma 3. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be an arbitrary execution of the BRP automaton and $t \in \mathcal{T}$ be a committed transaction, $node(t) = i$, then the following happens-before relations hold for the appropriate parameters: $\forall j \in \mathcal{N}\backslash\{i\}$: $begin_commit_i(t) \prec_\alpha receive_remote_j(t, \langle remote, t, WS\rangle) \prec_\alpha end_operation_j(t, WS.ops) \prec_\alpha end_commit_i(t) \prec_\alpha receive_commit_j(t, \langle commit, t\rangle)$.*

Proof. Let $t \in \mathcal{T}$, $node(t) = i$, be a committed transaction. By Property 1.1, it has previously been with $status_i(t) =$ active. As $status_i(t) =$ pre_commit has been also achieved, the $begin_commit_i(t)$ action has been executed. It multicasts to the rest of nodes the $\langle remote, t, DB_i.WS(t)\rangle$ message. $\forall j \in \mathcal{N}, j \neq i$ the message is in $channel_j$ and the $receive_remote_j(t, \langle remote, t, WS\rangle)$ action will be invoked. As an effect, the operation will be submitted to the DB_j. As t is a committed transaction, it will not be aborted neither by protocol itself or the DB_j (Lemma 1). Therefore, once the operation is done the $end_operation_j(t, op)$ will be the only action enabled for t at j. This last action will send the $\langle ready, t, j\rangle$ message to i. The reception of these messages (reliable channels) will successively call for the $receive_ready_i(t, \langle ready, t, k\rangle)$ action with $k \in \mathcal{N} \setminus \{i, j\}$. At that moment, the only enabled action for t at site i will be the $end_commit_i(t)$ action. This action will commit the transaction at i and multicast the $\langle commit, t\rangle$ message to the rest of nodes. The only action enabled for t at j (being $j \in \mathcal{N}, j \neq i$) is the $receive_commit_j(t, \langle commit, t\rangle)$ action that commits the transaction at that site. Hence, $\forall j \in \mathcal{N} \setminus \{i\}$, the Lemma holds following the previous causal chain. □

In order to define the correctness of our replication protocol we have to study the global history,

H, of committed transactions, $C(H)$ [BHG87]. We may easily adapt this concept to our BRP automaton. Therefore, a new auxiliary state variable, H_i, is defined in order to keep track of all the DB_i operations performed on the local DBMS at the i site. For a given execution α of the BRP automaton, $H_i(\alpha)$ plays a similar role to the local history H_i at site i as introduced in [BHG87] for the DBMS. In the following, only committed transactions are part of the history, deleting all operations that do not belong to transactions committed in $H_i(\alpha)$. The serialization graph for $H_i(\alpha)$, $SG(H_i(\alpha))$, is defined as in [BHG87]. An arc and a path in $SG(H_i(\alpha))$ are denoted as $t \rightarrow t'$ and $t \overset{*}{\rightarrow} t'$ respectively. Our local DBMS produces ANSI serializable histories [BBG+95]. Thus, $SG(H_i(\alpha))$ is acyclic and the history is strict. The correctness criterion for replicated databases is 1CS, which stands for a serial execution over the logical data unit (although there are several copies of this data among all sites) [BHG87]. Thus, for any execution resulting in local histories $H_1(\alpha), H_2(\alpha), \ldots, H_{\mathcal{N}}(\alpha)$ at all sites its serialization graph, $\cup_k SG(H_k(\alpha))$, must be acyclic so that conflicting transactions are equally ordered in all local histories.

Before showing the correctness proof, we need an additional Property relating transaction isolation level of the underlying DB modules to the automaton execution event ordering. Let us see first this with an example assuming a strict-2PL scheduler as the underlying DB_i. A transaction must acquire all its locks before committing. If there are two conflicting transactions, $t, t' \in \mathcal{T}$, such that $t \rightarrow t'$ then t' will acquire its locks after t has released them (which occurs at commit time). Therefore, in our case we have taht $status_i(t') =$ pre_commit will be subsequent to $status_i(t) =$ committed in the execution.

The following Property and Corollary establish a property about local executions of committed transactions.

Property 3. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the BRP automaton and $i \in \mathcal{N}$. If there exist two transactions $t, t' \in \mathcal{T}$ such that $t \overset{*}{\rightarrow} t'$ in $SG(H_i(\alpha))$ then $\exists z_1 < z_2 < z_3 < z_4$: $s_{z_1}.status_i(t) = pre_commit \wedge s_{z_2}.status_i(t) = committed \wedge s_{z_3}.status_i(t') = pre_commit \wedge s_{z_4}.status_i(t') = committed$.*

Proof. We firstly consider $t \rightarrow t'$. Thus, exists an operation op, issued by t, and another operation op', issued by t', such that op conflicts with op' and op executes before op'. Hence, by $H_i(\alpha)$ construction we have that $DB_i.notify(t, op) =$ run is prior to $DB_i.notify(t', op') =$ run. However, we have assumed that the DB_i is serializable as shown in [BBG+95]. In such a case, $H_i(\alpha)$ is strict serializable for write and read operations. Therefore, it is required that

$DB_i.notify(t, op)$ = run must occur before $DB_i.commit(t)$ and that the latter must be prior to $DB_i.notify(t', op')$ = run. The $DB_i.commit(t)$ operation is associated with $status_i(t)$ = committed. Considering t', $DB_i.notify(t', op')$ = run is associated with $status_i(t) \in$ {active, pre_commit}. Therefore, $\exists z_2 < z'_3$ in α such that $s_{z_2}.status_i(t)$ = committed and $s_{z'_3}.status_i(t') \in$ {active, pre_commit}. By Property 1 and by the fact that both transactions commit, $\exists z_1 < z_2 < z'_3 \leq z_3 < z_4$ in α such that $s_{z_1}.status_i(t)$ = pre_commit $\wedge$ $s_{z_2}.status_i(t)$ = committed $\wedge$ $s_{z_3}.status_i(t')$ = pre_commit $\wedge$ $s_{z_4}.status_i(t')$ = committed. Thus, the Property holds for $t \rightarrow t'$. The case $t \xrightarrow{*} t'$ is proved by transitivity. □

The latter Property reflects the *happens-before* relationship between the different *status* of conflictive transactions. The same order must hold for the actions generating the mentioned status. The next Corollary expresses this property.

Corollary 1. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the BRP automaton and $i \in \mathcal{N}$. If there exist two transactions $t, t' \in \mathcal{T}$ such that $t \xrightarrow{*} t'$ in $SG(H_i(\alpha))$ then the following happens-before relations, with the appropriate parameters, hold:*

1. *$node(t) = node(t') = i$: $begin_commit_i(t) \prec_\alpha end_commit_i(t) \prec_\alpha begin_commit_i(t') \prec_\alpha end_commit_i(t')$.*

2. *$node(t) = i \wedge node(t') \neq i$: $begin_commit_i(t) \prec_\alpha end_commit_i(t) \prec_\alpha end_operation_i(t', WS'.ops) \prec_\alpha receive_commit_i(t', \langle commit, t'\rangle)$.*

3. *$node(t) \neq i \wedge node(t') = i$: $end_operation_i(t, WS.ops) \prec_\alpha receive_commit_i(t, \langle commit, t\rangle) \prec_\alpha begin_commit_i(t') \prec_\alpha end_commit_i(t')$.*

4. *$node(t) \neq i \wedge node(t') \neq i$: $end_operation_i(t, WS.ops) \prec_\alpha receive_commit_i(t, \langle commit, t\rangle) \prec_\alpha end_operation_i(t', WS'.ops) \prec_\alpha receive_commit_i(t', \langle commit, t'\rangle)$.*

Proof. By Property 3, $\exists z_1 < z_2 < z_3 < z_4$: $s_{z_1}.status_i(t)$ = pre_commit $\wedge$ $s_{z_2}.status_i(t)$ = committed $\wedge$ $s_{z_3}.status_i(t')$ = pre_commit $\wedge$ $s_{z_4}.status(t')$ = committed. Depending on $node(t)$ and $node(t')$ values the unique actions whose effects modify their associated *status* are the ones indicated in this Corollary. □

In the following, we prove that the BRP provides 1CS. Lemma 3 states the causal relationship between a remote transaction and its execution at the master site. Corollary 1 points out the

relationship between conflicting transactions. We try to show that it is not possible to obtain a different order at distinct sites where the conflicting transactions are executed. In other words, every conflictive transaction is executed in the same order at all available sites.

Theorem 1. *Let* $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ *be a fair execution of the BRP automaton. The graph* $\cup_{k\in\mathcal{N}} SG(H_k(\alpha))$ *is acyclic.*

Proof. By contradiction. Assume there exists a cycle in $\cup_{k\in\mathcal{N}} SG(H_k(\alpha))$. There are at least two different transactions $t, t' \in \mathcal{T}$ and two different sites $x, y \in \mathcal{N}$, $x \neq y$, such that those transactions are executed in different order at x and y. Thus, we consider (a) $t \overset{*}{\rightarrow} t'$ in $SG(H_x(\alpha))$ and (b) $t' \overset{*}{\rightarrow} t$ in $SG(H_y(\alpha))$; being $node(t) = i$ and $node(t') = j$. There are four cases under study:

(I) $i = j = x$.

(II) $i = x \wedge j = y$.

(III) $i = j \wedge i \neq x \wedge i \neq y$.

(IV) $i \neq j \wedge i \neq x \wedge i \neq y \wedge j \neq x \wedge j \neq y$.

In the following, we simplify the notation. The action names are shortened, i.e. *begin_commit*$_x(t)$ by $bc_x(t)$; *end_commit*$_x(t)$ by $ec_x(t)$; *receive_remote*$_x(t, \langle remote, t, WS\rangle)$ by $rr_x(t)$; *end_operation*$_x(t, WS.ops)$ by $eo_x(t)$; and *receive_commit*$_x(t, \langle commit, t\rangle)$ by $rc_x(t)$. Besides, for all cases we graphically show a proof of its contradiction in Figures 3.5-3.7.

CASE (I) By Corollary 1.1 for (a): $bc_x(t) \prec_\alpha ec_x(t) \prec_\alpha bc_x(t') \prec_\alpha ec_x(t')$. (i)

By Corollary 1.4 for (b): $eo_y(t') \prec_\alpha rc_y(t') \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. (ii)

By Lemma 3 for t: $bc_x(t) \prec_\alpha rr_y(t) \prec_\alpha eo_y(t) \prec_\alpha ec_x(t)$ followed by (i) $\prec_\alpha$ (via Lemma 3) $bc_x(t') \prec_\alpha rr_y(t') \prec_\alpha eo_y(t')$. Thus, $eo_y(t) \prec_\alpha eo_y(t')$ in contradiction with (ii), as it can be seen in Figure 3.5.

CASE (II) By Corollary 1.2 for (a): $bc_x(t) \prec_\alpha ec_x(t) \prec_\alpha eo_x(t') \prec_\alpha rc_x(t')$. (i)

By Corollary 1.2 for (b): $bc_y(t') \prec_\alpha ec_y(t') \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. (ii)

By Lemma 3 for t: $bc_x(t) \prec_\alpha rr_y(t) \prec_\alpha eo_y(t) \prec_\alpha ec_x(t)$; by (i) $\prec_\alpha eo_x(t')$, and by Lemma 3 for t', $\prec_\alpha ec_y(t')$. Thus $eo_y(t) \prec_\alpha ec_y(t')$ in contradiction with (ii) which it is shown in Figure 3.6.

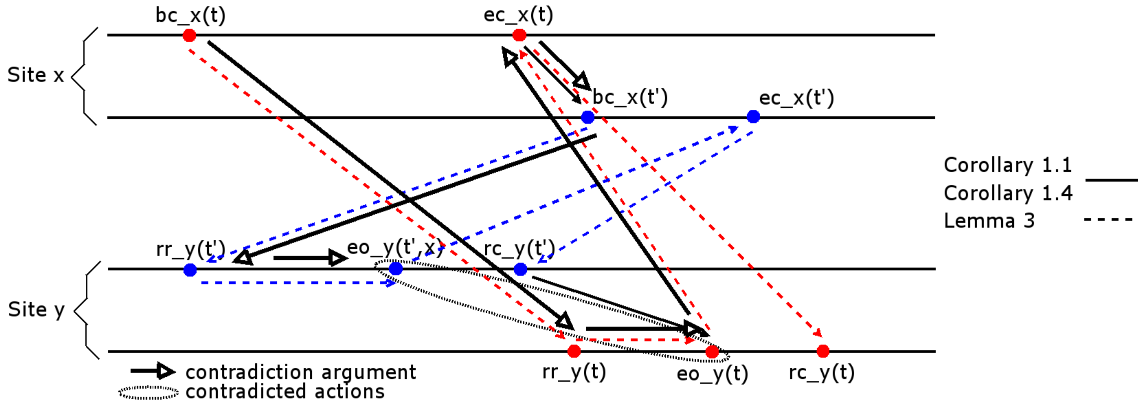


Figure 3.5: CASE (I): $node(t) = node(t') = x$

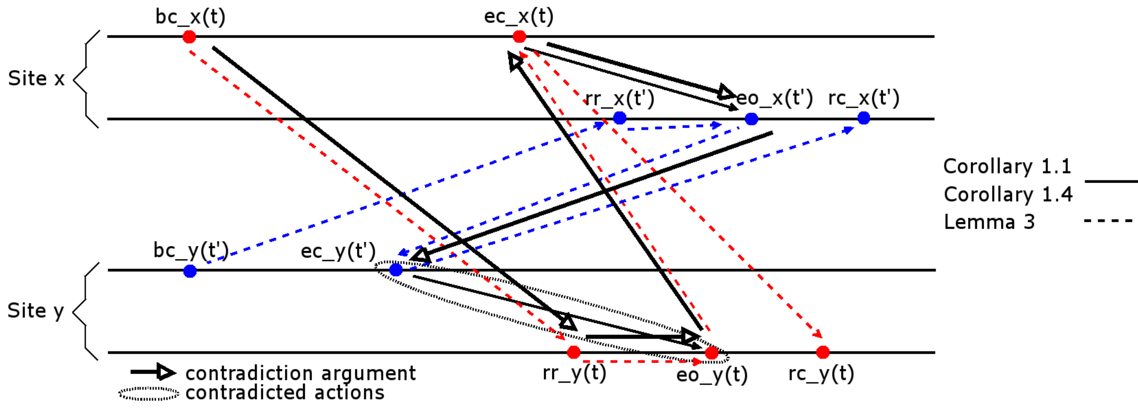


Figure 3.6: CASE (II): $node(t) = x$ and $node(t') = y$

CASE (III) As x and y are different sites from the transaction master site, only one of them will be executed in the same order as in the master site. If we take into account the different one with the master site then we will be under assumptions considered in CASE (I).

CASE (IV) By Corollary 1.4 for (a): $eo_x(t) \prec_\alpha rc_x(t) \prec_\alpha eo_x(t') \prec_\alpha rc_x(t')$. (i)

By Corollary 1.4 for (b): $eo_y(t') \prec_\alpha rc_y(t') \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. (ii)

By Lemma 3 for t at site y: $bc_i(t) \prec_\alpha rr_y(t) \prec_\alpha eo_y(t) \prec_\alpha ec_i(t) \prec_\alpha rc_y(t)$. Applying Lemma 3 for t at x: $bc_i(t) \prec_\alpha rr_x(t) \prec_\alpha eo_x(t) \prec_\alpha ec_i(t) \prec_\alpha rc_x(t)$. Therefore, we have that $eo_y(t) \prec_\alpha rc_x(t)$. Let us apply Lemma 3 for t' at y: $bc_j(t') \prec_\alpha rr_y(t') \prec_\alpha eo_y(t') \prec_\alpha ec_j(t') \prec_\alpha rc_y(t')$ and for site x: $bc_j(t') \prec_\alpha rr_x(t') \prec_\alpha eo_x(t') \prec_\alpha ec_j(t') \prec_\alpha rc_x(t')$. Thus, we have $eo_x(t') \prec_\alpha rc_y(t')$. Taking into account Lemma 3 for t we have: $eo_y(t) \prec_\alpha rc_x(t)$ (via (i)) $\prec_\alpha eo_x(t')$ (via Lemma 3 for t') $\prec_\alpha rc_y(t')$, in contradiction with (ii), see Figure 3.7.

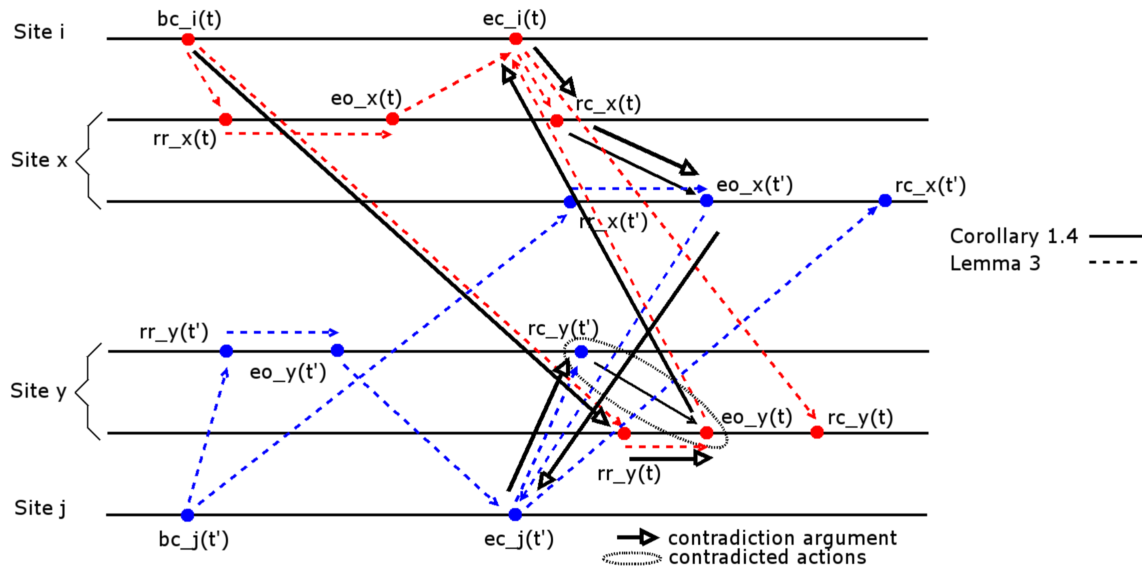


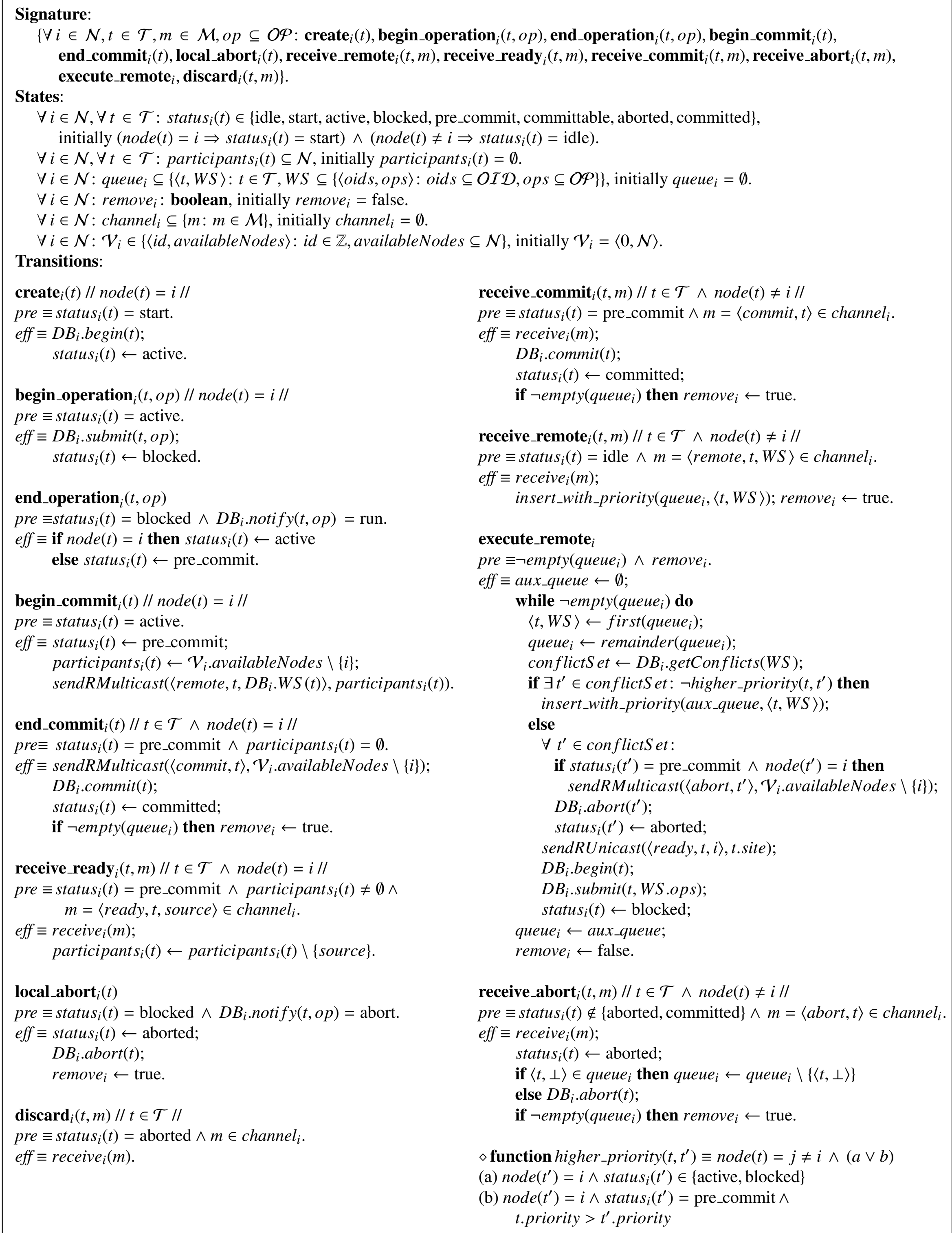
Figure 3.7: CASE (IV): $node(t) = i$ and $node(t') = j$

□

3.4 Enhanced Replication Protocol

This Section deals with variations of BRP so as to increase its performance. The comprehension of the correctness proof allows us to study and consider several variations while keeping the correctness (or the consistency level intended by the replication protocol).

The first modification consists in changing the behavior of remote transactions since they do not have to wait for sending the *ready* message until the end of the updates execution. Hence, we reduce the transaction response time. Another modification deals with the abortion rate reduction. As BRP does not impose any message ordering, *remote* messages of conflicting transactions may be received at different sites in distinct order, as a result both transaction will be aborted and none will proceed. To solve this inconvenience a prioritized queue is added. This queue allows delivered remote transactions to wait when there are currently executing transactions in the database whose priority is higher than them. When the remote transactions contained in the queue reach the proper priority to be submitted to the DBMS, or their master sites decide to rollback the transaction, they leave the queue. Thus, remote transaction abortions may be decreased, since BRP does not allow remote transactions to wait. These modifications lead to the definition of the ERP state transition system which is shown in Figure 3.8.



Signature:
$\{\forall\, i \in \mathcal{N}, t \in \mathcal{T}, m \in \mathcal{M}, op \subseteq \mathcal{OP}$: **create**$_i(t)$, **begin_operation**$_i(t, op)$, **end_operation**$_i(t, op)$, **begin_commit**$_i(t)$, **end_commit**$_i(t)$, **local_abort**$_i(t)$, **receive_remote**$_i(t, m)$, **receive_ready**$_i(t, m)$, **receive_commit**$_i(t, m)$, **receive_abort**$_i(t, m)$, **execute_remote**$_i$, **discard**$_i(t, m)\}$.

States:
$\forall\, i \in \mathcal{N}, \forall\, t \in \mathcal{T}$: $status_i(t) \in$ {idle, start, active, blocked, pre_commit, committable, aborted, committed}, initially $(node(t) = i \Rightarrow status_i(t) =$ start$) \wedge (node(t) \neq i \Rightarrow status_i(t) =$ idle$)$.
$\forall\, i \in \mathcal{N}, \forall\, t \in \mathcal{T}$: $participants_i(t) \subseteq \mathcal{N}$, initially $participants_i(t) = \emptyset$.
$\forall\, i \in \mathcal{N}$: $queue_i \subseteq \{\langle t, WS\rangle : t \in \mathcal{T}, WS \subseteq \{\langle oids, ops\rangle : oids \subseteq \mathcal{OID}, ops \subseteq \mathcal{OP}\}\}$, initially $queue_i = \emptyset$.
$\forall\, i \in \mathcal{N}$: $remove_i$: **boolean**, initially $remove_i =$ false.
$\forall\, i \in \mathcal{N}$: $channel_i \subseteq \{m : m \in \mathcal{M}\}$, initially $channel_i = \emptyset$.
$\forall\, i \in \mathcal{N}$: $\mathcal{V}_i \in \{\langle id, availableNodes\rangle : id \in \mathbb{Z}, availableNodes \subseteq \mathcal{N}\}$, initially $\mathcal{V}_i = \langle 0, \mathcal{N}\rangle$.

Transitions:

create$_i(t)$ // $node(t) = i$ //
$pre \equiv status_i(t) =$ start.
$eff \equiv DB_i.begin(t)$;
  $status_i(t) \leftarrow$ active.

begin_operation$_i(t, op)$ // $node(t) = i$ //
$pre \equiv status_i(t) =$ active.
$eff \equiv DB_i.submit(t, op)$;
  $status_i(t) \leftarrow$ blocked.

end_operation$_i(t, op)$
$pre \equiv status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ run.
$eff \equiv$ **if** $node(t) = i$ **then** $status_i(t) \leftarrow$ active
  else $status_i(t) \leftarrow$ pre_commit.

begin_commit$_i(t)$ // $node(t) = i$ //
$pre \equiv status_i(t) =$ active.
$eff \equiv status_i(t) \leftarrow$ pre_commit;
  $participants_i(t) \leftarrow \mathcal{V}_i.availableNodes \setminus \{i\}$;
  $sendRMulticast(\langle remote, t, DB_i.WS(t)\rangle, participants_i(t))$.

end_commit$_i(t)$ // $t \in \mathcal{T} \wedge node(t) = i$ //
$pre \equiv status_i(t) =$ pre_commit $\wedge\ participants_i(t) = \emptyset$.
$eff \equiv sendRMulticast(\langle commit, t\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$;
  $DB_i.commit(t)$;
  $status_i(t) \leftarrow$ committed;
  if $\neg empty(queue_i)$ **then** $remove_i \leftarrow$ true.

receive_ready$_i(t, m)$ // $t \in \mathcal{T} \wedge node(t) = i$ //
$pre \equiv status_i(t) =$ pre_commit $\wedge\ participants_i(t) \neq \emptyset \wedge$
  $m = \langle ready, t, source\rangle \in channel_i$.
$eff \equiv receive_i(m)$;
  $participants_i(t) \leftarrow participants_i(t) \setminus \{source\}$.

local_abort$_i(t)$
$pre \equiv status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ abort.
$eff \equiv status_i(t) \leftarrow$ aborted;
  $DB_i.abort(t)$;
  $remove_i \leftarrow$ true.

discard$_i(t, m)$ // $t \in \mathcal{T}$ //
$pre \equiv status_i(t) =$ aborted $\wedge\ m \in channel_i$.
$eff \equiv receive_i(m)$.

receive_commit$_i(t, m)$ // $t \in \mathcal{T} \wedge node(t) \neq i$ //
$pre \equiv status_i(t) =$ pre_commit $\wedge\ m = \langle commit, t\rangle \in channel_i$.
$eff \equiv receive_i(m)$;
  $DB_i.commit(t)$;
  $status_i(t) \leftarrow$ committed;
  if $\neg empty(queue_i)$ **then** $remove_i \leftarrow$ true.

receive_remote$_i(t, m)$ // $t \in \mathcal{T} \wedge node(t) \neq i$ //
$pre \equiv status_i(t) =$ idle $\wedge\ m = \langle remote, t, WS\rangle \in channel_i$.
$eff \equiv receive_i(m)$;
  $insert_with_priority(queue_i, \langle t, WS\rangle)$; $remove_i \leftarrow$ true.

execute_remote$_i$
$pre \equiv \neg empty(queue_i) \wedge remove_i$.
$eff \equiv aux_queue \leftarrow \emptyset$;
  while $\neg empty(queue_i)$ **do**
    $\langle t, WS\rangle \leftarrow first(queue_i)$;
    $queue_i \leftarrow remainder(queue_i)$;
    $conflictSet \leftarrow DB_i.getConflicts(WS)$;
    if $\exists\, t' \in conflictSet : \neg higher_priority(t, t')$ **then**
      $insert_with_priority(aux_queue, \langle t, WS\rangle)$;
    else
      $\forall\, t' \in conflictSet$:
        if $status_i(t') =$ pre_commit $\wedge\ node(t') = i$ **then**
          $sendRMulticast(\langle abort, t'\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$;
        $DB_i.abort(t')$;
        $status_i(t') \leftarrow$ aborted;
      $sendRUnicast(\langle ready, t, i\rangle, t.site)$;
      $DB_i.begin(t)$;
      $DB_i.submit(t, WS.ops)$;
      $status_i(t) \leftarrow$ blocked;
  $queue_i \leftarrow aux_queue$;
  $remove_i \leftarrow$ false.

receive_abort$_i(t, m)$ // $t \in \mathcal{T} \wedge node(t) \neq i$ //
$pre \equiv status_i(t) \notin$ {aborted, committed} $\wedge\ m = \langle abort, t\rangle \in channel_i$.
$eff \equiv receive_i(m)$;
  $status_i(t) \leftarrow$ aborted;
  if $\langle t, \perp\rangle \in queue_i$ **then** $queue_i \leftarrow queue_i \setminus \{\langle t, \perp\rangle\}$
  else $DB_i.abort(t)$;
  if $\neg empty(queue_i)$ **then** $remove_i \leftarrow$ true.

$\diamond$ **function** $higher_priority(t, t') \equiv node(t) = j \neq i \wedge (a \vee b)$
(a) $node(t') = i \wedge status_i(t') \in$ {active, blocked}
(b) $node(t') = i \wedge status_i(t') =$ pre_commit $\wedge$
  $t.priority > t'.priority$

Figure 3.8: State transition system for the Enhanced Replication Protocol (ERP) automaton

3.4.1 Performance

The response time, $\theta_r(t)$, of a transaction t ($node(t) = i$) in our system is determined by the sum of the following times: the transaction processing at the master site, $\theta_{DB_i}(t)$; multicasting the *remote*

message to the rest of sites, $\theta_{MC}(t)$; transaction updates processing at the rest of available sites $\theta_{DB_j}(t)$, with $j \in \mathcal{N} \setminus \{i\}$; and, finally, each remote site sending the *ready* message back to the master site, $\theta_{UC_j}(t)$. Therefore, we have $\theta_r(t) \approx \theta_{DB_i}(t) + \theta_{MC}(t) + \max_j(\theta_{DB_j}(t)) + \max_j(\theta_{UC_j}(t))$. This response time is a consequence of the 2PC origin of the O2PL [CL91]. The response time is limited by the slowest remote transaction execution, since it must wait for applying the updates at all sites before committing the transaction. On the other hand, database unilateral aborts are coped using this approach. Hence, there exists a trade-off between safety and performance.

If we take a look at the proof of Theorem 1, it does not require or establish the time when the *ready* message must be sent by remote transactions. It only establishes that $receive_remote_j(t, \langle remote, t, WS\rangle \prec_\alpha end_operation_j(t, WS.ops)$, so the *ready* message may be sent at any time during this two actions execution. Thus, if we send the *ready* message back once the transaction has overcome the $higher_priority(t, t')$ function and before it has been submitted to the database, we approximately reduce the transaction response time to the following : $\theta_r(t) \approx \theta_{DB_i}(t) + \theta_{MC}(t) + \max_j(\theta_{UC_j}(t))$. The transaction response time is decreased because it does not need to wait for the execution of the remote transactions, therefore we get rid of $\max_j(\theta_{DB_j}(t))$. Nevertheless, the transaction atomicity is compromised as the $local_abort_i(t)$ action must never be invoked for a remote transaction. It implies that there must not be unilateral aborts for already submitted remote transactions. In other words, once a remote transaction is submitted to the database, as there are no conflicts, it successfully ends. The BRP submits a remote transaction to the database after all conflicting transactions have been aborted. Therefore, it will never enter in a local deadlock. However, this is not enough to prevent that a database submitted remote transaction is aborted by a new delivered remote transaction. Hence, the $higher_priority(t, t')$ function must also be modified so that a remote transaction must never be aborted by a new incoming remote transaction.

3.4.2 Decreasing Abortion Rate

Our BRP lacks of fairness due to the fact that we do not allow to wait a remote transaction. The worst case occurs whenever two conflicting transactions arrive at distinct order at two different sites and both reached the *pre_commit* state at their respective first-delivered sites. When the second *remote* message arrives, it will send a *rem_abort* message (see Figure 3.2) to its transaction master site. Hence, both transactions will be rolled back and neither one will commit. This problem may be solved by the use of queues storing remote transactions pending to apply.

Each time a transaction t is delivered at a site. It is firstly enqueued (arranged by $t.priority$) and then checked to see whether its priority is greater than any other conflicting transaction executing at that site. If its priority is less it will remain enqueued. Otherwise, it will be submitted to the database. Moreover, each time a transaction commits or rolls back at a given site, the queue is checked again so that awaiting transactions must proceed or not in the DBMS. The main difference with the BRP is that only the transaction master site will decide if the transaction will be committed or rolled back.

