Universidade do Minho
Escola de Engenharia

Alfrânio Tavares Correia Júnior

Practical Database Replication

Outubro de 2010

Alfrânio Tavares Correia Júnior **Practical Database Replication**

UMinho | 2010

Universidade do Minho
Escola de Engenharia

Alfrânio Tavares Correia Júnior

Practical Database Replication

Tese de Doutoramento em Informática

Trabalho efectuado sob a orientação do
Professor Doutor Rui Carlos Oliveira

Outubro de 2010

Acknowledgments

Over this six-year journey, many people have crossed my path. New friendships have been born and others have become stronger. So I want you to know that this work would not be possible without your support. Most importantly, I want you to know that these years were crucial for me not only due to the knowledge that I have acquired with your help but also due to the life experience that they have represented. I have no words to thank you for this opportunity.

I want to thank everybody from the GSD for the wonderful work environment, in particular, Luí Soares, José Orlando, and António Sousa for the support when needed and the insightful discussions. Thank you, Rui, for this opportunity and for your friendship and advice.

Thank God and Alfrânio and Nadil to allow me to be alive; this thesis is yours too.

Abstract

Software-based replication is a cost-effective approach for fault-tolerance when combined with commodity hardware. In particular, shared-nothing database clusters built upon commodity machines and synchronized through eager software-based replication protocols have been driven by the distributed systems community in the last decade.

The efforts on eager database replication, however, stem from the late 1970s with initial proposals designed by the database community. From that time, we have the distributed locking and atomic commitment protocols. Briefly speaking, before updating a data item, all copies are locked through a distributed lock, and upon commit, an atomic commitment protocol is responsible for guaranteeing that the transaction's changes are written to a non-volatile storage at all replicas before committing it. Both these processes contributed to a poor performance.

The distributed systems community improved these processes by reducing the number of interactions among replicas through the use of group communication and by relaxing the durability requirements imposed by the atomic commitment protocol. The approach requires at most two interactions among replicas and disseminates updates without necessarily applying them before committing a transaction. This relies on a high number of machines to reduce the likelihood of failures and ensure data resilience. Clearly, the availability of commodity machines and their increasing processing power makes this feasible.

Proving the feasibility of this approach requires us to build several prototypes and evaluate them with different workloads and scenarios. Although simulation environments are a good starting point, mainly those that allow us to combine real (e.g., replication protocols, group communication) and simulated-code (e.g., database, network), full-fledged implementations should be developed and tested. Unfortunately, database vendors usually do not provide native support for the development of third-party replication protocols, thus forcing protocol developers to either change the database engines, when the source code is available, or construct in the middleware server wrappers that intercept client requests otherwise. The former solution is hard to maintain as new database releases are constantly being produced, whereas the latter represents a strenuous development effort as it requires us to rebuild several database features at the middleware.

Unfortunately, the group-based replication protocols, optimistic or conservative, that had been proposed so far have drawbacks that present a major hurdle to their practicability. The optimistic protocols make it difficult to commit transactions in the presence of hot-spots, whereas the conservative protocols have a poor performance due to concurrency issues.

In this thesis, we propose using a generic architecture and programming interface, titled GAPI, to facilitate the development of different replication strategies. The idea consists of pro-

viding key extensions to multiple DBMSs (Database Management Systems), thus enabling a replication strategy to be developed once and tested on several databases that have such extensions, i.e., those that are replication-friendly. To tackle the aforementioned problems in group-based replication protocols, we propose using a novel protocol, titled AKARA. AKARA guarantees fairness, and thus all transactions have a chance to commit, and ensures great performance while exploiting parallelism as provided by local database engines. Finally, we outline a simple but comprehensive set of components to build group-based replication protocols and discuss key points in its design and implementation.

Resumo

A replicação baseada em software é uma abordagem que fornece um bom custo benefício para tolerância a falhas quando combinada com hardware commodity. Em particular, os clusters de base de dados "shared-nothing" construídos com hardware commodity e sincronizados através de protocolos "eager" têm sido impulsionados pela comunidade de sistemas distribuídos na última década.

Os primeiros esforços na utilização dos protocolos "eager", decorrem da década de 70 do século XX com as propostas da comunidade de base de dados. Dessa época, temos os protocolos de bloqueio distribuído e de terminação atómica (i.e. "two-phase commit"). De forma sucinta, antes de actualizar um item de dados, todas as cópias são bloqueadas através de um protocolo de bloqueio distribuído e, no momento de efetivar uma transacção, um protocolo de terminação atómica é responsável por garantir que as alterações da transacção são gravadas em todas as réplicas num sistema de armazenamento não-volátil. No entanto, ambos os processos contribuem para um mau desempenho do sistema.

A comunidade de sistemas distribuídos melhorou esses processos, reduzindo o número de interacções entre réplicas, através do uso da comunicação em grupo e minimizando a rigidez os requisitos de durabilidade impostos pelo protocolo de terminação atómica. Essa abordagem requer no máximo duas interacções entre as réplicas e dissemina actualizações sem necessariamente aplicá-las antes de efectivar uma transacção. Para funcionar, a solução depende de um elevado número de máquinas para reduzirem a probabilidade de falhas e garantir a resiliência de dados. Claramente, a disponibilidade de hardware commodity e o seu poder de processamento crescente tornam essa abordagem possível.

Comprovar a viabilidade desta abordagem obriga-nos a construir vários protótipos e a avaliá-los com diferentes cargas de trabalho e cenários. Embora os ambientes de simulação sejam um bom ponto de partida, principalmente aqueles que nos permitem combinar o código real (por exemplo, protocolos de replicação, a comunicação em grupo) e o simulado (por exemplo, base de dados, rede), implementações reais devem ser desenvolvidas e testadas. Infelizmente, os fornecedores de base de dados, geralmente, não possuem suporte nativo para o desenvolvimento de protocolos de replicação de terceiros, forçando os desenvolvedores de protocolo a mudar o motor de base de dados, quando o código fonte está disponível, ou a construir no middleware abordagens que interceptam as solicitações do cliente. A primeira solução é difícil de manter já que novas "releases" das bases de dados estão constantemente a serem produzidas, enquanto a segunda representa um desenvolvimento árduo, pois obriga-nos a reconstruir vários recursos de uma base de dados no middleware.

Infelizmente, os protocolos de replicação baseados em comunicação em grupo, optimistas ou conservadores, que foram propostos até agora apresentam inconvenientes que são um grande obstáculo à sua utilização. Com os protocolos optimistas é difícil efectivar transacções na presença de "hot-spots", enquanto que os protocolos conservadores têm um fraco desempenho devido a problemas de concorrência.

Nesta tese, propomos utilizar uma arquitetura genérica e uma interface de programação, intitulada GAPI, para facilitar o desenvolvimento de diferentes estratégias de replicação. A ideia consiste em fornecer extensões chaves para múltiplos SGBDs (Database Management Systems), permitindo assim que uma estratégia de replicação possa ser desenvolvida uma única vez e testada em várias bases de dados que possuam tais extensões, ou seja, aquelas que são "replication-friendly". Para resolver os problemas acima referidos nos protocolos de replicação baseados em comunicação em grupo, propomos utilizar um novo protocolo, intitulado AKARA. AKARA garante a equidade, portanto, todas as operações têm uma oportunidade de serem efectivadas, e garante um excelente desempenho ao tirar partido do paralelismo fornecido pelos motores de base de dados. Finalmente, propomos um conjunto simples, mas abrangente de componentes para construir protocolos de replicação baseados em comunicação em grupo e discutimos pontos-chave na sua concepção e implementação.

Contents

List of Figures xi
List of Tables xiii

1 Introduction **1**
1.1 Problem Statement 2
1.2 Contributions 3
1.3 Organization 4

2 On Relational Database Replication **7**
2.1 Introduction 7
2.2 Relational Database Concepts 7
2.2.1 Relational Database 7
2.2.2 Transactions 9
2.2.3 Isolation Levels 9
2.2.4 Concurrency Control 9
2.3 RDBMS Replication Protocols 10
2.3.1 Distributed Execution 10
2.3.2 Spectrum of Availability 11
2.3.3 Spectrum of Durability 11
2.3.4 Group Communication 12
2.3.5 1-Copy Equivalence 13
2.3.6 Caching & Replication 13
2.3.7 Handling Transactions 14
2.3.8 Full vs. Partial Replication 15
2.4 Survey of RDBMS Replication Protocols 16
2.4.1 Academic Protocols 16
2.4.2 Mainstream Protocols 20
2.5 A Generic Architecture for Replication 25
2.5.1 Reflector Component 26
2.5.2 Replicator Component 28
2.6 Summary 29

3 AKARA: Replication Protocol **35**
3.1 Introduction . . . 35
3.2 Group-based Protocols . . . 36
3.2.1 NODO: Replication Protocol . . . 37
3.2.2 Active replication . . . 39
3.2.3 Database State Machine (DBSM) and Postgres-R (PGR) . . . 40
3.3 The AKARA Protocol . . . 42
3.3.1 Intuition . . . 42
3.3.2 Algorithm . . . 44
3.4 Evaluation . . . 45
3.4.1 Simulation Environment . . . 45
3.4.2 Results . . . 47
3.5 Open Issues . . . 50
3.6 Conclusion . . . 51

4 GAPI: GORDA API **53**
4.1 Introduction . . . 53
4.2 Related Work . . . 54
4.2.1 Reflective Architectures . . . 54
4.2.2 Design Patterns . . . 56
4.3 Reflector: Replication-friendly Database Support . . . 57
4.3.1 Target Reflection Domain . . . 57
4.3.2 Processing Stages . . . 57
4.3.3 Processing Contexts . . . 59
4.3.4 Synchronization . . . 59
4.3.5 Base-level and Meta-level Calls . . . 60
4.3.6 Exception Handling . . . 61
4.3.7 Attachment, Referenceable, Equality, and Hash Code . . . 61
4.4 Case Studies . . . 62
4.4.1 Primary-Backup . . . 62
4.4.2 State-machine . . . 64
4.4.3 Certification-based Approaches . . . 65
4.4.4 Satellite Database . . . 66
4.4.5 Database Caching . . . 68
4.5 Evaluation . . . 69
4.5.1 Implementation Effort . . . 70
4.5.2 Performance . . . 72
4.6 Conclusions . . . 73

5 Designing and Integrating Components **75**
5.1 Introduction . . . 75
5.2 Pluggable Replication Protocols . . . 76
5.2.1 Group-based Replicator . . . 76

5.2.2 GCS: Communication and Coordination Support . . . 77
5.3 Transaction's Read-set and 1-SR . . . 79
5.3.1 Model . . . 80
5.3.2 Locks . . . 81
5.3.3 Read-set, Write-set, and Write-values . . . 84
5.3.4 Open Issues . . . 87
5.4 Reading Your Writes . . . 88
5.5 Conclusion . . . 89

6 Conclusion **91**
6.1 Research Assessment . . . 91
6.2 Future Directions and Open Issues . . . 92

References **94**

A Requirements for Replication-friendly Databases **105**
A.1 Requirements . . . 105

B Code for the Query Cache Plugin **107**

List of Figures

2.1 Application/Broadcast Layering 12
2.2 Proposal of a Generic Architecture for Database Replication 26

3.1 Notation of States, Transitions and Queues. 37
3.2 States, Transitions, and Queues in NODO. 38
3.3 States, Transitions, and Queues in DBSM. 41
3.4 States, Transitions, and Queues in PGR. 42
3.5 States, Transitions, and Queues in AKARA. 43
3.6 AKARA Algorithm. 46
3.7 Performance of DBSM, PGR, and NODO. 47
3.8 Latency & Abort in DBSM, PGR, and NODO. 48
3.9 Performance of DBSM, NODO and AKARA-25. 49

4.1 Major Meta-level Interfaces: Processing Stages and Contexts. 58
4.2 Primary-backup Replication using GAPI. 63
4.3 State-machine Replication using GAPI. 64
4.4 Certification-based Replication using GAPI. 66
4.5 Satellite Databases and Plugins using GAPI. 68
4.6 Performance Results using GAPI. 73

5.1 Replicator Architecture 76
5.2 Hierarchical Path. 80

List of Tables

2.1 Differences between Replication and Caching 13
2.2 Replication Taxonomy 30
2.3 Classification of Replication Protocols: Concepts 31
2.4 Architecture and Database's Features Exploited. 32
2.5 Notes on the Replication Protocols 33

3.1 Analysis of AKARA. 49

Chapter 1

Introduction

Redundancy is a key element to provide fault-tolerant applications with increased performance. Particularly in environments where commodity machines may easily be deployed, fault tolerance by means of software components has attractive features.

Shared-disk clusters provided by mainstream solutions such as MS-SQL Server Cluster, DB2 Cluster, and Oracle RAG improve database server performance but require a centralized storage system to ensure data resilience. Although the database engine can be accessed through different machines, storage issues can endanger the entire approach.

Shared-nothing clusters are a well-known and cost-effective approach to database server scalability, in particular, with highly intensive read-only workloads typical of many 3-tier web-based applications. The common reliance on a centralized component and a simplistic propagation strategy employed by mainstream solutions, however, lead to poor scalability with traditional on-line transaction processing (OLTP), where the update ratio is high. Such approaches also pose an additional obstacle to high availability while introducing a single point of failure.

Mainstream approaches rely on lazy replication through a primary replica to improve performance and fault tolerance. Generally, update transactions are processed by a primary replica and upon commit, clients are immediately notified. Eventually, updates are disseminated to backup replicas, a process that does not have impact on the response time perceived by clients. Unfortunately, this technique does not allow automatic failover as backup replicas may get behind.

A variety of eager database replication protocols has been proposed to circumvent this problem, and two different aspects can be used to analyze them: (*i*) the number of messages exchanged to disseminate updates and, (*ii*) the degree of durability provided.

Some protocols send a message for each updated data item. In most cases, such protocols exchange a large volume of messages among replicas and fall prey to the high number of deadlocks. Gray et al., in the paper "Dangers of Replication and a Solution," point out that a replicated database with n copies stored over n sites can have a deadlock rate proportional to n^3, which is impractical.

Regarding durability, some protocols ensure that transactions' updates are written into a non-volatile storage at all available replicas before relaying to clients their success – commit. Despite being quite interesting from the viewpoint of fault tolerance, this approach clearly may present severe performance problems.

On the other hand, database replication protocols based on group communication, or simply group-based replication protocols, attempt to achieve both performance and fault tolerance by reducing the number of interactions amongst replicas, eliminating distributed deadlocks and relaxing durability. In such systems, interactions occur upon request to commit and require two group communication steps at most. At the same time, network durability is provided, which means that updates are ensured to be disseminated to all available replicas but not readily applied before committing a transaction and replying to a client.

1.1 Problem Statement

The increasing interest in providing eager replication on database clusters is reflected in the number of researches and prototypes that have been developed in this area. However, the lack of native support from database vendors for third-party replication forces proponents of those solutions to either modify database engines or to develop, in middleware, server wrappers that intercept client requests. Unfortunately, the modification of the database engine is hard to maintain and, in many cases, simply impossible due to unavailability of source code. On the other hand, a middleware wrapper, which implements replication and redirects requests to the actual underlying database, represents a large development effort and introduces an additional communication step, thereby decreasing the performance.

Furthermore, although group-based replication protocols have shown to be a feasible approach to provide both performance and fault tolerance, in practice, the available proposals present hurdles that are difficult to overcome.

In short, the proposed protocols can be classified into two sets: optimistic and pessimistic. In the optimistic set, a transaction is submitted to a replica and during its execution is synchronized with concurrent transactions submitted to the same replica, according to the concurrency control mechanism provided by the database engine. Upon commit, a termination protocol is initiated to decide whether the transaction should abort due to conflicts with concurrent transactions running on remote replicas or should commit nevertheless. Clearly, this makes it difficult to commit long-running transactions or transactions in workloads with hot-spots, thus resulting in high abort rates.

On the other hand, the conservative set guarantees that transactions do not abort, unless this had been requested by a client. The protocol ensures this by establishing an ordered execution amongst conflicting transactions. For instance, before starting executing, a transaction is labeled according to its conflict classes, which represent tables that are going to be read and updated during its execution. Upon begin, it acquires a total order position according to its conflict classes and starts executing at the replica it was submitted to when it is at the head of all its conflict classes. Clearly, conflict classes based on tables have a negative impact on performance, but a fine-grained class may be hard to determine and may easily lead to classification mistakes, thus resulting in database integrity issues.

Both approaches have issues with specific subsets of real-world workloads and do not handle data definition statements (e.g., create table, add column), which is a major hurdle for their acceptance. Such limitations are deep-rooted, and working around them requires in-depth under-

standing of protocols and changes to applications.

Despite the easiness introduced in the application development on replicated databases with One-Copy Serializability by avoiding the need for programmers to handle interleaving amongst transactions to ensure data integrity, this fact has not drawn much attention from the distributed systems community. Consequently, existing proposals are limited as they either require each replica to provide Serializability (SR) or are prone to generate barely traceable integrity issues, ultimately risking the entire system. Furthermore, the growing interest in databases with Snapshot Isolation (SI) makes it extremely important to understand how to provide 1-SR and at the same time increase performance by not blocking or aborting read-only transactions.

1.2 Contributions

This thesis has been developed in the context of the GORDA Project funded by the European Community and targets database clusters with commodity hardware under OLTP workloads. It presents three major contributions.

AKARA First, it proposes a novel protocol titled AKARA, that combines multiple transaction execution mechanisms and replication techniques and then shows how AKARA avoids identified pitfalls in previously proposed group-based replication protocols. Experimental results show that AKARA is a promising approach to providing both performance and fault tolerance on database clusters.

GAPI Second, this thesis proposes a replication-friendly database support named GORDA Architecture and Programming Interface (GAPI) and enables different replication strategies to be implemented once and be deployed in multiple DBMSs (Database Management Systems). This is achieved by proposing a reflective interface for transaction processing instead of relying on client interfaces or ad-hoc server extensions. This research further shows that the proposed approach is cost-effective that it enables the reuse of replication protocols or components in multiple DBMSs as well as potentially efficient, as it allows close coupling with DBMS internals.[1]

Building blocks Finally, a generic and simple set of building blocks which proposes to faciliate the development of group-based replication protocols is introduced. Key implementation details, decisions, and orchestration of this set are discussed, particularly on how common abstract processes in the database community, i.e., capture, apply, and distribution processes, should be designed and integrated to provide a full-fledged solution. It also defines the read-set and discusses how to extract it in order to provide 1-SR on databases with different consistency criteria (e.g. SR and SI).[2]

[1]The rendering of the GAPI in both the SleepyCat and PostgreSQL was done by the author whereas the other renderings were a joint work with the GORDA's team.

[2]The implementation of the jGCS was done by one of the GORDA's partners.

1.3 Organization

This thesis is organized as follows. Chapter 2 surveys database replication protocols from both industry and academia perspectives to understand the set of extensions that a DBMS should provide to have a generic interface for replication. Chapter 3 analyzes in detail group-based replication protocols and presents AKARA, a novel replication protocol that combines multiple transaction execution mechanisms and replication techniques to circumvent identified shortcomings of previously proposed group-based replication protocols. Chapter 4 presents the key concepts behind the GAPI and shows several examples on how it might be used not only to build different replication protocols but also to extend the DBMSs in general. Chapter 5 describes building blocks to provide full-fledged replication protocols and discusses key implementation details. Finally, Chapter 6 concludes this thesis and puts forward the proposition of further work.

Bibliography

[1] A. Sousa, L. Soares, A. Correia, F. Moura, and R. Oliveira. Evaluating database replication in ESCADA (Position Paper). In *Proc. of the Workshop on Dependable Distributed Data Management, SRDS*, 2004.

[2] A. Correia, A. Sousa, L. Soares, F. Moura, and R. Oliveira. Revisiting Epsilon Serializability to improve the Database State Machine (Extended Abstract). In *Proc. of the Workshop on Dependable Distributed Data Management, SRDS*, 2004.

[3] A. Correia, A. Sousa, L. Soares, J. Pereira, F. Moura, and R. Oliveira. Group-based Replication of On-line Transaction Processing Servers. In *LADC*, 2005.

[4] A. Correia. Towards an Architecture for Generic Database Replication. In *WTD - Third Workshop on Theses and Dissertations in Dependable Computing, collocated at LADC*, 2005.

[5] A. Sousa, J. Pereira, L. Soares, A. Correia, L. Rocha, R. Oliveira, and F. Moura. Testing the Dependability and Performance of GCS-Based Database Replication Protocols. In *DSN*, 2005.

[6] A. Sousa, A. Correia, F. Moura, J. Pereira, and R. Oliveira. Evaluating Certification Protocols in the Partial Database State Machine. In *ARES*, 2006.

[7] R. Oliveira, J. Pereira, A. Correia Jr, and E. Archibald. Revisiting 1-Copy Equivalence in Clustered Databases. In *ACM SAC*, 2006.

[8] J. Grov, L. Soares, A. Correia Jr., J. Pereira, R. Oliveira, and F. Pedone. A Pragmatic Protocol for Database Replication in Interconnected Clusters, 2006. In *PRDC*, 2006.

[9] N. Carvalho, A. Correia Jr., J. Pereira, L. Rodrigues, R. C. Oliveira, and S. Guedes. On the use of a reflective architecture to augment database management systems. *J. UCS*, 13(8):1110–1135, 2007.

[10] A. Correia, J. Orlando, L. Rodrigues, N. Carvalho, R. Oliveira, and S. Guedes. Gorda: An Open Architecture for Database Replication. In *IEEE NCA*, 2007.

[11] M. Matos, A. Correia Jr., J. Pereira, and R. C. Oliveira. Serpentine: adaptive middleware for complex heterogeneous distributed systems. In *SAC*, 2008.

[12] A. Correia, J. Pereira, and R. C. Oliveira. AKARA: A Flexible Clustering Protocol for Demanding Transactional Workloads. In *DOA*, 2008.

[13] A. Correia, J. Pereira, L. Rodrigues, R. Oliveira, and N. Carvalho. Practical Database Replication. In *Replication: Theory and Practice*. Springer, 2010.

[14] Distributed System Group (UMinho). GORDA API. http://gorda.di.uminho.pt/community/gapi/, 2008.

[15] Distributed System Group (UMinho). GORDA API on PostgreSQL. http://gorda.di.uminho.pt/community/pgsqlg/, 2008.

[16] Distributed System Group, (UMinho). New version of the GORDA API. https://launchpad.net/gapi, 2008.

[17] Distributed System Group, (UMinho). Perfekt - Framework for benchmarks. https://launchpad.net/perfekt, 2008.

Chapter 2

On Relational Database Replication

2.1 Introduction

Replication is a key factor for improving performance, by leveraging parallel computation with different replicas, and for increasing availability by redirecting clients to operational replicas. Many database replication protocols have been proposed and implemented, and this work surveys existing proposals, from both industry and academia perspectives. We do not attempt, however, to present a full list of all existing replication approaches.

Both lazy and eager replication protocols are presented, and their implementations for mainstream database management systems and existing proposals of algorithms are briefly surveyed. The former protocol is known as asynchronous or deferred update protocol [58]. Although this protocol provides only a limited form of fault-tolerance, it is available for the vast majority of databases and accounts for most installations of replicated databases. The latter is known as synchronous replication and shows up in a number of commercial offers, ranging from conventional distributed database protocols to the RAIDb approach [33] known as Sequoia from Continuent [35].

The rest of this chapter is organized as follows. Section 2.2 and Section 2.3 present concepts on relational databases and replication protocols. Section 2.4 surveys mainstream replication protocols such as Sequoia and Sybase Replication Server and research on both eager and lazy protocols. Section 2.5 provides a description of an architecture for practical database replication. Section 2.6 classifies the database replication protocols and presents conclusions of the chapter.

2.2 Relational Database Concepts

2.2.1 Relational Database

The term *Relational Database* was coined by Edgar Codd in 1970 and defines a model where data is organized in relations, which are typically known as tables. A Relational Database Management System (RDBMS or simply DBMS) manages the access to relations and provides an interface used by clients to send statements (i.e., queries, inserts, updates, and deletes) written in

a high level language such as SQL (Structured Query Language) [51].

The execution of a query goes through the following stages [51, 86]:[1]

- *Parser* - At this stage, a query written in a high-level language (e.g., SQL) passes through a lexical, syntactic, and semantic analysis. On completion of the process, the query is transformed into an internal representation based on some sort of query graph [112], which is more suitable for the next processing stage. Predicates (i.e., logical expressions) are transformed into a normal form, that is, a conjunctive normal form or a disjunctive normal form. In the former, a sequence of conjuncts is connected with the $\wedge$ operator where each conjunct contains one or more terms connected with the $\vee$ operator. In the latter, a sequence of disjuncts is connected with the $\vee$ operator where each disjunct contains one or more terms connected with the $\wedge$ operator.

- *Query Rewriter* - It aims at optimizing queries by rewriting expressions, subqueries, and views [57]. In other words, it simplifies expressions, unnests subqueries, and expands views. Note that this stage does not take into account, where data is stored (i.e., replicas, cache), resource usage (i.e. processor and network), and the physical state of the system (i.e., size of data items, relations, and indices). On completion of the process, the graph is transformed into an internal representation based on a relational algebra tree, which is more suitable for the next processing step. The relational algebra tree (or, simply, query tree) is produced as follows:

  - A leaf node is constructed for each base relation in the query;
  - A non-leaf node is created for each intermediate relation produced by relational algebra operations over a base relation or another intermediate result.

  The execution flow happens from the leaves to the root, which represents the result of the query.

- *Query Optimizer* - At this stage, the relational algebra tree is optimized regarding the data localization, resource usage, and physical state of the system [36]. On completion of the optimization process an execution plan is produced. In basic terms, the optimizer, by means of algorithms based on dynamic programing, greedy programming, or simulated annealing, changes the order of operations, substitutes group of operations for equivalent ones, and assesses physical operations (e.g., index scan, table scan) to minimize either resource consumption or response time.

- *Code Generation* - At this stage, the chosen plan is transformed into a low-level representation, allowing an efficient evaluation of the operations and predicates [119].

- *Query Execution* - This stage is responsible for the query execution according to the operations defined in the plan.

[1] Most DBMSs combine the parser and query rewriter stages, do not provide code generation, and use dynamic programming at the optimizer stage.

2.2.2 Transactions

A set of statements may be encompassed in a transaction, which is ended by either a commit or rollback command and executed against a DBMS. The use of a transaction eases the development of database applications as a developer can take advantage of a set of properties, titled ACID properties:

- *Atomicity* - It provides guarantees that a committed transaction has all its changes applied. On the other hand, an aborted transaction has none of its changes applied.
- *Consistency* - On completion of a transaction, the database is left in a consistent state. Otherwise, if an illegal state is detected, the transaction is aborted.
- *Isolation* - It provides guarantees regarding the interferance between concurrently executed transactions.
- *Durability* - The DBMS guarantees that committed changes are non-volatile and will survive crashes.

2.2.3 Isolation Levels

The strictest correctness criterion to handle concurrent transactions in centralized databases is known as Serializability (SR). In basic terms, this defines that a concurrent execution of a set of transactions must be equivalent to a serial execution of such set [23].

Most databases that offer SR do so by means of the Two-Phase Locking (2PL) mechanism, which grants simultaneous access to objects (e.g., data items, relations, etc.) for non-conflicting operations and blocks conflicting operations in the event of a conflict. Intuitively, two operations conflict if they are executed on behalf of concurrent transactions, access the same object, and at least one of them is a write operation.

Some databases (e.g., Oracle [39], PostgreSQL [60], MS SQL-Server [91]) trade consistency for performance by providing a weaker criterion named Snapshot Isolation (SI) [22]. By doing so, transactions execute on a database snapshot and reads never block. In other words, a transaction sees the effects of previously committed transactions but does not see the effects of concurrent transactions. Write operations are managed by the first-updater-wins rule, which means that a transaction trying to update an object must block and abort whenever a concurrent transaction updates the same object and commits first.

2.2.4 Concurrency Control

Concurrency control aims at providing isolation and the mechanisms used to do so can be classified as [5, 136]:

- *Optimistic* - A transaction is executed without *a priori* coordination among concurrent transactions. Upon commit, a certification procedure is required to check if the transaction violates a consistency criterion. If it does, the transaction is aborted. Otherwise, it is committed [81].

- *Pessimistic* - Operations of a transaction that conflict with concurrent transactions are blocked to avoid violating a consistency criterion.

Concurrency control in relational databases has been the subject of a large body of reaserch. For a comprehensive reference, see [23, 141].

2.3 RDBMS Replication Protocols

2.3.1 Distributed Execution

The need to increase availability during failures and the fact that data and its access may be inherently distributed led to the use of Distributed Database Management Systems (DDBMS) and distributed executions [86] along with replication. In such cases, however, the challenges to provide concurrency control, primarily due to communication and computer latency, increase.

In initial attempts, a concurrency control was provided through the use of a distributed lock manager. When a data item needed to be accessed, either to be updated or read, a message was sent to the distributed lock manager that acquired the appropriate locks. Upon commit, the system ensured that all replicas would commit the changes. For example, due to the lack of resources (e.g., memory or disk space), a replica might need to rollback the changes. Thus, a Two-Phase commit (2-PC) protocol [23,141] would guarantee that either all sites commit or none of them do so. The use of a distributed lock manager might result in distributed deadlocks [80] and thus harm the performance of the system as a whole [58].

In the 2-PC, when a transaction is about to commit, a coordinator, an external entity or the replica that started the transaction (i.e., delegate replica), contacts the other replicas and initiates a poll. Each replica votes "yes" or "no," notifying the coordinator whether it will be able to commmit or not commit the transaction. If a replica votes "yes," this means that it has prepared itself to commit the transaction if asked to by writing its changes to non-volatile storage. However, if a replica votes "no", the transaction had to be locally aborted for some reason, .e.g., due to a deadlock. The coordinator collects the votes, and, if all replicas voted "yes," it decides to commit the transaction and notifies the replicas asking them to commit it. Otherwise, it asks them to abort it.