The modifications we have just introduced allows us to send the *ready* message just when the transaction is submitted to the underlying database. We have avoided the *rem_abort* message usage and deadlock among transactions executing at different sites is prevented due to the priority associated for each transaction. This will be outlined in the following. We must show that the ERP proposal is still correct. Hence, if we appropriately modify Properties and Lemmas of the correctness proof for the BRP automaton, then we will see that our new solution is correct too.

3.4.3 The ERP Automaton

We introduce the modifications of the BRP into the new ERP automaton. The $receive_remote_i(t, \langle remote, t, WS \rangle)$ message inserts the delivered remote transaction into a queue, $queue_i$. Its insertion position depends on the $t.priority$ field. The key action of the new replication protocol is the $execute_remote_i$ one. This action is invoked at least each time a transaction t is delivered, due to the fact that is enabled also by the new bolean state variable $remove_i$. Furthermore, this action will be invoked several times (as it is called after a *commit*, *abort* or a *remote* delivery of another transaction) until t is submitted to the DB_i.

The remote updates for a given $WS(t)$ will only be applied if there is no conflicting transaction at node i having a higher priority than the received one. The $higher_priority(t, t')$ function has been modified from its BRP counterpart. It still depends on the state of the transaction ($status_i(t)$) and its priority, but a submitted conflicting remote transaction has always higher priority than any new incoming remote transaction. Therefore, a new incoming conflicting remote transaction whose priority is lower than any other executing transaction will be inserted again in $queue_i$.

The correctness of our solution is not compromised by the queue usage, since only the transaction master site decides whether a transaction aborts or not. Finally, if the remote transaction is the one with the highest priority among all at i, then it will send the *ready* message to the mas-

ter site at the same time it is submitted to the DB_i module. Before its submission to the DB_i, it will abort every local conflictive transaction and submit t to DB_i. Hence, we eliminate any possibility of local deadlocks. Aborted local transactions in *pre_commit* state with lower priority will multicast an *abort* message to the rest of sites. The finalization of the remote transaction ($end_operation_i(t, WS.ops)$) changes its $status_j(t)$ = pre_commit, $j \neq node(t)$. It has to wait for the reception of the *commit* message from the master site (as it has received all *ready* messages), or straightly commit if the message has arrived. The reception of this message commits the transaction at the remainder sites ($receive_commit_i(t, \langle commit, t\rangle)$). Recall that each time a transaction commits or rolls back, the $queue_i$ is inspected in order to wake up waiting remote transactions. Again, transactions in the *pre_commit* state are committable at any point from the DBMS point of view.

Finally, it is important to note that with our assumption that unilateral aborts do not occur for remote transactions, the ERP will not invoke the $local_abort_i(t)$ for a transaction t with $node(t) \neq i$. It follows, for the same reasoning and the queue usage, that the $receive_rem_abort_i(t, \langle rem_abort, t\rangle)$ action of the BRP has been suppressed in the ERP automaton.

3.5 ERP Correctness Proof

This Section contains the proofs (atomicity and 1CS) of the ERP state transition system (introduced in Figure 3.8) in a failure free environment.

Let us start showing that the ERP is deadlock free, assuming that deadlocks involving exclusively local transactions at a given site are directly resolved by the underlying local DBMS executing the $local_abort_i(t)$ action. In case of remote transactions, we assume that as the $execute_remote_i$ action aborts all local conflicting transactions whose priority is lower than t, then the remote transaction will never be aborted by the DBMS during its execution. Besides, we have modified the $higher_priority(t, t')$ function so that any remote transaction executing at a given site will never be rolled back. If a delivered remote transaction has lower priority than any other transaction executing at its delivered site, then it will be inserted in $queue_i$ (sorted by $t.priority$). Hence, this is a potential deadlock source, as the protocol permits remote transactions to wait. However, as channels are reliable, all *remote* messages of enqueued transactions will reach the transaction master sites of all of them. The abortion decision is performed only at the enqueued transaction master

site. If a transaction is finally aborted then all sites will receive the *abort* message, as channels are reliable, and execute the $receive_abort_i(t, \langle abort, t\rangle)$.

The ERP must guarantee the atomicity of a transaction; that is, the transaction is either committed at all available sites or aborted at all sites. One can note that priority rules (see the $higher_priority(t, t')$ function in Figure 3.8) ensure that a remote transaction is never aborted by a local transaction nor a remote one, provided that there are no unilateral aborts; therefore, it will eventually reach the *pre_commit* state, given that it does not receive an *abort* message from its master site.

In the following, we will continue with the same correctness proof philosophy as we did in Section 3.3. Hence, we only introduce those new proofs that will be affected by the modifications introduced to the ERP.

Property 4. *Let* $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ *be an arbitrary execution of the ERP automaton and* $t \in \mathcal{T}$. *Let* $\beta = s_0.status_j(t)\, s_1.status_j(t) \ldots s_{z'}.status_j(t)$ *be the sequence of status values of* t *at site* $j \in \mathcal{N}$, *obtained from* α *by removing the consecutive repetitions of the same* $status_j(t)$ *value and maintaining the same order apparition in* α. *The following Property holds:*

1. *If* $node(t) = j$ *then* β *is a prefix of the regular expression:*
   - $start \cdot active \cdot (blocked \cdot active)^* \cdot pre_commit \cdot committed$
   - $start \cdot active \cdot (blocked \cdot active)^* \cdot pre_commit \cdot aborted$
   - $start \cdot (active \cdot blocked)^+ \cdot aborted$
2. *If* $node(t) \neq j$ *then* β *is a prefix of the regular expression:*
   - $idle$
   - $idle \cdot blocked \cdot pre_commit \cdot committed$
   - $idle \cdot blocked \cdot pre_commit \cdot aborted$
   - $idle \cdot blocked \cdot aborted$
   - $idle \cdot aborted$

A *status* transition for a t transaction in Property 4 is associated with an operation on the *DB* module where the transaction was created, i.e. *pre_commit* to *committed* involves the $DB.commit(t)$ operation. These aspects are straightforward from the ERP automaton inspection in Figure 3.8.

The following Property is needed to prove Lemma 4. This Property states the invariant properties of the ERP. If a transaction t with $node(t) = i$ is committed at $j \neq i$, it is because it was already committed at its master site. A remote transaction currently being executed at its DB_j module ($status_j(t)$ = blocked) may only change its *status* if its execution is completed or by an *abort* message coming from its master site. In other words, it will never be aborted by the DB_j module. In the same way, a remote transaction in the *pre_commit* state may only change its *status* if it receives a *commit* or an *abort* message. Finally, if a transaction is *committed* at its master site then at the rest of available sites it will be either *committed* (it has already received the *commit* message), *pre_commit* (it is waiting to receive the *commit* message) or *blocked* (it is still applying the updates at that site).

Property 5. *Let* $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ *be an arbitrary execution of the ERP automaton and* $t \in \mathcal{T}$*, with* $node(t) = i$*.*

1. *If* $\exists\, j \in \mathcal{N} \setminus \{i\}\colon s_z.status_j(t) = committed$ *then* $s_z.status_i(t) = committed$*.*

2. *If* $\exists\, z' < z\colon s_{z'}.status_j(t) = s_z.status_j(t) = blocked$ *for any* $j \in \mathcal{N} \setminus \{i\}$ *then* $\forall\, z''\colon z' < z'' \leq z\colon \pi_{z''} \notin \{receive_abort_j(t, \langle abort, t\rangle), end_operation_j(t, WS.ops)\}$*.*

3. *If* $\exists\, z' < z\colon s_{z'}.status_j(t) = s_z.status_j(t) = pre_commit$ *for any* $j \in \mathcal{N} \setminus \{i\}$ *then* $\forall\, z''\colon z' < z'' \leq z\colon \pi_{z''} \notin \{receive_commit_j(t, \langle commit, t\rangle), receive_abort_j(t, \langle abort, t\rangle)\}$*.*

4. *If* $s_z.status_i(t) = committed$ *then* $\forall\, j \in \mathcal{N}\colon s_z.status_j(t) \in \{blocked, pre_commit, committed\}$*.*

Proof.

1. The Proof is the same as in Property 2.1.

2. We proof this Property by contradiction. If we assume that $\pi_{z''} = receive_abort_j(t, \langle abort, t\rangle)$ action, then by its effects $s_{z''}.status_j(t)$ = aborted and by Property 4.2 its *status* never changes in contradiction with our initial assumption. If we suppose that $\pi_{z''} = end_operation_j(t, WS.ops)$ action is executed then, by its effects, $s_{z''}.status_j(t)$ = pre_commit. By Property 4.2 it never goes to *blocked* again in contradiction with our assumption.

3. We proof this by contradiction. If we assume that $\pi_{z''} = receive_commit_j(t, \langle commit, t\rangle)$ by this action effects $s_{z''}.status_j(t)$ = committed and by Property 4.2 its *status* never changes it

contradiction with our initial assumption. Besides, if we assume that the executed action is $\pi_{z''} = receive_abort_j(t, \langle abort, t\rangle)$ then $s_{z''}.status_j(t) =$ aborted and again, by Property 4.2, its *status* never changes in contradiction with our assumption.

4. If $s_z.status_i(t) =$ committed, by Property 4.1 we have for all z' that $s_{z'}.status_i(t) \neq$ aborted and $\langle abort, t\rangle \notin s_{z'}.channel_j$ for all $j \in \mathcal{N}$. Thus, the $receive_abort_j(t, m)$ action is disabled at any state of α. As $s_z.status_i(t) =$ committed then all $j \in \mathcal{N} \setminus \{i\}$ has sent the *ready* message to i which implies, by the $execute_remote_j$ action effects, that $status_j(t) =$ blocked and that it has been submitted to the DB_j module or $status_j(t) =$ pre_commit if the $end_operation_j(t, WS.ops)$ has been already executed. Thus by Property 4.2, $s_z.status_j(t) \in \{blocked, pre_commit, committed, aborted\}$. We must show that $status_j(t) =$ aborted will never be achieved. This is easy to proof since it will need an *abort* message coming from the transaction master site, and this is not possible since for all z' we have that $\langle abort, t\rangle \notin s_{z'}.channel_j$. On the other hand, as we assume that the DB_j module does not abort a submitted remote transaction, once $execute_remote_j$ is executed for t, then it will never execute the $local_abort_j(t, WS.ops)$ and $status_j(t) =$ aborted will never be possible. Hence, the Property holds. □

The following Lemma states the atomicity of committed transactions.

Lemma 4. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the ERP automaton and $t \in \mathcal{T}$ with $node(t) = i$. If $\exists\, j \in \mathcal{N}\colon s_z.status_j(t) = committed$ then $\exists\, z' > z\colon s_{z'}.status_j(t) = committed$ for all $j \in \mathcal{N}$.*

Proof. If $j \neq i$ by Property 5.1 (or $j = i$) $s_z.status_i(t) =$ committed. By Property 5.4, $\forall\, j \in \mathcal{N} \setminus \{i\}\colon s_z.status_j(t) \in \{\text{blocked}, \text{pre_commit}, \text{committed}\}$. Without loss of generality, assume that s_z is the first state where $s_z.status_i(t) =$ committed and $s_z.status_j(t) =$ pre_commit (if $s_z.status_j(t) =$ blocked it is because of its submission to the DB_j module, due to the $execute_remote_j$ for t. By weak fairness of action execution, the $end_operation_j(t, WS.ops)$ will be eventually invoked and $s_z.status_j(t) =$ pre_commit). By the effects of $\pi_z = end_commit_i(t)$, we have that $\langle commit, t\rangle \in s_z.channel_j$. By Property 5.4 invariance either $s_z.status_j(t) =$ committed or $s_z.status_j(t) =$ pre_commit and $\langle commit, t\rangle \in s_z.channel_j$. In the latter case the $receive_commit_j(t, \langle commit, t\rangle)$ action is enabled. By weak fairness assumption, it will be eventually delivered, thus $\exists\, z' > z\colon \pi_{z'} = receive_commit_j(t, \langle commit, t\rangle)$. By its effects, $s_{z'}.status_j(t) =$ committed. □

In a similar way, if a transaction is aborted, it will be aborted at all sites. This is stated in the following Lemma. The proof is very simple by inspection of the ERP action shown in Figure 3.8.

Lemma 5. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the ERP automaton and $t \in \mathcal{T}$ with $node(t) = i$. If $s_z.status_i(t) = aborted$ then $\exists z' \geq z$: $s_{z'}.status_j(t) = idle$ for all $j \in \mathcal{N} \setminus \{i\}$ or $s_{z'}.status_j(t) = aborted$ for all $j \in \mathcal{N}$.*

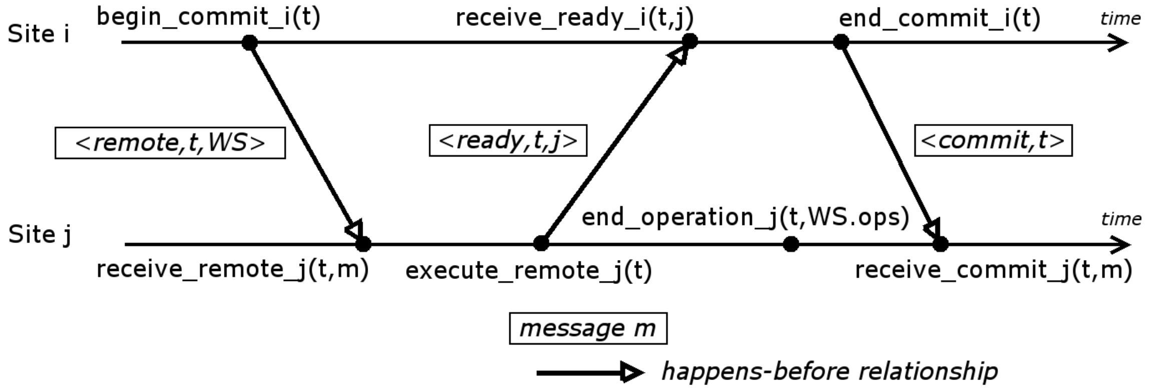


Figure 3.9: *Happens-before* relationship for a given transaction t between its execution at the master site and the rest of nodes

The ERP has some set of actions that *happens-before* other actions, i.e. they are causally related. For example, assuming t is a committed transaction with $node(t) = i \neq j$, the following *happens-before* relationship $begin_commit_i(t) \prec_\alpha receive_remote_j(t, \langle remote, t, WS \rangle)$ is held, see Figure 3.9. This is clearly seen by the effects of the $begin_commit_i(t)$ action: it sends a $\langle remote, t, DB_i.WS(t)\rangle$ to all $j \in \mathcal{N} \setminus \{i\}$. This message will be eventually received by j that enables the $receive_remote_j(t, \langle remote, t, WS \rangle)$ action, since $status_j(t) =$ idle, and, by weak fairness of actions, it will be eventually executed. However, this fact is delegated to the $execute_remote_j$ action. The following Lemma indicates that a transaction is *committed* if it has received every *ready* message from its remote transaction ones. These remote transactions have been executed as a consequence of the $execute_remote_j$ action execution.

For the sake of clearness and understanding, we are going to add t as a parameter to the $execute_remote_j$ action, provided that t is one of the submitted transactions to the DB_j module by its execution ($execute_remote_j(t)$). This may be done without loss of generality as transactions we consider in the following proofs correspond to committed transactions.

Lemma 6. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the ERP automaton and $t \in \mathcal{T}$ be a committed transaction, $node(t) = i$, then the following happens-before relations hold: $\forall j \in$*

$\mathcal{N} \setminus \{i\}$: $begin_commit_i(t) <_\alpha receive_remote_j(t, \langle remote, t, WS \rangle) <_\alpha execute_remote_j(t) <_\alpha receive_ready_i(t, \langle ready, j \rangle) <_\alpha end_commit_i(t) <_\alpha receive_commit_j(t, \langle commit, t \rangle)$.

Proof. Let $t \in \mathcal{T}$, $node(t) = i$, be a committed transaction. By Property 4.1, it has previously been with $status_i(t) =$ active. As $status_i(t) =$ pre_commit has been also achieved, the $begin_commit_i(t)$ action has been executed. It multicasts to the rest of nodes the $\langle remote, t, DB_i.WS(t) \rangle$ message. $\forall j \in \mathcal{N}, j \neq i$ the message is in $channel_j$ and the $receive_remote_j(t, \langle remote, t, WS \rangle)$ action will be invoked, what inserts the delivered transaction in $queue_j$. When t becomes the transaction with the highest priority among all in $queue_j$, the $execute_remote_j$ action (this action will be enabled each time a transaction is committed or rolled back, or a remote transaction is delivered) will be invoked for t ($execute_remote_j(t)$). Then, by its effects it will send the *ready* message to i and the operation will be submitted to the DB_j module. By channel reliability it will eventually invoke the $receive_ready_i(t, \langle ready, j \rangle)$ action at the transaction master site. Respectively, the only action enabled at site i (when $participants_i(t) = \emptyset$) will be the $end_commit_i(t)$ action. This action will commit the transaction at i and multicast the $\langle commit, t \rangle$ message to the rest of nodes that leads to transaction commitment at the rest of sites. The only actions enabled for t at j (being $j \in \mathcal{N}, j \neq i$) are the $end_operation_j(t, WS.ops)$ or $receive_commit_j(t, \langle commit, t \rangle)$ actions depending whether $status_j(t) =$ pre_commit or $status_j(t) =$ blocked, respectively. If the transaction is still *blocked*, assuming weak fairness for action execution and due to the fact that there are no unilateral aborts, the $end_operation_j(t, WS.ops)$ will be eventually enabled. By its effects, $status_j(t) =$ pre_commit. As the $\langle commit, t \rangle \in channel_j$ then the $receive_commit_j(t, \langle commit, t \rangle)$ action, by weak fairness of actions, will be eventually executed. Hence, $\forall j \in \mathcal{N} \setminus \{i\}$, the Lemma holds following the causal chain. □

The following Lemma emphasizes the *happens-before* relationship for remote transactions. It is based on Property 4.2 which establishes the relationship between status transitions for remote transactions to their respective algorithm actions. This will serve in order to set up the relationship for a transaction t, $node(t) = i \neq j$ between the $execute_remote_j$, that submits t to the DB_j module, and the pair $end_operation_j(t, WS.ops)$ and $receive_commit_j(t, \langle commit, t \rangle)$ actions. This is needed due to the fact that with the previous Lemma there is no point where this causal relationship may be put in.

Lemma 7. *Let* $\alpha = s_0 \pi_1 s_1 \ldots \pi_z s_z \ldots$ *be a fair execution of the ERP automaton and* $t \in \mathcal{T}$ *be a*

committed transaction, $node(t) = i$, then the following happens-before relations hold: $\forall j \in \mathcal{N} \setminus \{i\}$: $receive_remote_j(t, \langle remote, t, WS\rangle) \prec_\alpha execute_remote_j(t) \prec_\alpha end_operation_j(t, WS.ops) \prec_\alpha receive_commit_j(t, \langle commit, t\rangle)$.

Proof. As t is a committed transaction. By Property 4.1, it has previously been with $status_i(t) =$ active. As $status_i(t) =$ pre_commit has been also achieved, the $begin_commit_i(t)$ action has been executed. It multicasts to the rest of nodes the $\langle remote, t, WS\rangle$ message. The reception of this message will invoke the $receive_remote_j(t, \langle remote, t, WS\rangle)$ that inserts t into $queue_j$. When t reaches the highest priority among all delivered transactions at j, it will be submitted to DB_j and the *ready* message will be sent to i. The collection of all *ready* messages at i will invoke the $end_commit_i(t)$ action that multicasts the $\langle commit, t\rangle$ message to all nodes excluding i. The remote transaction will eventually finish ($DB_j.notify(t, WS.ops) =$ run), either before or after the $end_commit_i(t)$ action that executes the $end_operation_j(t, WS.ops)$ action. By its effects, $status_j(t) =$ pre_commit and by weak fairness action execution, the $receive_commit_j(t, \langle commit, t\rangle)$, as $\langle commit, t\rangle \in channel_j$, will be executed. Then this Lemma holds for all remote transactions that finally commit. □

The following Property and Corollary establish a property about local executions of committed transactions. They are the same as in Section 3.3.

Property 6. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be an arbitrary execution of the ERP automaton and $i \in \mathcal{N}$. If there exist two transactions $t, t' \in \mathcal{T}$ such that $t \xrightarrow{*} t'$ in $SG(H_i(\alpha))$ then $\exists z_1 < z_2 < z_3 < z_4$: $s_{z_1}.status_i(t) = pre_commit \wedge s_{z_2}.status_i(t) = committed \wedge s_{z_3}.status_i(t') = pre_commit \wedge s_{z_4}.status_i(t') = committed$.*

The latter Property reflects the *happens-before* relationship between the different *status* of conflictive transactions. The same order must hold for the actions generating the mentioned status. The next Corollary expresses this property.

Corollary 2. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the ERP automaton and $i \in \mathcal{N}$. If there exist two transactions $t, t' \in \mathcal{T}$ such that $t \xrightarrow{*} t'$ in $SG(H_i(\alpha))$ then the following happens-before relations, with the appropriate parameters, hold:*

1. *$node(t) = node(t') = i$: $begin_commit_i(t) \prec_\alpha end_commit_i(t) \prec_\alpha begin_commit_i(t') \prec_\alpha end_commit_i(t', \langle commit, t'\rangle)$.*

2. $node(t) = i \wedge node(t') \neq i$: $begin_commit_i(t) <_\alpha end_commit_i(t) <_\alpha end_operation_i(t', WS'.ops) <_\alpha receive_commit_i(t', \langle commit, t'\rangle)$.

3. $node(t) \neq i \wedge node(t') = i$: $end_operation_i(t, WS.ops) <_\alpha receive_commit_i(t, \langle commit, t\rangle) <_\alpha begin_commit_i(t') <_\alpha end_commit_i(t')$.

4. $node(t) \neq i \wedge node(t') \neq i$: $end_operation_i(t, WS.ops) <_\alpha receive_commit_i(t, \langle commit, t'\rangle) <_\alpha end_operation_i(t', WS'.ops) <_\alpha receive_commit_i(t', \langle commit, t'\rangle)$.

If we have two conflicting transactions, $t, t' \in \mathcal{T}$ with $node(t) \neq i$ and $node(t') \neq i$, such that $t \rightarrow t'$ in $SG(H_i(\alpha)$, then the $execute_remote_i(t')$ action that submits t' to the database must be executed after the commitment of t, via the $receive_commit_i(t, \langle commit, t\rangle)$ action. The next Lemma states how the *happens-before* relationship affects to two committed transactions executing at a remote node.

Lemma 8. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the ERP automaton and $i \in \mathcal{N}$. If there exist two committed transactions $t, t' \in \mathcal{T}$ with $node(t) = j \neq i$ and $node(t') = k \neq i$ such that $t \overset{*}{\rightarrow} t'$ in $SG(H_i(\alpha))$ then the following happens-before relations hold: $\forall\, i \in \mathcal{N} \setminus \{k, j\}$: $execute_remote_i(t) <_\alpha receive_commit_i(t, \langle commit, t\rangle) <_\alpha execute_remote_i(t') <_\alpha receive_commit_i(t', \langle commit, t'\rangle)$.*

Proof. Let $t \in \mathcal{T}$ be a committed transaction, with $node(t) \neq i$. By Property 4.2, it has previously been $status_i(t) =$ blocked which has been caused by the $execute_remote_i$ action. Although this action may have been called several times, we consider the one that was successfully completed for t ($execute_remote_i(t)$). This action submits the transaction to the DB_i module and sends the *ready* message to the transaction master site. Provided that t is a remote transaction, all local conflicting transactions have been rolled back earlier. Moreover, this remote transaction will never be aborted by the underlying database (recall that we do not consider unilateral aborts for remote transaction), only the protocol itself may consider whether to abort or not a remote transaction. As t has been committed, the $end_operation(t, WS.ops)$ had been invoked and finally, with the *commit* message coming from the master site, the $receive_commit(t, \langle commit, t\rangle)$ action would have been invoked. Let us assume another committed transaction $t' \in \mathcal{T}$, with $node(t') \neq i$, such that $t \overset{*}{\rightarrow} t'$. This implies that t' committed after t. By Corollary 2.4, $end_operation_i(t, WS.ops) <_\alpha receive_commit_i(t, \langle commit, t\rangle) <_\alpha end_operation_i(t', WS'.ops) <_\alpha receive_commit_i(t', \langle commit, t'\rangle)$. Now

we have to set up the *happens-before* relationship between the $execute_remote_i(t')$ and the $receive_commit_i(t, \langle commit, t\rangle)$. If t' is delivered to i after t has committed, via the $receive_remote_i(t', \langle remote, t'\rangle)$ action, the Lemma will trivially hold for this case. Otherwise, the message is delivered after the $receive_remote_i(t, \langle remote, t\rangle)$ and before the $receive_commit_i(t, \langle commit, t\rangle)$ actions execution, and so more cases will be taken into account. The $receive_remote_i(t', \langle remote, t'\rangle)$ action will insert into $queue_i$ the delivered transaction t'. By the effects of this action, the $execute_remote_i$ action will be invoked. This action will check all the set of conflicting transactions currently executing at DB_i. There will be at least one, t, that conflicts with t'. By the invocation of the *higher_priority* function for t and t', it results that t has higher priority than t'. This fact will not change, even though several invocations of the $execute_remote_i$ action will take place, as long as t does not perform the commit, or, in other words, until the execution of the $receive_commit_i(t, \langle commit, t\rangle)$ action takes place. This can be derived by inspection of the *higher_priority* function that returns false if the compared transaction is a non-committed remote transaction currently being executed at the DB_i module. Hence, the Lemma holds. □

The same may be applied to two conflicting transactions, $t, t' \in \mathcal{T}$ with $node(t) = i$ and $node(t') \neq i$, such that $t \to t'$ in $SG(H_i(\alpha))$. The $execute_remote_i$ action that submits t' to the database must be executed after the commitment of t, via the $end_commit_i(t)$ action. The next Lemma states how the *happens-before* relationship affects to a committed transaction executing at a remote node.

Lemma 9. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the ERP automaton and $i \in \mathcal{N}$. If there exist two transactions $t, t' \in \mathcal{T}$ with $node(t) = i$ and $node(t') \neq i$ such that $t \xrightarrow{*} t'$ in $SG(H_i(\alpha))$ then the following happens-before relations hold: $\forall\, i \in \mathcal{N}$: $begin_commit_i(t) \prec_\alpha end_commit_i(t) \prec_\alpha execute_remote_i(t') \prec_\alpha end_operation_i(t', WS'.ops) \prec_\alpha receive_commit_i(t', \langle commit, t'\rangle)$.*

Proof. Let us assume two committed transactions $t, t' \in \mathcal{T}$ with $node(t) = i \neq node(t')$ such that $t \xrightarrow{*} t'$. By Corollary 2.2 we have: $begin_commit_i(t) \prec_\alpha end_commit_i(t) \prec_\alpha end_operation_i(t', WS'.ops) \prec_\alpha receive_commit_i(t', \langle commit, t'\rangle)$. As t' is a committed remote transaction at i, via Lemma 7, it has executed the following actions: $receive_remote_i(t', \langle remote, t', WS'\rangle) \prec_\alpha execute_remote_i(t') \prec_\alpha end_operation_i(t', WS'.ops) \prec_\alpha receive_commit_i(t', \langle commit, t'\rangle)$. Thus, we have to establish the *happens-before* relation between the $execute_remote_i(t')$ and the $end_commit_i(t)$ actions. Again, we have two options: if the $receive_remote_i(t', \langle remote, t', WSt'\rangle)$ action

is executed after the $end_commit_i(t)$ action then the Lemma holds. The remainder case is when the $receive_remote_i(t', \langle remote, t', WS'\rangle)$ and the successful completion of the $execute_remote_i$ action for t' happen between the $begin_commit_i(t)$ and the $end_commit_i(t)$ actions. The successful completion of $execute_remote_i$ for t' will never happen under this interval. This is easily shown by inspection of the $execute_remote_i$ action. As $t \xrightarrow{*} t'$, we have that the $getConflicts$ function will return at least t as a conflicting transaction. However, t has $status_i(t)$ = pre_commit and by hypothesis it has been executed before t'. This means that $t.priority > t'.priority$ for read-write conflicts and that it must wait enqueued. □

In the following, we prove that the ERP protocol provides 1CS [BHG87].

Theorem 2. *Let* $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ *be a fair execution of the ERP automaton. The graph* $\cup_{k\in\mathcal{N}} SG(H_k(\alpha))$ *is acyclic.*

Proof. By contradiction. Assume there exists a cycle in $\cup_{k\in\mathcal{N}} SG(H_k(\alpha))$. There are at least two different transactions $t, t' \in \mathcal{T}$ and two different sites $x, y \in \mathcal{N}$, $x \neq y$, such that those transactions are executed in different order at x and y. Thus, we consider (a) $t \xrightarrow{*} t'$ in $SG(H_x(\alpha))$ and (b) $t' \xrightarrow{*} t$ in $SG(H_y(\alpha))$; being $node(t) = i$ and $node(t') = j$. There are four cases under study:

(I) $i = j = x$.

(II) $i = x \wedge j = y$.

(III) $i = j \wedge i \neq x \wedge i \neq y$.

(IV) $i \neq j \wedge i \neq x \wedge i \neq y \wedge j \neq x \wedge j \neq y$.

In the following, we simplify the notation. The action names are shortened, i.e. $begin_commit_x(t)$ by $bc_x(t)$; $end_commit_x(t)$ by $ec_x(t)$; as each invocation of the $execute_remote_x$ action may execute a set of transactions, $\mathcal{K} \subseteq \mathcal{T}$, we denote it by $er_x(k)$, with $k \in \mathcal{K}$; $receive_ready_x(t, \langle ready, t, l\rangle)$, with $l \in \mathcal{N}$, by $rr_x(t, l)$; $end_operation_x(t, op)$ by $eo_x(t)$; and, $receive_commit_x(t, \langle commit, t\rangle)$ by $rc_x(t)$.

CASE (I) By Corollary 2.1 for (a): $bc_x(t) \prec_\alpha ec_x(t) \prec_\alpha bc_x(t') \prec_\alpha ec_x(t')$. (i)

By Corollary 2.4 for (b): $eo_y(t') \prec_\alpha rc_y(t') \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. Applying Lemmas 7 and 8 for t and t': $er_y(t') \prec_\alpha eo_y(t') \prec_\alpha rc_y(t') \prec_\alpha er_y(t) \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. (ii)

For (i), via Lemma 6 for t, we have the following: $bc_x(t) \prec_\alpha er_y(t) \prec_\alpha rr_x(t,y) \prec_\alpha ec_x(t) \prec_\alpha bc_x(t') \prec_\alpha ec_x(t')$. Taking into account Lemma 6 for t' and Lemma 8 for t and t': $bc_x(t) \prec_\alpha er_y(t) \prec_\alpha rr_x(t,y) \prec_\alpha ec_x(t) \prec_\alpha bc_x(t') \prec_\alpha er_y(t') \prec_\alpha rr_x(t',y) \prec_\alpha ec_x(t') \prec_\alpha rc_y(t')$. Therefore, we have that $er_y(t) \prec_\alpha rc_y(t')$ in contradiction with (*ii*) as it can be seen in Figure 3.10.

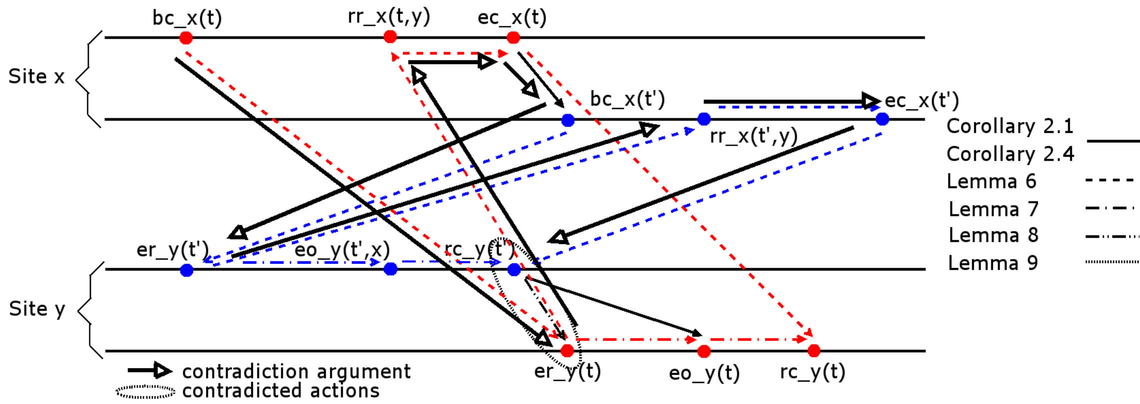


Figure 3.10: CASE (I): $node(t) = node(t') = x$

CASE (II) By Corollary 2.2 for (*a*): $bc_x(t) \prec_\alpha ec_x(t) \prec_\alpha eo_x(t') \prec_\alpha rc_x(t')$. By Lemma 9 for t and t': $bc_x(t) \prec_\alpha ec_x(t) \prec_\alpha er_x(t') \prec_\alpha rc_x(t')$. (*i*)

By Corollary 2.2 for (*b*): $bc_y(t') \prec_\alpha ec_y(t') \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. Applying Lemma 9 for t' and t: $bc_y(t') \prec_\alpha ec_y(t') \prec_\alpha er_y(t) \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. (*ii*)

By Lemma 6 for t: $bc_x(t) \prec_\alpha er_y(t) \prec_\alpha rr_x(t,y) \prec_\alpha ec_x(t)$, via (*i*), $\prec_\alpha er_x(t') \prec_\alpha rr_y(t',x) \prec_\alpha ec_y(t') \prec_\alpha rc_x(t')$. Thus $er_y(t) \prec_\alpha ec_y(t')$ in contradiction with (*ii*), see Figure 3.11.