There are two drawbacks with this protocol. If a replica votes "yes" and thus prepares itself to commit the transaction but does not receive any information about the coordination's decision, it holds the locks acquired on behalf of the transaction until it gets a decision from the coordinator. Therefore the longer it takes to get the coordinator's decision, the higher the likelihood of having concurrent transactions trying to access the data items on which the locks are held. Note that a replica cannot unilaterally come up with a decision because it knows nothing about the other replicas. To circumvent this problem, the 3-PC protocol was devised. For further details, see [141].

Secondly, if one of the replicas crashes before voting, the coordinator acts as if it had voted "no" and eventually decides to abort the transaction. Therefore the higher the number of replicas, the higher the likelihood that the system makes no progress, and thus availability may be harmed.

2.3.2 Spectrum of Availability

A major issue is the strategy used to update replicas while ensuring that read operations return current values [4, 23, 28, 45, 53, 55, 135, 137]. Recent results [109] indicate that under realistic conditions, the best performance is offered by protocols that eagerly update replicas.

The simplest of these is the Read One/Write All (ROWA) protocol [137], which allows read operations to be executed by reading a single replica and ensures that write operations are done by every replica. Yet, in such an environment, the failure of a single replica is sufficient to disallow write operations in the whole system.

A practical alternative is the Read One/Write All Available (ROWAA) protocol [55], in which the system continues to operate as long as there are replicas that do not fail. Each replica is in one of two possible states: on-line or off-line. A replica is on-line when its state is up to date and it is ready to execute transactions. Whenever it fails, it is excluded from the set of available replicas, and its state changes to off-line. To become on-line again, it has to go through an incarnation process, which will bring it up to date. The available replica management protocol is a separate protocol from the replication protocol and only interacts with the replication protocol in the definition of the currently available replicas.

2.3.3 Spectrum of Durability

The 2-PC or 3-PC enforce a transaction's durability by guaranteeing that the voting replicas logged its changes and therefore will be able to commit it if asked to. So when a client receives a reply, a transaction t committed at the delegate replica will eventually commit at the other available replicas. In this context, if all replicas crash, t is not lost. This approach, however, imposes too much overhead on the system and can be relaxed taking into account the degree of failures that the system may tolerate [59, 144]:

- *1-safe* - The client receives a reply as soon as the delegate replica has committed t but before t has been delivered to other replicas. If the delegate replica crashes before delivering t to the other replicas and a new delegate replica is chosen, manually or automatically, it may happen that new transactions conflict with t. Therefore, we either forget about the delegate's contents and t is lost, or bring the delegate replica on-line before the system is able to accept transactions.

- *Group-safe* - The client receives a reply as soon as t is processed at the delegate replica and delivered to other replicas but is not yet committed [72]. This allows the replication protocols to exploit asynchronous write and group commit [59].

- *2-safe* - The client receives a reply as soon as t has been logged at all available replicas and committed at the delegate replica. In this context, the system can survive a massive failure without getting t lost.

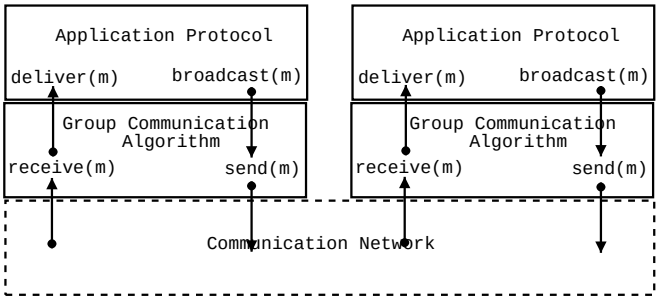


Figure 2.1: Application/Broadcast Layering

2.3.4 Group Communication

Group communication is a fundamental building block for developing fault-tolerant distributed applications. It offers strong properties on communication reliability despite failures. Besides reliability, it usually offers message ordering such as FIFO, causal, or total order [65].

Upon receiving a message, the group communication protocol processes the message and delivers it to the application, according to Figure 2.1. The stage before delivering the message imposes the desired requirements [65].

Group communication toolkits [92] provide a uniform reliable broadcast primitive satisfying the following properties [65]:

- **Validity**: If a correct[2] process $broadcasts$ a message m, then it eventually $delivers$ m.
- **Uniform Agreement**: If a process $delivers$ a message m, then all correct processes eventually $deliver$ m.
- **Integrity**: For any message m, every correct process delivers m at most once, and only if m was previously broadcast by some process.

Informally, it guarantees that all the correct processes eventually agree on the set of messages to deliver despite failures.

They also provide a total order primitive satisfying the following property [65]:

- **Total Order**: If correct processes p and q both deliver messages m and m', then they do so in the same order.

Group-based replication protocols take advantage of the specific properties of the group communication primitives, such as ordering and atomicity of messages, to eliminate the possibility of deadlocks, reduce message overhead, and increase performance. These primitives are key components to foster the adoption of software-based replication on shared-nothing architectures with commodity machines, and Chapter 3 describes in detail this family of protocols (e.g., [42, 76, 84, 99, 104, 111, 129])

[2] A correct process is a process that does not crash.

2.3.5 1-Copy Equivalence

A replicated system is 1-copy equivalent to a centralized system, i.e., non-replicated databse, if it provides the same consistency criterion. In particular, the strictest correctness criterion to handle concurrent transactions is 1-Copy Serializability (1-SR) [23] and states that a concurrent execution of a set of transactions is equivalent to a serial execution on a non-replicated database. To improve performance, however, some protocols provide 1-Copy Snapshot Isolation [84], which means that a concurrent execution of a set of transactions is equivalent to an execution on a non-replicated database providing SI.

2.3.6 Caching & Replication

Caching mechanisms exploit the principle of locality, minimizing access to slow resources by using the disk to avoid network access or memory to avoid disk access. Early optimization techniques cached information such as base relations, indexes, or a part of them. Caching of intermediate results, however, can also dramatically increase performance [64, 66]. This approach is called *materialized view* and some mainstream products such as Oracle [21, 39] and Microsoft SQL Server [54, 91] use it.

Materialized views can be used to avoid the overhead of computing complex queries. Instead of processing the query when the view is referenced, the results are automatically retrieved from the cache. Besides, it is also possible to substitute portions of a query using the view, whenever the optimizer identifies a chance to reduce computation cost.

Replication also exploits locality and, at the same time, increases resiliency. The differences between replication and caching (not only materialized views) are presented in what follows [79]:

	replication	caching
placement	server	client, server or intermediate layer
granularity	coarse	fine
storage device	usually disk	usually main memory
update protocol	propagation	usually invalidation
remove copy	explicit	implicit
mechanism	separate fetch	keep copy after use

Table 2.1: Differences between Replication and Caching

- *Placement* - The replication technique deals with relations or a subset of relations (See 2.3.8) that need to be stored on the server. On the other hand, the caching technique manipulates subsets of relations or result sets of queries that can be stored anywhere.
- *Granularity* - The relation is the grain of the replication whereas caching deals with sets of data items.

- *Storage device* - As a result of the coarse grain used in replication, information is stored on disk. In contrast, caching usually stores information in memory. Nevertheless, this decision is highly influenced by the amount of memory necessary to store the information, and even caching can use the disk as a storage area (e.g., materialized views).

- *Update protocol* - When information is updated, the changes must be propagated to all replicas. In a caching mechanism, the cache is usually invalidated.

- *Remove copy* - The "copied information" is transparently removed from the cache whereas in using replication, the removal requires coordination amongst the replicas.

- *Mechanism* - To access the replicated information, it is necessary to refer to the replica explicitly or implicitly. The use of the information in cache is the consequence of successive operations instead.

DBMSs such as Oracle [39] use materialized views' techniques to provide replication solutions. When a materialized view is defined and published as a replication object, the replication protocol along with the DBMS are responsible for keeping the materialized view updated. Generally speaking, when a relation is updated, either an incremental or a complete refresh is done to keep the materialized view updated. In the former case, the DBMS computes the changes to the materialized views from the changes to the relation and updates the materialized views using the computed changes.

2.3.7 Handling Transactions

2.3.7.1 Propagating Changes

According to [58], replication can be classified as synchronous (*eager replication*) or asynchronous (*lazy replication*). Eager replication protocols provide strong consistency and fault-tolerance by ensuring that updates are stable at multiple replicas before replying to clients. This may however have a noticeable impact on user-visible performance. On the other hand, lazy replication works by executing and committing each transaction at a single replica without synchronization. Other replicas are updated later by capturing, distributing, and applying the changes. Impact on user-visible performance, especially on transaction latency, is therefore reduced. Lazy replication, however, is not suitable for fault-tolerance by fail-over while ensuring strong consistency because updates can be lost after the failure of a replica.

2.3.7.2 Processing Update Transactions

According to [58], replication can also be classified as *primary copy* or *update-anywhere*. In the former case, update transactions are executed at a single replica, which results in scalability problems. In the latter case, updates can be sent to any replica at the expense of requiring complex protocols to guarantee consistency among the replicas.

2.3.7.3 Receiving Requests

From the point of view of the distributed systems community, replication can be classified according to the set of replicas that receives and processes requests from a client [128, 143]: *active*, *passive*, *semi-active* or *semi-passive*.

In the active replication, also known as the state-machine approach [82, 125], requests from a client are sent to all replicas which process them in the same deterministic way and reply back to the client. The determinism, however, is hard to achieve with complex interactive transactions and precludes the use of concurrent executions. The high resource usage due to redundancy of processing and the impossibility of concurrent executions affect the performance of the system as whole. On the other hand, response time is usually slow and failures are transparent to the clients.

In the passive replication, also known as primary-backup [58], one of the replicas (i.e., a primary or delegate) receives the requests from a client, processes them, and replies back to it. Changes gathered in the execution are propagated to other replicas (backup) either within the boundaries of a transaction (i.e., eager approach) or afterwards (i.e., lazy approach). In contrast to the active replication, failures are not transparent to the clients. It is necessary to guarantee that updates sent by the new and the faulty primary are received and applied in the same order in all replicas.

To exploit the easiness of handling failures and at the same time cope with non-deterministic requests, the semi-active and semi-passive replication techniques were proposed [116].

The semi-active behaves as the active replication when deterministic requests are processed. However, each time replicas receive a non-deterministic request, a replica called the *leader* processes it and sends the result to the other replicas, called *followers*.

The semi-passive replication technique masks failures by sending requests from a client to all replicas [128]. However, requests are processed by the primary, and backups are restricted to apply the changes gathered during the execution and disseminated to them. Both the primary and backups reply to the client.

2.3.8 Full vs. Partial Replication

A replication also can be classified as *full* or *partial*. In the full replication scenario, all sites have copies of all the relations in a database. In contrast, partial replication has the database split according to application semantics and each fragment replicated at a subset of the available sites. It exploits access locality allowing each transaction to require only a small subset of all sites to execute and commit, thus reducing processing and communication overhead associated with replication. Partial replication is an alluring technique to ensure the reliability of very large and geographically distributed databases while, at the same time, offering good performance. The advantages of partial replication have, however, to be weighed against the added complexity that is required to manage it. In fact, if the chosen configuration cannot make transactions execute locally or if the overhead of consistency protocols offsets the savings of locality, potential gains cannot be realized.

2.4 Survey of RDBMS Replication Protocols

2.4.1 Academic Protocols

Lazy Protocols

A number of research groups have worked on database replication based on the lazy propagation of updates. We present some of the most representative proposals [6, 9, 14, 27, 113].

2.4.1.1 Breitbart et al.

Breitbart et al. [27] present two lazy protocols that release the central certification and ordering but instead rely on a precise placement of the replicas.

This work assumes that all replicas implement strict two-phase locking [23] and that communication is through FIFO reliable multicast [65] (see Section 2.3.4 for further information on group communication). A copy graph is defined as a directed graph in which a set of vertices corresponds to the set of sites, and there is an edge between two vertices only if one of them has a primary copy of an item and the other one stores a secondary copy. A set of edges in the copy graph are called *backedges* if their deletion eliminates all the cycles in the copy graph.

A forest is constructed out of the copy graph and used to propagate replica updates along the edges of the forest. Transactions execute at a single site, and when they commit, the transaction's updates are forwarded to the children of the site in the given tree. Updates are applied in the order in which they are received at the site. Since updates might need to be routed through a number of intermediate sites, the protocol might result in significant message overload in the network and unnecessary propagation delays. Thus, it is also proposed that a protocol propagates updates along the edges of the copy graph itself instead of the constructed forest. To guarantee correct order of updates at commit time, transactions are assigned a system-wide unique timestamp.

2.4.1.2 Anderson et al.

In [14], Anderson et al. present a solution targeted towards large-scale replication. The authors assume a star-topology where one central site is used to handle global concurrency control. Two variations of the protocol are considered, one with optimistic and the other with pessimistic transaction execution. In general, the optimistic approach outperforms the pessimistic one. In that case, a transaction is first executed at the site receiving the request, subject to local concurrency control there. After local execution, the read-set and write-set of the transaction are submitted to the central site for global validation. Using the read- and write-sets, this validator site maintains a replication graph, which essentially represents a compressed version of a conflict graph [23, 24] for running transactions.

If the replication graph remains acyclic, after tentatively applying the changes required by the transaction being validated, the execution is successful. Thus, the write-set is multicast to all replicating sites, which can commit the updates as soon as they are received. The approach is regarded *lazy* since a positive reply to the client can be sent as soon as the execution is validated by the certifying site, i.e., some replicas may not yet have received the updates at the time of

commit. 1-SR is ensured by aborting transactions failing the validation procedure. The protocol is shown to provide good performance compared with a traditional eager replication protocol based on distributed locks, but we are not aware of any other studies comparing it with more modern replication protocols.

2.4.1.3 Amaza et al.

Conflict-aware scheduling was introduced as a way to implement lazy read-one/write-all replication that guarantees 1-SR [9]. The key to this approach is the extension of a scheduler to include conflict awareness: Transactions are directed to the replicas in such a way that the number of conflicts is reduced.

The authors consider a cluster architecture for a dynamic content site, where a scheduler distributes incoming requests to a cluster of databases and delivers responses to application servers. The scheduler forwards write operations to all the replicas and replies to the client as soon as the first reply from the replicas is received. Read operations are executed only at a single replica.

By knowing in advance the set of relations accessed on behalf of a transaction and by assigning a unique sequence number to each incoming transaction, the conflict-aware scheduler approach maintains 1-SR. In basic terms, lock requests are sent to all replicas and executed in an assigned sequence order, thus forcing conflicting transactions to execute in the same order at all replicas and enforcing strong consistency. The scheduler itself can be replicated for availability and fault tolerance reasons.

2.4.1.4 Plattner et al.

Ganymed [113] is a recent middleware-based replication approach that has the following two key aspects. The first is the separation between update and read-only transactions: Update transactions are handled by a primary replica and lazily propagated to the slaves, whereas queries are processed by any replica. The second is the use of Snapshot Isolation [22] (SI) concurrency control to improve performance of read-only transactions.

The main component of the Ganymed system is a lightweight scheduler that implements the replication protocol and load-balances transactions between a set of database replicas. Update transactions are executed at the primary replica; read-only transactions are scheduled to the slaves according to the least-pending-request, first balancing algorithm. If the latest database version is not yet available at the chosen slave node, the execution of the transaction is delayed. Furthermore, a client application can choose to execute read-only transactions at the primary replica or set a staleness threshold and access outdated versions of the database.

To ensure One-Copy Snapshot Isolation [84] (1-SI), the scheduler makes sure that write-sets of update transactions are applied at all replicas in the same order. The distribution of write-sets is handled by a FIFO update queue for every replica.

2.4.1.5 Akal et al.

Conventional lazy replication techniques do not necessarily provide up-to-date data; recent work in [6] addresses the issue of data freshness. The main idea of the proposed protocol is to exploit distributed versioning together with freshness locking to guarantee efficient replica maintenance that provides consistent execution of read-only transactions.

This work assumes a replicated database that contains objects that are both distributed and replicated among the server nodes. Each object has a primary site; updates of an object can occur only at its primary site, thus write operations on the same object are ordered according to the order given at the primary site. All database servers are partitioned into read-only sites and update sites. Read-only transactions can run at any read-only site. Read operations of the same transaction can run at different read-only sites. Local changes are propagated to secondary copies by propagation transactions. Refresh transactions bring the secondary copies at read-only sites to the freshness level specified by a read-only transaction.

Read-only transactions place freshness locks on the objects at the read-only sites to ensure that concurrent updates do not overwrite the required versions by ongoing read-only transaction. Freshness locks keep the objects accessed at a certain freshness level during the execution of the transaction. If a read-only transaction is scheduled to a site that does not yet fulfill the freshness requirements, a refresh transaction updates the objects at that site.

Regardless of the number of read-only sites, read-only transactions always see a consistent database state.

Eager Protocols

In this section, we present research studies that focus on eager replication protocols. Group-based replication protocols with optimistic concurrency control will be analyzed later in Chapter 3.

2.4.1.6 Amir et al.

By taking advantage of the Spread Group Communication Toolkit that provides atomic broadcast and deals with network partitions, Amir et al. [10, 11, 13] propose an active replication architecture that tolerates network partitions.

The architecture is structured into two layers: a Replication Server and Spread Toolkit as Group Communication Toolkit. The prototype of the Replication Server is an extension of PostgreSQL and consists of three components: (*i*) a Postgres Interceptor, (*ii*) a Semantics Optimizer and (*iii*) a Replication Engine. The Spread Group Communication Toolkit provides basic services for reliable, FIFO, total order and safe delivery as it overcomes message omission faults and notifies the replication server of changes in the membership of the currently connected servers.

Postgres Interceptor interfaces the Replication Engine with the DBMS client-server protocol. The interceptor listens for client connections, and once a client message is received, it is passed to the Semantics Optimizer. The Semantics Optimizer consists of: (*i*) a basic SQL parser that identifies queries, which are executed locally without any replication overhead and (*ii*) a primary

component checker. The Replication Engine includes the recovery module, dealing with both node and network failures, and the synchronizer algorithm [13] handling delivered actions.

The replication implements a symmetric distributed algorithm to determine the order of actions to be applied to the database where each server builds its own knowledge about the order of actions in the system. In the presence of network partitions, a set of servers is elected as the *primary component*, and the remaining servers will belong to the *non-primary component*. Actions (queries and writes) are handled depending on the server state, i.e., whether the server belongs to the primary component or not, as only servers in the primary component are allowed to execute actions against the database.

2.4.1.7 Pedone et al.

The Pronto protocol presented in [101] by Pedone and Frølund was among the first to consider building a database replication protocol without requiring modifications to the database engine, allowing it to be deployed in heterogeneous environments. The protocol is based on the primary-backup replication model and on atomic broadcast primitives.

The main idea behind the protocol is a hybrid approach that has elements of both primary-backup and active replication. A primary replica receives the transaction and executes it, but instead of sending only resulting state to backups, it broadcasts the transaction itself together with ordering information. Like an active replication, every database processes all transactions. However, the backups process transactions after the primary, which allows the backups to make the same non-deterministic choices as the primary.

The protocol proposed has a mechanism that handles replica failures. Transaction execution evolves as a sequence of epochs. When a backup replica suspects the primary to have crashed, it broadcasts a message to all database sites to change the current epoch, which will result in another database site becoming the primary. To prevent database inconsistencies, a transaction passes a validation test before committing, which ensures a transaction is only committed by a database site if the epoch in which the database site delivers the transaction and the epoch in which the transaction is executed are the same.

The client accesses the primary replica through an augmented driver that tries to connect to a new primary replica if the current one has failed.

2.4.1.8 Mishima et al.

In [93], a middleware-based replication protocol, titled Pangea, is proposed. Clients send requests to Pangea, which is responsible for correctly scheduling the requests in order to achieve 1-SI, and never contact a DBMS directly. When a request is received, a global snapshot is started by sending an innocuous read operation to all replicas. While a snapshot is being taken, concurrent transactions willing to commit need to wait, thus guaranteeing that the transaction on behalf of which the request was sent has the same snapshot at all replicas. Concurrent transactions willing to acquire a snapshot or execute an operation can proceed in "parallel."

To guarantee that all replicas will process conflicting transactions in the same order while non-conflicting transactions are executed in "parallel," Pangea selects a leader amongst the set of

replicas and exploits the leader's scheduler (an SI scheduler). Write operations are sent first to the leader, and as soon as the leader replies back, the operation is sent to the follower replicas. Then, it collects all replies and sends a reply back to the client. Conflicting operations are blocked by the leader's local scheduler, which uses the first-updater-wins rule. Read-only operations can be sent to any replica taking advantage of the global snapshot.

Any operation is parsed before being sent to any replica in order to distinguish read and write-operations, and to rewrite non-deterministic statements (e.g. statements with *random()*), thus avoiding synchronization issues among the replicas. Stored procedures are not handled by the protocol as they may hide non-deterministic operations.

2.4.1.9 Rodrigues et al.

In [120], the authors propose a replication protocol that combines group-based replication, in particular [76], and quorum-based replication to provide linearizability [67]. A transaction t is submitted to a delegate replica, executed optimistically, and subjected to the replica's concurrency control mechanism (i.e., 2-PL). Upon commit, t's id along with its write-set are disseminated through an atomic multicast, and a total order among concurrent transactions is established. When there is no transaction $t\prime$ ordered before t, write locks are atomically acquired for all data items in its write-set and an acknowledgment message is sent back to the delegate replica using a point-to-point channel. When the delegate replica has received acknowledgments from a write-quorum of replicas, a commit message is sent through a reliable broadcast. The delegate replica replies to the client as soon as the commit message is delivered, and the other replicas may asynchronously proceed with the updates. If t read or updated a data item and there is a transaction $t\prime$ ordered before t that wants to obtain a lock on the same data item, t is aborted and an abort message is sent through a reliable broadcast. Each data item has an associated version number, which is monotonically increased after an update.

A read-only transaction t is also executed optimistically at a delegate replica. Upon commit, t's read-set is disseminated through a reliable broadcast and a read-quorum of replicas is checked to see if the reads are still valid. When the message is delivered, read locks are atomically acquired for all data items in t's read-set. If there is a transaction $t\prime$ with a write lock on a data item read by t, the lock acquisition is delayed until $t\prime$ finishes its execution. When the read locks have been acquired, the versions are checked and the locks are released. If the reads are not outdated (i.e., the versions match), a positive acknowledgment is sent back to the delegate node. Otherwise, a negative acknowledgment is sent back. When the delegate node has received positive acknowledgment from a read-quorum of replicas, it commits t. Otherwise, it aborts t.

2.4.2 Mainstream Protocols

Lazy Protocols

Lazy replication is a standard feature of all mainstream database management systems [37, 38, 91, 115, 133, 139]. Usually, such systems adopt a publisher/subscriber model, i.e., the changes are

sent out by one database server ("publisher") and are received by others ("subscribers"), where six abstract processes are responsible for the replication.

The capture process extracts changes or published information from a source database or publisher and passes them to the distribution process, which routes these changes to the appropriate destination databases or subscribers. Then, a coordination process is used to detect integrity issues by checking on conflicts among concurrent changes. The apply process injects remote changes into a database. The recovery process is responsible for doing a state transfer every time a database is brought up on-line. The management process, which may be spread over the processes, is used to configure and administer the replication infra-structure.

Implementations differ in which replicas can process and publish updates to different data objects, how updates are captured, distributed, filtered, and applied, and finally how the entire system is managed.

2.4.2.1 MS SQL Server 2008

The MS SQL Server 2008 provides three replication solutions: (*i*) transactional replication, (*ii*) snapshot replication, and (*iii*) merge replication. The transactional replication is a primary-backup replication where the changes are later replicated per transaction. The snapshot replication is used during state transfers when a replica subscribes to a published object. The merge replication allows update-anywhere replication by use of conflict detection mechanisms, which are built according to the application semantic.

In what follows, we introduce the ideas behind the MS SQL Server 2008 replication architecture.

Snapshot Replication In a snapshot replication, the entire published object is captured. This solution does not monitor the updates made against an object and is normally used as a state transfer for the transactional and merge replication. Generally, the capture process is implemented by a "Snapshot Agent." Periodically, it copies the replicated object's schema[3] and data from the publisher to a snapshot folder for future use by a "Distribution Agent," which also acts as an apply process.

Transactional Replication The transactional replication uses the log as the source to capture incremental changes that were made to published objects. The capture process is implemented by a "Log Reader Agent," which copies transactions marked for replication from the log to the "Distribution Agent." Basically, the capture process reads the transaction log and queues the committed transactions marked for replication. The capture process takes place at the distribution's site, and there is one process (i.e., a log reader agent) for each database that has an object configured to be published. Then, the "Distribution Agent" reads the queued transactions and applies them to the subscribers.

[3]The schema is the object's structure and is used to rebuild it.

Merge Replication The merge replication allows the publisher and the subscribers to make updates whether connected or disconnected. When both are connected, it merges the updates. Whenever a conflict arises, a user-defined conflict resolution is started. The capture process tracks the changes at the publishers and is implemented using triggers. The distribution process is made by a "Merge Agent," which also acts as an apply process.

2.4.2.2 Oracle 10g

Oracle 10g offers two replication solutions: stream replication and advanced replication [39, 139, 145].

Stream Replication Oracle Stream enables the propagation and management of data, transactions, and events in a data stream either within a database, or from one database to another. The stream transfers published information to subscribed destinations. Based on this feature, it is possible to use Oracle Strem to provide replication.

The Oracle stream uses the log as input for a capture process, which formats the changes into modifier events and enqueues them. It captures data manipulation and definition commands.

Then, a distribution process reads the previous queue, which is located on the same database, and enqueues the events in a remote database. The queue from which the events are propagated is called the source queue, and the queue that receives the events is called the destination queue. There can be a one-to-many, many-to-one, or many-to-many relationships between source and destination queues.

At a destination database, an apply process consumes the events, applying them directly; alternatively, the apply process may dequeue the events and send them to an apply handler. Basically, the apply handler performs customized processing of the event and then applies it to the replicated object.

Advanced Replication When the replication is enabled in a database, a set of support triggers and procedures is created on the new replica. The triggers enqueue calls to remote procedures according to the executed commands, and the entries in queue are consumed by the Oracle implementation of the distribution process. See [39] for further explanation about triggers and the special triggers create for replication.

The distribution process pushes and pulls the deferred calls, propagating the changes from sources to destination databases. Then, the apply process dequeues this information and updates the subscriber. If an asynchronous configuration is used along with update-anywhere replication, a conflict detection mechanism compares the old value of the source object with the actual value of the destination object. If a conflict is detected, a conflict resolution procedure is called. Otherwise, if a synchronous configuration is used, a two-phase commit protocol guarantees consistency.

The management process performs operations such as the addition of replicas or modifications to replicated objects. Management is allowed only from a specific site know as "master definition site," and when the replication is in a suspended mode, it means that the environment does not accept transactions.

2.4.2.3 Sybase

The Sybase Replication Server is a separate product designed specifically for replication. In particular, its components are clearly separated from the activities of a source database. The Sybase Replication Server architecture, which does not use triggers, does not burden the source database.

On the Sybase Replication Server, the capture process runs as a thread ("Replication Agent") inside the database that reads the transaction log to detect changes to the objects. When it detects a modification in a relation or a stored procedure that has been marked for replication, it passes it on to the Replication Server. During this propagation, it translates the logged information into data-source independent commands understood by the Replication Server.

The distribution process is available as a service for the Replication Server and allows it to use multiple Replication Servers to create routes from the source to the destination. The direct or indirect routes that replicated changes take are pre-configured by systems administrators. Routes allow administrators to make the most efficient use of corporate networks in accordance with the constraints of their networks and the requirements of their applications.

To apply the changes, the Replication Server connects to the subscriber and then delivers the replicated information to its destination. There is no special process running at the destination. Furthermore, the Replication Server does provide data translation vehicles, called function strings, and user-defined data types that the customer can use to change the data within a command or even change the actual command.

The management tool (Replication Management Server) allows creation of connections, routes, replication definitions, etc. Furthermore, the configuration and status information on direct and indirect routes can be monitored from a central location by the Replication Server Manager.

2.4.2.4 IBM DB2

The IBM DB2 DataPropagator is a replication product for relational data, which consists of three main components: administration interfaces, capture and apply processes.

The capture process runs at the publishers and uses the transactional log to get changed data. Each published object (i.e., relation) has a corresponding "change data table" where the gathered changes are temporally stored. When the changes are applied by the subscribers, they are automatically deleted from the table. An apply process reads the "change data table" and applies the changes to subscribers.

The management is made using control tables. The replication components use control tables to communicate with each other and to manage replication tasks such as managing replication sources and targets, capturing changes, replicating changes, and tracking how many changes are replicated and how many remain to be done.

2.4.2.5 PostgreSQL

The Slony-I is the open-source "replication product" available for the PostgreSQL and implements lazy replication with a primary-backup architecture.

The capture process is designed using triggers that log the changes made against the published objects.

These changes are periodically distributed using a replication daemon that connects directly to the publisher, reads the logged changes, and forwards it to the subscribers. It allows to connect several subscribers in cascade. The subscribers connect as regular clients to a PostgreSQL site and apply the updates. The management process is neither integrated nor centralized. For that reason, the maintenance tasks must be done on each replica.

2.4.2.6 MySQL

MySQL includes a built-in lazy replication protocol that can be configured to propagate updates between replicas. Although the system only provides primary-backup replication, each slave can be configured to relay updates to other replicas, or even to behave as a primary for a partition. There is, however, no conflict resolution, and thus, it is the responsibility of the users to avoid inconsistency. Schema updates are also propagated, and thus administration can be centralized.