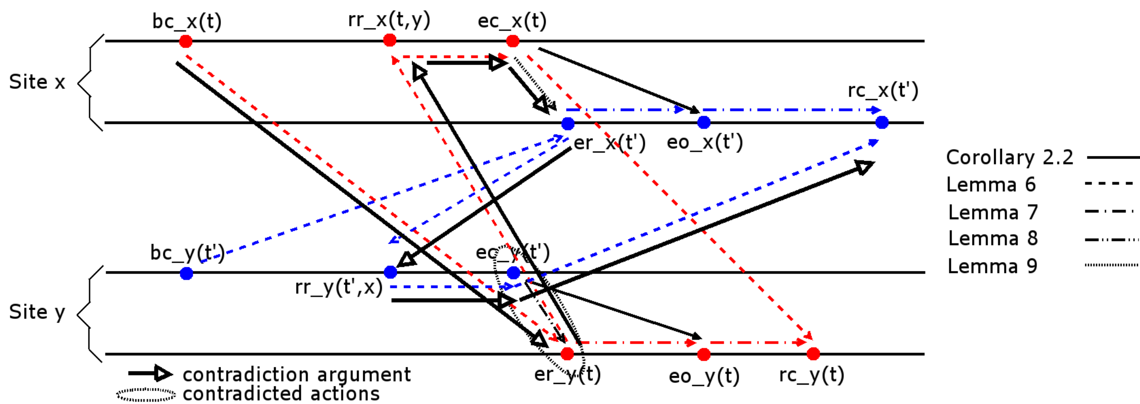


Figure 3.11: CASE (II): $node(t) = x$ and $node(t') = y$

CASE (III) As x and y are different sites from the transaction master site, only one of them will be executed in the same order as in the master site. If we take into account the different one with the master site we will be under assumptions considered in CASE (I).

CASE (IV) By Corollary 2.4 for (a): $eo_x(t) \prec_\alpha rc_x(t) \prec_\alpha eo_x(t') \prec_\alpha rc_x(t')$. Applying Lemmas 7 and 8 for t and t' at x: $er_x(t) \prec_\alpha eo_x(t) \prec_\alpha rc_x(t) \prec_\alpha er_x(t') \prec_\alpha eo_x(t') \prec_\alpha rc_x(t')$. (i)

By Corollary 2.4 for (b): $eo_y(t') \prec_\alpha rc_y(t') \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. If we apply Lemmas 7 and 8 for t' and t at y: $er_y(t') \prec_\alpha eo_y(t') \prec_\alpha rc_y(t') \prec_\alpha er_y(t) \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. (ii)

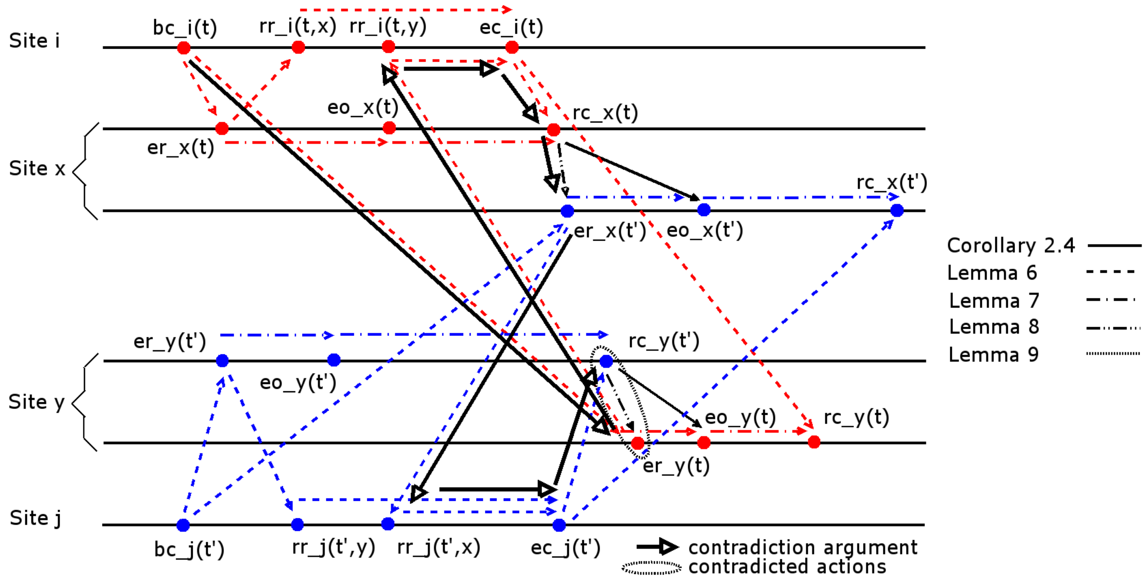


Figure 3.12: CASE (IV): $node(t) = i$ and $node(t') = j$

By Lemma 6 for t at x and y: $bc_i(t) \prec_\alpha er_y(t) \prec_\alpha rr_i(t,y) \prec_\alpha ec_i(t) \prec_\alpha rc_x(t)$. Via Corollary 2.4 for (a): $bc_i(t) \prec_\alpha er_y(t) \prec_\alpha rr_i(t,y) \prec_\alpha ec_i(t) \prec_\alpha rc_x(t) \prec_\alpha er_x(t') \prec_\alpha rr_j(t',x) \prec_\alpha ec_j(t') \prec_\alpha rc_y(t')$. Therefore, we have that $er_y(t) \prec_\alpha rc_y(t')$ in contradiction with (ii), as it is depicted in Figure 3.12. □

3.6 The TORPE Replication Protocol

In this Section we introduce a modification of the previous protocols called TORPE. It uses the total order multicast primitive provided by the GCS to order transactions executed in the system. Hence, distributed deadlocks are avoided. However, additional verifications have to be done whenever a *remote* message is delivered at a site in order to detect deadlocks between executing local

transactions and the write set of the transaction delivered. TORPE is described in this Section because it is the natural comparison with BRP and ERP protocols; it uses GCS facilities to order transactions instead of transaction priorities. We will formalize the TORPE as a state transition system [Sha93], although we are not going to introduce its correctness proof. In Figure 3.13 the TORPE state transition system [Sha93] is introduced.

Transactions are globally ordered by the total order delivery guarantee provided by the GCS. However, it is still necessary to maintain an update queue (*queue*$_i$) where delivered transactions must wait until all conflicting transactions, only those already delivered by the GCS, finish. Otherwise, they may become involved in a local deadlock with local and remote transactions whose resolution depends on the *DB* module, who does not know anything about local nor remote transactions.

TORPE is a ROWAA algorithm whose behavior is very similar to the BRP. Transactions are initially executed at their master site. The remaining sites enter in the context of this transaction when the client wishes to commit the given transaction. TORPE multicasts the *remote* message to all available sites using the total order multicast primitive. When the transaction master site receives the *remote* message, it checks if there is another write set contained in the update queue. In such a case, it aborts and multicasts, with the reliable multicast, an *abort* message to the rest of sites; otherwise, it commits the transaction and multicasts, using the reliable multicast, a *commit* message to the rest of sites. If the message arrives at another site it will check if its write set is in conflict with another already stored transaction in the update queue then it will be inserted into the last position of the queue. In any other case, it checks if there is another conflicting transaction, local or remote that has been previously delivered by the GCS, currently executing in the corresponding *DB* module, in such a case it will be inserted into the queue, otherwise it will be submitted to the *DB* module. Meanwhile, the transaction master site may have multicast the *commit* message. Hence, we still need another set to store the set of committable transactions (*committable*$_i$), those that have been delivered but has not finished applying the updates on a given replica. Respectively, an *abort* message may have been generated by the master site which removes the transaction from the queue or rollbacks its changes from the underlying database. In both cases, the update queue is checked if there are pending transaction to be applied in the node. It is important to note that once a remote transaction is submitted to the *DB* module it will not be aborted by the *DB*, in other words, TORPE does not support unilateral aborts [Ped99]; recall that

Signature:
$\{\forall i \in \mathcal{N}, t \in \mathcal{T}, m \in \mathcal{M}, op \in \mathcal{OP}$: $\mathbf{create}_i(t)$, $\mathbf{begin_operation}_i(t, op)$, $\mathbf{end_operation}_i(t, op)$, $\mathbf{begin_commit}_i(t)$,
$\mathbf{end_commit}_i(t, m)$, $\mathbf{local_abort}_i(t)$, $\mathbf{receive_remote}_i(t, m)$, $\mathbf{execute_remote}_i$, $\mathbf{receive_commit}_i(t, m)$,
$\mathbf{receive_abort}_i(t, m)$, $\mathbf{discard}_i(t, m)$, $\mathbf{commit}_i(t)\}$.

States:
$\forall i \in \mathcal{N} \wedge t \in \mathcal{T}$: $status_i(t) \in$ {idle, start, active, blocked, pre_commit, committable, committed, aborted},
initially $(node(t) = i \Rightarrow status_i(t) =$ start$) \wedge (node(t) \neq i \Rightarrow status_i(t) =$ idle$)$.
$\forall i \in \mathcal{N}$: $channel_i \subseteq \{m : m \in \mathcal{M}\}$, initially $channel_i = \emptyset$.
$\forall i \in \mathcal{N} \wedge t \in \mathcal{T}$: $committable_i \subseteq \mathcal{T}$, initially $committable_i = \emptyset$.
$\forall i \in \mathcal{N}$: $queue_i \subseteq \{\langle t, WS\rangle : t \in \mathcal{T}, WS \subseteq \{\langle oids, ops\rangle : oids \subseteq \mathcal{OID}, ops \subseteq \mathcal{OP}\}\}$, initially $queue_i = \emptyset$. // *Ordered Set* //
$\forall i \in \mathcal{N}$: $remove_i$: **boolean**, initially $remove_i =$ false.
$\forall i \in \mathcal{N}$: $\mathcal{V}_i \in \{\langle id, availableNodes\rangle : id \in \mathbb{Z} \wedge availableNodes \subseteq \mathcal{N}\}$, initially $\mathcal{V}_i = \langle 0, \mathcal{N}\rangle$.

Transitions:

$\mathbf{create}_i(t)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ start.
eff $\equiv DB_i.begin(t)$; $status_i(t) \leftarrow$ active.

$\mathbf{begin_operation}_i(t, op)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ active.
eff $\equiv DB_i.submit(t, op)$; $status_i(t) \leftarrow$ blocked.

$\mathbf{end_operation}_i(t, op)$
pre $\equiv status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ run.
eff $\equiv$ **if** $node(t) = i$ **then** $status_i(t) \leftarrow$ active
else $status_i(t) \leftarrow$ pre_commit.

$\mathbf{begin_commit}_i(t)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ active.
eff $\equiv status_i(t) \leftarrow$ pre_commit;
$sendTORMulticast(\langle remote, t, DB_i.WS(t)\rangle,$
$\mathcal{V}_i.availableNodes)$.

$\mathbf{end_commit}_i(t, m)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ committable $\wedge$.
$m = \langle commit, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$; // *Remove m from channel*
$DB_i.commit(t)$; $status_i(t) \leftarrow$ committed.

$\mathbf{local_abort}_i(t)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ abort.
eff $\equiv DB_i.abort(t)$; $status_i(t) \leftarrow$ aborted.

$\mathbf{discard}_i(t, m)$
pre $\equiv status_i(t) =$ aborted $\wedge\ m = \langle *, t, *\rangle \in channel_i$.
eff $\equiv receive_i(m)$.

$\mathbf{receive_commit}_i(t, m)$ // $node(t) \neq i$ //
pre $\equiv m = \langle commit, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$; $committable_i \leftarrow committable_i \cup \{t\}$.

$\mathbf{commit}_i(t)$
pre $\equiv status_i(t) =$ pre_commit $\wedge\ t \in committable_i$.
eff $\equiv committable_i \leftarrow committable_i \setminus \{t\}$;
$DB_i.commit(t)$; $status_i(t) \leftarrow$ committed;
$remove_i \leftarrow$ true.

$\mathbf{receive_abort}_i(t, m)$
pre $\equiv status_i(t) \notin$ {aborted, committed} $\wedge$
$m = \langle abort, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$; $status_i(t) \leftarrow$ aborted;
if $\langle t, \perp\rangle \in queue_i$ **then** $queue_i \leftarrow queue_i \setminus \{\langle t, \perp\rangle\}$
else $DB_i.abort(t)$; $remove_i \leftarrow$ true.

$\mathbf{receive_remote}_i(t, m)$
pre $\equiv status_i(t) \in$ {idle, pre_commit} $\wedge\ m = \langle remote, t, WS\rangle \in channel_i$.
eff $\equiv receive_i(m)$;
if $queue_conflicts(t, WS)$ **then**
if $node(t) = i$ **then**
$DB_i.abort(t)$;
$sendRMulticast(\langle abort, t\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$;
$status_i(t) \leftarrow$ aborted;
else $queue_i \leftarrow queue_i \cup \{\langle t, WS\rangle\}$
else
if $node(t) = i$ **then**
$status_i(t) \leftarrow$ committable;
$sendRMulticast(\langle commit, t\rangle, \mathcal{V}_i.availableNodes)$
else // *Remote transaction with no queue conflict* //
$conflictSet \leftarrow DB_i.getConflicts(WS)$;
if $(\exists t' \in conflictSet$: $(node(t') = i \wedge$
$status_i(t') =$ committable$) \vee node(t') \neq i)$ **then**
$queue_i \leftarrow queue_i \cup \{\langle t, WS\rangle\}$
else
$\forall t' \in conflictSet$:
if $node(t') = i \wedge status_i(t') \neq$ committable **then**
if $status_i(t') =$ pre_commit **then**
$sendRMulticast(\langle abort, t'\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$;
$DB_i.abort(t')$; $status_i(t') \leftarrow$ aborted
$DB_i.begin(t)$;
$DB_i.submit(t, WS)$;
$status_i(t) \leftarrow$ blocked.

$\diamond$ **function** $queue_conflicts(t, WS) \equiv \exists \langle t', WS'\rangle \in queue_i$:
$WS'.oids \cap WS.oids \neq \emptyset$

$\mathbf{execute_remote}_i$
pre $\equiv \neg empty(queue_i) \wedge remove_i$.
eff $\equiv aux_queue \leftarrow \emptyset$;
while $\neg empty(queue_i)$ **do**
$\langle t, WS\rangle \leftarrow first(queue_i)$; $queue_i \leftarrow remainder(queue_i)$;
$conflictSet \leftarrow DB_i.getConflicts(WS)$;
if $\exists t' \in conflictSet$: $(node(t') = i \wedge$
$status_i(t') =$ committable$) \vee node(t') \neq i$ **then**
$aux_queue \leftarrow aux_queue \cup \{\langle t, WS\rangle\}$
else
$\forall t' \in conflictSet$:
if $node(t') = i \wedge status_i(t') \neq$ committable **then**
if $status_i(t') =$ pre_commit **then**
$sendRMulticast(\langle abort, t'\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$;
$DB_i.abort(t')$; $status_i(t') \leftarrow$ aborted
$DB_i.begin(t)$;
$DB_i.submit(t, WS)$;
$status_i(t) \leftarrow$ blocked;
$queue_i \leftarrow aux_queue$; $remove_i \leftarrow$ false.

Figure 3.13: State transition system for TORPE, where updates are propagated using the total order group communication primitives provided by the GCS

this is something that we also assume for the ERP.

The state variables used by TORPE are very similar to the ones introduced in the BRP or ERP, the new ones and its behavior have been already introduced. The *status* variable with its new valid transitions is shown in Figure 3.14. The same can be applied to the actions used in TORPE, most of them are the same as the BRP excepting those that govern the update queue ($execute_remote_i$) and the transaction commitment in the *DB* module ($commit_i(t)$).

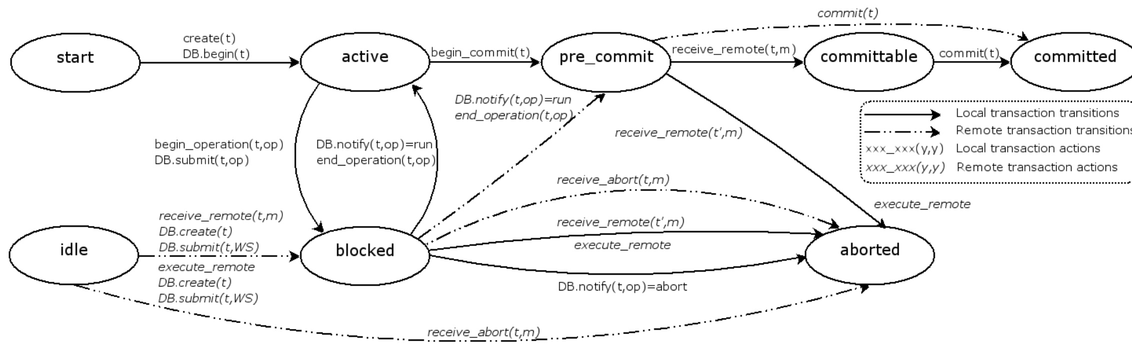


Figure 3.14: Valid transitions for a given $status_i(t)$ of a transaction $t \in \mathcal{T}$

3.7 Discussion

We have started with the BRP, its main disadvantage is that it must wait for applying the updates at all available sites before committing the transaction. On the other hand, the BRP is able to manage unilateral aborts. The BRP does not assume any message ordering delivery for any kind of messages. Therefore, distributed deadlock may appear between sites. The BRP has avoided this defining a dynamic deadlock prevention function. This function depends on both the transaction priority itself and its state, and does not allow incoming conflicting transactions to wait. Hence no distributed deadlock occur and, as there is no message ordering, this will provoke more abortions than those strictly necessary.

It has been shown that the BRP proposal is correct. We have proved that it satisfies the strongest correctness criteria for replicated databases, 1CS [BHG87]. This correctness proof allowed us to find the spots where the BRP can be modified to improve its performance. At this point, it is important to highlight that the deadlock prevention function establishes the order in which the transactions are executed in the system. This function is invoked each time a transaction is submitted to the DBMS, while all conflicting transactions have been aborted (i.e. no unilateral

aborts may occur), as it is shown in Figure 3.2. Hence, the *ready* message may be sent before the database submission of the remote transaction. Therefore, we reduce the transaction response time. Nevertheless, the priority function must be modified so that any remote transaction currently being executed at the DBMS will have greater priority than any other conflicting transaction. This modification does not directly reduce the transaction abortion rate, but if we allow remote transactions to wait and its abortion or commitment is left to its master site, we reduce the transaction abortion rate.

The ERP introduces all those key points in its behavior, as it is depicted in Figure 3.8. The ERP introduces a new *execute_remote* action that manages the queue where remote transactions are stored until they become the highest priority ones or there are no conflicts at all. The queue management shown in Figure 3.8 is not the best one but we have left it that way for a better understanding of its behavior. Finally, it has been shown that the ERP proposal is correct too.

Finally, we have presented the TORPE protocol. This replication protocol is the alternative approach used in database replication and it has been introduced so as to be compared with BRP and ERP in MADIS [IBDdJM+05]; hence, no correctness proof is given. It orders transactions using the total order delivery guarantee featured by the GCS. However, it needs additional checks with current transactions executing at the site where it has been delivered as the delivered transaction may be involved in a local deadlock and the *DB* module will abort it. In other words, TORPE does not support unilateral aborts.

As we have mentioned before, the 1CS is the strongest correctness criteria for replicated databases. However, this is almost impossible to reach with current commercial databases, as most of them provide SI [BBG+95], like PostgreSQL. SI is the result of a certain type of MultiVersion Concurrency Control (MVCC). It extends the multiversion mixed method described in [BHG87], which allowed snapshot reads by readonly transactions. In which a transaction reads data from a snapshot of the (committed) data as of the time the transaction started, *Start-Timestamp*. This time may be any time before the transaction performs its first read. A transaction running in SI is never blocked attempting a read as long as the snapshot data from its *Start-Timestamp* can be maintained. Write operations performed by the transaction will also be reflected in this snapshot, to be read again if the transaction accesses the data a second time. Updates by other transactions active after the transaction *Start-Timestamp* are invisible to the transaction. As read operations do not block, the 1CS cannot be reached with DBMSs providing SI. When the transaction T_1 is

ready to commit, it gets a *Commit-Timestamp*, which is larger than any existing *Start-Timestamp* or *Commit-Timestamp*. The transaction successfully commits only if no other transaction T_2 with a *Commit-Timestamp* in the execution interval of T_1: [*Start- Timestamp*, *Commit-Timestamp*] wrote data that T_1 also wrote. Otherwise, T_1 will abort. This feature, called first-committer-wins, prevents lost updates. When T_1 commits, its changes become visible to all transactions whose *Start-Timestamps* are larger than the *Commit-Timestamp* of T_1.

There are some recent research works [LKPMJP05, EPZ05] trying to establish a correctness criteria for replicated databases providing SI. In [LKPMJP05] a correctness criteria very similar to 1CS, called 1CSI, is shown. 1CSI establishes an SI-schedule $\mathcal{S}$ produced by a centralized scheduler over the set $\mathcal{T}$ of committed transactions. Each local SI-scheduler $\mathcal{S}_i$ must be somehow equivalent to this global schedule. Let us denote as $\mathcal{T}_i$ the execution of the set $\mathcal{T}$ at site i. As $\mathcal{T}$ and $\mathcal{T}_i$ share the same write sets, each local SI-scheduler $\mathcal{S}_i$ should set up the same order as $\mathcal{S}$ to all conflicting write sets. As only local transactions at site i have nonempty read sets they must have the same reads-from relationship [BHG87] as in $\mathcal{S}$. The *Start-Timestamp* for remote transactions is irrelevant since they have empty readsets. The same can be applied for the *Commit-Timestamp* of read-only transactions as their write set is empty. Another correctness criteria is GSI [EPZ05]. This criteria allows the use of older snapshots, instead of the latest one, in order to facilitate its implementation. Two conflicting transactions whose write set intersection is nonempty follow the first-committer-wins rule. A read operation performed by a transaction only reads a committed snapshot of the database. This snapshot could be any snapshot that has been taken before the transaction starts.

The BRP and ERP proposals may be easily ported to a replicated database system with DBMS with SI-schedulers in order to guarantee GSI [EPZ05]. As read operations never get blocked, we have to change the *getConlicts*(WS) function definition, more precisely: $getConflicts(WS) = \{t' \in \mathcal{T} \colon WS(t').oids \cap WS(t).oids \neq \emptyset\}$. Hence, each time a new remote transaction is received in the BRP (see Figure 3.2) or the *execute_remote* action in the ERP submits a remote transaction to the DBMS (see Figure 3.8) its write set must be checked with this new function.

By inspection of the correctness proof for the BRP, Lemma 3 (Lemma 6 for the ERP) states that the protocol behavior is not influenced by the underlying database. On the other hand, Property 3 (Property 6 for the ERP) asserts that the execution depends on the transaction isolation level that imposes a determined order. Let $t, t' \in \mathcal{T}$ be two conflictive committed transactions as

$WS(t).oids \cap WS(t').oids \neq \emptyset$. Then the Property 3 (Property 6 for the ERP) holds. It can be shown that with Lemma 3 and Property 3 (Lemma 6 and Property 6 for the ERP) all write sets are applied at all sites following the same order. This fact does not assure, due to network latency, that a read or write transaction may obtain a different snapshot from the current snapshot; this leads to a similar behavior to GSI.

3.7.1 Comparison with Related Works

The protocols included in this Chapter are based on the O2PL protocol [CL91], which is a 2PC protocol. It is one of the first optimistic replication protocols proposed that uses the ROWAA technique. It only needs to multicast the write set of a transaction to the rest of sites in order to commit the transaction. Nevertheless, it presents some drawbacks that we have coped with our BRP and ERP proposals: (*a*) It is never stated the transaction isolation level achieved with the O2PL; (*b*) it does not show its correctness proof; (*c*) it does not introduce how to handle the unilateral aborts [Ped99]; and, (*d*) it does not establish the communication protocols needed to accomplish the replication. The ERP has slightly modified the 2PC rule, since it does not wait for applying the updates at the rest of nodes before committing the transaction.

As the mechanism of group communication primitives was being improved, it has been shown that total order based replication protocols show a good behavior [AT02, EPZ05, JPPMKA02, Kem00, KA00b, KPA+03, LKPMJP05, Ped99, PMJPKA00, PMJPKA05, RMA+02, WK05]. In [Kem00, KA00b] the SER-D and SER-CS are introduced. All transactions are firstly performed on their closest site (*Local Read Phase*). Read operations are executed immediately, while write operations are treated differently depending on the protocol considered; the SER-D defers them until the end of the transaction whilst SER-CS performs them on a shadow copy. When the user wishes to commit a transaction (*Send Phase*), if it is read-only it will directly commit, otherwise, write operations will be multicast to the rest of sites using the total order service. All write operations are atomically performed at this moment (*Lock Phase*), aborting all conflicting transactions in the local read phase and those that have entered in the send phase, or will be enqueued if a conflicting transaction has previously reached the lock phase. If the message is delivered at the transaction master site, and it has not been aborted yet, it commits the transaction in the local database and multicasts a commit message to the rest of sites using a reliable multicast.

This total order replication protocol presents some good advantages, since the master site

does not have to wait for applying the updates at the rest of nodes. The total order delivery sets up the order in which transactions are going to be executed in the system, avoiding distributed deadlocks (however, additional verifications with local transactions have to be done once a write set is delivered at a site). As the write set is collected from the *redo log* of the RDBMS it is faster to apply than local user transactions. On the other hand, it does not deal with unilateral aborts as the BRP does. As there is no way to provide a semantic about transactions to the GCS, i.e. in case of a distributed application that may generate concurrent transactions, the application cannot establish a total order based on transaction information [Kem00, Sec. 10.2]. Comparatively, the BRP and ERP have associated priorities to transactions and these priorities may be selected by the user applications. So we are able to add some semantic information to the transaction order. Moreover, non-conflicting transactions must be total order delivered whilst using the BRP and ERP will be concurrently executed. Finally, it was implemented modifying the DBMS internals which has problems of portability even to new version releases of the same DBMS [WK05]; we have implemented our protocols using the middleware approach that ensures DBMS portability but this introduces additional overhead that causes a decrease on its performance. An optimization of this protocol using the optimistic atomic broadcast primitive [Ped99] was introduced in [KPA+03]. Another approach for achieving database replication in a middleware architecture based on the optimistic atomic broadcast ideas introduced in [PGS98] for WAN is presented in [RMA+02].

A more aggressive version of the optimistic atomic broadcast in a middleware architecture is presented in [JPPMKA02, PMJPKA05]. Database stored procedures are executed as transactions defining a conflict class. Transactions issued by users are delivered using the optimistic atomic broadcast to all sites but the outcome of transactions is only decided at the master site of the respective conflict class. Hence, remote sites do not even have to wait for the definitive total order to execute a transaction. It additionally provides good scalability results. The BRP and ERP may accept any SQL statement, hence they are more flexible; however, ours present a higher overhead since they propagate the SQL statements and we do not try to balance the workload.

Another way for achieving database replication is by means of epidemic algorithms as introduced in [HSAA03]. They propose three different replication protocols over 2PL DBMSs which are intended to be used on WANs or the Internet, where the number of replicas is small to moderate and all replicas can execute update operations. All protocols follow the ROWAA policy, all operations are firstly performed at a given site and when the user wishes to commit, the transac-

tion enters in the *pre-commit* state and updates are inserted into a log, maintaining the write locks. Replicas exchange their respective logs to keep each other informed about the operations that have occurred on their sites by way of epidemic propagation. Epidemic algorithms are generally implemented using vector clocks [Lam78] to ensure this property. When a replica receives a log record containing a transaction t, it checks in its copy of the log if there exist a concurrent non-committed transaction t' whose read set and write set intersection with the write set of t is nonempty. In such a case, t and t' are aborted and both aborts are inserted in the copy of its log. Otherwise, write locks are acquired on behalf of t and updates are applied, reaching the *pre-commit* state too. As epidemic messages exchange continues, the master site will know if all sites have completely applied the updates done by t and will commit the transaction and then inserts a commit in the copy of its log. The same can be applied to the rollback of a transaction. The second protocol is a more optimistic approach as transactions commit as soon as they terminate locally and inconsistencies are detected later. The reconciliation process is left to the application. The last protocol use quorums to resolve conflicts avoiding that both conflicting transactions abort. The first solution presents the same problem as the BRP, it must wait for the application of the updates at all available nodes. They have solved this problem by the use of quorums which overloads the protocol, instead of the priority function defined by us that led to the definition of the ERP. We are not concerned with the optimistic approach as it is a lazy replication protocol.

We have introduced a modification to our protocols so as to support SI DBMSs and provide GSI [EPZ05]. However, it is not our goal to compare our proposal with solutions proposed in [EPZ05, LKPMJP05, WK05]. We have simplified ours in order to achieve the desired consistency but this does not achieve the same performance as them, because we still need two message rounds to commit a transaction. They are a good approach since most of commercial DBMSs offer this isolation level. They only need one total order message per transaction. They do not transfer the read set and it is enough to have a version counter on each node. The certification phase is very easy. The only drawback is how to handle its associated write set queue, but this may also be kept so as to implement a simple recovery protocol.

Chapter 4

MADIS: A Slim Middleware for Database Replication

This Chapter describes the middleware architecture used in the MADIS project for maintaining the consistency of replicated databases. In the proposed architecture, most of the effort is spent using basic resources provided by conventional database systems (e.g. triggers, views, etc). This allows the underlying database to perform more efficiently many tasks needed to support any replication protocol, and simplifies the implementation of such protocols. The system design allows the database to maintain simultaneously the metadata needed for making several replication protocols work as pluggable modules.

4.1 Introduction

MADIS has been designed with the aim of isolating the Consistency Manager (CM) from the underlying DBMS particularities. It takes advantage of database resources to perform efficiently part of the tasks needed to complete the consistency management in a replicated database. The CM provides an interface where a wide range of replication protocols, like the ones depicted in Chapter 3, may be plugged in.

Our architecture, puts an emphasis on its database support for replication and consistency. MADIS provides a flexible middleware designed as a two-layer service conceived to enhance the availability and performance of networked databases. The lower layer consists of an extension of the original schema of an existing database. The defined extension only uses standard SQL

features, such as triggers, rules and functions, providing to the upper layer (i.e. the middleware that includes all consistency management) the information needed to carry out its tasks, minimizing the overhead of such collection. For instance the metadata management, such as the set of records read, written, created or deleted in each transaction is automatically stored in one of the tables of this database extension. When needed, the replication protocol retrieves that information avoiding the use of complex routines to do so. As a result, the mechanisms of the middleware to manage the collection, retrieval and removal of such meta-data have been simplified, when compared to those needed in other middleware-based systems, such as COPLA [IMDBA03].

Of course, the performance of a middleware-based database replication system will be worse than that of a core-based one, as Postgres-R [Kem00, KA00a], but its advantage is to be easily portable to different DBMSs. Moreover, the upper layer of our middleware can be implemented in any programming language, since the support it needs is directly installed in the DBMS using standard SQL.

Our MADIS middleware supports pluggable protocols that ensure the consistency of transactions involving replicas. MADIS may support different kinds of pluggable protocols, whose paradigms range from eager [AAES97, KA00b] to lazy update propagation [FMZ94], from optimistic [Sch81] concurrency control to pessimistic [BK91], from primary copy to update everywhere [WSP+00], etc. In particular, its API allows to switch from one replication protocol to another, as needed. We emphasize that this switch can be performed without the need of recalculating the required metadata for the newly plugged-in protocol. This ensures that the process of switching protocols is seamless and fast. Additionally, it enhances the modularity and openness of the system.

This Chapter provides some contributions: (*a*) A detailed description of the schema modifications needed to provide support for the CM. (*b*) All such modifications only need SQL standard features. Hence, we have automatized most consistency management tasks. (*c*) A very simplified architecture for consistency management as MADIS serves as a benchmark for different replication protocols. (*d*) A study of the overhead introduced by MADIS and its comparison with other similar approaches. (*e*) The BRP and ERP have been implemented in MADIS and we present some experimental results where it can be shown the enhancements introduced by the ERP in relation to BRP. (*f*) We have implemented TORPE in MADIS so as to compare a total order replication protocol with our BRP and ERP proposals.

The rest of this Chapter is structured as follows. Section 4.2 describes the structure and functionality of MADIS. Section 4.3 describes the schema modification that MADIS proposes to aid a local CM. Section 4.4 outlines a Java implementation of the CM, in the form of a standard JDBC driver. The interface between the CM and a generic replication protocol that may be implemented in this architecture is introduced in Section 4.5. In Section 4.6 a performance analysis is included, it presents several experimental results: The overhead introduced by MADIS compared with PostgreSQL and other middleware architectures; and, the evaluation of the implementation of BRP, ERP and TORPE in MADIS. Finally, some discussions about MADIS and a review of recent works are included in Section 4.7.

4.2 The MADIS Architecture

The MADIS architecture is composed by two main layers. Figure 4.1 shows the overall layout of the MADIS architecture. The bottom one generates some extensions to the relational database schema, adding fields to some relations and also tables to maintain the collected write sets and (optionally) read sets of each transaction, also known as the transaction report. The modifications done in the transaction report tables are managed inside the same transactional context as the transaction which these modifications refer to. These columns and tables are automatically filled by triggers and stored procedures that must be installed. Thus, the application layer will see no difference between the MADIS JDBC driver and the native JDBC driver.

The upper layer of the MADIS architecture is positioned between the client applications and the database. It acts as a database mediator. The top layer of the MADIS architecture is the CM, which is positioned between the client applications and the database. Its implementation can be done regardless of the underlying database.