The capture process runs in the same thread executing operations on behalf of a client and stores information in a replication-log. The need for a special log relies on the fact that a transaction may access different storage engines (i.e., databases) with different transactional-logs, thus requiring a place where information is stored through a two-phase commit [38]. The distribution and apply processes are two threads running on each slave, which are responsible for contacting the primary replica to get the updates and applying them accordingly.

The information stored in the replication-log can have a statement-based or row-based format. In the former case, the slave process plain queries as an active replication. To preserve determinism, the master re-writes the queries or appends additional information that is used by the slave. For example, if a *random()* is used, the seed is logged with the query. Log entries in the row-based format, on the other hand, do not need any extra information and can be directly applied.

Eager Protocols

2.4.2.7 Volume Replication Protocols

Replication of disk volumes performed at the block I/O level is a straightforward and general-purpose approach to replication. By intercepting each block written by the application-designated volumes and shipping it over to a network, a remote copy is maintained ready for fail-over. The replication process is thus completely transparent to the application.

As with local mirroring, waiting for confirmation that both copies have been written before notifying the application of completion ensures consistency upon failure. This is specially relevant for database management systems that rely on write ordering, enabling them to recover by replaying logs.

The downside of this approach is that remote updates are performed with a block granularity, which, depending on the application, could represent a large network overhead. The approach is

also restricted to fail-over, as the backup copy cannot usually be used even for read-only access, due to lack of cache synchronization at the application level.

Examples of this approach can be found in the Veritas Volume Replicator [40], which is available for a number of different operating systems and in the open-source DRBD for Linux [85]. In particular, the Veritas Volume Replicator [134] shows that with sufficient network resources available and with a typical OLTP load, the approach results in overhead within 5% of a single local volume and is thus viable.

2.4.2.8 RAIDb Protocols

A Redundant Array of Inexpensive Databases (RAIDb) aims to provide better performance and fault tolerance than a single database, at low cost, by combining multiple database instances into an array of databases. Like RAID to disks, different RAIDb levels provide various cost/performance/fault tolerance trade-offs. RAIDb-0 features full partitioning; RAIDb-1 offers full replication; and RAIDb-2 introduces an intermediate solution, called partial replication, in which the user can define the degree of replication of each database relation.

RAIDb hides the distribution complexity and provides the database clients with the view of a single database like in a centralized architecture. As for RAID, a controller sits in front of the underlying resources. The clients send their requests directly to the RAIDb controller that distributes them amongst the set of RDBMS backends. The RAIDb controller gives the illusion of a single RDBMS to the clients. RAIDb controllers may provide various additional services such as load balancing, dynamic back-end addition and removal, caching, connection pooling, monitoring, debugging, logging, or security management services.

The RAIDb guarantees that replicas are updated through an active replication. Read-only operations can be executed by any replica independently, thus providing parallelism.

The RAIDb was implemented as a software solution in Sequoia middleware [32,33,35], originally known as C-JDBC. Active replication of databases was implemented in EMIC Application Cluster (EAC) owned by Continuent, which was discontinued [35, 46].

2.5 A Generic Architecture for Replication

In Figure 2.2, we outline an architecture for practical database replication. Although the architecture is widely known in mainstream lazy replication solutions, it perfectly fits any replication solution and thus can be adopted as the basis for a generic architecture for database replication as we shall describe in detail in both Chapters 4 and 5. The components of the architecture are [41, 56]:

- The application, which might be the end-user or a tier in a multi-tiered application.
- The driver, which provides a standard interface for the application and accesses the DBMS using a communication mechanism that is hidden from the application, and can be proprietary.

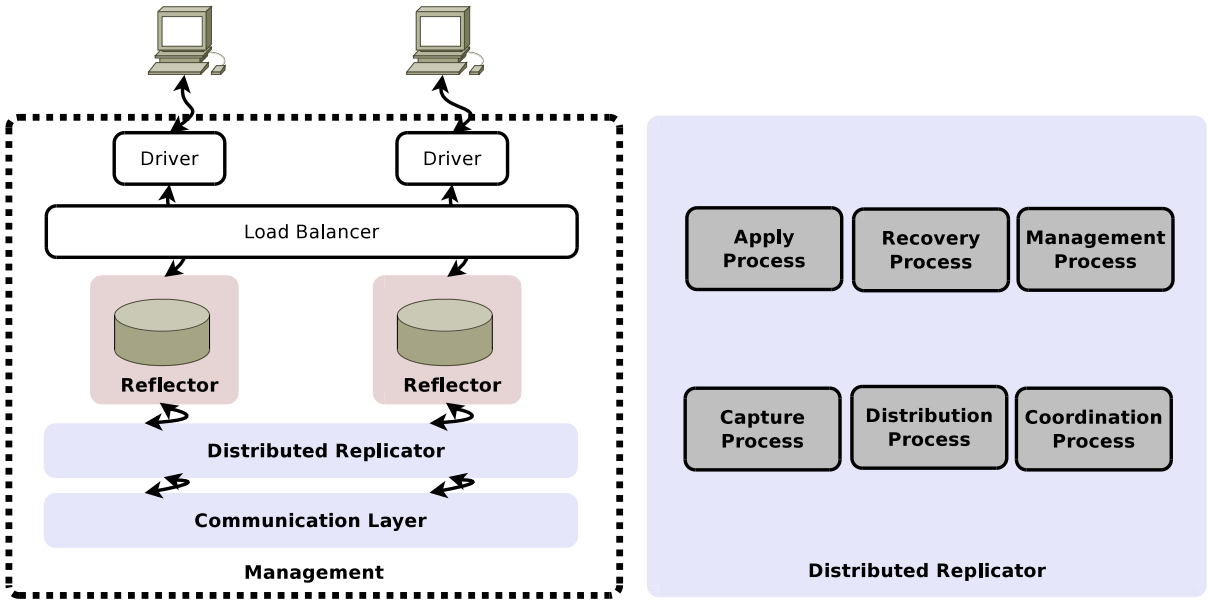


Figure 2.2: Proposal of a Generic Architecture for Database Replication

- The load balancer, which dispatches client requests to database replicas using a suitable load-balancing algorithm.
- The DBMS, which holds the database content and handles requests which query and modify data.
- The management tools, which, independent of the application, are able to control the driver, the DBMS components, and the replicator.
- The reflector, which is attached to each DBMS and allows inspection and modification of on-going transaction processing.
- The replicator, which somehow must inspect and modify on-going transactions and coordinate multiple database replicas to ensure consistency among the replicas.
- The communication layer, which is used to disseminate the changes among the replicas. See Chapter 5 for further details.

An important component of the architecture is the interface among the building blocks, which allows them to be reused in different contexts. To support as much as possible off-the-shelf and third-party tools, the call-level and SQL interfaces, and the remote database accesses protocol adhere to existing standards. For instance, the architecture can be easily mapped to a Java system, using JDBC as the call-level interface and driver specification.

2.5.1 Reflector Component

Multiple solutions have been used and proposed for interface replication protocols with DBMSs.

Replication implemented as a normal client In this approach, both the application and the replication protocol interact with the DBMS independently and exclusively through client interfaces, e.g., JDBC. This strategy is however very limited as the replication protocol is confined to propagate the updates performed by the application-initiated transactions without any control over their execution. As a consequence, the inability to suspend a third-party-initiated transaction and synchronously update the database replicas only allows it to perform asynchronous replication. An example of this approach is Slony-I [115], which provides asynchronous replication of PostgreSQL. Typically, these solutions resort to installing triggers in the underlying DBMS in order to update meta-information and gather updates.

Replication implemented as a server wrapper These implementations rely on a wrapper to the database server that intercepts all client requests by sitting between clients and the server. An example of an application of this approach is Sequoia [35]. The middleware layer presents itself to clients as a virtual database. Compared to the previous approach, implemented as a regular DBMS client, this solution offers improved functionality as it is able to intercept, parse, delay, modify, and finally route statements to target database servers. Nonetheless, it imposes additional overhead as it duplicates some of the work of the database server. The development of such infrastructure also represents a large undertaking and prevents clients from connecting directly to database servers using native privileged interfaces. It also has to rely on triggers installed in the underlying DBMS to capture relevant control information such as updates.

Replication implemented as a server patch This solution requires changes to the underlying database server. This approach has been used to implement group-based replication protocols such as the Postgres-R prototypes [76, 146]. Given that it is implemented in the DBMS kernel, the replication protocol has easy access to control information such as read- and write-sets, transaction life-cycle events, etc. It has, however, the disadvantage of requiring access to the database server source code. It also imposes a significant obstacle to portability, not only to the multiple database servers but also as the implementation evolves.

Replication using custom interfaces Databases that natively support asynchronous replication usually do so using a well-defined and documented interface. Although proprietary, this interface allows some customization and integration with third-party products when asynchronous propagation is desired but is of little use otherwise. An example of this approach is the Oracle Streams interface [15, 139], which is based on existing standards and confined to asynchronous propagation of updates.

Replication using a plugin The idea is augmenting the standard database interfaces with additional primitives that provide abstractions and reflect the database internal states such as transaction life-cycle events and processing stages (e.g., transaction parsing, optimization and execution) inside the DBMS engine. This the subject of Chapter 4, and additional information can be found at [41, 56].

2.5.2 Replicator Component

The replicator can be divided in to six abstract processes: (*i*) capture, (*ii*) distribution, (*iii*) coordination, (*iv*) apply, (*v*) recovery, and (*vi*) management.

Capture The capture process involves obtaining changes performed on replicated objects. Depending on the implementation, this process is triggered by events such as the number of changes in a published object (e.g., relation), the number of committed transactions in a publisher, and time intervals involved.

This can be easily implemented without modifications to the database management system by setting up triggers in update operations, although it will have some impact on performance. The overhead can be reduced by using the database transactional log. By retrieving information from "stable parts of the log," the impact on performance is reduced, provided that the impacting of reading from it does not have an effect upon flushing information into it.

It is also responsible for extracting data definition statements (i.e., DDL), thus working alongside the management process to change database meta-information.

Distribution The distribution process propagates changes in published objects to relevant replicas. Complex distribution scenarios, involving multiple hops, filtering, and staging areas are better addressed by implementations that decouple distribution from capture and apply.

Distribution must deal with failures in replicas and in the network, possibly including long periods of disconnected operation. Failures must not cause loss or duplication of changes. Filtering is often required due to visibility and performance reasons when coping with very large amounts of data and different access control policies. Filtering can also be required to enforce that changes to each replicated object are performed only at a designated primary replica, thus avoiding database integrity issues.

Coordination The main concern when applying changes is to ensure that no database integrity issues arise and thus database integrity is preserved. This can be easily ensured by restricting modification of each data item to a designated primary copy. Otherwise, a coordination procedure is required.

The coordination procedure may be executed before applying updates to a database or after it. For lazy replication protocols, the coordination procedure is executed after applying the updates and is rendered as an application-specific procedure for reconciliation, which is supported by sorting conflicting updates and triggering compensation actions to preserve database integrity.

Apply The apply process is responsible for injecting remote updates into the subscriber. These updates may be in a canonical format (e.g., XML) produced by a capture process, thus requiring additional processing. Finally, the apply process may do further in order to reduce processing in the publishers that may be responsible for handling several subscribers.

Management The management process is concerned with two different aspects: changes to database meta-information and changes to the replication configuration. Management can be restricted by requiring that operations are performed at a master replica and thus forbidding concurrent access by other management operations or even by user transactions.

Recovery The main issue when reconfiguring the system is the initial synchronization of new replicas. In general, the state of the subscriber's object is compared with the publisher's object and the differences are applied to the subscriber. However, to avoid an unfeasibly large amount of data being propagated between the replicas during the state transfer, a previously taken snapshot from the publisher's object could be applied off-line to the future subscriber's object before bringing up the subscriber on-line.

2.6 Summary

In this chapter, we briefly analyzed several database replication protocols, varying from mainstream solutions to academic proposals. Then, we outlined a generic architecture for database replication that could be used to model such protocols. In what follows, we highlight important aspects in the set of protocols discussed in order to understand how they fit into the concepts presented in Sections 2.2 and 2.3, and which features a DBMS should provide to support them. These are key points that guided the development of GAPI in Chapter 4.[4]

Table 2.2 presents the taxonomy used. There are, however, other taxonomies such as [58, 69, 142, 143]. There are terms presented in table that were discussed in Sections 2.2 and 2.3, but others that were not and deserve some explanation:

- Architecture - Defines how the replication protocol is integrated with the DBMS. This was already discussed in Section 2.5.1, but a different set of terminology was used:
  - *White-box*: tightly coupled and usually more efficient by exploiting database internals.
  - *Black-box*: middleware approach where the interaction with the DBMS is through standard features (e.g., triggers, procedures) and interfaces (e.g., JDBC). Usually, this allows us to use off-the-shelf databases.
  - *Gray-box*: Some key features are exposed to the middleware level in order to provide more efficient, generic solutions and specific protocols. Protocols that do not have an implementation are classified within this alternative.
- Observe Requests - The replication protocol must be able to *intercept*, *parse*, and if necessary change SQL statements, for instance, to remove non-deterministic operations such as *random()*.

[4] See in Appendix A a detailed set of features that a DBMS should provide to support a variety of replication protocols.

- Capture Data Access - The replication protocol must be able to capture the *read-set* and the *write-set* by means of non-standard interfaces (e.g., triggers).
- Transaction Life-Cycle - The replication protocol must be able to hold the execution of a *begin*, *commit*, and *abort*.
- Apply Updates - Remote replicas must be able to *apply* updates, most likely through a highly optimized interface, and sometimes with a *high priority*, which means that local transactions that conflict with it must be aborted.
- Inject Information - While applying updates, it must be possible to assign a *timestamp* to every updated object (e.g. data item) and a mapping between a *global id* and the local transaction's id to do a recovery.

Dimension	Possible Values
Isolation Level (IL)	1-SR, 1-SI, Strong 1-SR, Strong 1-SI
Concurrency Control (CC)	Optimistic, Pessimistic
Degree of Replication (DR)	Full, Partial, Caching, Materialized Views
Propagating Changes (PC)	Eager, Lazy
Processing Update Transactions (PUT)	Primary copy, Update-anywhere
Receiving Requests (RR)	Active, Passive, Semi-Active, Semi-Passive
Spectrum of Availability (SA)	ROWA, ROWAA, ROWO, Quorum
Spectrum of Durability (SD)	0-safe, 1-safe, group-safe, 2-safe
Architecture (AR)	Blackbox, White-box, Gray-box
Observe Requests (OR)	Intercept, Parse, Change
Capture Data Access (CA)	read-set, write-set
Transaction Life-Cycle (TL)	Begin, Commit, Abort
Apply Updates (AU)	Apply, High Priority
Inject Information (II)	Version, Global Identification

Table 2.2: Replication Taxonomy

The following research papers [48, 74–76, 84, 102–104, 110, 111, 146] were not discussed in Section 2.4 and will be subject of study in Chapter 3. They share the roots of a certification procedure, and for the sake of completeness are classified here. Notice, however, that Lazy Mainstream protocols are not presented as their design patterns are well-known. We also have omitted [13] Amir et al. as they consider transactions with single statements and were presented in this survey because developed one of the first prototypes to combine replication and group communication and introduced the key notion of primary partition.