A database replication and recovery protocol has to be plugged into this CM. A replication protocol has to provide a `ConsistencyProtocol` interface to the CM. Besides, it may implement some `Listener` interfaces in order to be notified about several events related to the execution of a given transaction. This functionality will be described in Section 4.4. Take into account that the replication protocol can also gain access to the incremented schema of the underlying database to obtain information about transactions, thus performing the actions needed to provide the required consistency guarantees. The replication protocol can also manipulate the incremented schema,

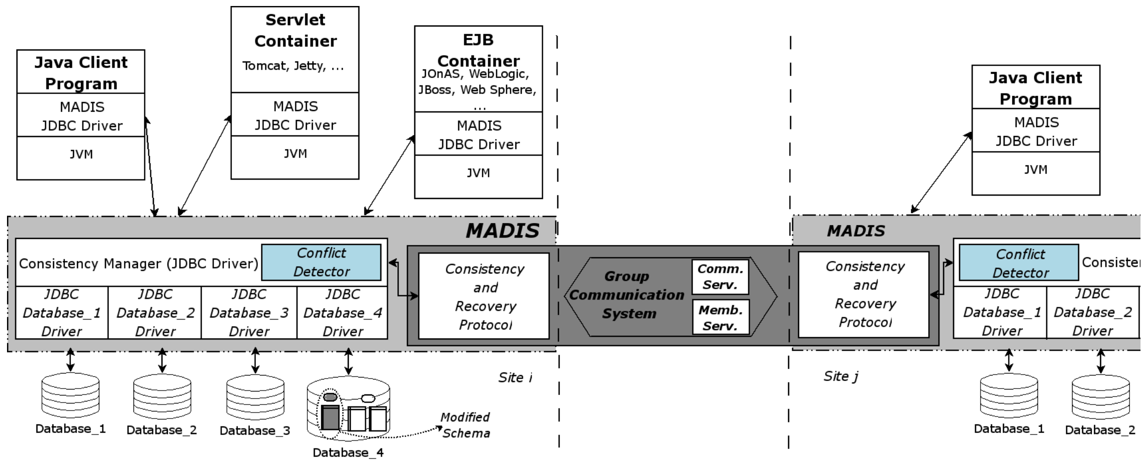


Figure 4.1: The MADIS Architecture

making use of the provided database procedures when needed. Finally, the replication protocol is also responsible of managing the communication among the database replicas. To this end, it has to use a GCS that provides at least the set of communication primitives depicted in Chapter 2, as well as others, such as total or causal order [CKV01].

The CM is composed by a set of Java classes that provide a JDBC-compliant interface. These classes implement the following JDBC interfaces: `Driver`, `Connection`, `Statement`, `CallableStatement`, `ResultSet`, and `ResultSetMetaData`. They are used to intercept all invocations that could be relevant for a database replication protocol. The invocations made on other interfaces or operations are directly forwarded to the native JDBC driver (the PostgreSQL one, in our case). Besides these classes there exists a CM class (or `RepositoryMgr`) that is also able to provide a skeleton for this layer that maintains the rest of classes and gives also support for parsing the SQL sentences in order to modify them in some cases.

In the rest of this Chapter, we describe a Java implementation, designed to be used by the client applications as a common JDBC driver and how to plug a replication protocol to this architecture. This driver functionality introduces a consistency control over a replicated database that is provided in a transparent way to user applications.

4.3 Schema Modification

The lower layer of the MADIS architecture consists of a modification in the schema of the existing database. The process for distributing an existing centralized database starts with the execution of

a program that performs a schema migration at each replicated node. This migration consists of the inclusion of tables, views, triggers and database procedures designed to maintain, automatically, a number of reports about the activity performed during the lifetime of a transaction. That way, the schema modification allows the database to automatically perform:

- Collection and management of the information pertaining to the write operations performed by any local transaction. In other words, collection of write sets.
- Management of the metadata pertaining to the different records in the database. Different metadata are needed by the different replication protocols the upper layer may manage [MEIBG+01, RMA+02]. Therefore, each site must store this kind of information.
- Optionally, collection and management of the information relating to any read operation performed by any local transaction. That is, collection of extensive read set (including the information read to perform queries). If this information is not generated, a replication protocol requiring the set of obtained objects must perform some additional work to obtain this information from the upper layer.

The operations needed by the replication protocols can be performed through a number of added database procedures, thus enabling an *ad-hoc* management (not always required) of the information automatically maintained in the database.

4.3.1 Modified and Added Tables

Modified Tables

For each existing table T_i in the original database schema, MADIS defines a number of modifications, relating field additions, view definitions, and others. Therefore, a new field is added for metadata purposes so as to identify a record on T_i, this new field is called `LOCAL_`T_i`_OID`. To this end, a field is added, defining a link to the metadata associated with each record in the table T_i. The reason for this nomenclature can be found in the possibility for the initial schema to include JOIN clauses, using the form "`SELECT * FROM T1 LEFT JOIN T2`", where the fields used to perform the matching are those with the same name in both tables.

The `LOCAL_`T_i`_OID` holds the local object identifier for the record that is local to a particular node in the system. Thus, it is possible for an object (identified by a *unique* `GLOBAL_OID`) to have

different LOCAL_T_i_OID's within the system. A GLOBAL_OID is required for the different nodes in the system, to agree in the identity of each record, regardless the local identification (sensible to local information).

Added Tables

In addition, MADIS creates for each table in the original schema (T_j) an extra table (named MADIS_META_T_j), containing the metadata needed for any protocol pluggable in the consistency manager. When a protocol is activated, MADIS executes a start-up process, to initialize each "META" table in the database. The primary key of the table consists of a unique object identifier. A typical "META" table is described as:

- LOCAL_OID. *(primary key)* The local identifier for the object.
- GLOBAL_OID. A global object identifier. Unique in the entire system. This identifier is composed of the object's creator node identifier and the local object identifier in that node.
- VERSION. The version number for the object.
- TRANSACTION_ID. The identifier for the last transaction that updated the object.
- TIMESTAMP. The most recent time the object was locally updated.

These MADIS_META_T_j tables contain part of the information needed by a wide range of replication protocols, at least those that were implemented in COPLA [AGME05, MEIBG+01, RMA+02]. Hence, as all the fields are automatically maintained by the database manager, any of such protocols is suitable to be activated at will. It is important to note that a particular protocol will only make use of a reduced set of metadata. However, the database manager must always maintain all metadata so that switching from one protocol to another may be done at any time.

In addition to these metadata tables, MADIS defines a table MADIS_TR_REPORT containing a log including the activity of each transaction of the database. The table is as follows:

- TRID (part of the primary key). Transaction identifier.
- GLOBAL_OID (part of the primary key). The global object identifier.
- FIELD_ID *(optionally)* (part of the primary key). The identifier for the accessed field.

- MODE. The mode the object was accessed (read/insert/delete/modify).

For each transaction, only one record per field-of-object is maintained in the MADIS_TR_REPORT table. In addition, once the transaction is terminated, the CM eliminates from this table any record relating the concluded transaction. Note that several MVCC-based DBMSs do not use locks with record granularity, but locks that block access to entire pages or even tables. Such systems must use multiple "per transaction" temporary *TrReport* tables, including the transaction in the table name (i.e., these tables have a <TRID>_TR_REPORT name).

4.3.2 Triggers

As mentioned, MADIS introduces a set of new triggers in the database schema definition. These triggers can be classified in three main groups:

- *Write set managers.* They are responsible for the collection of the information relating the objects written by the executing transactions.
- *Read set managers.* Collect the information related to the objects read by executing transactions. Their inclusion in the schema is optional, and when included, it is requested to be implemented by creating views.
- *Metadata automation.* These triggers are executed when the metadata stored in the MADIS extension tables must be updated. The collection and maintenance of such information is performed automatically by the triggers.

Write Set Collection

The write set collection is performed defining three triggers for each table T_i in the original schema. They insert in the MADIS_TR_REPORT table the information related to any write-access to the table performed by the executing transactions.

The write set collector (WSC) triggers are named WSC_I_T_i, WSC_D_T_i, and WSC_U_T_i, and their definition allows to intercept any write access (insert, delete or update respectively) to the T_i table, recording the event in the transaction report table (MADIS_TR_REPORT).

```
CREATE TRIGGER WSC_I_MY_TABLE BEFORE INSERT ON MY_TABLE FOR EACH ROW
   EXECUTE PROCEDURE TR_INSERT(MY_TABLE, GET_TRID(), NEW.L_MY_TABLE_OID);
```

The example shown above defines a basic WSC trigger that is related to the insertion of a new object. Note that the trigger executes the procedure GET_TRID() to obtain the current transaction identifier that corresponds to the transaction identifier assigned by PostgreSQL to the transaction. Respectively, the field L_MY_TABLE_OID contains the identifier of this record in the database. The example inserts a single row in the MADIS_TR_REPORT table for each insertion in the table MY_TABLE. The execution of the invoked procedure causes the DBMS to insert into the MADIS_TR_REPORT table the adequate rows, in order to keep track of the transaction activities.

Deletions and updates must also be intercepted by means of analogous triggers. However, as described above, the accessed fields can be optionally included in the transaction report (depending on the configuration of the MADIS middleware). To this end, a WSC trigger managing the updates should be *split* into a number of triggers, one for each field contained in the managed table (WSC_U_MY_TABLE_FIELD**1**, ...WSC_U_MY_TABLE_FIELD**N**). Here is an example of update and delete triggers:

```
CREATE TRIGGER WSC_U_MY_TABLE_FIELD1 BEFORE INSERT ON MY_TABLE FOR EACH
   ROW WHEN OLD.FIELD1<>NEW.FIELD1 EXECUTE PROCEDURE TR_UPDATE(
      'MY_TABLE', 'FIELD1', GET_TRID(), NEW.L_MY_TABLE_OID);
CREATE TRIGGER WSC_U_MY_TABLE_FIELD2 BEFORE INSERT ON MY_TABLE FOR EACH
   ROW WHEN OLD.FIELD2<>NEW.FIELD2 EXECUTE PROCEDURE TR_UPDATE(
      'MY_TABLE', 'FIELD2', GET_TRID(), NEW.L_MYTABLE_OID);
...
CREATE TRIGGER WSC_D__MY_TABLE_FIELD1 AFTER DELETE ON MY_TABLE FOR EACH
   ROW EXECUTE PROCEDURE TR_DELETE_MY_TABLE()'';
CREATE TRIGGER WSC_D__MY_TABLE_FIELD2 AFTER DELETE ON MY_TABLE FOR EACH
   ROW EXECUTE PROCEDURE TR_DELETE_MY_TABLE()'';
```

Read Set Collection

The second group of triggers is responsible for the read set collection of transactions. As already mentioned, this collection is optional, due to its high cost, and the fact that some replication protocols can be accomplished without using read sets. To implement this collection, a view must be included for each table in order to compensate the lack of TRIGGER ... BEFORE SELECT in the SQL-99 standard. The original table must be renamed and replaced by the new view. The following example shows the view inserted for a table MY_TABLE, in order to collect any read access performed on it:

```
CREATE VIEW MY_TABLE (FIELD1, FIELD2, L_MY_TABLE_OID) AS SELECT
   TR_SELECT('MY_TABLE', 'FIELD1', GET_TRID(), L_MY_TABLE_OID),
```

```
    TR_SELECT('MY_TABLE','FIELD2', GET_TRID(), L_MY_TABLE_OID),
       L_MY_TABLE_OID FROM MY_TABLE;
```

As views cannot be updated in several DBMSs, it becomes also necessary for the WSC triggers to be modified, in order to redirect the write accesses to the renamed original table. This can be done by implementing the WSC triggers as 'INSTEAD OF EVENT' triggers, (in contrast to the basic BEFORE EVENT detailed above). Finally, the TR_INSERT, TR_UPDATE and TR_DELETE procedures should be modified, in order to include the required redirection.

Metadata Management

The last group of triggers added by MADIS is that responsible for the metadata management. In fact, this management can be disseminated in the WSC triggers detailed in this Section. However, we describe here the metadata management implementation as independent triggers, in order to simplify the discussion. Whenever a new record is inserted, the DBMS must automatically insert the corresponding row in the metadata table. To this end, MADIS includes, for each table T_i, a trigger that inserts a row in the corresponding MADIS_META_T_i table. As the GLOBAL_OID is established based on the creator node identifier (i.e. the node where the object was created), and the local object identifier in the creator node, all fields contained in the MADIS_META_T_i table can be filled without intervention of any replication protocol.

Following the life-cycle of a row, when a row is accessed in write mode, the DBMS must intercept the access, and the metadata (e.g. version, timestamp and so on) of such object must be updated. To this end, a specialized metadata maintainer (MM) trigger is included for each table. The MM trigger updates the version, the transaction identifier, and timestamp of the record in the given metadata table. As an example, the following shows the implementation of a basic MM trigger for MY_TABLE:

```
CREATE TRIGGER MM_MY_TABLE BEFORE UPDATE ON MY_TABLE FOR EACH ROW
   EXECUTE PROCEDURE META_UPDATE('MY_TABLE', GET_TRID(),
      NEW.L_MY_TABLE_OID)
```

Finally, when an object is deleted, the corresponding metadata row must be also deleted. To this end, an additional trigger is also included for each table in the original schema.

Summarizing the tasks performed by the described triggers, it is easy to see that, for each table, only three triggers must be included: BEFORE INSERT, BEFORE UPDATE, BEFORE DELETE. Their implementation includes both the transaction report management, and the metadata maintenance. If the read set management is a requirement, it is necessary to replace the definition of the triggers, implementing

INSTEAD OF triggers, in contrast to BEFORE triggers. This allows the DBMS to redirect any write access to the adequate table, as well as to perform the metadata maintenance and the transaction management.

Database Procedures

As it has been depicted, data management of the MADIS_TR_REPORT table as well as the respective MADIS_META_T_j tables are carried out by database functions. Their purpose is multiple; they vary from obtaining the transaction identifier to a row insertion in a given metadata table. These functions are implemented using the PL/pgSQL procedural language provided by PostgreSQL. It permits to add control structures to the SQL language. Therefore, it will be easier to automatize the filling of different metadata tables. The PL/pgSQL is easy to use since it is very intuitive to programmers.

These functions are specially designed to be performed as long as a transaction is performing operations. However, the aim of the MADIS_TR_REPORT table is to collect the objects written and read by a given transaction. MADIS has defined a set of functions that perform this task. They can be implemented using the PL/pgSQL language but the response time of this implementation will be unacceptable in certain cases. Thus, we have defined the functions that return the objects read and written by a transaction as native functions implemented in C.

Detecting Conflicting Transactions

In many database replication protocols we may need to apply the updates propagated by a remote transaction. If several local transactions are accessing the same data items that this remote update, such remote update will remain blocked until those local transactions terminate. Moreover, if the underlying DBMS uses a MVCC combined with a SI level [BBG+95], such a remote update is commonly aborted, and it has to be reattempted until no conflicts arise with any local transaction. Additionally, in most cases, the replication protocol will end aborting also such conflicting local transactions once they try to commit. As a result of this, it seems appropriate to design a mechanism that notifies to the replication protocol about conflicts among transactions, at least when the replication protocol requires so. Once notified, the replication protocol will be able to decide which of the conflicting transactions must be aborted and, once again, a mechanism has to be provided to make possible such abortion.

To this end, we have included in the underlying database some support for detecting transaction conflicts that have produced a transaction blocking. It consists of a stored procedure, provided that the underlying DBMS is PostgreSQL [Pos05], named GET_BLOCKED() that looks for blocked transactions in the view PG_LOCKS placed in the PostgreSQL system catalog. It returns a set of pairs composed by the identifier of a blocked transaction and the identifier of the transaction that has caused such a block.

Transaction Termination

A replication protocol may abort an ongoing transaction canceling all its statements. This implicitly rollbacks such a transaction, and may be requested using standard JDBC operations. If the transaction is currently executing a statement, it may be aborted using another thread to request such a cancelation. Respectively, the replication protocol may decide to commit a transaction. This procedure closely follows the commit procedure used with standard JDBC operations, i.e. there will not be any ongoing statement in the DBMS belonging to the transaction.

4.4 Consistency Manager

The architecture proposed by MADIS makes use of the database as the manager for most information related to consistency management. Moreover, the DBMS also provides the collected information to the CM with standardized structures. Thus, the consistency management can be ported from a platform to another with a minimal effort. The rest of this Section shows a Java implementation of a CM making use of the described schema modification.

Our Java implementation of the CM allows a generic replication protocol to intercept any access to the underlying database, in order to coordinate both local accesses, and update propagation of local transactions (and, respectively, the local application of remote transactions).

In our basic implementation of MADIS, we implement a JDBC driver that encapsulates an existing PostgreSQL driver, intercepting the requests performed by the user applications. The requests are transformed, and a new request is elaborated in order to obtain additional information (as metadata). The user perception of the result produced by the requests is also manipulated, in order to hide to the user applications the additionally recovered information. This mechanism allows the plugged replication protocol to be notified about any access performed by the application to the database, including query execution, row recovery, transaction termination requests (i.e. commit/rollback), etc. Therefore, the protocol has a chance to take specific actions during the transaction execution so as to accomplish its tasks.

4.4.1 Connection Establishment

The JDBC standard recommends the use of the `DriverManager` class to obtain connections in order to be used by an application. The implementation of `DriverManager` selects the particular JDBC Driver to be invoked in base to the URL specified by the user application to determine the target database. Thus, the URL "`jdbc:postgresql://myserver.mydomain/mydatabase`" indicates the `DriverManager` to use the PostgreSQL Driver to obtain the connection. The DriverManager performs the selection invoking each registered JDBC Driver in the system. As we want to provide a JDBC interface in our MADIS architecture, the user requests a `MADISConnection` by way of an appropriate

URL (e.g. "jdbc:madis:postgresql://myserver.mydomain/mydatabase") in order to select the MADIS Driver, as well as the underlying JDBC Driver to be used by the middleware to access the underlying database.

More About Detecting Conflicting Transactions

An execution thread is associated per transaction. It is used each time its associated transaction begins any operation. It may become blocked due to the concurrency control policy of the underlying DBMS. Take into account that in MVCC DBMSs the read-only operations cannot be blocked.

Thus, once a database connection is created, a thread is also created and associated to it. Each time the current transaction in a given connection initiates an updating operation, its associated thread is temporarily suspended, with a given timeout. If such an updating operation terminates before that timeout has expired, the thread is awakened and nothing else needs to be done. On the other hand, if the timeout is exhausted and the operation has not been concluded, the thread is reactivated and then makes a call to the GET_BLOCKED() procedure. As a result, the replication protocol is able to know if the transaction associated to this thread is actually blocked and which other transaction has caused its stop.

This mechanism can be combined with the transaction priority scheme defined for the BRP and ERP protocols. Two priority classes are defined, with values 0 and 1. Class 0 is assigned to local transactions that have not started their commit phase. Class 1 is for local transactions that have started their commit phase and for those transactions associated to delivered write sets that have to be locally applied. Once a conflict is detected, if the transactions have different priorities, then the one with the lowest priority will be aborted. Otherwise, i.e., when both transactions have the same priority, the *higher_priority* function is applied for transactions at level 1, but nothing is done with transactions belonging to level 0. Similar approaches may be followed in other replication protocols that belong to the update everywhere with constant interaction class [WPS+00].

4.4.2 Common Query Execution

Query executions are also intercepted by MADIS encapsulating the Statement class. As response of user invocation to createStatement() or prepareStatement() the MADIS Connection generates Statements that manage user queries execution. When the user application requests a query execution, the request is sent to the CM class, which calls the processStatement() operation of the plugged replication protocol.

Once this is done, the replication protocol may modify the statement, adding to it the patches needed to retrieve some metadata, or collect additional information into the transaction report. However, this statement modification is only needed by a few replication protocols, which also have the opportunity to retrieve these metadata using additional sentences (on the "report-tables") once the original query has been

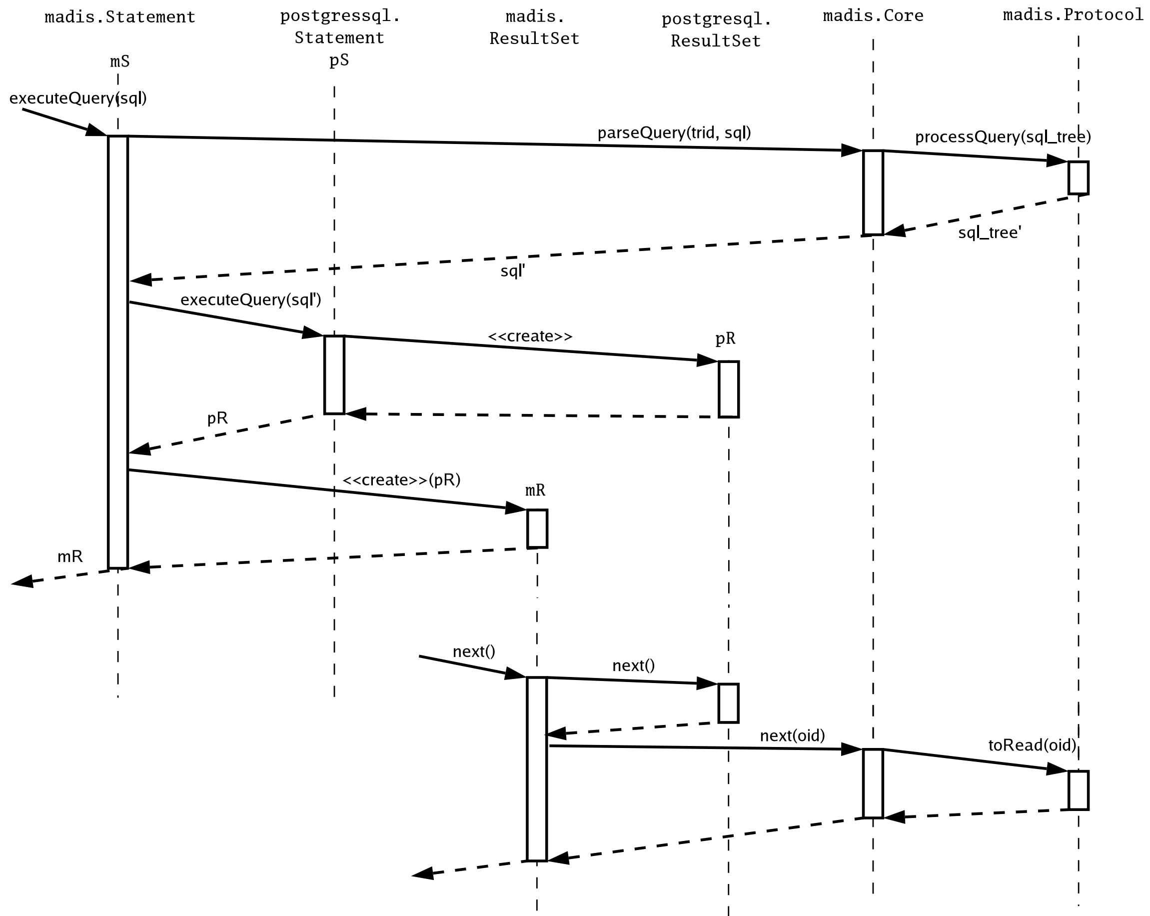


Figure 4.2: Query execution

completed. Optimistic replication protocols do not need such metadata (like current object versions, or the latest update timestamps for each accessed object) until the transaction has requested its commit operation. So, they do not need these statement modifications on each query. The process for queries is depicted in Figure 4.2 whilst Figure 4.3 describes the update control flow. It is important to note that the `ResultSet` should be also encapsulated in order to hide such included metadata.

We recommend to access the metadata using a separate query. Otherwise, the following additional steps are needed:

1. The resulting SQL statement is executed, performing a common invocation to the encapsulated JDBC Statement instance, and a `ResultSet` is obtained as a response. The obtained `ResultSet` is also encapsulated by MADIS, returning to the user application an instance of a MADIS `ResultSet`. It contains the `ResultSet` returned by the JDBC Statement.

2. When the application tries to obtain a new record from the `ResultSet`, MADIS intercepts the request. It notifies about the new obtained object to the replication protocol. Consequently, in order to keep the required guarantees, the protocol may modify the database, the state of the MADIS

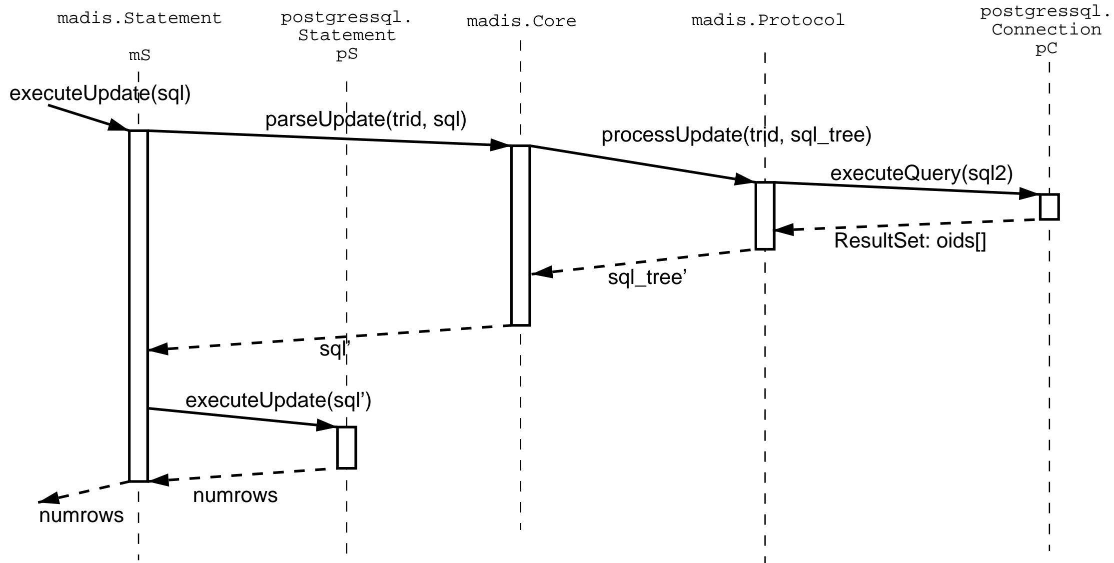


Figure 4.3: Update Execution

`ResultSet`, or even abort the current transaction. In addition, the MADIS `ResultSet` tasks also hide the metadata (included in the query) when the application requests the different fields of the current row.

4.4.3 Commit/Rollback Requests

The termination of a transaction is also requested by the user application. Either when the application requests a *commit* or when a *rollback* is invoked, MADIS must intercept the invocation, and take additional actions.

When the user application requests a commit operation (see Figure 4.4), the MADIS `Connection` redirects the request to the CM instance. Then, the plugged protocol is notified, having then the chance to perform any action involving other nodes, access to the local database, etc.

If the protocol concludes this activity with a positive result, then the transaction is suitable to commit in the local database, and the CM responds affirmatively to the MADIS `Connection` request. Finally, the MADIS `Connection` completes locally the commit, and returns the completion to the user application after the notification to the CM using the `doneCommit()` operation defined in its interface. On the other hand, a negative result obtained from the protocol activity will be notified directly to the application, after the abortion of the local transaction.

Take into account that the `doneCommit()` method is also able to notify a unilateral abort, generated by the underlying database, and that this may allow that the plugged protocols were able to manage such unilateral aborts, too. This is the case of the BRP protocol described in Chapter 3. Finally, `rollback()` requests received from the user application must be also intercepted, redirected to the CM, and notified to

the plugged protocol.

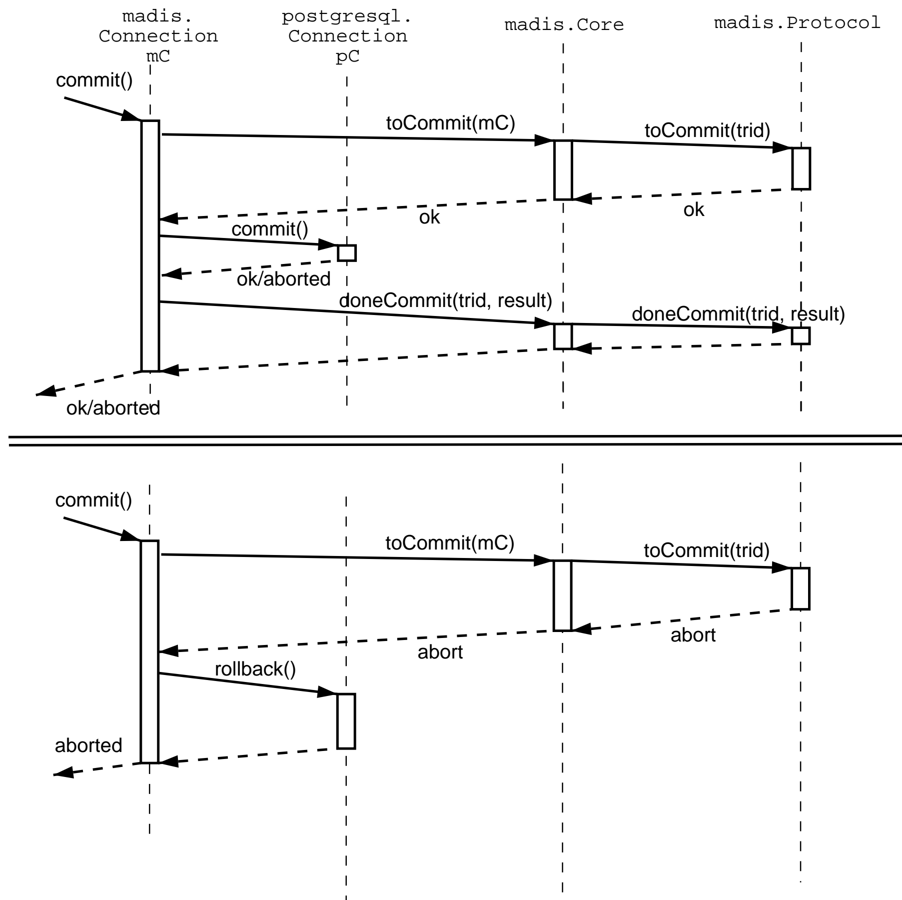


Figure 4.4: Commit suceeded vs aborted

4.5 Protocol Interface

The interaction between our consistency manager and the plugged replication protocol is ruled by an interface with operations to complete the following tasks:

- **Protocol registration**. The protocol has to be plugged into the CM using a registration method. In this registration procedure it has to specify with a parameter the set of events it is interested in. Some of these events depend on the information that has been put into the `MADIS_TR_REPORT` table that was described in Section 4.3. The available events are:

  1. `RECOVERED`: Some objects have been recovered in a `ResultSet`. The protocol will receive an extended `ResultSet` that also contains the OIDs of the objects being recovered, and may use this information for building the transaction read set, if needed.

2. UPDATED: This event is similar to the previous one, but reports the objects that have been updated.
3. UPDATE_PRE: The protocol will be notified when the current transaction is going to initiate an updating operation on the database. Thus, the protocol may modify the update sentence at will, if needed.
4. UPDATE_POST: The protocol will be notified after an update sentence has been executed. Thus, it may read the current transaction report for obtaining the set of updated objects. This is an alternative way of doing the same as in the event number 2 described above.
5. QUERY_PRE: The protocol will be notified before a SELECT statement is initiated in the database. It may modify the query, if needed.
6. QUERY_POST: The protocol will be notified once a query has been completed. It may access then the transaction report, if needed.
7. ACCESSED: The protocol will get all the objects accessed by the latest SQL statement, instead of the objects being recovered in its associated ResultSet. This set is normally different from the set of recovered objects and usually the latter is a subset of the former. Some protocols may need the set of objects accessed, instead of the recovered ones, as the strictly serializable ones.
8. TREE: The protocol requests that the CM builds a parsing tree for each sentence being executed. Later, the protocol may ask for such a tree, modifying it when needed.

- **Event requesting**. There are also a set of explicit operations that the protocol may use for requesting those events that were not set at protocol registration time.
- **Event cancelation**. A set of operations for eliminating the notification of a given event to the currently plugged-in protocol.
- **Access to transaction write set and metadata**. A set of operations that allow the full or partial recovery of the current write set or metadata for a given transaction. Most of the protocols will need the transaction write set only at commit time in its master node, for its propagation to the rest of replicas, but others may need such data before and these operations allow this earlier recovery, too.

This interface is general enough to implement most of the replication protocols currently developed for databases.

4.5.1 Connection Establishment

Figure 4.5 shows a UML diagram sequence for a connection establishment. The sequence starts with a request to the DriverManager, and the selection of the MADIS JDBC Driver. Then, the MADIS

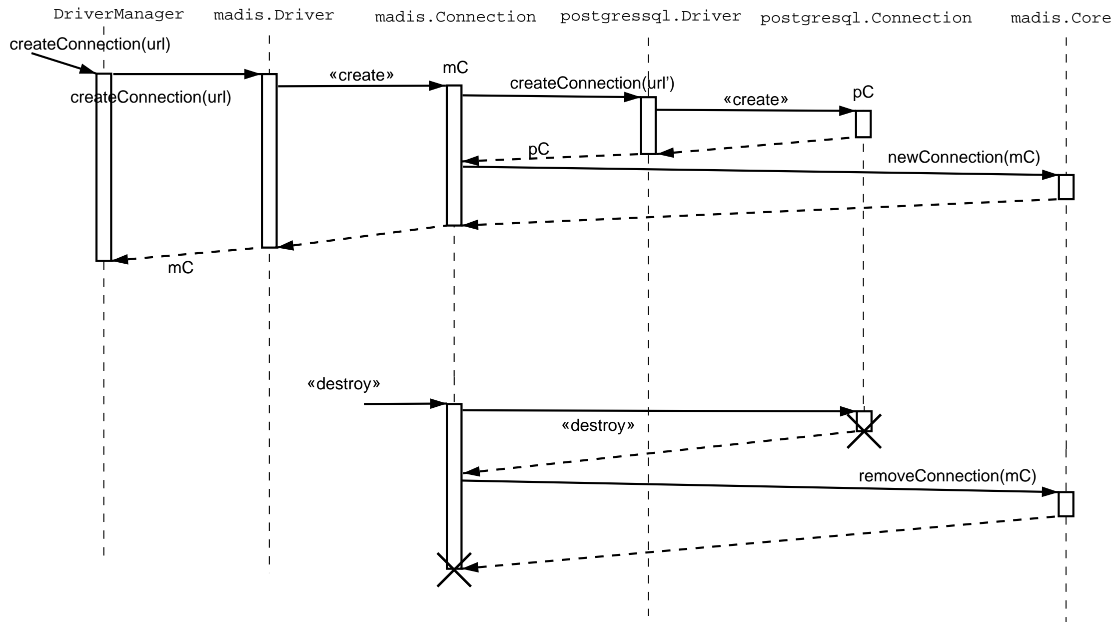


Figure 4.5: Connection Establishment

Driver invokes the MADIS Connection to be built, indicating the underlying PostgreSQL connection URL to be used. The constructor of the MADIS Connection builds a PostgreSQL Connection, and includes it as an attribute. Finally, the MADIS Driver returns the new MADIS Connection. During the construction of the MADIS Connection, the middleware is notified. Then, specific actions can be done by the middleware. For instance, the replication protocol is notified, in order to annotate a new transactional context in the system.

4.6 Experimental Results

The important question to be discussed in this Section is the cost to be paid by the system from drifting to the DBMS the generation and maintenance of the information needed by a generic replication protocol to accomplish the tasks of consistency maintenance, concurrency control and update propagation. This question, for our architecture, corresponds with the degree of performance degradation of the underlying database manager due to the overload introduced by the schema modification (i.e. triggers, procedures, added tables, etc) in the database.

4.6.1 Overhead Description

In spatial terms, the overhead introduced by the schema modification is easy to be determined. Considering the trigger and procedure definitions as irrelevant, the main overload in space is produced by each MADIS_META_T_j table. These tables contain at least two identifiers (local and global object identifier) and

the rest of fields are used by each one of the pluggable protocols. We consider that most of protocols can be implemented with the support of a transaction identifier, a timestamp, and a sequential version number. Finally, the transaction report maintains the information regarding to the executing transactions just during the lifetime of such transactions. Thus, in global terms, this does not constitute a spatial overhead itself.

With regard to computational overhead, our architecture introduces a number of additional SQL sentences and calculations for each access to the database.

This overhead can be classified into four main categories:

- INSERT. The overhead is mainly caused by the insertion of a row into the MADIS_TR_REPORT table for registering such insertion. An additional row is also inserted in the MADIS_META_T_j. Thus, for each row inserted in the original schema, two additional rows are inserted by the schema extension.
- UPDATE. When updating a row of the original schema, there will be also inserted an additional row in the MADIS_TR_REPORT table. However, in this case there will not be needed to insert into the MADIS_META_T_j table any row, but just an update is needed.
- DELETE. In this case, an additional row must be inserted in the MADIS_TR_REPORT table to register the deletion, and the deletion of the corresponding row in MADIS_META_T_j should be also deleted (although in a deferred mode).
- SELECT. When selecting a row from the original schema, there is no need to alter the MADIS_META_T_j table at all. In addition, depending on the particular replication protocol plugged in the system. It can also be avoided any insertion in the MADIS_TR_REPORT table, provided that replication protocols just based on the write set will not need records about the objects read by a transaction.

Summarizing, INSERT, UPDATE and DELETE operations need additional insertions on the MADIS_TR_REPORT table, and other operations with the corresponding MADIS_META_T_j table. In contrast the SELECT overhead varies depending on the plugged protocol. The read set collection may be performed in most of the cases by the middleware, just including the LOCAL_T_j_OID in the SQL sentences executed in the database. Thus, this inexpensive OID inclusion is often the overhead introduced in the SELECT statements. In this Section, we discuss the overhead introduced in **I**NSERT, **U**PDATE and **D**ELETE operations, due to the relevance of the overhead in these operations. We are using a dummy replication protocol, in order to calculate just the overhead introduced by the architecture.

4.6.2 Overhead in the MADIS Architecture

The experiments consisted of the execution of a Java program, performing database accesses via JDBC. The schema used by the program contains four tables and are shown in Figure 4.6. A program execution starts with the database connection, and schema creation. Afterwards, a number of "training" transactions are executed, ensuring that any Java required class is loaded. Then, three measurements are performed. Each

- **CUSTOMER**:(**ID**: INTEGER, NAME: VARCHAR(30), ADDR: VARCHAR(30))
  - **Primary Key**: ID
- **SUPPLIER**:(**ID**: INTEGER, NAME: VARCHAR(30), ADDR: VARCHAR(30))
  - **Primary Key**: ID
- **ARTICLE**:(**ID**: INTEGER, DESCR: VARCHAR(30), PRICE: DOUBLE, ID_SUP:VARCHAR(30))
  - **Primary Key**: ID
  - **Foreign Key**:ID_SUP → SUPPLIER
- **ORDER**:(**ID**: INTEGER, ID_CUS: INTEGER, ID_ART: INTEGER, QUANT: INTEGER)
  - **Primary Key**: ID
  - **Foreign Keys**: ID_CUS → CUSTOMER ID_ART → ARTICLE

Figure 4.6: Database tables description of the experiment

measurement calculates the time taken by *numtr* sequential transactions (performing a number of **I**NSERT, **U**PDATE or **D**ELETE operations depending on the required measurement).

- *Transactions for the first measurement*: they consist of *numrows* insertions of rows into CUSTOMER, SUPPLIER, and ARTICLE tables (referencing the new SUPPLIER), and ten additional insertions into the ORDER table, referencing the rows inserted in this transaction.
- *Transactions for the second measurement*: they consist of *numrows* updates of such rows introduced by the previous transactions, using four different SQL sentences.
- *Transactions for the third measurement*: they consist of *numrows* deletions of such rows introduced by the previous transactions, using four different SQL sentences.

For each measurement, the experiment provides three values: the total cost of the *numtr* transactions of type **I**, **U** and **D** respectively, each one acting with *numrows* rows per table. The experiments were performed in an Intel(R) Pentium(R) IV, at 2.80 GHz, 1GB of RAM, of a Fedora Core 2, Linux 2.4.22 kernel. The DBMS was PostgreSQL 7.4.1.

We observed that deletions are the most costly operations in our core implementation. For a more accurate description of the overhead we calculated the time cost per transaction (Figures 4.7 and 4.8). The results stabilized with a few number of transactions, which indicates that the system does not suffer appreciable performance degradation along the time. In addition, it is shown in Figure 4.7 that the overhead per transaction is always lower than 80 ms in our experiments. Besides, Figure 4.8 shows that the sensitivity for *numrows* is inappreciable (the system scales well in relation to managed rows) for any of the transaction types (**I**, **U**, and **D** respectively).

We concluded that our implementation of the MADIS database core introduces bounded overheads for Insertion and Update operations. However, Delete operations cause the schema modification to produce a

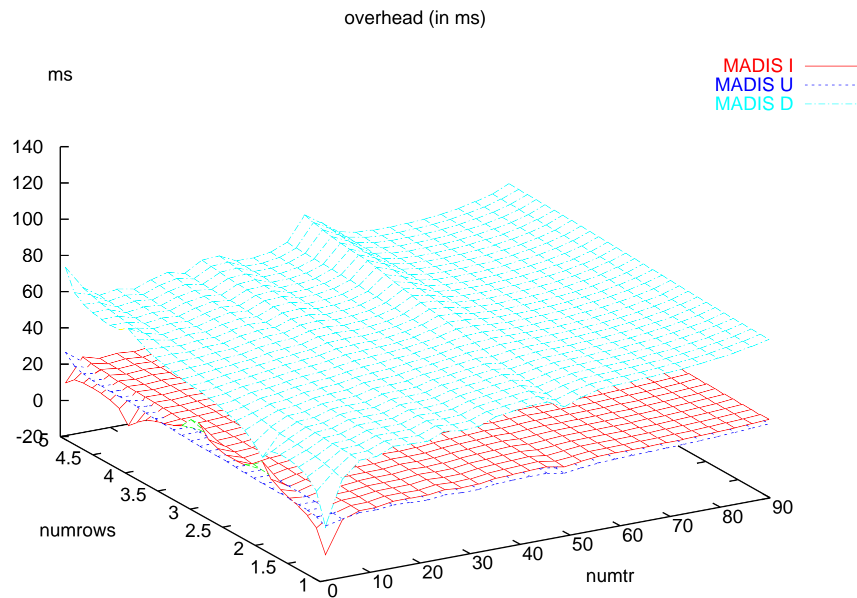


Figure 4.7: MADIS absolute overhead (in ms)

dangerous, although logarithmic degradation of the performance (600% for 6000 rows deleted).

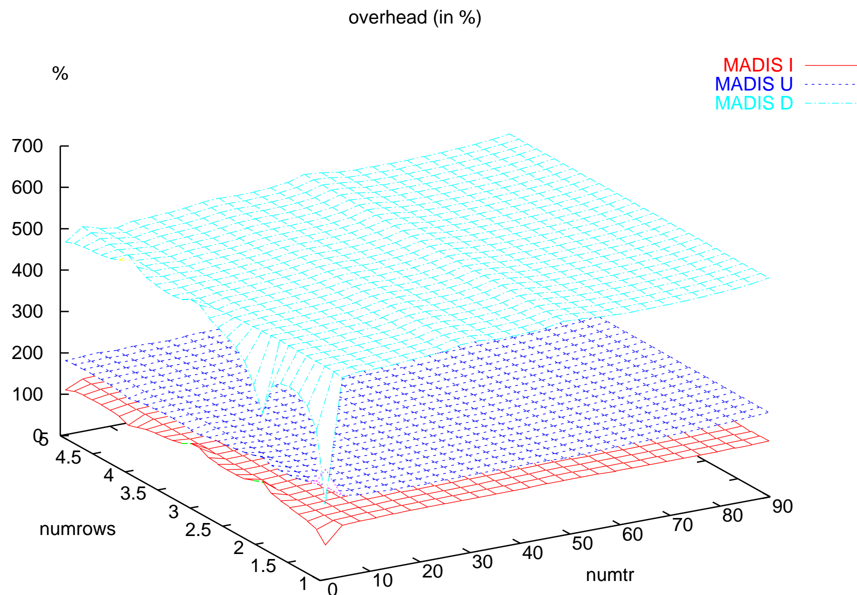


Figure 4.8: Relative MADIS/JDBC overhead (in %)

4.6.3 Comparison of Overheads with Other Architectures

In GlobData [RMA+02, IMDBA03, AGME05], a middleware was developed to be used as a research tool in the field of replication protocols. In fact, several protocols were designed, developed, and implemented using this middleware. However, the architecture used in Globdata (COPLA) did not be conceived to provide low overheads in order to provide the required metadata to the plugged protocols. We include a comparison with COPLA. In the same conditions as the ones depicted in the previous Subsections, we executed an equivalent test using COPLA. The conclusions, depicted in Figure 4.9, were that COPLA has a poor scalability for Update and Delete operations (50 and 200 times more costly than the standard schema).

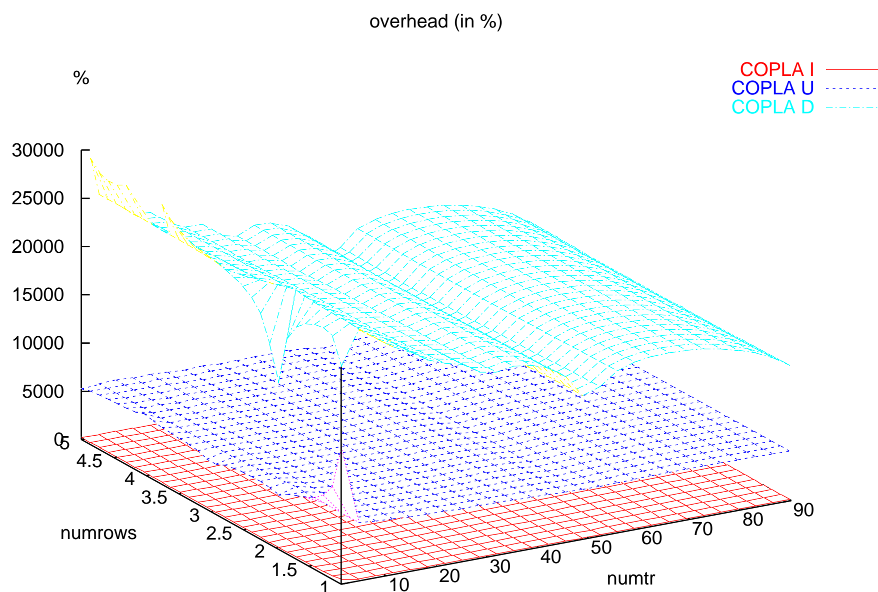


Figure 4.9: Relative COPLA/JDBC overhead

Finally, the MADIS architecture was compared with RJDBC [EPMEIBBA04] as a lower bound of the achievable results. In RJDBC, there is no metadata maintained in the system. In contrast, all the requests to the database are just broadcast to any node in the system. When there is a unique node (as in our experiments), the system introduces a minimal overhead, consisting in the management of the requests. The experiments show (Figure 4.10) that the system overhead remains stable proportionally to the number of rows processed. However, it is also shown that the overhead introduced for `I` and `U` operations is comparable to the one introduced by MADIS. Thus, as the RJDBC architecture only allows a unique eager, pessimistic, and linear replication protocol, it will not scale well with regard to the number of connected nodes.

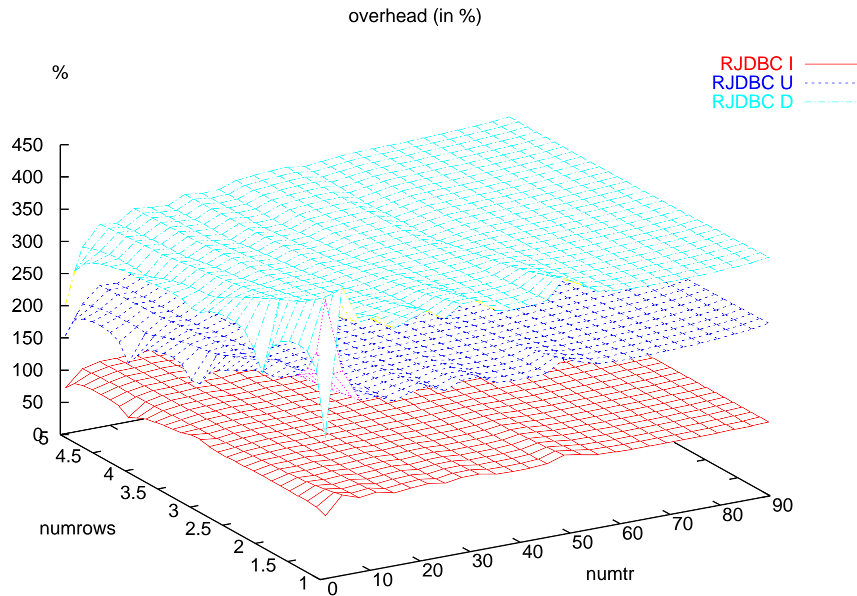


Figure 4.10: Relative RJDBC/JDBC overhead

4.6.4 Experimental Results of the Replication Protocols Implementation in MADIS

The BRP and ERP protocols, introduced in Chapter 3, have been also implemented in MADIS. Right now, we have theoretically shown that: both protocols are correct, and the ERP presents lower response times and abortion rates than the BRP. These assertions are going to be tested by their implementation in MADIS. This set of experiments tries to provide a representative performance analysis of both protocols in MADIS.

Besides, the TORPE protocol has also been implemented in MADIS. As seen before, the BRP and ERP protocols do not rely on strong group communication primitives, but on a deadlock prevention schema based on transaction priorities that prevents from distributed deadlocking. However, total order based replication protocols, like TORPE, manage replica consistency by the total order message delivery featured by GCSs. Therefore, we are very interested in comparing our replication proposals with, so as to see how they behave.

Database Schema and Transaction Types. In all the experiments, the database consists of 30 tables each containing 1000 tuples. Each table contains the same schema: two integers, one being the primary key (TABLE_ID) and the other one denoted as (ATTR). Update transactions consist of a number of operations of the type:

```
UPDATE TABLE_NAME SET ATTR = ATTR + 1 WHERE TABLE_ID = Y;
```

Where **TABLE_NAME** is a randomly chosen table between the 30 tables composing the database and **Y** is another randomly chosen number between 1 and 1000.

Clients. Transactions are submitted by clients that are evenly distributed among the servers. PostgreSQL is very inefficient from the client management point of view, as it creates a new process and sockets for each

new client connection. Hence, we have decided to maintain client connections once they are connected to their master site and submit transactions until the end of the given test.

Workload. The interarrival time between the submission of two consecutive transactions is uniformly distributed. The submission rate (also referred to as workload) varies through the experiments and is determined by the number of clients and the mean interarrival rate for each client. The workload is denoted by the number of transactions submitted per second (TPS). Except for a few experiments, in which resources were completely saturated, the system throughput is equal to the submission rate.

Hardware Configuration. For the experiments, we used a cluster of 8 workstations (Fedora Core 1, Pentium IV 2.8GHz, 1GB main memory, 80GB IDE disk) connected by a full duplex Fast Ethernet network. PostgreSQL 7.4 was used as the underlying DBMS. Finally, Spread 3.17.3 was in charge of the group communication for TORPE. Besides, we have implemented a basic reliable multicast using TCP in order to feature the reliable broadcast communication primitives needed by BRP and ERP, since we have not found an efficient GCS implementation that performs a reliable multicast.

PostgreSQL Configuration. We have modified the `fsync` option. By default this option is enabled and the PostgreSQL server will use the `fsync()` system call in several places to make sure that updates are physically written to disk. This insures that the database will recover to a consistent state after an operating system or hardware crash. However, using `fsync()` results in a performance penalty: when a transaction is committed, PostgreSQL must wait for the operating system to flush the write-ahead log to disk. When `fsync()` is disabled, the operating system is allowed to do its best in buffering, ordering, and delaying writes. This can result in significantly improved performance. However, if the system crashes, the results of the last few committed transactions may be lost in part or whole. In the worst case, unrecoverable data corruption may occur. Hence, violating the ACID properties of transactions [BHG87]. Our sole concern is to compare replication protocols. Regarding this comparison, there is no need to incorporate the `fsync()` option in the DBMS. Since the current implementation of MADIS is not optimized and the overhead included by the physical writes in the DBMS is the same for the three replication protocols, the response time will be higher. Hence, if we get rid of a common factor for all the protocols, then the experiments will run faster without loosing each of the protocol intrinsic characteristics that are placed at the middleware level.

Set up of the Test Runs. Table 4.1 gives an overview of the parameters used to carry out the different proposed experiments. We analyze the interval time between a client transaction submission and the commit reception of the client application. Each experiment shows the update time of a transaction at a single node, referred as `UPDATE_TIME`. All tests were run until 2000 transactions were executed. As it uses a multiversion system, data access time continuously grows since the primary index contains more and more entries for each tuple and all version of the tuple have to be checked in order to find the valid version. Each time we executed a new test suite, we executed the `vacuumdb` command for the given database to perform

Experiments	WL1	WL2	WL3	Conflicts
Database Size	30 tables of 1000 tuples each			
Tuple Size	appr. 100 bytes			
Number of Servers	5	2-8	5	5
Number of Updates Operations	5	10	5-25	5
Number of Clients	1-20	2-8	5	5
Submission Rate in TPS	10-35	10	10	20
Hot Spot Size	0%	0%	0%	varying

Table 4.1: Parameters of experiments

a full vacuuming of it in order to obtain comparable results.

Set up of the Replication Protocols The BRP was fully implemented in MADIS. However, ERP and TORPE were implemented without their associated queues as most of the test benchmarks present no conflict at all. Hence, the conflict analysis was only done with BRP and ERP where abortions may be handled in an easier way even without queue management.

Workload Analysis 1 (WL1): Response Time of the Protocols Varying The System Load and the Number of Clients

In this experiment, we try to see how the three replication protocols are able to cope with increasing workloads and increasing number of users. These tests were carried out in a configuration of 5 servers. Transactions consist of 5 update operations. Workload was increased steadily from 10 to 35 TPS and, for each workload, several tests were executed varying the number of clients from 1 to 20 in the whole system. This means, the more clients exist the less transactions each client submits to achieve a specific workload.

Figures 4.11 - 4.13 show the response time for the BRP, ERP and TORPE protocols in this experiment respectively. Generally, the maximum throughput is limited when working with a small number of clients since a client can only submit one transaction at a time, and hence, the submission rate per client is limited by the response time. For instance, one client would have to submit a transaction each 40 ms to achieve a throughput of 25 TPS. With one client TORPE and ERP response times are below 25 and 28 ms respectively, but BRP ones are close to 50 ms and it is not possible to achieve the desired throughput. In general, the response times obtained were normally shorter than the required interarrival time to achieve a given throughput, except for a few tests where the system was saturated that we considered to be valid since the difference between the submission rate and the achieved throughput was not significant.

BRP presents the worst behavior of the presented protocols. The reason for this is that the master site of a transaction has to wait for the update execution of the slowest site, since remote sites have to execute the received transaction before sending the *ready* message, as the commitment process determines. However, ERP and TORPE do not have to wait so long: in the first one the remote sites sends the *ready* message once

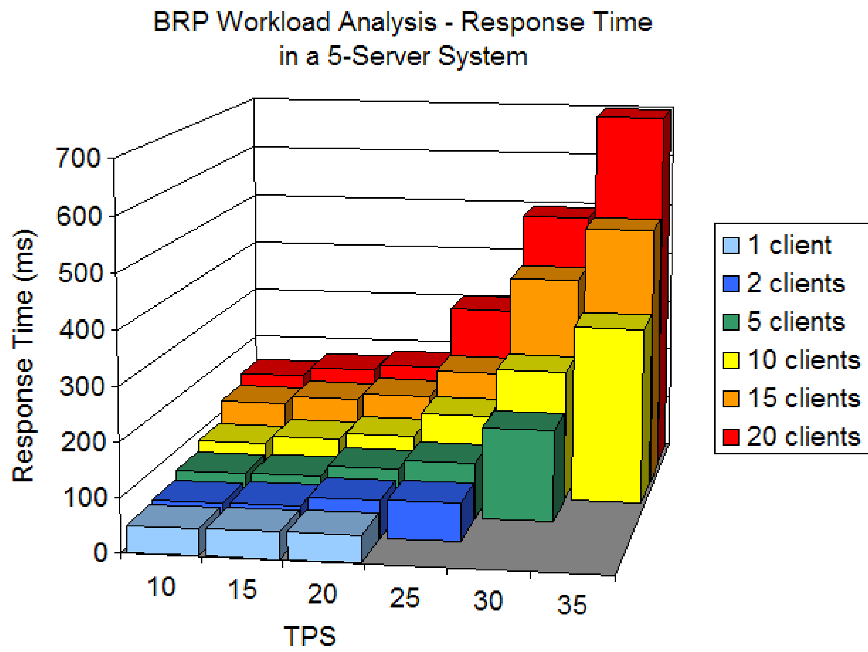


Figure 4.11: BRP response time in a 5-server system varying the submission rate and the number of clients

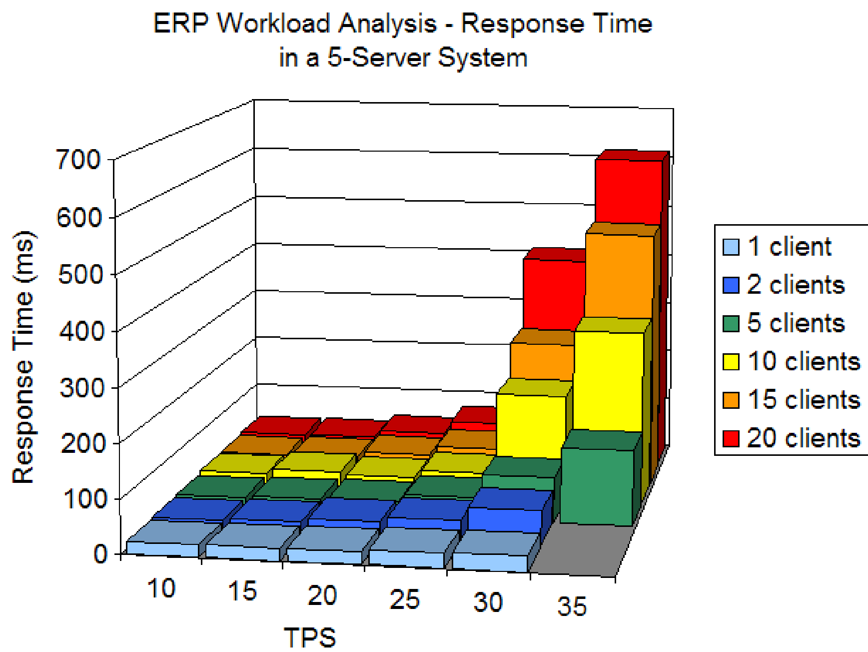


Figure 4.12: ERP response time in a 5-server system varying the submission rate and the number of clients

conflicts and priority rules are checked before executing the updates and in the second one the master site does not wait any response from remote sites, only waits for the delivery of the messages in total order. This fact implies that, in the BRP protocol, a transaction remains longer at its master site, thus increasing the

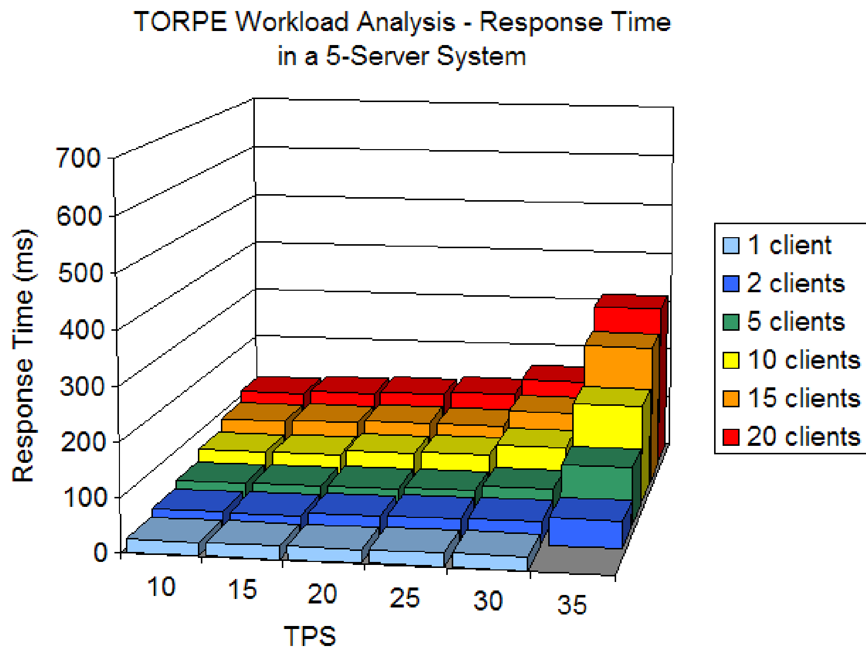


Figure 4.13: TORPE response time in a 5-server system varying the submission rate and the number of clients

administration overhead for the operating system. As a concluding remark, increasing the workload and the multiprogramming level only results in higher response times, due to the higher administration overhead (process switches, communication to/from the client) and contention at specific resources.

ERP is also somehow limited by the CPU, but not up to the level that BRP is. The master site must wait for the *ready* message coming from the remote sites at commitment time, so that, when the resources of the available remote sites become saturated at high workloads and multiprogramming levels, these have no enough time to process all the delivered messages and the response times becomes greater. As it has been pointed out before for the ERP protocol, remote sites do not wait for the update execution before sending the *ready* message, what results in a shorter transaction execution time in the respective transaction master sites. Due to this fact, not only shorter response times are achieved but also the system remains less saturated in general terms, allowing a better performance than the one obtained with BRP. As seen in Figure 4.12, response times keep between 20 and 70 ms for a given number of clients with workloads up to 25 TPS. Beyond this workload, and specially with a high multiprogramming level, response times grow quickly as a consequence of the increasing saturation of the sites, as said before.

In the TORPE protocol, the master site only waits for the local delivery of a total order message to complete the commitment process. This means that TORPE is not so limited by the CPU as BRP and ERP are, as can be seen in Figure 4.13. For that reason, TORPE response times do not increase so much as in the other protocols at high workloads and multiprogramming levels. Working with intermediate loads, response times remain between 20 ms with 1 client and 125 ms with 20 clients. As shown in Figure 4.13, increasing

the system workload for a given number of clients does not involve longer response times. However, the system response times increase as the number of clients grows. Having more clients in the system means more messages to be exchanged between the sites. As a result of this increase in the number of messages, the GCS takes more time to establish the total order in the message delivery to the available sites.

Finally, Figure 4.14 presents a direct comparison between the implemented protocols for a cluster of 5 servers with 5 clients issuing nonconflicting transactions. These transactions contain each 5 update operations. Figure 4.14 highlights what we have discussed in Chapter 3: ERP has a lower response time than BRP as it does not have to wait for applying the changes to the rest of nodes before committing the transaction and TORPE is less dependent on the CPU saturation than ERP and BRP are. Hence, TORPE is able to cope perfectly with workloads of 35 TPS with the lowest response times (106 ms). On the other hand, with a workload of 30 TPS, the BRP protocol is completely saturated (165 ms) and the system throughput can hardly keep up with the submission rate. ERP behaves in a similar way, but with around 50 ms lower response times, what allows this protocol to saturate (162 ms) at a higher workload(35 TPS).

To summarize, BRP is the worst solution for database replication, the 2PC rule highly penalized its behavior in all possible scenarios. ERP presents a better approach to achieve database replication, thanks to its modification of the 2PC rule. The experiments corroborate what their formal analysis, introduced in Chapter 3, have shown. The last replication protocol is the best option in general for achieving database replication. It does not present problems of scalability nor increased workloads. However, the overhead introduced by the GCS turns the ERP into a valuable option in low loaded scenarios.

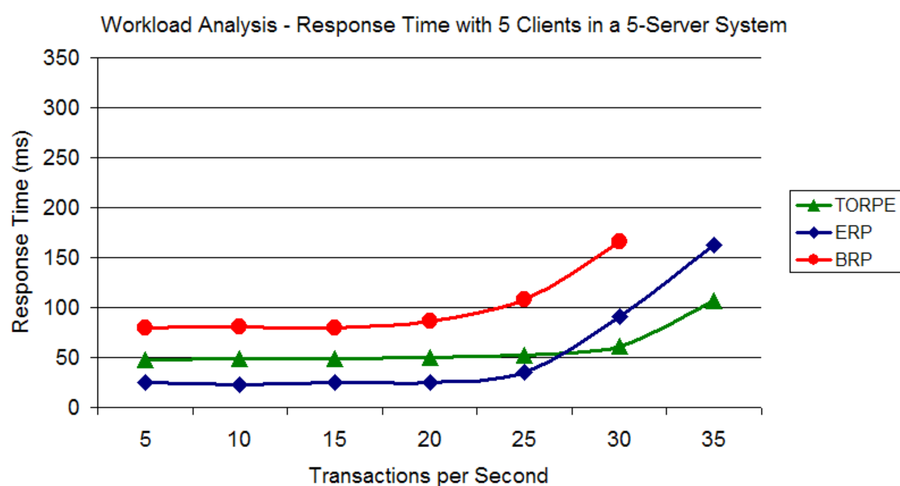


Figure 4.14: Response time of the replication protocols in a 5-server system varying the submission rate

Workload Analysis (WL2): Response Time Varying The Number of Servers

This second experiment tries to test the scalability of the implemented protocols. The analysis shown in [GHOS96] demonstrates that conventional replication protocols do not cope very well with increasing the system size. We have evaluated their performance varying the number of servers from 2 to 8 and performing the same test suites in each configuration. One client is allocated in each server and the load introduced into the system remains constant to 10 TPS in all the performed tests.

As shown in Figure 4.15, the response time increases as the system size grows. As expected, ERP shows a better response time than BRP, they basically differ in the time spent before sending the *ready* message and the communication overhead due to the presence of new sites. The gap between the BRP and ERP protocol remains around 50 ms which is twice the UPDATE_TIME of the transaction at a single node. On the other hand, TORPE behaves better than BRP but worse than ERP, since total order message delivery time increases also as the number of sites grows.

Figure 4.15: Response time of the replication protocols for a submission rate of 10 TPS varying the number of servers

Workload Analysis (WL3): Response Time Varying The Transaction Length

The aim of this experiment is showing how the transaction length affects the response time of the system. We expect that as we add more operations to the transaction its response time will be incremented in the implemented protocols. Figure 4.16 shows the results obtained in a system composed by 5 servers with 5 clients submitting a workload of 10 TPS with no conflicts varying the number of operations executed by transactions from 5 to 25 updates.