Protocols	IL	CC	DR	PC	PUT	RR	SA	SD
[14] Anderson et al.	1-SR	Pessimistic, Optmistic	Partial	Lazy	Pr.-Copy	Pr.-Backup	ROWAA	1-safe
[27] Breitbart et al.	1-SR	Pessimistic	Partial	Lazy	Pr.-Copy	Pr.-Backup	ROWAA	1-safe
[9] Amaza et al.	1-SR	Pessimistic	Full	Lazy	-	Active	ROWA	1-safe
[113] Plattner et al.	1-SR	Pessimistic	Full	Lazy	Pr.-Copy	Pr.-Backup	ROWAA	1-safe
[6] Akal et al.	1-SR	Pessimistic	Partial	Lazy	Pr.-Copy	Pr.-Backup	ROWAA	1-safe
[101] Pedone et al.	1-SR	Pessimistic	Full	Eager	Pr.-Copy	Pr.-Backup + Active	RAAWAA	1-safe
[93] Mishima et al.	1-SI	Pessimistic	Full	Eager	Pr.-Copy	Semi-Active	ROWAA	2-safe
[120] Rodrigues et al.	1-SR	Optimistic	Full	Eager	Up.-anywhere	Pr.-Backup	Quorum	1-safe
[40] Volume Replication Protocols	-	-	Full	Eager	Pr.-backup	Pr.-Copy	ROWAA	2-safe
[33] RAIDb Protocols	1-SR	Pessimistic	Full, Partial	Eager	-	Active	ROWAA	0-safe, 1-safe, 2-safe
[110] [111] [48]	1-SR	Pessimistic	Full	Pr.-backup	Up.-anywhere	Passive	ROWAA	Group-safe + 1-safe
[102], [103], [104]	1-SR	Optimistic	Full	Pr.-backup	Up.-anywhere	Passive	ROWAA	Group-safe + 1-safe
[74], [75]	1-SR, 1-SI, others	Optimistic	Full, Partial	Pr.-backup	Up.-anywhere	Passive	ROWAA	Group-safe + 1-safe
[76]	1-SR	Optimistic	Full, Partial	Pr.-backup	Up.-anywhere	Passive	ROWAA	Group-safe + 1-safe
[146], [84]	1-SI	Optimistic	Full	Pr.-backup	Up.-anywhere	Passive	ROWAA	Group-safe + 1-safe

Table 2.3: Classification of Replication Protocols: Concepts

Protocols	AR	OR	CA	TL	AU	II
[14] Anderson et al.	Any as a gray-box		read-set, write-set	begin, commit, abort	apply high-priority	
[27] Breitbart et al.	Any as a gray-box		read-set, write-set	begin, commit, abort	apply high-priority	
[9] Amaza et al.	MySQL as a black-box	intercept, parse	-	-	-	-
[113] Plattner et al.	Any as a black-box	intercept, parse	-	-	-	-
[6] Akal et al.	Any as a gray-box		read-set, write-set	begin, commit, abort	apply high-priority	
[93] Mishima et al.	PostgreSQL as a black-box	intercept, parse	-	-	-	-
[101] Pedone et al.	Any as a black-box	intercept, parse	-	-	-	-
[120]	Any as a white-box	-	read-set, write-set	commit, abort	apply, high priority	timestamp
[40] Volume Replication Protocols	Any as a black-box	-	-	-	-	-
[33] RAIDb Protocols	Any as a black-box	intercept	-	-	-	-
[110]	Any as a black-box	intercept	-	-	-	-
[111]	PostgreSQL as a black-box	intercept	-	-	-	-
[48]	PostgreSQL as a gray-box	intercept	write-set	-	apply	-
[102], [103], [104]	Any as a gray-box	-	read-set, write-set	apply, high priority	commit, abort	global id
[74], [75], [76]	PostgreSQL as a white-box	-	write-set	apply, high priority	commit, abort	-
[146]	PostgreSQL as a white-box	-	write-set	apply high priority	commit, abort	global id
[84]	PostgreSQL as a black-box	intercept	-	-	-	-

Table 2.4: Architecture and Database's Features Exploited.

Protocols	Notes
[9] Amaza et al.	Although the protocol is built upon MySQL, it does not exploit any particular MySQL's feature and as such any DBMS could be used. The protocol uses an active replication approach but does not mention how to cope with non-deterministic operations (e.g., *random()*), and for that reason we introduced the need to parse queries at the middleware level.
[93] Mishima et al.	We classify it as pessimistic because conflicting concurrent transactions obey a SI local scheduler and as primary-copy because the leader replica processes write operations before the follower backup replicas, which have the order of operations given by the leader. The semi-active classification comes from the fact that the protocol is a hybrid between primary-backup replication and active-replication similarly to the leader-follower approach [116]. Furthermore, it does not require any failure detector mechanism along with a consensus to designate a new leader.
[101] Pedone et al.	The protocol does not specify exactly how read-only, transactions should be executed. We think, however, that such transactions do not need to be processed by all replicas and a primary might just reply to the client as soon as it detects that a read-only transaction is about to commit. This can be easily implemented in the client driver as part of the changes required to run the protocol. For that reason, we assume ROWAA. The protocol assumes 1-safe as the client is notified as soon as a replica commits a transaction. However, we might augment the client driver to wait for the majority of replicas and thus provide 2-safe. Finally, it is not clear how the protocol handles non-deterministic operations (e.g., *random()*).
[120] Rodrigues et al.	The Quorums are used to provide session consistency or the strongness property that is considered in our current definition of 1-SR and 1-SI. In other words, the protocol provides strong 1-SR, or simply 1-SR. The work seems to be based on [76], which uses a white-box approach, and as such we classify [120] as a white-box too. Notice that the pre-locking phase described in the protocol can be easily provided in the middleware and as such does not require any specific DBMS's feature or extension to be implemented.
[111]	Although the protocol is implemented on top of the PostgresQL, it does not exploit any specific PostreSQL's features and might be used on top of any engine.
[48]	The adaptive algorithm discussed in the paper does not need any specific DBMS's feature or extension and is completely implemented in the middleware.

Table 2.5: Notes on the Replication Protocols

Chapter 3

Scalable Replication under Demanding Workloads

3.1 Introduction

Database replication based on group communication [76, 84, 104, 111, 129] supplies the foundation for affordable and scalable clusters that eschew a shared storage infrastructure. By enabling the use of commodity machines and a variety of database solutions, it helps reduce costs associated with building fault-tolerant architectures and eases the scale-out factor [33, 113]. The result also improves on reliability when compared with most mainstream solutions, as these often reduce to lazy update or rely on centralized components.

Generically, the approach is eager and builds on the classical replicated state machine [125]: The same sequence of update operations is applied to the same initial state, thus producing a consistent replicated output and final state. The problem is then to ensure deterministic processing without overly restricting concurrent execution, which would dramatically reduce throughput.

This is achieved by executing the bulk of the transaction at a single replica and then propagating raw updates, as in a passive replication, which has the additional advantage of avoiding re-execution. A single total order broadcast for each transaction suffices for coordination, thus being able to achieve a close to linear scalability even with write-intensive loads [70]. In contrast, eager replication, based on distributed locking and atomic commit protocols, requires much finer-grained coordination and falls prey of deadlocks [58].

Protocols differ mainly in whether transactions are executed optimistically [76, 104] or conservatively [111]. In the former case, a transaction is executed by a receiving replica without a *priori* coordination with other replicas. Just before committing, replicas coordinate and check for conflicts between concurrent transactions. Transactions that would locally commit may abort due to conflicts with remote concurrent transactions. In the conservative approach, all replicas first agree on the execution order for (potentially) conflicting transactions assuring that when a transaction executes, there is no concurrent conflicting transaction being executed remotely and therefore its success depends entirely on the local database engine. Clearly, two transactions conflict if both access the same conflict class (e.g., table) and one of them updates it.

Despite the promising benchmark results, the practicality of such protocols is limited as all have disappointing performance with specific subsets of demanding real-world workloads. Namely, the optimistic approach may become impractical as long-running transactions may experience unacceptable abort rates. This makes it very hard to commit such transactions in a heavily loaded server, even when resubmission is possible, thus compromising liveness. This issue does not arise with conservative protocols. However, achieving good performance may require a careful application-specific definition of conflict classes, if possible at all, without changes to application semantics [70]. This is particularly troublesome as a labeling mistake can lead to inconsistency.

Furtheremore, simple statements that update a large number of items result in heavy network traffic in both approaches, while saving little in avoiding re-execution. The same is true for DDL statements (e.g., create index, alter table), in which extracting and applying updates may require intimate knowledge of database internals and also yield large updates. In these situations, active replication is desirable.

The challenge is, therefore, to combine the ease of use of the optimistic approach, with the fairness of the conservative approach and the straightforward implementation of the active replication. In this chapter, we answer to this challenge presenting a novel protocol target at general-purpose demanding workloads. Section 3.2 surveys the current protocols focusing on dynamic aspects, namely, on queuing that happens in different parts of the system and on the amount of concurrency that can be achieved. Section 3.3 proposes AKARA database replication protocol based on group communication. As happens with conservative protocols, AKARA is fair and takes advantage of a judicious definition of conflict classes to maximize concurrency. However, to attain the performance level of optimistic protocols, AKARA exploits the tentative execution of potentially conflicting transactions as allowed by the underlying system, i.e., by any database management system. Furthermore, as in active replication protocols, AKARA allows any deterministic statement to be actively replicated. Section 3.4 evaluates AKARA using the workload from TPC-C and shows that it provides, even with a generic-application independent (i.e., strictly syntactic) definition of conflict classes, peak performance comparable to a purely optimistic protocol while at the same time enforcing fairness. Section 3.5 discusses open issues, and Section 3.6 concludes the chapter.

3.2 Group-based Protocols

In this section, we present an overview of major approaches to database replication using group communication. We do this survey, however, with a novel twist. We focus on dynamic aspects, namely, (i) on queuing that happens in different parts of the system and (ii) on the amount of concurrency that can be achieved. Then, we contrast the original assumptions underlying such protocols with our experience with actual implementations and using the TPC-C workload [70]. This is extremely relevant, as both NODO and DBSM have been proposed as exploiting optimistic assumptions on system dynamics that we are not able to confirm in our realistic setting. The conclusion sets the motivation for proposing the AKARA protocol in Section 3.3.

Figure 3.1 introduces the notation used to depict protocol state-machines. Given the emphasis

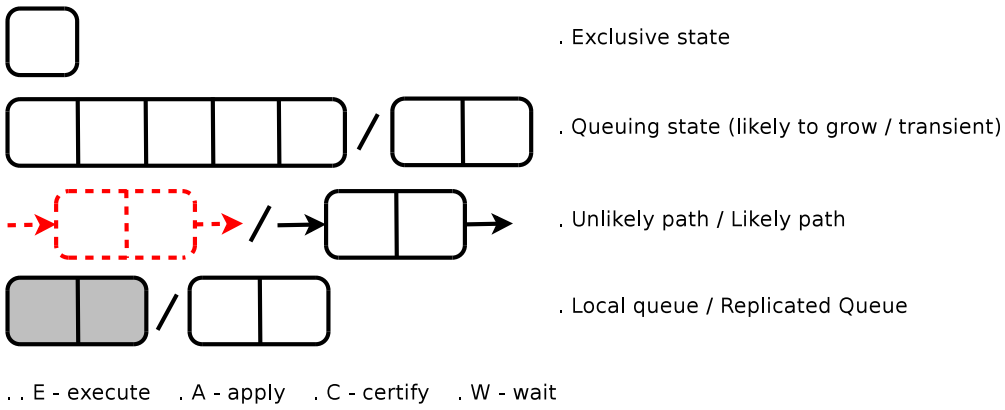


Figure 3.1: Notation of States, Transitions and Queues.

on dynamic aspects, we use different symbols for states that represent queuing and states in which, at most, a single non-conflicting transaction can be at any given time. We show also which queues are likely to grow without bound when the system is congested. When alternative paths exist, due to optimistic execution, we show which is considered to be the more likely to be executed. We make a distinction between local and replicated queues and identify relevant actions: execute, apply, certify, and wait.

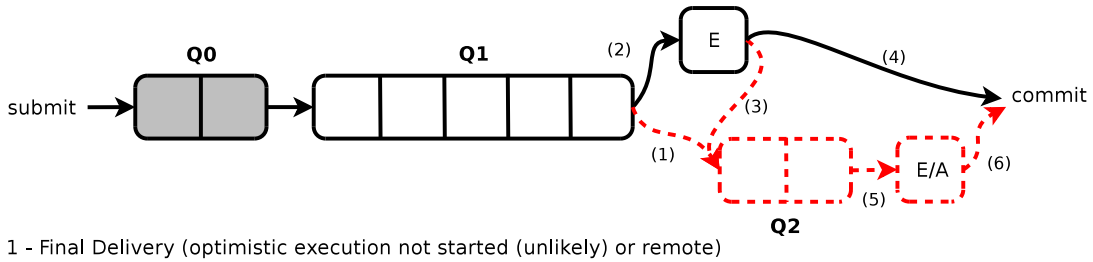
Since all protocols involve an atomic broadcast step, we use a consistent naming for queues in different protocols. $Q0$ is before the abcast, $Q1$ is between abcast and delivery, and $Q2$ is after delivery. Protocols with an optimistic execution use messages in $Q1$, which has messages with a tentative order, i.e., messages that have been optimistically delivered. Messages in $Q2$ have a final order.

3.2.1 Non-Disjoint Conflict Classes and Optimistic Multicast (NODO)

In NODO, data is a *priori* partitioned in conflict classes, not necessarily disjoint. Each transaction has an associated set of conflict classes (the data partitions it accesses), which are assumed to be known in advance. This approach requires to know the entire transaction before its execution precluding the processing of interactive transactions. Read-only transactions, however, are handled locally.

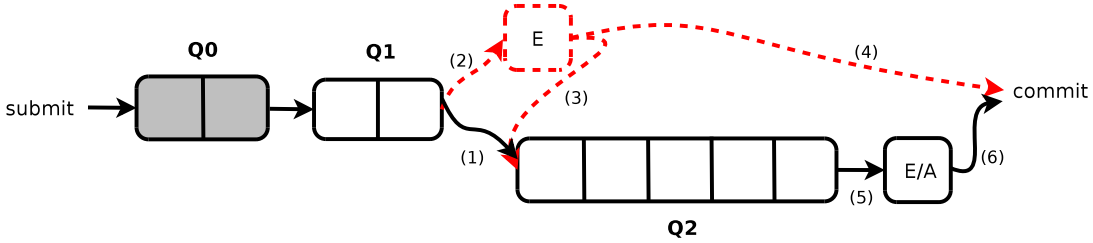
Upon submitting a transaction ($Q0$), its id and conflict classes are atomically multicast to all replicas obtaining a total order position ($Q2$). Each replica has a queue associated with each conflict class, and, once delivered, a transaction is classified according to its conflict classes and enqueued in all corresponding queues. As soon as a transaction reaches the head of all of its conflict class queues, it is executed. Transactions are executed by the replica to which they are submitted.