As we have previously mentioned, as PostgreSQL is a MVCC-based DBMS, the database size grows continuously. As the number of transactions increases, and also the number of operations associated to

each one, there are more versions of the data items that have to be checked. This update time presents a non-linear increment in a centralized system, as it is shown in Figure 4.16 with the line `LOCAL`, as we augment the number of operations in the transaction. Hence, the non-linear increase in the response time of all protocols is also reflected in Figure 4.16. We can also observe that BRP again shows a worse behavior than ERP and TORPE, since it has to wait for the *ready* message coming from the slowest node in the system before committing the transaction. Thus, as the number of operations is increased, the difference in response time between BRP and the other protocols becomes greater, since the execution time of remote transactions is longer. Besides, the longer response times when increasing the transaction length, the greater saturation of the site resources since transactions take more time to finish and consequently to release the reserved resources previously taken. For this reason, with longer transactions BRP response times grows much faster than ERP or TORPE does.

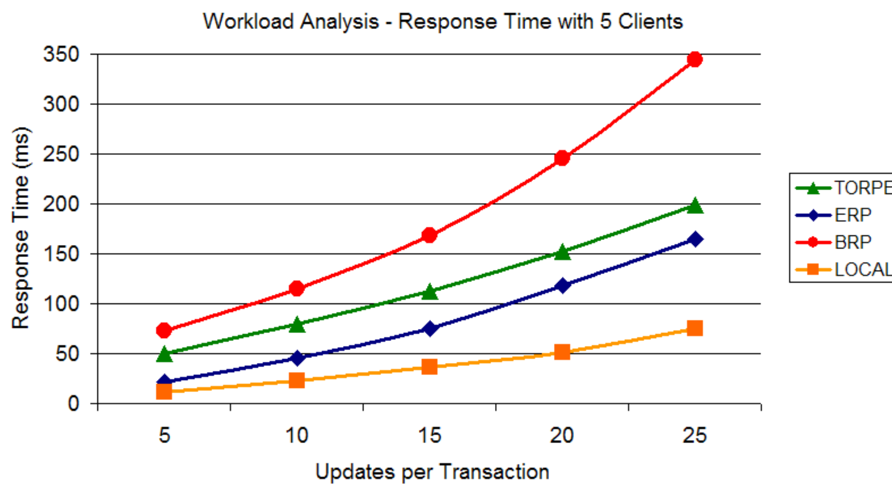


Figure 4.16: Response time of the replication protocols in a 5-server system for a submission rate of 10 TPS varying the transaction length

Conflict Rate for the BRP and ERP protocols

Usually, database access is not equally distributed among all data but there exist hot-spot areas that are accessed by most transactions leading to high conflict rates. In the previous experiments, conflict rates were rather small because we modeled a uniform data access distribution, so that the probability of conflict between two transactions was low. Therefore, considering that databases contain usually hot-spot areas that are accessed by most transactions, a hot-spot area of 1000 tuples was defined in the database and then we run a suite of test varying the access distribution pattern to that area.

The access distribution is determined by the probability of operations accessing to the hot-spot area (from 50% to 90%) and the percentage of tuples that will be accessed from the defined hot-spot area (from

50% to 5%). Table 4.2 depicts the tested configurations. The tests were performed with a 5 server configuration and 5 concurrent clients submitting 20 TPS to the system. Each transaction consists of 5 update operations.

Data Configuration	I	II	III	IV	V	VI
Hot-spot access frequency in %	50	60	80	80	90	90
Hot-spot data size in %	50	20	20	10	10	5
Hot-spot data size in total # tuples	-	6000	6000	3000	3000	1500

Table 4.2: Conflict Rate: Data access distribution to achieve different conflict rates

As it was expected, the higher the access frequency and the smaller the data area, the higher is the conflict rate, since there is a higher probability that two o more transactions tries to access to a same tuple. Figure 4.17 shows that ERP has a lower abort rate than BRP when system is stressed. The main reason for this behavior is again the higher response times of the BRP protocol. Transactions remain longer in the master site waiting for the reply of the available sites, which must execute the updates of the transaction before sending the *ready* message to the master site. Due to this fact, it is more probable that other transactions can conflict with the existing one. Hence, when conflict probability increases, ERP behaves better than BRP. It is due to the fact that for ERP transactions executed in its master site take shorter time to finish, thus reducing the probability of conflict between transactions.

Taking into account that the implementation of ERP used in these experiments does not include the queue mechanism explained in Chapter 3, the obtained abort rates for the ERP protocol could be reduced slightly in the full implementation. In that case, only the master site of a transaction would be in charge of its abortion and therefore unnecessary aborts would be avoided in certain situations.

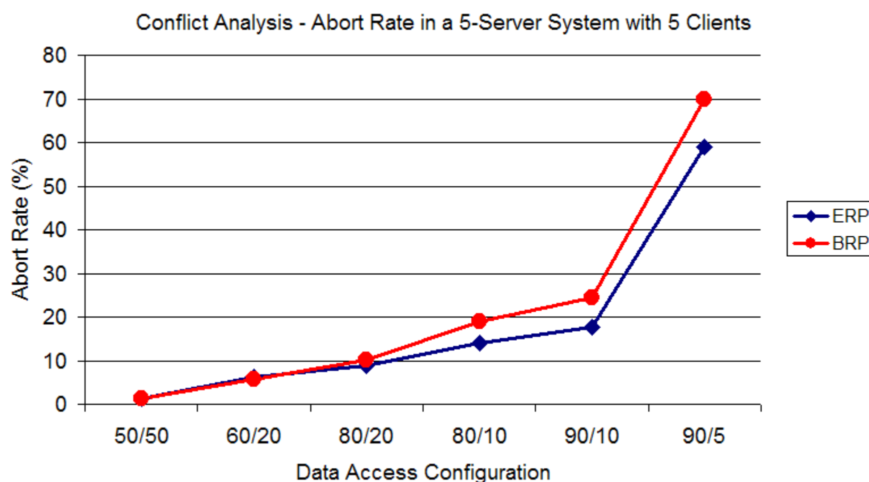


Figure 4.17: Conflict Rate: Abort rate for BRP and ERP in a 5-server system for a submission rate of 20 TPS

4.7 Discussion

In this Chapter a new MADIS is a *middleware-based* database replication architecture. MADIS implementation allows user applications to access in a standard way a replicated database without needing to include changes in their code. This is a very important matter that we learnt from our experience in COPLA [IMDBA03, MEIBG+01] where users see a full object oriented architecture even though objects were stored in RDBMSs. As most of enterprise data is stored following the relational model, it was hard to change their application to a proprietary object oriented definition language. MADIS avoids this difficulty by allowing user applications to be unmodified as it provides a standard JDBC interface.

MADIS is designed to give support to a wide range of replication protocols, using a minimal database schema extension with triggers, functions and rules in order to collect the metadata needed by such protocols; hence, it is a control-table based replication architecture according to [Pee02]. The CM makes use of the automatically collected information in the database, notifying such accesses to a plugged replication protocol. Therefore, replication protocols can be developed and implemented much more efficiently than in comparable middleware packages, where the meta data management for maintaining the consistency of replicated data either is opaquely intertwined in the protocol's code or, worse, is hidden in the application code. Moreover, the structure of the protocol becomes much more elegant and concise when the meta data management is largely delegated to the underlying DBMS. These conceptual advantages have been verified by experimental measurements. It is possible to include a wide range of protocols in the system, each one providing different guarantees and behaviors to the user applications. The implementation of the CM is simple enough to be ported from one platform to another with a minimal cost.

Finally, an implementation of the replication protocols formalized in Chapter 3 has been done in MADIS. The experiments performed verify what it has been formally proved for the BRP and ERP which is the 2PC penalization for BRP and the decrease of the abortion rate in ERP. Their comparison against TORPE shows that total order replication protocols are the best alternative for database replication. However, the overhead introduced by the GCS in low workload environments makes ERP a better solution under that circumstances.

4.7.1 Comparison with Related Works

There are other approaches to develop a middleware architecture for replicated databases. The first architecture we are going to consider is COPLA, it is a full object oriented replicated architecture that provides object state persistence and it is introduced in [IMDBA03]. This architecture was our previous experience in the design and implementation of a middleware database replication architecture. This architecture provided several levels of consistency and may be managed by different replication protocols such as [MEIBG+01, RMA+02, AGME05]. Each one of these protocols has different metadata needs and they

cannot be easily switched if the user wants to use a different replication protocol for the same application. Besides, replication protocols have to re-implement several features that are included in a standard DBMS, such as a lock table or a version system. User applications are defined using a proprietary object definition language GODL [AACV03]. This presents some difficulties as existing user applications have to be modified and, furthermore, as the underlying DBMS is a relational one an object-relational mapping has to be done. Client applications access the system by way of transactions using two different levels of consistency: *plain*, no consistency at all; and, *checkout*, serializable transaction isolation level. Objects may be fetched in the context of a transaction by way of a proprietary object query language GOQL [AACV03]. In MADIS we provide a standard JDBC interface where applications do not need to be modified and we have isolated the replica control from the concurrency control provided by the underlying DBMS. All, of this reduces the overhead introduced in the system compared with COPLA, as it has been previously shown.

Recent middleware-based database replication architectures provide a JDBC interface to interact with user applications. The first one we are going to introduce is Clustered JDBC (C-JDBC) [CMZ04] which is an open-source middleware solution for database clustering on a shared-nothing architecture built with commodity hardware. C-JDBC uses the ROWAA approach, the routing of queries to different backends can be done following several number of strategies or by a user-defined policy. It supports query caching and fault tolerance. This architecture offers a C-JDBC driver to user application that replaces the specific JDBC driver. The C-JBDC controller is a Java program that acts as a proxy between the C-JDBC driver and the database backend. The scheduling of transaction is as follows: the beginning, commitment or abortion of a transaction are sent to all backends. Reads are sent to a single backend. Updates are sent to all backends where the affected tables are located. All operations are synchronous with respect to the client. The C-JDBC waits until it has received responses from all backends involved in the operation before it returns a response to the client. It is important to note that at any given time only a single update, commit or abort is in progress on a particular database schema. Multiple reads from different transactions can be going on at the same time. Updates, commits and aborts are sent to all backends in the same order. Dealing with fault-tolerance issues they propose a recovery protocol that uses checkpoints and database logs. A very similar approach to the previous one is Replicated JDBC (RJDBC) [EPMEIBBA04] which stands for a simple, easy to install middleware, placed between the application and the DBMS, intercepting all database operations and forwarding them among all replicas of the system using a total order multicast. As in the previous case a RJDBC driver is provided to the user application. This RJDBC driver wraps all JDBC invocations and forward them to the RJDBC core. Not all operations are required to be multicast but only those affecting the database state. When an RJDBC node receives a new operation from the communications protocol, it is enqueued for subsequent processing that along with the total order delivery maintains data consistency. Both approaches share a lot of similarities, MADIS overcome both since it provides a JDBC interface where different replication and recovery protocols may be implemented in order to maintain consistency. Hence,

user applications will not have to wait for the response coming from all nodes each time they issue an update operation inside a transaction. However, the overhead introduced by these architectures is less than in MADIS.

In Postgres-R and Postgres-R(SI) [Kem00, WK05], a DBMS core is modified to support distribution. This approach strongly depends on the underlying DBMS thus being not portable, and must be reviewed for each new DBMS release. However, its performance is generally better than a middleware architecture. Recently, these systems have been ported to a middleware architecture [LKPMJP05], introducing a minimal support in the DBMS core in order to access its internal redo-logs for obtaining (or applying) the writesets of the currently executing transactions (or those of the remote transactions that have been locally delivered, respectively). This core support also simplifies a lot the work to be done in the middleware, reducing the overall costs needed for such a management, at least when compared to our MADIS approach that uses triggers to this end. As a result, this Postgres-R evolution has better performance than MADIS, but MADIS only uses standard SQL features and is easier to port to other DBMSs.

Progress DataXtend RE (formerly known as PeerDirect) [Pee02] is a commercial tool that proposes a solution suited for enterprise data distribution. PeerDirect uses a technique based on triggers and procedures to replicate a database. It is transparent to the application and DBMS, there is no need to change the application code nor the structure of existing database tables. Furthermore, this solution must be DBMS heterogeneous and not operate differently if we switch from one DBMS vendor to another. It monitors all possible changes of the database state, not only those coming from interactive SQL statements issued by the application. It does not need a lot of the intervention of the database administrator. However, the system only includes one replication protocol well fitted for managing real-time event stream data such as algorithmic trading. Additionally, the new editions of this software have migrated their focus to mobile environments, being able to provide different levels of consistency among the set of replicas of a given piece of data.

Finally, ORION Integrator [Ori05] is another commercial tool that also provides support for data replication and integration. However, its aim is not exactly the same as that of the previously discussed research projects or commercial products. It is similar to them, since it also provides support for replication, but it needs that each replicated item has a *source* replica and one or many *target* replicas, configured as such. On the other hand, this product is able to achieve an easy integration of different DBMSs, i.e., different replicas may use different underlying DBMSs. Its core engine is able to translate the data when it is being propagated in order to store it in the appropriate format for its target DBMS.

Chapter 5

About Failures and the Recovery Process

Up to now we have considered that our system is free of failures, i.e., we have not taken into account any recovery issue for our replication protocol proposal. Dealing with site failures is not an easy task, nevertheless the recovery facility increases system availability and its fault-tolerant guarantees. There exists a tradeoff between recovery issues and system performance, since there are additional tasks to be done throughout the lifetime of a transaction to ensure recoverability. As it will be depicted in this Chapter, it is necessary to strength the delivery guarantees, as well as additional metadata information in the database.

Hereafter, we change the assumption of a failure free model to a *partial amnesia crash* [Cri91] failure model which is needed to perform the recovery of a failed site. This model assumes that after a site fails, some part of its components will maintain its state while other parts will completely loss their associated state. In our case, we consider that the *DB* module maintains its state but the state variables associated with the state transition system considered (BRP or ERP) are completely lost.

The system will try to continue executing as long as it involves a primary partition [Bar04, CKV01]. If we take this into consideration, we will have to give a rough outline of a possible recovery process for this algorithm; the recovery process is based on ideas depicted in [AGME05]. The key idea is the definition of dynamic recovery partitions associated to recovering nodes.

If we consider failures, then GCS must provide a uniform reliable multicast [Bar04, CKV01, HT94]. The GCS will group messages delivered in views. These views have a global unique identifier which allows every site to determine the view when a given site joined the group, since they are fired each time a site joins or crashes. A view serves as a synchronization point to set up updates missed by crashed nodes; as a matter of fact we group crashed nodes and missed updates by views in this recovery proposal.

As a rough outline of our recovery proposal, once a site joins the system after a failure, a recoverer

site is elected among the previously available nodes. This recoverer "*tells*" the joining node the updated objects missed during its failure. This establishes a recovery partition of the database in the recovering and the recoverer nodes. The recoverer node will hold the partition as long as the data transfer of that partition is taking place. Previously alive sites will access objects belonging to the partition, whereas the recovering node will get blocked and the recoverer will only block for update operations. As a final remark, local transactions may start in a recovering node after its associated recovery partitions have been established.

The recovery protocol introduced in this Chapter is based on the ERP protocol. Therefore, we will introduce the specific recovery actions and modifications for the ERP state transition system.

5.1 Replication Protocol Modifications

Our ERP proposal has to be modified in order to support failures. The protocol must be able to wait for the delivery of the *commit* message (even at the transaction master site) before committing the transaction. Otherwise, if a crash occurs during the delivery of the *commit* message to any node (see Figure 3.8) then an inconsistency among nodes will occur. The failed node has committed a transaction the rest of nodes has not, as it is shown in Figure 5.1.

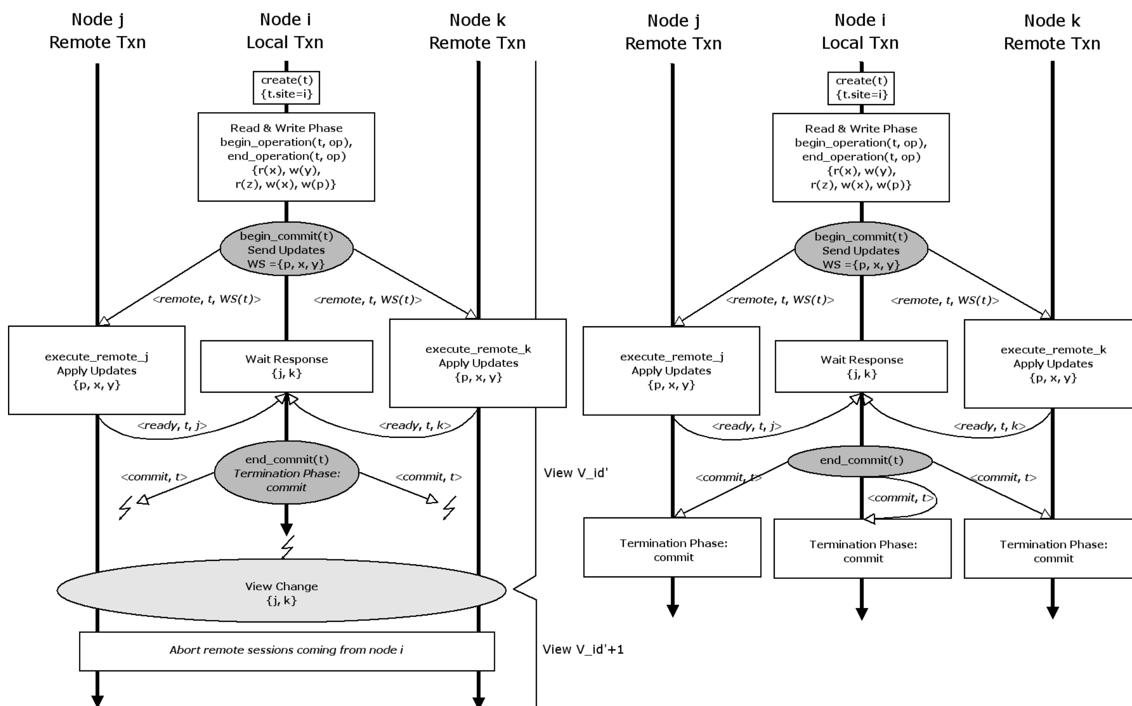


Figure 5.1: In a failure free environment, the transaction master site directly commits. This is not a correct approach in the presence of failures. A site may commit a transaction that the rest of sites do not commit (left). The solution is to use the uniform reliable multicast and the delivery of the message to the master site (right)

Therefore, a new transaction state *committable* has been introduced in the ERP state transition system. This new state reflects that a local transaction, i.e., a transaction executing at its transaction master site, has

received all the *ready* messages coming from the available nodes at a given view but it has not received the *commit* message yet. Up to now (failure free environment), the local transaction t was committed once the $end_commit_i(t)$, being $node(t) = i$, action was executed; in other words, when all available nodes had sent the *ready* message to the transaction master and $participants_i(t) = \emptyset$. As it has been pointed out before, this is not enough to prevent inconsistencies, since the transaction master site may fail before delivering the *commit* message to any other site. This problem is solved by the usage of a uniform reliable multicast facility provided by the GCS [HT94] for all multicast messages sent by the replication protocol.

We have changed the $end_commit_i(t)$ and $receive_commit_i(t, \langle commit, t\rangle)$ actions. These slight modifications are shown in Figure 5.2. The $end_commit_i(t)$ action multicasts, using the $sendURMulticast(m, group)$ primitive from Chapter 2, the $\langle commit, t\rangle$ message to all available nodes (including itself). Besides, it sets $status_i(t)$ = committable. As a direct consequence, the $receive_commit_i(t, \langle commit, t\rangle)$ action will be enabled at i, as well as the rest of available nodes. The precondition has been modified so as to be activated for remote transactions in the *pre_commit* state and local transactions in the *committable* state.

Up to now, we were only considering problems dealing with the GCS. However, we have to consider data needed for the recovery of failed nodes. As far as there are *crashed* nodes, the ERP must store the data items updated by committed transactions. Therefore, each time a transaction commits, it must store the items that have been modified by a given transaction. Afterwards, when a node recovers from a failure, the recovery protocol must determine the updated items missed by the recovering node and transfer their current states to it. All these tasks will be described in detail on the sequel.

5.2 Recovery Protocol Outline

Our goal is to try to harm the fewer number of user transactions during the recovery process. We want to maintain the system availability even though there are nodes being recovered in our system. We have accomplished this by the definition of dynamic recovery partitions on the database. Besides, we also want user transactions to be performed on the recovering node as soon as the site joins the system, even though it has not been recovered yet.

The GCS groups messages delivered in views [CKV01]. The uniform reliable multicast facility [HT94] ensures that if a multicast message is delivered by a site (faulty or not) then it will be delivered to all available sites in that view. All these characteristics permit us to know which objects are updated in the context of an installed view. This information will be stored in the database of available sites as recovery metadata information. Hence, each time a transaction commits, provided that there are failed nodes, all data items identifiers updated by it will be stored in the database. If a failed node rejoins the system, it will be easy to determine the partitions of missed objects, as we know the view where it failed and the one when it rejoined the group. All these aspects are outlined in Figure 5.3.

Signature:
$\{\forall\, i \in \mathcal{N}, t \in \mathcal{T}, m \in \mathcal{M}, op \subseteq \mathcal{OP}$: **create**$_i(t)$, **begin_operation**$_i(t, op)$, **end_operation**$_i(t, op)$, **begin_commit**$_i(t)$, **end_commit**$_i(t)$, **local_abort**$_i(t)$, **receive_remote**$_i(t, m)$, **receive_ready**$_i(t, m)$, **receive_commit**$_i(t, m)$, **receive_abort**$_i(t, m)$, **execute_remote**$_i$, **discard**$_i(t, m)\}$.

States:
$\forall\, i \in \mathcal{N}, \forall\, t \in \mathcal{T}$: $status_i(t) \in$ {idle, start, active, blocked, pre_commit, committable, aborted, committed}, initially $(node(t) = i \Rightarrow status_i(t) =$ start$) \wedge (node(t) \neq i \Rightarrow status_i(t) =$ idle$)$.
$\forall\, i \in \mathcal{N}, \forall\, t \in \mathcal{T}$: $participants_i(t) \subseteq \mathcal{N}$, initially $participants_i(t) = \emptyset$.
$\forall\, i \in \mathcal{N}$: $queue_i \subseteq \{\langle t, WS\rangle : t \in \mathcal{T}, WS \subseteq \{\langle oids, ops\rangle : oids \subseteq \mathcal{OID}, ops \subseteq \mathcal{OP}\}\}$, initially $queue_i = \emptyset$.
$\forall\, i \in \mathcal{N}$: $remove_i$: **boolean**, initially $remove_i =$ false.
$\forall\, i \in \mathcal{N}$: $channel_i \subseteq \{m : m \in \mathcal{M}\}$, initially $channel_i = \emptyset$.
$\forall\, i \in \mathcal{N}$: $\mathcal{V}_i \in \{\langle id, availableNodes\rangle : id \in \mathbb{Z}, availableNodes \subseteq \mathcal{N}\}$, initially $\mathcal{V}_i = \langle 0, \mathcal{N}\rangle$.

Transitions:

create$_i(t)$ // $node(t) = i$ //
$pre \equiv status_i(t) =$ start.
$eff \equiv DB_i.begin(t)$;
  $status_i(t) \leftarrow$ active.

begin_operation$_i(t, op)$ // $node(t) = i$ //
$pre \equiv status_i(t) =$ active.
$eff \equiv DB_i.submit(t, op)$;
  $status_i(t) \leftarrow$ blocked.

end_operation$_i(t, op)$
$pre \equiv status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ run.
$eff \equiv$ **if** $node(t) = i$ **then** $status_i(t) \leftarrow$ active
  else $status_i(t) \leftarrow$ pre_commit.

begin_commit$_i(t)$ // $node(t) = i$ //
$pre \equiv status_i(t) =$ active.
$eff \equiv status_i(t) \leftarrow$ pre_commit;
  $participants_i(t) \leftarrow \mathcal{V}_i.availableNodes \setminus \{i\}$;
  $sendURMulticast(\langle remote, t, DB_i.WS(t)\rangle, participants_i(t))$.

end_commit$_i(t)$ // $node(t) = i$ //
$pre \equiv status_i(t) =$ pre_commit $\wedge\ participants_i(t) = \emptyset$
$eff \equiv status_i(t) \leftarrow$ committable;
  $sendURMulticast(\langle commit, t\rangle, \mathcal{V}_i.availableNodes)$.

receive_ready$_i(t, m)$ // $node(t) = i$ //
$pre \equiv status_i(t) =$ pre_commit $\wedge\ participants_i(t) \neq \emptyset\ \wedge$
  $m = \langle ready, t, source\rangle \in channel_i$.
$eff \equiv receive_i(m)$;
  $participants_i(t) \leftarrow participants_i(t) \setminus \{source\}$.

local_abort$_i(t)$
$pre \equiv status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ abort.
$eff \equiv status_i(t) \leftarrow$ aborted;
  $DB_i.abort(t)$;
  $remove_i \leftarrow$ true.

discard$_i(t, m)$
$pre \equiv status_i(t) \in$ {aborted, committed} $\wedge\ m \in channel_i$.
$eff \equiv receive_i(m)$.

receive_remote$_i(t, m)$ // $t \in \mathcal{T} \wedge node(t) \neq i$ //
$pre \equiv status_i(t) =$ idle $\wedge\ m = \langle remote, t, WS\rangle \in channel_i$.
$eff \equiv receive_i(m)$;
  $insert_with_priority(queue_i, \langle t, WS\rangle)$; $remove_i \leftarrow$ true.

receive_commit$_i(t, m)$
$pre \equiv status_i(t) \in$ {pre_commit, committable} $\wedge$
  $m = \langle commit, t\rangle \in channel_i$.
$eff \equiv receive_i(m)$;
  $DB_i.commit(t)$;
  $status_i(t) \leftarrow$ committed;
  if $\neg empty(queue_i)$ **then** $remove_i \leftarrow$ true.

execute_remote$_i$
$pre \equiv \neg empty(queue_i) \wedge remove_i$.
$eff \equiv aux_queue \leftarrow \emptyset$;
  while $\neg empty(queue_i)$ **do**
    $\langle t, WS\rangle \leftarrow first(queue_i)$;
    $queue_i \leftarrow remainder(queue_i)$;
    $conflictSet \leftarrow DB_i.getConflicts(WS)$;
    if $\exists\, t' \in conflictSet$: $\neg higher_priority(t, t')$ **then**
      $insert_with_priority(aux_queue, \langle t, WS\rangle)$;
    else
      $\forall\, t' \in conflictSet$:
        if $status_i(t') =$ pre_commit $\wedge\ node(t') = i$ **then**
          $sendURMulticast(\langle abort, t'\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$;
        $DB_i.abort(t')$;
        $status_i(t') \leftarrow$ aborted;
      $sendRUnicast(\langle ready, t, i\rangle, node(t))$;
      $DB_i.begin(t)$;
      $DB_i.submit(t, WS.ops)$;
      $status_i(t) \leftarrow$ blocked;
  $queue_i \leftarrow aux_queue$;
  $remove_i \leftarrow$ false.

receive_abort$_i(t, m)$ // $t \in \mathcal{T} \wedge node(t) \neq i$ //
$pre \equiv status_i(t) \notin$ {aborted, committed} $\wedge\ m = \langle abort, t\rangle \in channel_i$.
$eff \equiv receive_i(m)$;
  $status_i(t) \leftarrow$ aborted;
  if $\langle t, \perp\rangle \in queue_i$ **then** $queue_i \leftarrow queue_i \setminus \{\langle t, \perp\rangle\}$
  else $DB_i.abort(t)$;
  if $\neg empty(queue_i)$ **then** $remove_i \leftarrow$ true.

$\diamond$ **function** $higher_priority(t, t') \equiv node(t) = j \neq i \wedge (a \vee b)$
(a) $node(t') = i \wedge status_i(t') \in$ {active, blocked}
(b) $node(t') = i \wedge status_i(t') =$ pre_commit $\wedge$
  $t.priority > t'.priority$

Figure 5.2: State transition system for the ERP so as to avoid data inconsistencies due to a site failure

These recovery partitions are grouped by the site identifier of the *recovering* site and the missed view identifier; this ensures a unique identifier throughout the whole system. Partitions are created at all sites by special transactions called *recovery transactions* that are started at the delivery of the recovery metainfor-

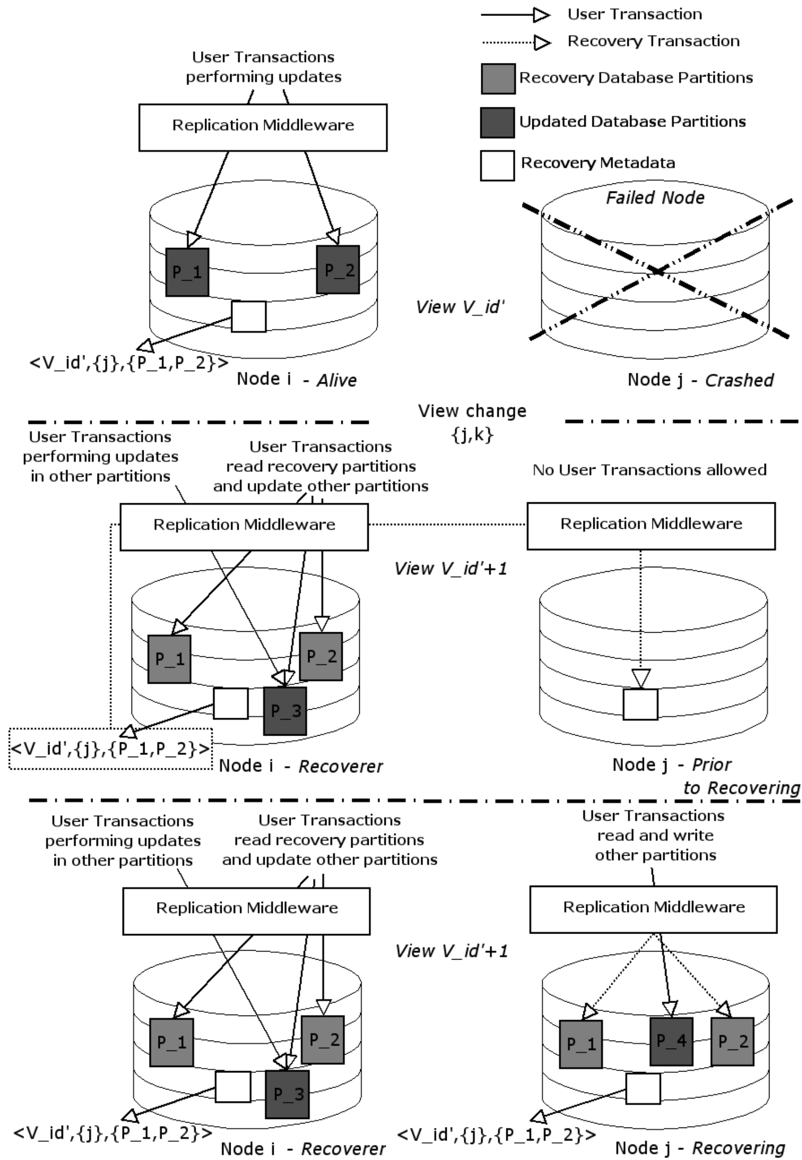


Figure 5.3: The rest of nodes store objects modified while a site is crashed. In this case, node j has failed and partitions (i.e. set of data items) P_1 and P_2 have been modified during view id'. When it rejoins the system, the recovery metadata is transferred to node j and the recovery protocol itself at node j can determine the partitions to be recovered

mation by a unique *recoverer* node. These transactions behave as normal transactions excepting that they have an additional field called *t.view_id*; this field will be equal to $\perp$ for local and remote user transactions. This last field is used to define, along with the *t.node* field, the recovery partition associated to a given recovery transaction.

Once a partition has been set up by its associated recovery transaction at the *recoverer* node, it sends the missed updates of that partition to the respective *recovering* site. Currently update transactions executing on the *recoverer* node will be rolled back if they conflict with the recovery partition as the partition is set up in the *recoverer* node. Afterwards, if a transaction executed at the *recoverer* attempts to modify a data item belonging to the partition then it will get blocked; in other words, read-only access is permitted in these

partitions at the *recoverer* node. Respectively, the recovery protocol will block the access to a partition for user transactions issued in its associated *recovering* node. Therefore, user transactions at *recovering* sites will even commit as long as they do not interfere with their own recovery partitions. Besides, local user transactions on *recovering* sites may start as soon as they reach the *recovering* state. This state is reached at a *recovering* node when all its associated recovery partitions have been set. When the partition is set at the *recoverer* node, it sends the state of the data items contained in the partition. Once they are sent, the partition is released at the *recoverer* site even though the *recovering* node has not finished its recovery process.