Clearly, the conflict classes have a direct impact on performance. The fewer the number of transactions with overlapping conflict classes, the better the interleave among transactions. Conflict classes are usually defined at the table level but can have a finer grain at the expense of a non-trivial validation process to guarantee that a transaction does not access conflict classes that



1 - Final Delivery (optimistic execution not started (unlikely) or remote)
2 - Submit transaction to optimistic exectuion
3 - Final Delivery (missed order and then rollback (unlikely))
4 - Final Delivery (correct order)
5 - Submit transaction to execution in order
6 - Execution finished in order

(a) Assumption of the designers.



1 - Final Delivery (optimistic execution not started or remote)
2 - Submit transaction to optimistic exectuion (unlikely)
3 - Final Delivery (missed order and rollback (unlikely, but it doesn't matter))
4 - Final Delivery (correct order)
5 - Submit transaction in order
6 - Execution finished in order

(b) Our assumption.

Figure 3.2: States, Transitions, and Queues in NODO.

were not previously specified.

Upon receiving the commit request, the outcome of the transaction is reliably multicast to all replicas along with the replica's changes (write-set) and a reply is sent to the client. Each replica applies the remote transaction's updates with the parallelism allowed by the initially established total order of the transaction.

The protocol ensures 1-copy serializability [23] as long as transactions are classified taking into account read/write conflicts. To achieve 1-copy snapshot isolation [84], transactions must be classified taking into account just write/write conflicts.

A transaction is scheduled optimistically if there is no conflicting transaction already ordered ($Q2$). This tentative execution may be done at the expense of an abort if a concurrent transaction is later ordered before it.

Figure 3.2(a) shows the states that a transaction goes through upon being submitted by a client. According to the designers' assumptions, the time spent in the queue ($Q1$) waiting for the total order is significant enough compared with time taken to actually execute such that it is worthwhile to optimistically execute transactions (transition 2 instead of transition 1). This

makes it possible that when a transaction is ordered, it is immediately committed (transition 4). Assuming that optimistic ordering is correct, a rollback (transition 3) is unlikely.

We have reasons, however, to believe that this assumption is invalid. The first hints for this are actually in the original description of the proposal [111]. First, the end-to-end transaction execution latency measured is larger than 50 ms, of which most certainly only less than 10% can be attributed to the latency of group communication. In fact, the protocol used for experiments, is very well known for its extremely good performance [17]. If this is true, then queuing will happen in queue $Q2$ and not in queue $Q1$. Thus, if $Q2$ is never empty, then no transaction in queue $Q1$ is eligible for optimistic execution. This is confirmed by the abort rate being always extremely low, even with a large share of conflicting update transactions [111], a hint that transition 2 is never taken.

The appropriate scenario for the NODO protocol is thus depicted in Figure 3.2(b): The optimistic path is seldom used and the protocol boils down to a coarse-grained distributed locking approach, which has a significant impact on scalability. Notice that if there are k (disjoint) conflict classes, there can be at most k transactions executing in the entire system. Again, this seems to be confirmed by the original presentation of the protocol (Figure 6 of [111]): With an update-intensive load and $k = 16$ distinct conflict classes, the scale factor for 15 nodes is just five-fold ($5\times$). Although this is attributed to saturating system resources, one cannot know for sure as it is not measured. One should expect that if k is smaller, this result is even worse.

Our experiments using the TPC-C workload confirm these hints. Figure 3.7 shows the NODO protocol saturating when there are still plenty of system resources available. Although our implementation does not have the optimistic functionality, queue $Q2$ is always large, and thus, the optimistic path would not be used anyway.

3.2.2 Active replication

Active replication is a technique to build fault-tolerant systems in which transactions are deterministically processed at all replicas and as such requires that each transaction's statement be processed in the same order at them. This might be ensured by means of a centralized or a distributed scheduler. Read-only transactions, however, are handled locally.

Sequoia 4.x [35], which was built after the C-JDBC [33], for instance, uses a centralized scheduler at the expense of introducing a single point of failure. Usually, any distributed scheduler would circumvent this resilience problem but would require a distributed deadlock detection mechanism. To avoid the distributed deadlocks, one might annotate transactions with conflict-classes and request distributed locks through an atomic multicast before starting executing a transaction. In contrast with NODO, however, a reliable message to propagate changes would not be needed as transactions would be actively executed. In both approaches, the consistency criteria would be similar to those provided by NODO.

The case against the active replication is shown in the NODO paper: unbearable contention with high write ratio. This technique additionally has the drawback of requiring a parser to remove non-deterministic information (e.g., random() or date()), thereby leading to re-implementing several features already provided by a database management system.

The active replication pays off when the overhead between transferring raw updates in a passive replication is higher than in re-executing statements. Naturally, it also makes it easy to execute DDL statements.

3.2.3 Database State Machine (DBSM) and Postgres-R (PGR)

In both protocols, transactions are immediately executed by the replicas to which they are submitted without any a priori coordination. Locally, transactions are synchronized according to the specific concurrency control mechanism of the database engine.

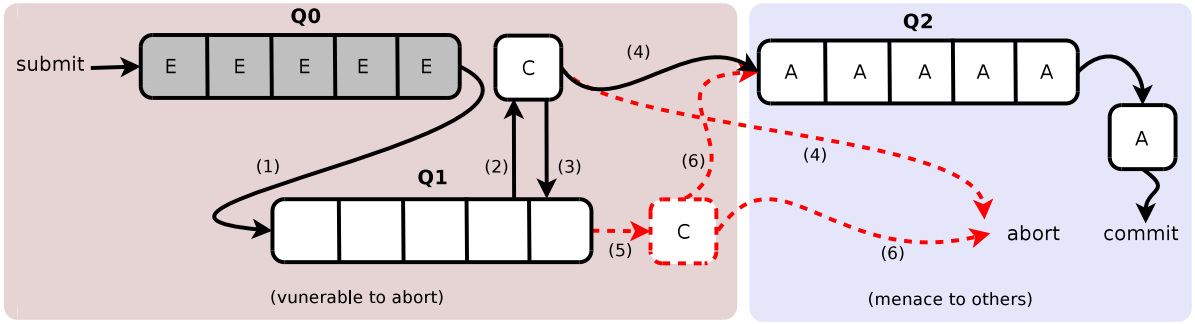
Upon receiving a commit request, a successful transaction is not readily committed. Instead, its changes (write-set) and read data (read-set) are gathered and a termination protocol initiated. The goal of the termination protocol is to decide the order and the outcome of the transaction such that a global correctness criterion is satisfied (e.g 1-copy serializability [23] or 1-copy snapshot isolation [84]). This is achieved by establishing a total order position for the transaction and certifying it against concurrently executed transactions. The certification of a transaction is done by evaluating the intersection of its read-set and write-set (or just write-set in case of the snapshot isolation) with the write-set of concurrent, previously ordered transactions. The fate of a transaction is therefore determined by the termination protocol and a transaction that would locally commit may end up aborted. On the other hand, read-only transactions are immediately committed.

These protocols differ on the termination procedure. Considering 1-copy serializability, both protocols use the transaction's read-set in the certification procedure. In the PGR, the transaction's read-set is not propagated and thus only the replica executing the transaction is able to certify it. In the DBSM, conversely, the transaction's read-set is propagated allowing each replica to autonomously certify the transaction.

In detail, upon the reception of the commit request for a transaction t, in PGR the executing replica atomically multicasts t's id and t's write-set. As soon as all transactions ordered before t are processed, the executing replica certifies t and reliably multicasts the outcome to all replicas. The certification procedure consists in checking t's read-set and write-set against the write-sets of all transactions ordered before t. The executing replica then commits or aborts t locally and replies to the client. Upon the reception of t's commit outcome each replica applies t's changes through the execution of a high priority transaction consisting of updates, inserts and deletes according to t's previously multicast write-set. The high priority of the transaction means that it must be assured of acquiring all required write locks, possibly aborting any locally executing transactions. In other words, if t does not end up aborted by a high priority transaction, it is transparently and indirectly certified what we entitle an in-core certification.

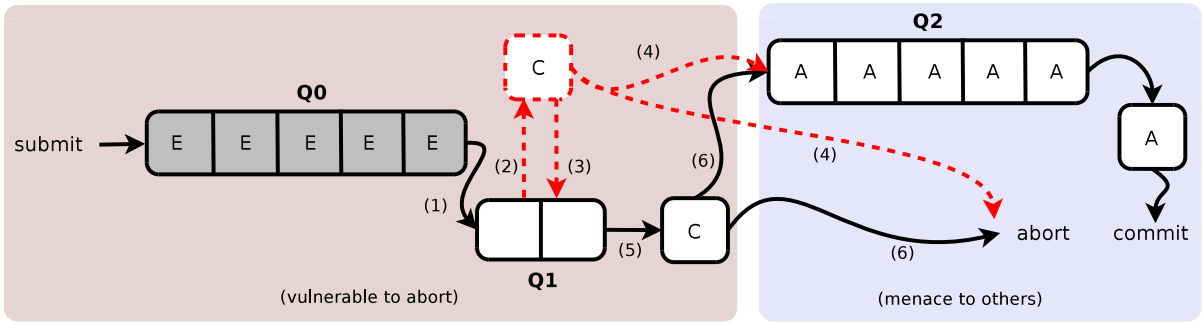
The termination protocol in the DBSM is significantly different and works as follows. Upon the reception of the commit request for a transaction t, the executing replica atomically multicasts t's id, the version of the database on which t was executed and t's read-set and write-set. As soon as t is ordered, each replica is able to certify t on its own. For the certification procedure, t's read-set and write-set are checked against the write-sets of all transactions committed since t's database version. If they do not intersect, t commits, otherwise t aborts. If t commits then its changes are applied through the execution of a high priority transaction consisting of updates,

inserts and deletes according to t's previously multicast write-set. Again, the high priority of the transaction means that it must be assured of acquiring all required write locks, possibly aborting any locally executing transactions. The executing replica replies to the client at the end of t.



1 - Multicast
2 - Certify Optmistically
3 - Final Delivery, missed order and then rollback (unlikely)
4 - Final Delivery, correct order
5 - Certify in order (unlikely)
6 - Commit/Abort in order (unlikely)

(a) Assumption of the designers.



1 - Multicast
2 - Certify Optmistically (unlikely)
3 - Final Delivery, missed order and then rollback (unlikely, but it doesn't matter)
4 - Final Delivery, correct order
5 - Certify in order
6 - Commit/Abort in order

(b) Our assumption.

Figure 3.3: States, Transitions, and Queues in DBSM.

In both protocols, transactions are queued while executing, as would happen in a non-replicated database, using whatever native mechanism is used to enforce ACID properties. This is queue $Q0$ in Figures 3.3 and 3.4. The most noteworthy feature of both protocols is that ever since a transaction starts until it is certified, it is vulnerable to being aborted by a concurrent transaction that gets to commit and write a conflicting item. On the other hand, from the instant that a transaction is certified until it finally commits on every node, it is a menace to other transactions which will be aborted if they touch a conflicting item. Latency in any processing stage is thus

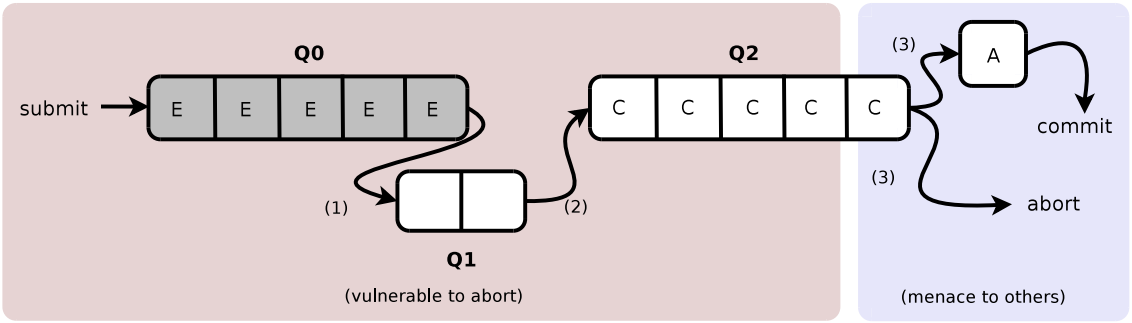


Figure 3.4: States, Transitions, and Queues in PGR.

bound to increase the abort rate. A side-effect of this is that the resulting system, when loaded, is extremely unfair to long-running transactions.

In the DBSM, the initial assumption was that the only added latency introduced by replication was in the atomic multicast step, similarly to NODO ($Q1$) in Figure 3.2(a). PGR [76] does not use optimistic delivery. However, this is only an issue in WANs. In clusters, latency comes from exhausting resources within each replica as queues build up in $Q0$ and $Q2$. It is thus no surprise that any contention whatsoever makes the abort rate shoot up.

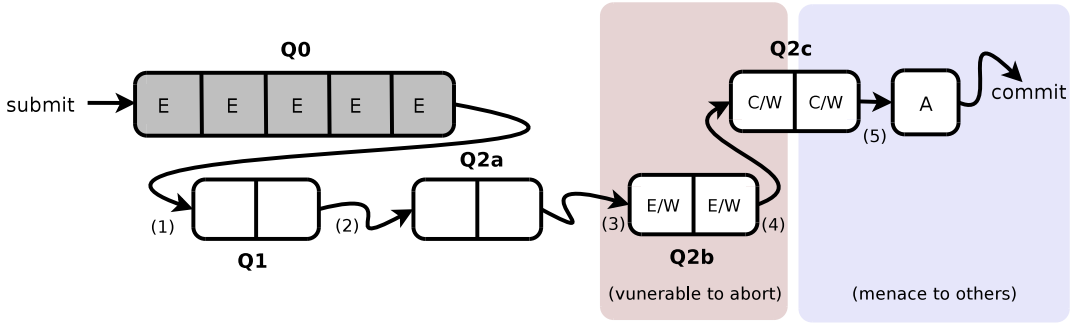
3.3 The AKARA Protocol

3.3.1 Intuition

The goal of AKARA is three-fold: maximize resource usage by scheduling sufficient concurrent executions (avoiding the pitfall of NODO) while at the same time keeping queuing outside the danger zones thus ensuring fairness (avoiding the pitfalls of DBSM) and overcome a profound limitation of both NODO and DBSM by allowing seamless active execution.

Figure 3.5 depicts the major states, transitions, and queues of this protocol. Let us assume that conflict classes are tables and, for simplicity, that all transactions access at least a common table. This assumption is completely realist as it is valid for the TPC-C, but in Section 3.3.2, we relax it and consider the case that transactions have no conflict classes in common.