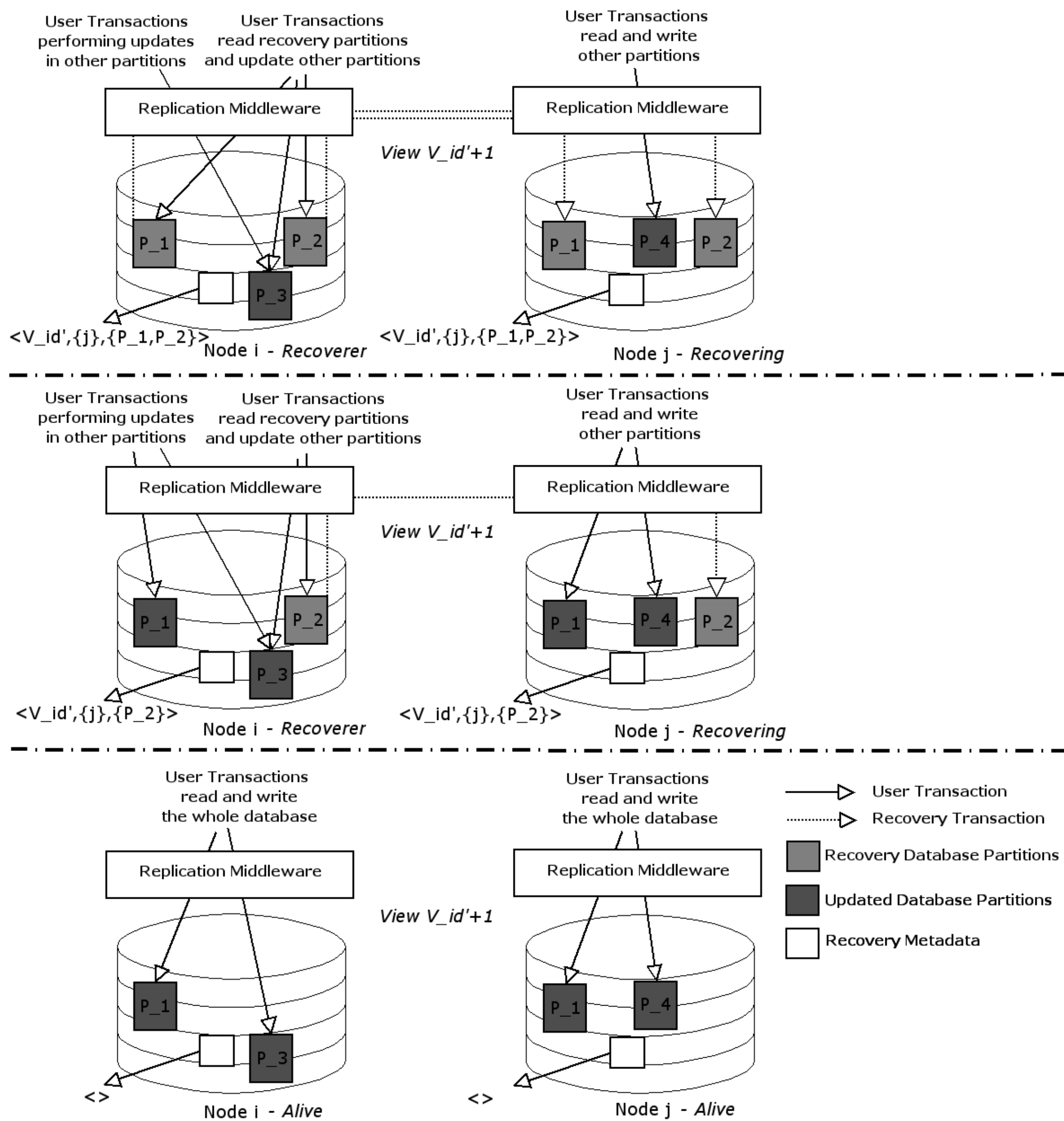


Figure 5.4: Object state transfer between the *recoverer* node *i* and a *recovering* node *j*. User transactions may perform their operations with no restrictions unless they try to access the recovery partitions. Partitions are released once changes are applied

As missed updates are applied in the underlying DBMS, the *recovering* node multicasts a message that notifies the recovery of a given view. The process continues until all missed changes are applied in the *recovering* node. During this recovery process a node may also fail. If it is a *recovering* site, then all its associated recovery partitions will be released on the *recoverer* node. In case of a failure of a *recoverer* site, the new oldest alive node will continue with the recovery process. A rough outline of the object state transfer and the finalization of the recovery process is shown in Figure 5.4.

Signature:
$\{\forall\, i \in \mathcal{N}, t \in \mathcal{T}, m \in \mathcal{M}, op \in \mathcal{OP}, \mathcal{W} \in \{\langle id, availableNodes\rangle : id \in \mathbb{Z}, availableNodes \subseteq \mathcal{N}\}$: $\mathbf{create}_i(t, op)$, $\mathbf{begin_operation}_i(t, op)$, $\mathbf{end_operation}_i(t, op)$, $\mathbf{begin_commit}_i(t)$, $\mathbf{end_commit}_i(t)$, $\mathbf{local_abort}_i(t)$, $\mathbf{receive_remote}_i(t, m)$, $\mathbf{receive_ready}_i(t, m)$, $\mathbf{receive_commit}_i(t, m)$, $\mathbf{receive_abort}_i(t, m)$, $\mathbf{execute_remote}_i$, $\mathbf{execute_remote}_i$, $\mathbf{discard}_i(t, m)$, $\mathbf{join}_i(\mathcal{W})$, $\mathbf{leave}_i(\mathcal{W})$, $\mathbf{receive_recovery_start}_i(m)$, $\mathbf{receive_view_recovered}_i(t, m)$, $\mathbf{receive_missed}_i(t, m)\}$.
States:
$\forall\, i \in \mathcal{N}: \mathcal{V}_i \in \{\langle id, availableNodes\rangle : id \in \mathbb{Z}, availableNodes \subseteq \mathcal{N}\}$, initially $\mathcal{V}_i = \langle 0, \mathcal{N}\rangle$.
$\forall\, i, j \in \mathcal{N}: sites_i(j) \in \{\langle state, age, to_recover, to_schedule\rangle : state \in$ {alive, crashed, pending_metadata, joining, recoverer, recovering}, $age \in \mathbb{Z}$, $to_recover \subset \mathcal{T}, to_schedule \subset \mathcal{T}\}$, initially $sites_i(j) = \langle \text{alive}, 0, \emptyset, \emptyset\rangle$.
$\forall\, i \in \mathcal{N}, \forall\, id \in \mathbb{Z}: missed_i(id) \in \{\langle sites, oids\rangle : sites \subset \mathcal{N}, oids \subset \mathcal{OID}\}$, initially $missed_i(id) = \emptyset$.

Figure 5.5: Signature and states for the ERP recovery protocol

5.3 Recovery Protocol Description

We have to consider different actions each time a view change event is fired by the GCS [CKV01] (due to a site $join_i(\mathcal{W})$ or $leave_i(\mathcal{W})$ action). These view change events are managed by the membership monitor [CKV01], which in our recovery protocol is represented by MM_i. We have defined another state transition system dealing with the ERP protocol recovery issues. It is introduced in Figures 5.5-5.7. Figure 5.5 shows the signature and states of this new state transition system. Specific recovery actions and modifications of the replication protocol actions are respectively given in Figures 5.6 and 5.7.

We must add an extra metadata table on the database in order to store all the information needed to recover nodes after their failure. This table, named `MISSED`, contains three fields: `VIEW_ID`, `NODES` and `OIDS`. `VIEW_ID` contains the view identifier which acts as an index to select the nodes crashed (or not recovered yet). `NODES` contains all sites that are still *crashed* or have not recovered yet the given view. Finally, `OIDS` contains the set of objects updated in that view. As we have depicted in Chapter 2 this metadata table is managed by the proper stored procedures.

Each time a node crashes ($leave_i(\mathcal{W})$) a new entry is added to this table. It fills the first two fields of this new row that is done via a database stored procedure ($DB_i.leave(view_id, nodes)$). Whenever a transaction commits, it appends its write set into the row corresponding to the current view via the $DB_i.missed_updates(view_id, oids)$ action. One can realize that this new table may indefinitely grow, however we have automatized its cleaning. Thus, at the end of a node missed view recovery process it is deleted from the respective entry via the $DB_i.recovered_view(view_id, node)$. If at the recovered view there

```
join_i(W) // W ∈ {⟨id, availableNodes⟩ : id ∈ ℤ ∧ nodes ⊆ N} //
pre ≡ MM_i.view_change = W ∧
    W.availableNodes \ V_i.availableNodes ≠ ∅.
eff ≡ nodes ← W.availableNodes \ V_i.availableNodes;
    if i ∈ V_i.availableNodes then
      ∀ t ∈ T :
        if status_i(t) ∈ {pre_commit, committable}∧
            node(t) = i then
          sendURMulticast(⟨remote, t, DB_i.WS(t)⟩, nodes);
          participants_i(t) ← participants_i(t) ∪ nodes;
    if W.availableNodes ≠ N then
      missed_i(W.id) ← ⟨N \ W.availableNodes, ∅⟩;
    ∀ j ∈ nodes : sites_i(j) ← ⟨pending_metadata, W.id, ∅, ∅⟩;
    oldest_alive ← min_age(sites_i(·), V_i, W);
    if i = oldest_alive then
      sendURMulticast(⟨recovery_start, i, nodes,
          sites_i(·), minimum_missed(nodes)⟩,
          W.availableNodes);
    V_i ← W.

◇ function min_age(sites(·), V, W) ≡
j ∈ N : j ∈ V.availableNodes ∩ W.availableNodes ∧
    (sites(j).state = recoverer ∨ (sites(j).state = alive∧
      (∀ k ∈ V.availableNodes ∩ W.availableNodes, k ≠ j :
        sites(k).state = alive ∧ (sites(j).age < sites(k).age ∨
          sites(j).age = sites(k).age ∧ j < k))))

◇ function minimum_missed(nodes) ≡
send_info ⊆ {⟨s, o⟩ : s ⊂ N ∧ o ⊆ OID} : send_info ← ∅
∀ k ∈ [0, V_i.id] :
    if ∃ j ∈ nodes : j ∈ missed_i(k).sites then
      ∀ l ∈ [k, V_i.id] : send_info ← send_info ∪ missed_i(l)
      break

receive_recovery_start_i(m)
pre ≡ sites_i(i).state ≠ crashed ∧ m = ⟨recovery_start,
        recov_id, nodes, m_sites(·), m_missed(·)⟩ ∈ channel_i.
eff ≡ receive_i(m);
    sites_i(·) ← m_sites(·);
    if i ∈ nodes then missed_i ← missed_i ∪ m_missed(·);
    sites_i(recov_id).state ← recoverer;
    ∀ j ∈ nodes :
      sites_i(j).state ← joining;
      if sites_i(i).state ∈ {joining, recoverer} then
        ∀ t' ∈ generate_rec_trans(j, missed_i(·)) :
          sites_i(j).to_recover ← sites_i(j).to_recover ∪ {t'};
          DB_i.begin(t'); objs ← missed_i(t'.view_id).oids;
          if j ≠ i then DB_i.recover_other(t', objs)
          else
            DB_i.recover_me(t', objs);
            sites_i(i).to_schedule ← sites_i(i).to_schedule ∪ {t'};
          status_i(t') ← blocked;
      // This updates obsolete recovery metadata info
      ∀ k ∈ V_i.availableNodes \ nodes ∧
          sites_i(k).state ∈ {alive, recoverer} :
        ∀ l ∈ {0, V_i.id ∧ k ∈ missed_i(l).sites :
          missed_i(l).sites ← missed_i(l).sites \ {k}.

◇ function generate_rec_trans(j, missed(·)) ≡
txns ⊆ T : txns ← ∅
∀ k ∈ [0, V_i.id) :
    if j ∈ missed_i(k).sites then // t = ⟨node, view⟩ //
      txns ← txns ∪ {recov_transaction(j, k)}

receive_missed_i(t, m)
pre ≡ sites_i(i).state ∈ {joining, recovering} ∧ status_i(t) =
        active ∧ m = ⟨missed, t, op⟩ ∈ channel_i.
eff ≡ receive_i(m); DB_i.submit(t, op); status_i(t) ← blocked.

receive_view_recovered_i(m)
pre ≡ sites_i(i).state ≠ crashed ∧
        m = ⟨view_recovered, view_id, id⟩ ∈ channel_i.
eff ≡ receive_i(m);
    missed_i(view_id).sites ← missed_i(t.view_id).sites \ {id}.
    if (∀ z ∈ {0, V_i.id} : id ∉ missed_i(z).sites then
      sites_i(id).state ← ⟨alive, V_i.id, ∅⟩;
    if (∀ j ∈ V_i.availableNodes : sites_i(j).state ∈
        {alive, recoverer}) then
      ∀ j ∈ V_i.availableNodes : sites_i(j).state ← alive.

leave_i(W) // W ∈ {⟨id, nodes⟩ : id ∈ ℤ, nodes ⊆ N} //
pre ≡ MM_i.view_change = W ∧
        V_i.availableNodes \ W.availableNodes ≠ ∅.
eff ≡ nodes ← V_i.availableNodes \ W.availableNodes;
    ∀ t ∈ T :
      if node(t) ∈ nodes then DB_i.abort(t); status_i(t) ← aborted
      else if node(t) = i ∧ status_i(t) = pre_commit then
        participants_i(t) ← participants_i(t) \ nodes;
      else if t ∈ {t' : t' ∈ missed_i(k).to_recover, k ∈ nodes} then
        DB_i.abort(t); status_i(t) ← aborted;
    ∀ k ∈ nodes, sites_i(k).state ∈ {joining, recovering, alive} :
      sites_i(k) ← ⟨crashed, V.id, ∅⟩
    if ∃ k ∈ nodes : sites_i(k).state = recoverer then
      sites_i(k) ← ⟨crashed, {V.id, ∅}⟩;
      oldest_alive ← min_age(sites_i(·));
      sites_i(oldest_alive).state ← recoverer;
      if i = oldest_alive then
        ∀ j ∈ {n ∈ N : sites_i(n).state ∈ {joining, recovering}} :
          ∀ t' ∈ generate_rec_trans(j, missed_i(·)) :
            sites_i(j).to_recover ← sites_i(j).to_recover ∪ {t'};
            DB_i.begin(t');
            objs ← missed_i(t'.view_id).oids;
            DB_i.recover_other(t', objs);
            status_i(t') ← blocked;
        ∀ j ∈ {j ∈ W.availableNodes :
            sites_i(n).state = pending_metadata} then
          nodes ← {j ∈ N : sites_i(j).state = pending_metadata};
          sendURMulticast(⟨recovery_start, i, nodes,
              sites_i(·), minimum_missed(nodes)⟩,
              W.availableNodes);
    V_i ← W.
```

Figure 5.6: Specific recovery state transition system for ERP

are no nodes left, then the row will be erased. As an implementation detail, it is important to note that the insertion of a new object identifier will check if that data item is included in previous views whose nodes are a subset of nodes included in the current view. This fact avoids to recover several times the same object in different views. We have included the $DB_i.get_missed_updates(nodes)$ database procedure

create$_i(t)$ // $node(t) = i$ //
$pre \equiv sites_i(i).state \notin \{\text{crashed, pending_metadata, joining}\} \wedge status_i(t) = \text{start}$.

begin_operation$_i(t, op)$ // $node(t) = i$ //
$pre \equiv sites_i(i).state \neq \text{crashed} \wedge status_i(t) = \text{active}$.

end_operation$_i(t, op)$
$pre \equiv sites_i(i).state \neq \text{crashed} \wedge status_i(t) = \text{blocked} \wedge DB_i.notify(t, op) = \text{run}$.
$eff \equiv$ **if** $t \in sites_i(node(t)).to_recover$ **then**
  if $sites_i(i).state = \text{recoverer}$ **then**
    $sendRUnicast(\langle missed, t, op\rangle, node(t))$;
    $sites_i(i).to_recover \leftarrow sites_i(i).to_recover \setminus \{t\}$;
    $status_i(t) \leftarrow \text{committed}$;
    $DB_i.commit(t)$;
    if $\neg empty(queue_i)$ **then** $remove_i \leftarrow \text{true}$
  else // $sites_i(i).state \in \{\text{joining, recovering}\}$ //
    if $op = \perp$ **then**
      $status_i(t) \leftarrow \text{active}$;
      $sites_i(i).to_schedule \leftarrow sites_i(i).to_schedule \setminus \{t\}$;
      if $sites_i(i).to_schedule = \emptyset$ **then**
        $sites_i(i).state \leftarrow \text{recovering}$;
    else
      $sites_i(i).to_recover \leftarrow sites_i(i).to_recover \setminus \{t\}$;
      $status_i(t) \leftarrow \text{committed}$;
      $DB_i.commit(t)$;
      if $\neg empty(queue_i)$ **then** $remove_i \leftarrow \text{true}$;
      $sendURMulticast(\langle view_recovered, t.view_id, i\rangle, \mathcal{V}_i.availableNodes)$
else
  if $node(t) = i$ **then** $status_i(t) \leftarrow \text{active}$
  else $status_i(t) \leftarrow \text{pre_commit}$.

begin_commit$_i(t)$ // $node(t) = i$ //
$pre \equiv sites_i(i).state \neq \text{crashed} \wedge status_i(t) = \text{active}$.

local_abort$_i(t)$
$pre \equiv sites_i(i).state \neq \text{crashed} \wedge status_i(t) = \text{blocked} \wedge DB_i.notify(t, op) = \text{abort}$.

receive_ready$_i(t, m)$ // $node(t) = i$ //
$pre \equiv sites_i(i).state \neq \text{crashed} \wedge status_i(t) = \text{pre_commit} \wedge participants_i(t) \neq \emptyset \wedge m = \langle ready, t, source\rangle \in channel_i$.

end_commit$_i(t)$ // $node(t) = i$ //
$pre \equiv sites_i(i).state \neq \text{crashed} \wedge status_i(t) = \text{pre_commit} \wedge participants_i(t) = \emptyset$.

receive_commit$_i(t, m)$
$pre \equiv sites_i(i).state \neq \text{crashed} \wedge status_i(t) \in \{\text{pre_commit, committable}\} \wedge m = \langle commit, t\rangle \in channel_i$.
$eff \equiv receive_i(m)$;
  $DB_i.commit(t)$;
  $status_i(t) \leftarrow \text{committed}$;
  if $\mathcal{V}_i.availableNodes \neq \mathcal{N}$ **then**
    $missed_i(\mathcal{V}_i.id).oids \leftarrow missed_i(\mathcal{V}_i.id).oids \cup DB_i.WS(t).oids$.

receive_remote$_i(t, m)$
$pre \equiv sites_i(i).state \neq \text{crashed} \wedge status_i(t) \neq \text{aborted} \wedge m = \langle remote, t, WS\rangle \in channel_i$.

execute_remote$_i$
$pre \equiv sites_i(i).state \notin \{\text{crashed, pending_metadata, joining}\} \wedge \neg empty(queue_i) \wedge remove_i$.

receive_abort$_i(t, m)$
$pre \equiv sites_i(i).state \neq \text{crashed} \wedge status_i(t) \neq \text{aborted} \wedge m = \langle abort, t\rangle \in channel_i$.

discard$_i(t, m)$
$pre \equiv sites_i(i).state \neq \text{crashed} \wedge status_i(t) = \text{aborted} \wedge m \in channel_i$.

$\diamond$ **function** $higher_priority(t, t') \equiv node(t) = j \neq i \wedge (a \vee b \vee c)$
(a) $node(t') = i \wedge status_i(t') \in \{\text{active, blocked}\}$
(b) $node(t') = i \wedge status_i(t') = \text{pre_commit} \wedge t.priority > t'.priority$
(c) $\nexists k \in \mathcal{N}: t' \in sites_i(k).to_recover$

Figure 5.7: Add-ons on the state transition system of ERP so as to support recovery features

that will be executed by the *recoverer* node so as to determine the metadata missed by the *nodes* that have just rejoint the system in order to send this information to them. Respectively, rejoining nodes will invoke $DB_i.set_missed_updates(view_id, nodes, oids)$ for each row contained in the metadata information transferred by the *recoverer* node to them.

As we are in a middleware architecture, we have added two database stored procedures for blocking or aborting transactions at the recovery startup time: *recover_me* and *recover_other*. The first one performs a "`SELECT FOR UPDATE`" SQL statement over the objects to be recovered in a given view by the *recovering* node. The second procedure is invoked at the *recoverer* node, it will rollback all previous transactions that were trying to update an object before performing the "`SELECT FOR UPDATE`" SQL statement over the given partition. This prevents local transactions from modifying the rows inside the partition. Afterwards, as we have mentioned before, transactions trying to update objects belonging to the partition will get blocked whereas read access is permitted.

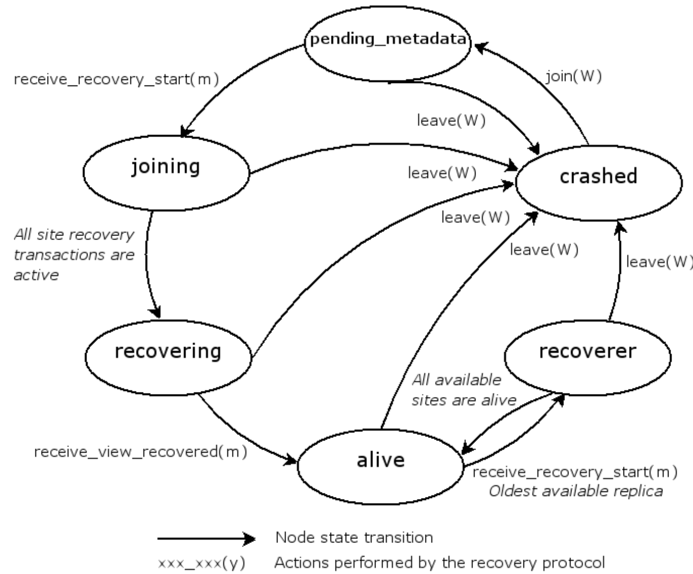


Figure 5.8: Valid transitions for a given $sites_i(j).state$ of a node $j \in N$ at node i

To continue with the recovery protocol, two new state variables, $sites_i(j)$ and $missed_i(id)$ have been added, at each site i. The first one stores the *state* of each node $j \in N$ (whether it is *pending_updates*, *joinining*, *crashed*, *recovering*, *recoverer* or *alive*; its possible transitions are shown in Figure 5.8), its respective *age* (the view identifier, $\mathcal{V}_i.id$, when it joined the system) and the recovering transactions associated to that node, as long as it is in the *recovering* state (*to_recover*). An additional field *to_schedule* monitors the execution of recovery transactions in a *recovering* node that will be described in detail afterwards. The second variable $missed_i$ represents the `MISSED` metadata table of the DBMS. For each view, it contains the set of nodes crashed (or not recovered yet) and the set of objects modified in the given view. In this protocol outline we have defined it as an array indexed by the view identifier whose associated values are the set of nodes crashed, or not recovered yet, and their respective missed updates for that view. Operations performed over the $missed_i(\cdot)$ state variable shown in Figure 5.6 may be easily ported to the respective database stored procedures as we have previously outlined and thoroughly explained in Chapter 2.

5.3.1 Site Failure

Initially all sites are up and *alive*. Afterwards, some site (or several with no loss of generality) may fail ($MM_i.view_change$). At this point, our recovery protocol starts running with the execution of the $leave_i(\mathcal{W})$ action, see Figure 5.6. All nodes change the state associated to that node to *crashed*; besides, a new entry in the `MISSED` table is added for this new view identifier containing (as it was mentioned before): the new view identifier and the failed node. User transactions will continue working as usual, nevertheless the respective transaction master sites will only wait for the *ready* message coming from these new set of available nodes (ROWAA). When a transaction commits, each available site inserts updated objects into the

entry associated with the current installed view contained in the MISSED metadata table.

Actions to be done when a node failure happens ($leave_i(\mathcal{W})$) involve several tasks in the recovery protocol executing at an available node, apart from the ones dealing with the variables and metadata management that we have mentioned before. It will rollback all remote transactions coming from the *crashed* node. Local transactions executing at an available node i with $status_i(t)$ = pre_commit must remove from their $participants_i(t)$ all *crashed* nodes. This process may imply a multicast of a *commit* message to all available nodes as all available nodes have previously answered they were *ready* to commit, this is shown in Figure 5.9. Besides, if the *crashed* site was *recovering*, all its associated recovery transactions in the *recoverer* node will be aborted too. If the failed node was the *recoverer*, then the protocol must choose another *alive* node to be the new *recoverer*. This new *recoverer* site will create the partitions pending to be transferred and then it will perform the object transfer to *recovering* and *joining* sites. If there exists any node whose state is *pending_metadata* it will start the recovery process for this node. This is depicted in Figure 5.10 and the whole recovery process will be described on the sequel.

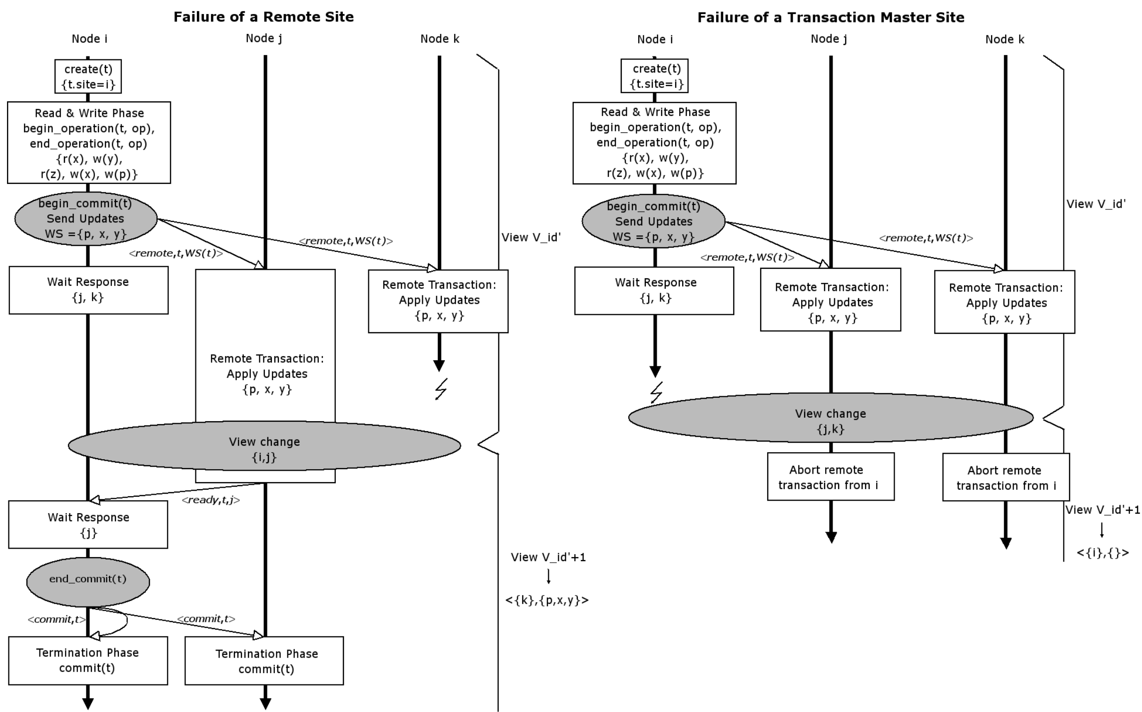


Figure 5.9: Actions to be done by the recovery protocol when a remote node (left) or a transaction master site (right) fails

5.3.2 Site Recovery

The membership monitor will enable the $join_i(\mathcal{W})$ action to notify that a node has rejoined the system (see Figure 5.6). These new nodes must firstly update their recovery metadata information. Hence, they are in the *pending_metadata* state, and may not start executing local transactions until they reach the *recovering* state.

The recovery protocol will choose one site as the *recoverer* by the function *min_age* shown in Figure 5.6, in our case, the oldest one. Once a site is elected, it multicasts its $sites_i$ state variable and the missed part of the variable $missed_i$ that corresponds to the oldest *crashed* node that has joint in this new installed view. One can note that there is no object state transfer at this stage, only recovery metadata information that is also outdated in these nodes.

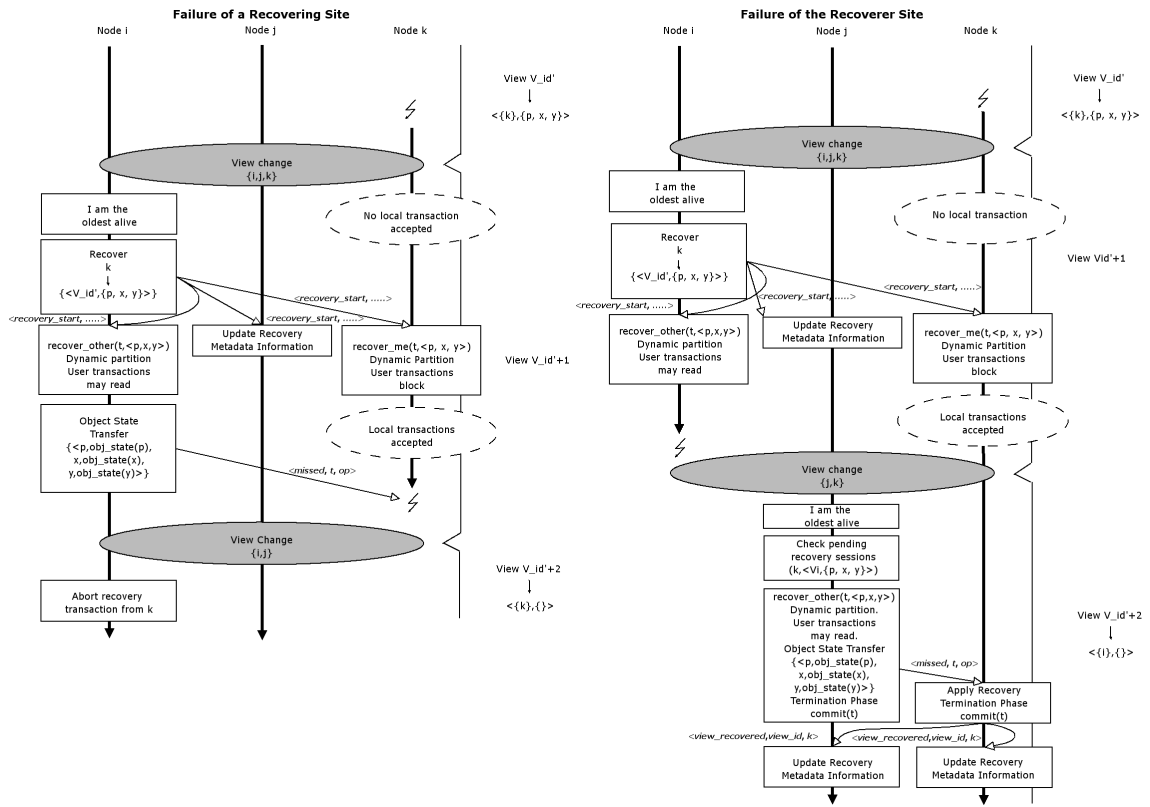


Figure 5.10: Actions to be done by the recovery protocol when a *recovering* node (left) or the *recoverer* node (right) fails

More actions have to be done while dealing with the join of new sites. Current local transactions of previously available sites in the *pre_commit* or *committable* states must multicast the *remote* message to all *pending_metadata* nodes which appropriately increase their associated $participants_i(t)$ variable. Otherwise, as these transactions are waiting for the *ready* message coming from previously available nodes, all new available nodes will receive a *commit* message from a remote transaction they did not know about its existence. This may be best viewed with the example shown in Figure 5.11. Let us consider a site that is only waiting for the *ready* message coming from a node available at the previous view that has not crashed in this new view. Assume that the *ready* message is delivered in this new installed view, then the transaction master site will multicast a *commit* message to all available nodes. This includes all new joining nodes that will never treat this message since its associated remote transaction has never been executed on their sites.

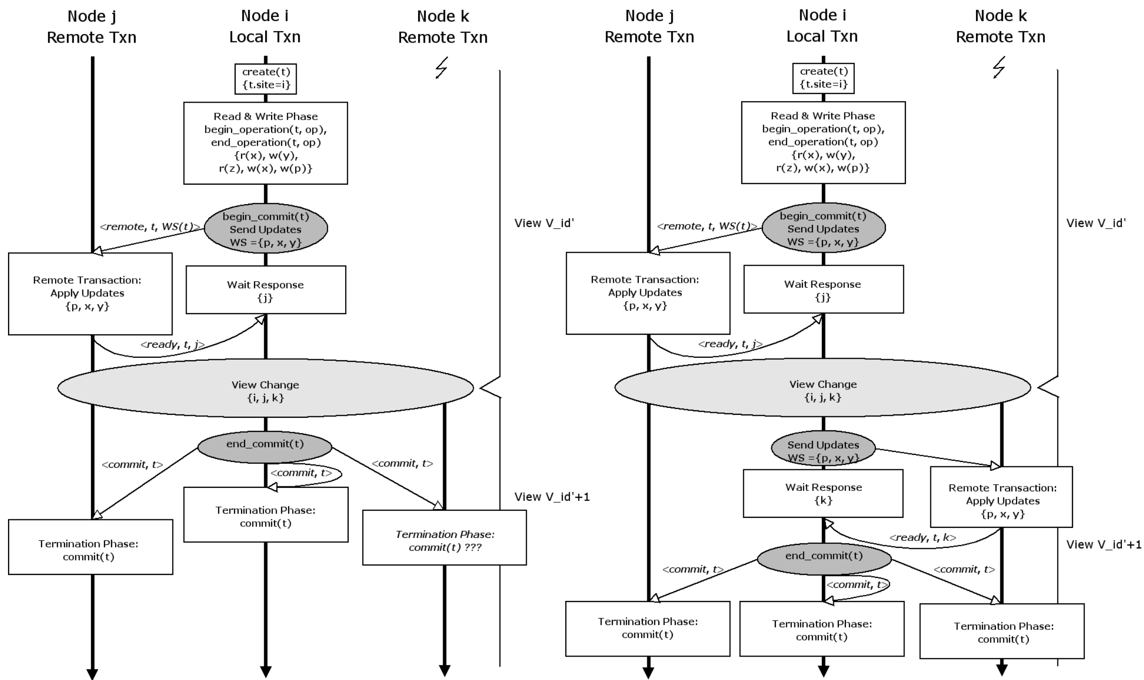


Figure 5.11: A new joining node may commit a transaction whose existence does not know (left). Its modification (right) allows joining nodes to execute that transaction, even though they have not recovered yet

The Beginning of the Recovery Process

The *recovery_start* message delivery, as its own name states, starts the recovery process which is performed by recovery transactions. First of all, those *pending_metadadata* nodes change their state to *joining*, as shown in Figure 5.8. The *joining* nodes update their metadata recovery information. As soon as the metadata information is updated the recovery protocol will set up the recovery partitions on *joining* nodes.

Let us focus on its own associated recovery transactions inserted in the respective *to_recover* and *to_schedule* fields. Each recovery transaction will submit two operations to the database. The first time, it will invoke the *recover_me* database stored procedure (recall that this procedure blocks any kind of access to the objects that comprise the partition) and the protocol will eliminate the respective recovery transaction entry from *to_schedule*. As all of them are removed, the *joining* node will change its state to *recovering*. Hence, user transactions will be able to start working on the *recovering* node, even though it has not ended its recovery process. The second operation will be executed during the missed updates transfer, as we will see next.

Meanwhile, the *recoverer* node generates the partitions by the invocation of another specific recovery database procedure called *recover_other*. This procedure aborts, during the partition set up time, all transactions already trying to update objects; afterwards, transactions trying to update data items belonging to a partition will get blocked. The recovery transaction performs two operations in the *recoverer* node, it sets up the partition and retrieves the missed updates that the recovery protocol transfers to the *recovering*, or

still *joining*, node. At this time, the recovery partition will be released (the recovery transaction committed) at the *recoverer* node. The rest of nodes will continue working as usual. If they execute transactions that update data items belonging to a recovery partition, they will be blocked in the *recoverer* and the *recovering* node, i.e. they will not receive the *ready* message from these nodes. This approach of implementing the *recover_other* procedure is intended for DBMS using locks as the concurrency control. However, if we use a MVCC DBMS there is no need to abort transactions at the *recoverer* site during the recovery partition set up, since the recovery transaction may read the values belonging to the partition while transactions updating items contained in the partition may propagate the changes at the same time. The recovery protocol will enqueue the *remote* message coming from these partitions until the recovery process is done. As a final remark, there is no need to abort any user transaction if we use a MVCC DBMS for carrying out the recovery process. Even more, transactions may update items belonging to recovery partitions, however their transaction master sites will not receive the *ready* message until the recovery process is done.

Transferring Missed Updates to the Recovering Node

This step of the recovery process is englobed inside the $receive_missed_i(t, \langle missed, t, op\rangle)$ and its respective $end_operation_i(t, op)$ action, since updates are submitted to the DB_i module. The op field contains the updates missed by i during $t.view$. This process will be repeatedly invoked if there are more than one missed view by the *recovering* node, as each execution of this pair of actions only involves a single view missed data transfer.

The $receive_missed_i(t, \langle missed, t, op\rangle)$ action will be enabled once its associated transaction has blocked the partition of the *recovering*, or still *joining*, node. The missed updates will be applied in the context of this recovery transaction. This operation is submitted to the DB_i module and we have to wait for its ending. As this transaction will not be aborted by the underlying database, it will eventually execute its associated $end_operation_i(t, op)$ action. This action will multicast a *view_recovered* message to all available sites.

Finalization of the Recovery Process

Once all missed updates are successfully applied, the recovery process finishes. In the following we will describe how the recovery process deals with the end of the recovery process of a given node. As we have said during the data transfer, all missed updates are grouped by views where the *recovering* node was *crashed*. Hence, each time it finishes the update, it will notify the rest of nodes about the completion of the view recovery.

The execution of the $receive_view_recovered_i(t, \langle view_recovered, view_id, j\rangle)$, with j as the node identifier being recovered and *view_id* as the recovered view identifier, is enabled once the message has been delivered by the GCS. This action updates the variable $missed_i$ as it removes the *recovering* node from the entry *view_id*. It also updates the *state* field of the $sites_i$ if it is the last recovery partition of the *recovering*

node setting it to *alive*. Besides, if there are no more partitions to be recovered by the *recoverer* node it is also set to *alive*.

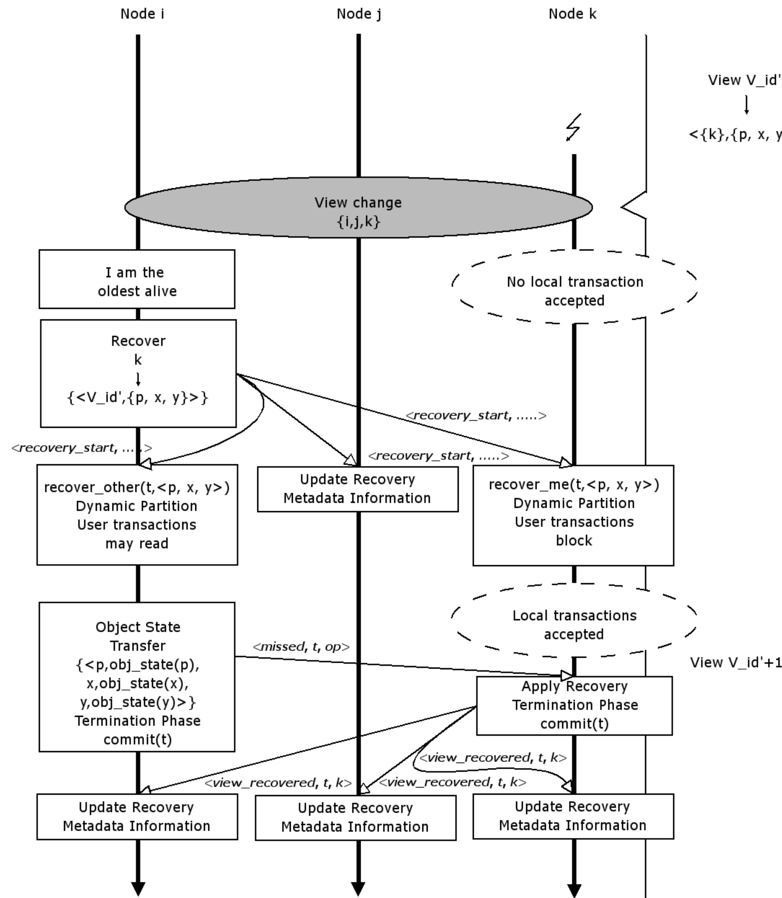


Figure 5.12: Description of the ERP recovery protocol. It recovers node k that has been *crashed* just for one view, id'. Data items $\{p, x, y\}$ have been updated in that installed view

All we have described till now is shown in Figure 5.12. Node k is *crashed* and it recovers at $\mathcal{V}.id = id' + 1$. During its failure, three different data items p, x and y have been updated. Once it rejoins the system, the *recoverer* site sends the recovery metadata to all nodes. This provokes the definition of a partition composed by the three data items. The *recoverer* sends the missed updates to the *recovering* site, which at the same time of its recovery may accept user transactions. When the recovery transaction of the *recovering* site finishes applying its updates, it sends a message saying that a view has been recovered. This message frees the associated partition and returns the state of k to *alive*. Respectively, as there are no more *recovering* nodes the site i returns to *alive* too.

5.4 Discussion

In this Chapter we have coped with site failures and recovery for the ERP replication protocol proposed in Chapter 3. We have considered the features provided by a GCS such as the view synchrony and the uniform

reliable multicast that facilitates tasks to be done when a site fails or recovers from a failure. Hence, we have modified the database schema so as to add recovery metadata information, mainly we have stored for each view: the set of nodes that have crashed (reported by the GCS when a new view is installed) and objects updated (filled by the replication protocol each time a transaction commits). We assume a partial amnesia crash failure for nodes belonging to our middleware architecture. Modifications of the ERP replication protocol and the recovery protocol itself are introduced as state transition systems. There is a wide range of possible failure and recovery scenarios to check, hence we have only considered a few with some figures that are intended to clarify the recovery protocol behavior. However, we have not provided its correctness proof.

We do not provide an explicit state transition system of the recovery protocol for the BRP replication protocol since it will consist in the same set of actions with slight modifications. It will have a queue where update transaction will be stored as long as there are recovery partitions on the *recovering* node. Once the partitions are totally released it will behave as the BRP. Nevertheless, specific recovery actions will remain the same.

We have based our recovery support idea in the view synchrony provided by the GCS. Each time a node fails or recovers from a failure, a new view change event is fired by the GCS notifying the current set of available nodes. Besides, the GCS features another set of interesting properties that are of great value from the recovery protocol point of view, as the uniform reliable multicast, which ensures that if a message has been delivered by a faulty node, then all node will eventually deliver the same message.

The GCS facilities also provide an easy way to feature site recovery as it is possible to group the updates missed by a faulty node by the installed view where they happened. We have stored the missed updates in the database by the definition of the proper additional metadata tables and functions. This data grouping serves as the definition of the basic recovery data unit for our replication protocol. Once a site recovers from a failure, it is established a set of partitions at all available sites (as many as views missed by the recovering node). Thus, those available nodes that are neither *recoverer* nor *recovering* nodes will access these partitions as usual. However they will get blocked when they propagate their updates and they conflict with a recovery partition at the *recovering* or the *recoverer* nodes. The *recovering* node will not be able to access these objects. The *recovering* node may start accepting user transactions as soon as the partitions are set up on it, even though it is not up to date.

As it can be seen from all above, there is a price to pay for dealing with recovery issues such as: more latencies on message deliveries (due to the uniform reliable multicast); data storage of missed updates at the DBMS; and, by blocking or, at worst, rolling back user transactions. All of this introduces additional overhead to the normal user transaction behavior that provokes a decrease in transaction response time although we are increasing our system robustness.

The recovery protocol depicted in this Chapter works well in environments where failures are not too

long or updates as long as a node is crashed are rare. Therefore, we may modify this recovery protocol in order to support a significatively large amount of updated objects. This modification, that is the most intuitive one, will consist of blocking the data repository and transfer the whole database to the recovering node. This solution penalizes system availability but it is the most effective one in such a case. As a final remark, we may set up a threshold in order to switch between the recovery idea outlined in this Chapter and the whole data transfer.

5.4.1 Comparison with Related Works

There are several works in the literature that propose several alternatives to accomplish database recovery. In [KBB01] several solutions to online reconfiguration in replicated databases are proposed. It discusses various alternatives for data transfer to joining sites, all of them allowing concurrent transaction processing, using the view synchrony provided by the GCS. It pursues solutions that admit cascaded reconfigurations during the data transfer itself; hence, a clear separation between the tasks of the GCS and the tasks of the database system must be done.

The different recovery options introduced in [KBB01] are commented in the following. The first proposal is the data transfer within the GCS; some GCSs are able to perform data transfer during the view change [BCJ+93]. However, this presents some drawbacks, as the GCS does not know the objects changed, it must transfer the whole database and during that transfer the system must remain in the same state. This blocks the whole system and violates the requirement of high availability. Therefore, the data transfer should be performed by the database system using the appropriate database techniques. This constitutes the second approach. The GCS should only provide the appropriate semantics to coordinate the data transfer. The data transfer is done via TCP and the updates performed after the view change are enqueued until data is transferred to the recovering node. There are two approaches for doing this: transferring the whole database, which is appropriate if the database is small or a great number of objects have been modified; or, checking the whole database and transferring items that have been modified (via version numbers). Another approach is to define a record table storing for each object modified the latest transaction identifier that has modified it. Once a site joins again, a read lock is set in the whole database that waits until all writing operation finishes. Then it will set read locks only on those objects contained in the record table that will be released once the data transfer has been finished. Another approach is the use of multiversion databases in order to avoid setting read locks for a long time. The last and the most interesting recovering idea is to decouple the view change as the synchronization point for data transfer. This last approach is referred as lazy data transfer. The recoverer node initially discards all incoming write sets until a given threshold is passed (such as given number of transactions or rounds). The recoverer transfers data to the recovering node belonging firstly to missed changes and then to pending updates executed in the given view until the threshold is passed. At that time, the recovering node is considered to be active and may start processing

transactions. It also explains steps to be done when a failure occurs during the data transfer. Our recovery proposal permits to start running transactions on the recovering node as soon as the recovery partition is set in the database.

In [Hol01], several recovery protocols have been also proposed using the virtual synchrony properties. The replication protocols used in the system are the ones introduced in [AAES97, KA00a]. The idea underlying each of these protocols is that since the communication subsystem guarantees totally ordered broadcasts, operations can be processed at every site in the same order. These protocols execute read operations locally and multicast only write operations. The first of the three recovery protocols proposed in [Hol01] is the *single broadcast recovery*, where some sites are defined as *Loggers*. A *Logger* is a site that stores log messages such as a view change, transaction commits and update messages. If a new view is installed in the system, no update transaction messages are delivered to any site until the new site has exchanged messages with a *Logger* and the *Logger* has signalled that the recovery process is finished. The *Logger* sees a view change message followed by a request from the *recovering* node to be brought up-to-date. The *Logger* looks up the last view of which the *recovering* site was a member and all the transactions which committed since the last view of which the *recovering* site was a member are sent to the *recovering* node in the order in which they committed. When the *recovering* node successfully completes global recovery, it acknowledges this to the *Logger* who then tells all sites to begin normal operations. The second approach is called *log update* where *Loggers* must check if there are on-going transactions at non-failed sites during a view change since the commit message is contained in the next view. Hence, when the node recovers again, data must be transferred from the first *remote* message missed. The drawback to this method is that the Loggers must keep the logs of previous views whether or not a site was missing from the group. The last method is the *augmented broadcast*, it shifts additional processing to the transaction master sites and requires a change to the recovery lock management algorithm. If there is an on-going transaction when a new view is installed, it modifies the protocol for committing transactions so that the write requests are included in the commit request for all transactions that broadcast their writes in an earlier view. In this method, write requests may be sent out of the original order to a recovering site. Therefore, during global recovery, the lock manager must be able to undo the writes of a not yet committed transaction if necessary to free the locks and perform the actions specified in the special *commit* message and then redo the effects of the first transaction. In our approach, we do not stop sending updates while a node is recovering as in the *single broadcast recovery* and we do not use logs to maintain the information to be transferred to the recovering node.

Another work that presents a recovery protocol is [JPPMA02]. The database is split into partitions where each site is the owner of one or several partitions. When a node joins the group, the partition master chooses one site as the recoverer of that partition which transfers the log of missed updates to the recovering node. The recoverer sends a message, using causal multicast, once it has finished transferring the missed

updates. The partition master site answers to that message with a causal multicast message containing the transaction identifier of the last committed transaction. This message causes that the recoverer node enters in the forwarding state. It sends all updates performed in the current view until the transaction identifier contained in the previous message. Besides, transactions are delivered at the recovering node that will not be applied until all the information transferred in the forwarding state have not being applied in the recovering node. At that time, the recovering node is considered to be active.

Transactions executed on active nodes are not blocked while a node is being recovered. As it has been pointed out, the node will start executing transactions as soon as the messages in the forwarding phase have been applied. This recovery approach is better than ours since there is no inactivity phase in any node. However, it presents other problems as transactions may only be executed on its partition master site. As it uses logs, when failures periods are considerably long, the information to be transferred will be large, while ours only transfer the latest version of data items.

Another recovery protocol that uses total order primitives from the GCS to perform recoveries is introduced in [RMA+02]. It presents the advantage that active nodes may continue working even though the recovery data transfer is being done, since the recovering node enqueues all write sets total order delivered until all missed updates have been applied. It does not admit transactions on the recovering node until all messages contained in the queue are applied. Our recovery protocol does not present that problem. We have a queue that stores conflicting write sets but some other write sets may be applied during the recovery process on the recovering node.

Finally, another recovery protocol proposed in the literature is the FOBr [CCIBGNME05]. The replication protocol used there is based in the object ownership concept in an object oriented architecture. Its performance is lower than the ERP. However, its recovery is more efficient as it is not necessary to block transactions on active nodes, even in the recovering one.

Chapter 6

Conclusions

6.1 Summary

This Thesis has proposed several research goals that include, among others, the development of correct new eager update everywhere replication protocols adapted to the MADIS middleware architecture. These protocols do not depend on strong group communication primitives. Besides, a recovery proposal for failed nodes has been included. In the following, the main research contributions of the Thesis are summarized in a more detailed manner:

• **The MADIS Middleware Architecture**. As it has been previously pointed out, we have collaborated in the design, development and implementation of the MADIS architecture. MADIS is a middleware architecture providing database replication, where availability and performance are increased due to the existence of multiple physical copies of a unique logical data item. We have focused in providing a standard interface for user applications, such as JDBC, while data consistency is intended to be managed by a wide range of replication protocols. We have defined a generic protocol interface so that protocols may listen to the events they are interested in, e.g. before performing an update, after a deletion operation and so on. This is achieved using database standard features, such as functions, stored procedures, triggers and tables that enhance the original database schema in order to facilitate the replication tasks to the protocols. We do not have modified any DBMS internals to ensure replication or to improve performance goals. Hence, the portability of this solution to different DBMSs will not be a very difficult task. Besides, due to the wide range of supported replication protocols, the price to pay is the amount of metadata needed for each transaction update. The MADIS version presented in this Thesis is its first release. Its optimization must be carefully analyzed, however this is not the main issue of this thesis. Although MADIS is not aimed for any concrete DBMS vendor, we have implemented it using PostgreSQL, as it is a free DBMS.

• **A Middleware Database Replication Protocol and its Enhanced Version**. Once our middleware architecture is designed and implemented, we may implement several database replication protocols. We

were particularly interested in developing eager update everywhere replication protocols, following the ROWAA [GHOS96] policy, that ensures 1CS [BHG87]. There are recent approaches [JPPMKA02, Kem00, KA00b, KPA+03, PMJPKA00, PMJPKA05] that ensure this correctness criteria by way of the total order delivery guarantees provided by the GCS [CKV01]. This is an interesting approach since transactions do not have to wait for applying the updates at the rest of sites in order to commit a transaction, as the 2PC rule states [BHG87]. Following this approach, the system performance is increased. However, we thought that relying on this strong group communication primitives, whose latencies and extra message rounds in environments where conflicts are rare, is a high price to pay [KPA+03]. Besides, the order of committed transactions is imposed by a "*black-box*" with an algorithm that does not have any information about transactions (such as number of objects read, written or number of restarts) that may cause several transaction patterns to be penalized by the total order based replication protocol.

The first replication protocol proposal, named BRP, is based on the O2PL [CL91] protocol which is the first approach done to develop a ROWAA [GHOS96] protocol. In order to avoid distributed deadlock, we have defined a dynamic deadlock prevention schema based on priorities: a global priority number associated to each transaction and the state of the transaction in the system. In this way, we may modify the transaction priority assignment and then change the order on which transactions may commit. The O2PL is a 2PC replication protocol, however, while developing its correctness proof, we realized how to circumvent the problem of waiting for applying the updates at the rest of nodes before committing the transaction. This is achieved thanks to the dynamic deadlock prevention schema we have defined. Up to our knowledge, no similar protocols have been described in the literature. This led us to the definition of the second replication protocol, named ERP. The ERP protocol may be considered as an intermediate between 2PC protocols and those based on total order. Finally, this ERP has been adapted to the MADIS architecture and some experimental results of its behavior are shown.

• **Correctness Proof**. The correctness proof associated to a middleware database replication protocol must consider all the components involved in our middleware architecture. In our case, it consists in the database module, the user transaction interface, the GCS and the interactions of the replication protocol itself with other instances of it running at different sites. We have formalized the definition of our replication and recovery protocols using a state transition system similar to the one proposed in [Sha93]. This approach may be viewed as the I/O automata [Lyn96] composition of all its components. This way of introducing the protocols, besides of its clearness, implies an easier way to define properties, that facilitates the correctness proof of our replication protocol proposal. Up to our knowledge, this is something that has not been done regarding the O2PL [CL91]. Moreover, as pointed out before, at the same time we were developing the correctness proof of the O2PL we realized how to optimize its behavior and modify its 2PC philosophy.

• **Experimental results of database replication protocols implementation in MADIS**. We have implemented the BRP and ERP protocols in MADIS. This implementation may serve to check what it has been

predicted by the correctness proof of both protocols. Besides, we were very interested in their comparison with total order replication protocols, a widely used approach in the literature [AT02, EPZ05, JPPMKA02, Kem00, KA00b, KPA+03, LKPMJP05, Ped99, PMJPKA00, PMJPKA05, RMA+02, WK05]. Hence we have designed and implemented a new replication protocol, called TORPE, that is an adaptation of the previous protocols but using the total order multicast primitive provided by the GCS. We also introduce this protocol as a state transition system [Sha93]; therefore this way of presenting protocols is not exclusive of the O2PL as we have done it with a total order replication protocol. The comparison between our proposals and TORPE brings out the best performance of total order based replication protocols. However, under our experimental settings, ERP shows a better behavior for low workloaded environments, due to the overhead introduced by the GCS to TORPE.

• **The Recovery Protocol**. Database replication may not be considered complete until we do not cope with site failures and its recovery. We have studied the recovery proposals for systems providing database replication. We have proposed a recovery protocol based on setting up database dynamic recovery partitions by way of standard procedures, such as SQL statements. This permits current active transactions to continue working in the rest of available sites and the execution of user transactions in the recovering node even though it is still being recovered. This recovery process is based on facilities provided by the GCS, such as view synchrony [CKV01] and uniform delivery of multicast messages [HT94]. This permits to group updates done in the repository by views which will lead to the definition of dynamic recovery partitions over the database during the recovery process of a site.

Once a node fails, the rest of nodes store objects being modified (grouped by views) while the given site is down. When the site joins again, the recoverer node exclusively imposes partitions in the recoverer and recovering nodes (those defined by the updated objects missed in each view). Current executing transactions at the recoverer site may be rolled back, at worst, if they are modifying a data item involved in a recovery partition while it is being set. Afterwards, like the rest of user transactions coming from previously available sites, it will get blocked while trying to modify an item belonging to the recovery partition. Once partitions are established in the joining node, it can accept user local transactions. Recovery partitions at the recoverer node will be released as soon as they transfer the data items contained in the partition to the joining node. Whenever a view is recovered, the recovering node notifies this fact to the rest of nodes and frees its associated partition.

The modifications proposed in the recovery process introduce an additional overhead that penalizes the system performance but increases its availability. Therefore, there is a tradeoff between availability and performance. The propagation of multicast messages must be uniformly performed that leads to a latency increase of message delivery.

6.2 Future Lines

This Thesis has revisited and adapted a well-known algorithm such as O2PL to provide database replication in a middleware architecture, that has brough out new questions that will serve as the guideline for future research.

ERP vs. TORPE Comparison and Their Implementation. The implementation of all the replication protocols have been something that it has been done at the same time MADIS was implemented. We have taken a stable version of MADIS and the replication protocols have been implemented on that version. Since that time, several enhancements have been done in MADIS that were not reflected in the experiments shown here. However, the implementation of ERP and TORPE is not finished as we only intended to show a pattern of their behavior in the system. Of course, it is necessary to adapt ERP and TORPE to the new MADIS, finish their implementations and compare again. Another interesting comparison between ERP and TORPE would be through the standard database benchmark TPC-W [TW05].

Adaptability of MADIS and Mobility. As different applications require different kinds of replication management, an adequate choice of appropriate protocols is due. Hence, MADIS is a middleware which provides flexible support for choosing, plugging in, operating and exchanging suitable protocols for user applications. It would be interesting that a global administrator will monitor the performance of the application (such as the response time, the network latency or the abortion rate among others) to adaptively change from one protocol to another.

The ease of developing new protocols in this architecture and the increase usage of mobile computers lead us to an extension of MADIS for mobile users. Adapting MADIS to mobile environments is an interesting topic to study. Changes to be done in MADIS may imply a totally different version of the architecture. It has to offer better group communication support for partitionable environments [CKV01, BG05] and nodes capacities will be much limited than those belonging to a fixed network environment.

Developing New Replication Protocols. Protocols presented in this Thesis are 1CS. However, it is required that the underlying DBMSs provide serializable transaction isolation level, as in [BBG+95]. As it has been pointed out in Chapter 3, most of commercial DBMSs provide SI [BBG+95] it would be interesting to develop new replication protocols taking into account this fact. [EPZ05, LKPMJP05] are a good starting point for developing new protocols.

It would also be interesting to develop new replication protocols for mobile environments. This topic is closely related to the previous one. As derived from [GHOS96], lazy replications are the best option for replicated databases in mobile environments. However, we would like to study the implementation of epidemic replication protocols. All of them have to cope with more unstable networks, narrower bandwidths, greater heterogeneity of platforms and devices, etc.

Bibliography

[AACV03] José Enrique Armendáriz, José Javier Astrain, Alberto Córdoba, and Jesús Villadangos. Implementation of an object oriented query language compiler for replicated architectures. In Ernesto Pimentel, Nieves R. Brisaboa, and Jaime Gómez, editors, *JISBD*, pages 441–450, 2003.

[AAES97] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. *LNCS*, 1300:496–503, 1997.

[ACZ03] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In Markus Endler and Douglas C. Schmidt, editors, *Middleware*, volume 2672 of *Lecture Notes in Computer Science*, pages 282–304. Springer, 2003.

[AGME05] J.E. Armendáriz, J.R. González de Mendívil, and F.D. Muñoz-Escoí. A lock-based algorithm for concurrency control and recovery in a middleware replication software architecture. In *HICSS*, page 291a. IEEE Computer Science, 2005.

[AT02] Yair Amir and Ciprian Tutu. From total order to database replication. In *ICDCS*, pages 494–503, 2002.

[Bar04] Alberto Bartoli. Implementing a replicated service with group communication. *Journal of Systems Architecture*, 50(8):493–519, 2004.

[BBD96] Özalp Babaoğlu, Alberto Bartoli, and Gianluca Dini. On programming with view synchrony. In *ICDCS*, pages 3–10, 1996.

[BBG+95] 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 Michael J. Carey and Donovan A. Schneider, editors, *SIGMOD Conference*, pages 1–10. ACM Press, 1995.

[BCJ+93] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. The ISIS - system manual, Version 2.1. Epfl-tech-rep, Dept. of Computer Science, Cornell University, Sep 1993.

[BG05] M.C. Bañuls and P. Galdámez. On-demand membership service for energy-aware networks. In *DEXA Workshops*, pages 315–319. IEEE Computer Society, 2005.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[BK91] N. S. Barghouti and G. E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.

[CCIBGNME05] Francisco Castro-Company, Luis Irún-Briz, Félix García-Neiva, and Francesc D. Muñoz-Escoí. FOBr: A version-based recovery protocol for replicated databases. In *PDP*, pages 306–313. IEEE Computer Society, 2005.

[CKV01] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[CL91] Michael J. Carey and Miron Livny. Conflict detection tradeoffs for replicated data. *ACM Trans. Database Syst.*, 16(4):703–746, 1991.

[CMZ04] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. C-jdbc: Flexible database clustering middleware. In *USENIX Annual Technical Conference, FREENIX Track*, pages 9–18. USENIX, 2004.

[Cri91] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):56–78, 1991.

[CT91] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems (preliminary version). In *PODC*, pages 325–340, 1991.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[EPMEIBBA04] J. Esparza-Peidro, F.D. Muñoz-Escoí, L. Irún-Briz, and J.M. Bernabéu-Aubán. Rjdbc: a simple database replication engine. In *Proc. of the 6th Int'l Conf. Enterprise Information Systems (ICEIS'04)*, 2004.

[EPZ05] Sameh Elnikety, Fernando Pedone, and Willy Zwaenopoel. Database replication using generalized snapshot isolation. In *SRDS*. IEEE Computer Society, 2005.

[FLO+05] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[FMZ94] F. Ferrandina, T. Meyer, and R. Zicari. Implementing lazy database updates for an object database system. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 261–272, Santiago, Chile, 1994.

[GHOS96] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In H. V. Jagadish and Inderpal Singh Mumick, editors, *SIGMOD Conference*, pages 173–182. ACM Press, 1996.

[Hol01] JoAnne Holliday. Replicated database recovery using multicast communication. In *NCA*, pages 104–107. IEEE Computer Society, 2001.

[HSAA03] JoAnne Holliday, Robert C. Steinke, Divyakant Agrawal, and Amr El Abbadi. Epidemic algorithms for replicated databases. *IEEE Trans. Knowl. Data Eng.*, 15(5):1218–1238, 2003.

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Dep. of Computer Science, Cornell University, Ithaca, New York (USA), May 1994.

[HT98] K. Hasegawa and M. Takizawa. Object-based locking protocol for replicated objects. In *13th International Conference on Information Networking (ICOIN '98)*, pages 398–402. IEEE Computer Society, Jun 1998.

[IBDdJM+05] Luis Irún-Briz, Hendrik Decker, Rubén de Juan-Marín, Francisco Castro-Company, Jose E. Armendáriz-Iñigo, and Francesc D. Muñoz-Escoí. MADIS: A slim middleware for database replication. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par*, volume 3648 of *Lecture Notes in Computer Science*, pages 349–359. Springer, 2005.

[IMDBA03] L. Irún, F. Muñoz, H. Decker, and J. M. Bernabéu-Aubán. COPLA: A platform for eager and lazy replication in networked databases. In *5th Int. Conf. Enterprise Information Systems (ICEIS'03)*, volume 1, pages 273–278, April 2003.

[JBE95] J. Jing, O. Bukhres, and A. Elmagarmid. Distributed lock management for mobile transactions. In *15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 118–126. IEEE Computer Society, Jun 1995.

[JPPMA02] Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Gustavo Alonso. Non-intrusive, parallel recovery of replicated data. In *SRDS*, pages 150–159. IEEE Computer Society, 2002.

[JPPMKA02] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Bettina Kemme, and Gustavo Alonso. Improving the scalability of fault-tolerant database clusters. In *ICDCS*, pages 477–484, 2002.

[KA00a] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB*, pages 134–143. Morgan Kaufmann, 2000.

[KA00b] 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.

[KBB01] Bettina Kemme, Alberto Bartoli, and Özalp Babaoglu. Online reconfiguration in replicated databases based on group communication. In *DSN*, pages 117–130. IEEE Computer Society, 2001.

[Kem00] B. Kemme. *Database Replication for Clusters of Workstations (ETH Nr. 13864)*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 2000.

[KPA+03] Bettina Kemme, Fernando Pedone, Gustavo Alonso, André Schiper, and Matthias Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Trans. Knowl. Data Eng.*, 15(4):1018–1032, 2003.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[LKPMJP05] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD Conference*, 2005.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151, 1987.

[Lyn96] Nancy A. Lynch. *Distributed Systems*. Morgan Kaufmann Publishers, 1996.

[MEIBG+01] F.D. Muñoz-Escoí, L. Irún-Briz, P. Galdámez, J.M. Bernabéu-Aubán, J. Bataller, and M.C. Bañuls. GlobData: Consistency protocols for replicated databases. In *YUFORIC'2001*, pages 97–104. IEEE Computer Society, 2001.

[Ora97] Oracle. Oracle8(tm) server replication, concepts manual. Technical report, 1997.

[Ori05] Orion. Vision solutions: Orion integrator. Accessible in URL: `http://www.orionintegrator.com`, 2005.

[PA04] 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.

[Ped99] F. Pedone. *The database state machine and group communication issues (Thèse N. 2090)*. PhD thesis, École Polytecnique Fédérale de Lausanne, Lausanne, Switzerland, 1999.

[Pee02] PeerDirect. Overview & comparison of data replication architectures (white paper), November 2002.

[PGS98] Fernando Pedone, Rachid Guerraoui, and André Schiper. Exlpoiting atomic broadcast in replicated databases. In *Euro-Par*, 1998.

[PMJPKA00] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. Scalable replication in database clusters. In Maurice Herlihy, editor, *DISC*, volume 1914 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2000.

[PMJPKA05] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. Consistent database replication at the middleware level. *ACM Transactions on Computers*, 2005. In Print.

[Pos05] PostgreSQL. The world's most advance open source database web site. Accessible in URL: `http://www.postgresql.org`, 2005.

[PST+97] Karin Petersen, Mike Spreitzer, Douglas B. Terry, Marvin Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, pages 288–301, 1997.

[RBSS02] Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, and Heiko Schuldt. Fas - a freshness-sensitive coordination middleware for a cluster of olap components. In *VLDB*, pages 754–765, 2002.

[RMA+02] Luís Rodrigues, Hugo Miranda, Ricardo Almeida, João Martins, and Pedro Vicente. The globdata fault-tolerant replicated distributed object database. In Hassan Shafazand and A. Min Tjoa, editors, *EurAsia-ICT*, volume 2510 of *Lecture Notes in Computer Science*, pages 426–433. Springer, 2002.

[RSB93] Aleta Ricciardi, André Schiper, and Kenneth P. Birman. Understanding partitions and the "no partition" assumption. In *Fourth Workshop on Future Trends of Distributed Systems*. IEEE Computer Society, Sep 1993.

[SAS+96] Jeff Sidell, Paul M. Aoki, Adam Sah, Carl Staelin, Michael Stonebraker, and Andrew Yu. Data replication in mariposa. In Stanley Y. W. Su, editor, *ICDE*, pages 485–494. IEEE Computer Society, 1996.

[Sch81] G. Schlageter. Optimistic methods for concurrency control in distributed database systems. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 125–130. IEEE Computer Society, 1981.

[Sha93] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Comput. Surv.*, 25(3):225–262, 1993.

[Sto79] Michael Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Trans. Software Eng.*, 5(3):188–194, 1979.

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[TW05] TPC-W. Transaction processing performance council. Accessible in URL: `http://www.tpc.org`, 2005.

[WK05] Shuqing Wu and Bettina Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*, pages 422–433. IEEE Computer Society, 2005.

[WPS+00] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In *ICDCS*, pages 464–474, 2000.

[WSP+00] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, pages 206–217, October 2000.

[XRHS99] M. Xiong, K. Ramamritham, J. Haritsa, and J.A. Stankovic. MIRROR: A state-conscious concurrency control protocol for replicated real-time databases. In *5th IEEE Real-Time Technology and Applications Symposium*, pages 100–110. IEEE Computer Society, Jun 1999.