Upon submission, transactions are classified according to a set of conflict classes and totally ordered by means of an atomic multicast primitive. This global order allows to prevent conflicting transactions to run concurrently. Once ordered, a transaction is queued into $Q2a$ waiting to be scheduled. Progression in $Q2a$ depends on an admission control policy. When a transaction reaches the top of $Q2a$ it is transferred to $Q2b$ and then executed. Transactions executed while in $Q2b$ are said to be run optimistically as they may end up aborting due to conflicts with concurrent transactions in $Q2b$ or $Q2c$. After execution, and having reached the top of $Q2b$, a transaction is transferred to $Q2c$. When a transaction reaches the top of $Q2c$ it may be ready to commit or



1 - Pre-classification and multicast
2 - Final Delivery
3 - Scheduler: scheduled to run optimistically or wait if remote or active
4 - Optmistic execution finished or simply next in line
5 - Next in line

Figure 3.5: States, Transitions, and Queues in AKARA.

not (because it had to abort due to conflicts). If it is ready to commit, its changes are propagated to all other replicas and the transaction commits. Otherwise, the transaction is re-executed conservatively by imposing its priority on any locally running transaction.

AKARA maximizes resource usage through the concurrent execution of potentially conflicting transactions by means of an admission control mechanism. It is worth noting, however, that an admission policy that only allows to execute non-conflicting transactions according to their conflict classes makes AKARA to fall short as a simple conservative protocol. The key is therefore to judiciously schedule the execution of each transaction in order to exploit idleness thus reducing contention introduced by a conservative execution while at the same time avoiding re-execution. We assume here a policy that just allows to optimistically execute n transactions in parallel. The analysis of more sophisticated policies is not target in this work as this simple policy suffices to show the effectiveness of our novel protocol.

Such optimistic executions however may lead to local deadlocks. Consider two conflicting transactions t and $t\prime$ that are ordered $< t, t\prime >$ and scheduled to run concurrently (both are in $Q2b$). If $t\prime$ grabs a lock first on a conflicting data item, it prevents t from running. However $t\prime$ cannot leave $Q2b$ before t without infringing the global commit order. Two extreme solutions for this problem are:

- Roll back right after execution, reapplying updates later on if no conflicts arise. This has a serious drawback as it imposes a severe overhead even when conflicts are unlikely or even nonexistent. And, when there are conflicts, it always implies a re-execution.

- The other solution is to abort a transaction that gets to the top of the queue (that is, reaches its commit order) if a subsequent transaction must finish execution before it. This has however the severe drawback that it prevents many non-conflicting transactions to be executed simultaneously, decreasing the value of the optimistic execution.

If both transactions have the same conflict classes and, of course, are locally executed at the same replica, a better alternative is to allow t' to overtake t in the global commit order. Notice that when a transaction t is totally ordered this ensures that no conflicting transaction will be executed concurrently at any other replica. Therefore, if t's order is swapped with that of $t\prime$ with the same conflict classes then it is still guaranteed that both t and $t\prime$ are executed without the interference of any remote conflicting transaction. In the experiments conducted in Section 3.4 with the TPC-C, the likelihood of having two transactions with the very same conflict classes is high as more than 85% of the occurrences are due to the *NewOrder* and *Payment* transactions.

Finally, the AKARA protocol also allows transactions to be actively executed thus providing a mechanism to easily replicate DDL statements and to reduce network usage. This execution steps are detailed in the next section.

3.3.2 Algorithm

The AKARA algorithm is presented in Figure 3.6. In that, a transaction is represented by a data structure containing the following information: seq - a global sequence number that corresponds to the total order established by the atomic multicast; cc - the transaction's estimated set of conflict classes; $type$ - whether the transaction should be passively or actively executed. Although not explicitly used in the algorithm of Figure 3.6, we assume that this data structure also contains the transaction's write-set.

Each replica maintains different sets: $Q0$, $Q2a$, $Q2b$ and $Q2c$ whose utilization was introduced in Sections 3.2 and 3.3 and shall be detailed next.

Once a transaction is submitted, $submit(t)$ is invoked. The transaction t is put into $Q0$, which used to store transactions before any coordination action is carried on. Right after, an external function (line 4) is used to compute the type of t: $passive$ or $active$. Then another external function (line 5) classifies t with respect to its conflict classes.[1] Once t is classified, it is atomically multicast to all replicas (line 6). Upon delivery (lines 8, 29 and 41), t is put into $Q2a$ and $t.seq$ is set. This gives t its commit order, which is total with respect to all its conflicting transactions. It is worth noticing that we omitted $Q1$ here as we do not exploit fast delivered transactions.

Assuming a passive execution (line 8), the initiating replica waits until t can be the next in $Q2a$ to be transferred to $Q2b$ and a scheduler decides to optimistically execute it (line 11). In particular, the function $next(Q, cc)$ (line 26) looks at a queue, in this case $Q2a$, and retrieves information on conflicting transactions. If there is a conflicting transaction ordered before t, i.e. $t \neq next(Q2a, t.cc)$, t waits for its turn. Otherwise, it can be removed from $Q2a$ and proceed.

Once the previous condition is achieved (line 10), t is put into $Q2b$ and its execution is started. From this moment until t can be removed from $Q2c$, it is vulnerable to be aborted by a remote high priority transaction. Therefore it may terminate its execution either upon requesting a commit or due to an abort requested by a conflicting and remote high priority transaction. In the former case, it is marked as ready to commit.

One needs to wait until t is executed and can be removed from $Q2b$ (line 15). However, due

[1] See Section 3.5 for a brief discussion on these functions.

to interleaves of concurrent events inside a database, a transaction $t\prime$ ordered before t may be blocked by t thus not being able to make progress and not allowing t to be removed from $Q2b$ and proceed. To overcome this problem, the algorithm (lines 52–58) allows t to overtake $t\prime$ in the global commit order, when both have the same conflict classes and belong to the same replica. Otherwise, it aborts t.

Once the previous condition is achieved (line 15), t is put into $Q2c$. When t can be removed from $Q2c$, its write-set is reliably multicast to all replicas if it is still ready to commit. Otherwise, t is executed as a high priority transaction and right after its write-set is reliably multicast to all replicas. Finally, t is committed at the initiating replica and removed from $Q2c$.

At a remote replica, the execution of a transaction t is straightforward (line 29). When t can be removed from $Q2a$, it is immediately moved to $Q2b$, and so forth, until it gets to $Q2c$. When t can proceed from $Q2c$ and its write-set is delivered, it is applied on the replica with a high priority, committed and then removed from $Q2c$.

A transaction t marked as active is executed at all replicas without distinction between a initiating or a remote replica, and its execution is straightforward (line 41). When t can be removed from $Q2a$, it is immediately moved to $Q2b$, and so forth, until it gets to $Q2c$. When t can proceed from $Q2c$, it is executed with a high priority, committed and then removed from $Q2c$. Active transactions are not executed optimistically to avoid different interleaves at different replicas.

3.4 Evaluation

3.4.1 Simulation Environment

The simulation environment is based on a centralized simulation model that combines real software components with simulated hardware, software and environment components to model a distributed system. This allows us to set up and run multiple realistic tests with slight variations of configuration parameters that would otherwise be impractical to perform, especially if one considers a large number of clients and replicas [131].

The key components, the replication and the group communication protocols, are real implementations while both the database engine and the network are simulated.

The simulation environment represents a LAN with 9 replicas connected by a network with a bandwidth of 1Gbps and a latency of 120μs. Each replica corresponds to a dual processor AMD Opteron at 2.4GHz with 4GB of memory, running the Linux Fedora Core 3 Distribution with kernel version 2.6.10. For storage we used a fiber-channel attached box with 4, 36GB SCSI disks in a RAID-5 configuration and the Ext3 file system. The database running is a PostgreSQL 7.4.6 with snapshot isolation and the global consistency criterion is 1-copy snapshot isolation [84].

Clients run an implementation that mimics the industry standard on-line transaction processing benchmark TPC-C [138]. TPC-C specifies five transactions: *NewOrder* with 44% of the occurrences; *Payment* with 44%; *OrderStatus* with 4%; *Delivery* with 4%; and *StockLevel* with 4%. The *NewOrder*, *Payment* and *Delivery* are update transactions while the others are read-only.

For the experiments in Section 3.4.2, we added to the benchmark three more transactions

```
1  Q0, Q2a, Q2b, Q2c: sets;
2  function submit(t)
3      put t into Q0;
4      t.type = compute_type(t);
5      t.cc = compute_classes(t);
6      abcast(t.type,t);
7  end
8  upon deliver(passive,t) to self
9      put t into Q2a;
10     wait (t = next(Q2a, t.cc) ∧
11       scheduled(t));
12     put t into Q2b;
13     execute t;
14 end
15 upon (t is local ∧
16   t is executed ∧
17   t = next(Q2b, t.cc))
18     put t into Q2c;
19     wait (t = next(Q2c, t.cc));
20     if (t is not ready to commit) then
21         execute t with priority;
22     rbcast (t.updates, t);
23     commit t;
24     remove t from Q2c;
25 end
26 function next(Q,cc) ≡ t ∈ Q st.
27     t.seq = min
       ({t'.seq | t' ∈ Q ∧ t'.cc ∩ cc})
28 end
29 upon deliver(passive,t) to others
30     put t into Q2a;
31     wait (t = next(Q2a, t.cc));
32     put t into Q2b;
33     wait (t = next(Q2b, t.cc));
34     put t into Q2c;
35     wait (t = next(Q2c, t.cc) ∧
36       t.updates were delivered);
37     apply t.updates with priority;
38     commit t;
39     remove t from Q2c;
40 end
41 upon deliver(active,t)
42     put t into Q2a;
43     wait (t = next(Q2a, t.cc));
44     put t into Q2b;
45     wait (t = next(Q2b, t.cc));
46     put t into Q2c;
47     wait (t = next(Q2c, t.cc));
48     execute t with priority;
49     commit t;
50     remove t from Q2c;
51 end
52 upon (t, t' are local ∧ t ≠ t' ∧
53   t is ready to commit ∧
54   t' = next(Q2b, t.cc))
55     if (t.cc == t'.cc) then
56         swap(t.seq, t'.seq);
57     else abort t;
58 end
```

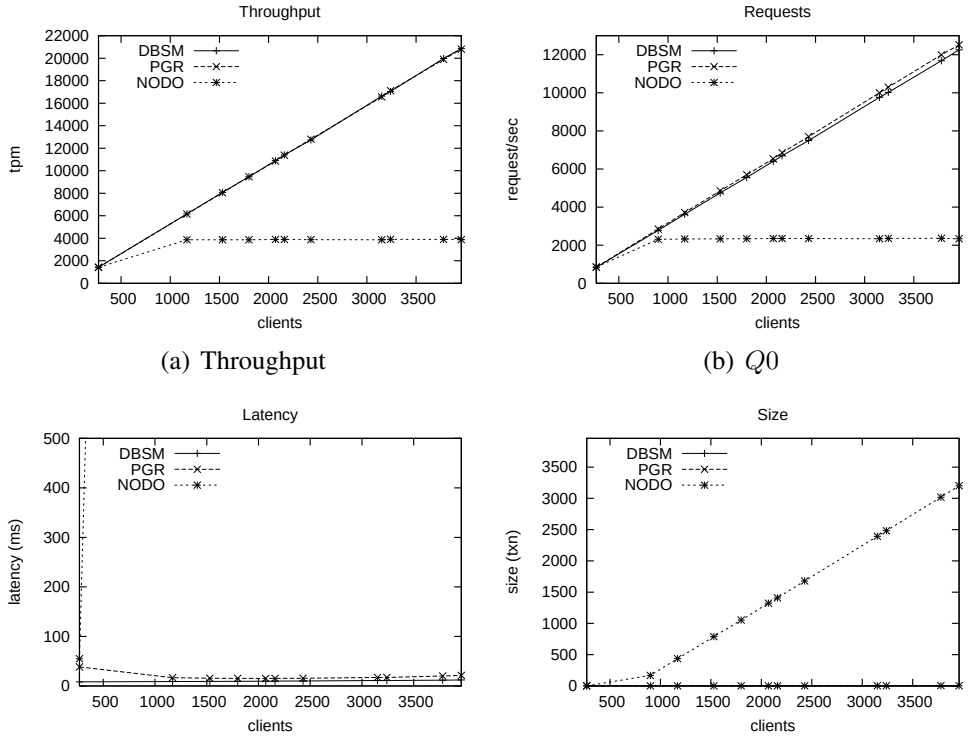
Figure 3.6: AKARA Algorithm.

that mimic maintenance activities such as adding users, changing indexes in tables or updating taxes over items. Specifically, the first transaction *Light-Tran* creates a constraint on a table if it does not exist or drops it otherwise. The second transaction *Active-Tran* increases the price of products and is actively executed. Conversely, *Passive-Tran* does the same maintenance activity but its changes are passively propagated. These transactions are never executed in the same run, have a probability of 1% and when are executing the probability of the *NewOrder* is reduced to 43%.

We varied the total number of clients from 270 to 3,960 and distributed them evenly among the replicas and each run has 150,001 transactions.

3.4.2 Results

The first set of experiments evaluate the DBSM, NODO and PGR approaches. In the NODO approach, we use the simple definition of a conflict class for each table, which can be easily extracted from the SQL code. Figures 3.7 and 3.8 compare the DBSM, PGR and NODO.



(a) Throughput (b) $Q0$

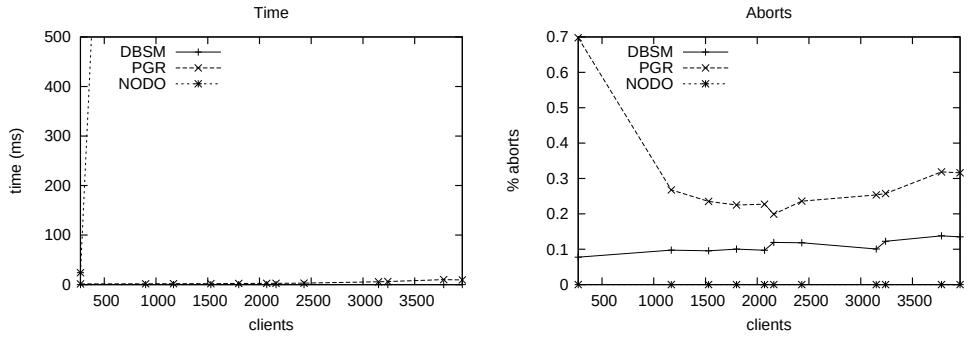
(c) Latency (d) $Q2$

Figure 3.7: Performance of DBSM, PGR, and NODO.

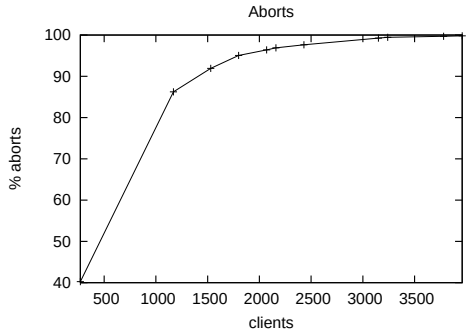
The DBSM and PGR show a throughput higher than 20000 tpm (Figure 3.7(a)). In fact, both present similar results and the higher the throughput the higher the number of requests per second inside the database (Figure 3.7(b)). These requests represent access to the storage, CPU, lock manager, and the replication protocol. Clearly, the database is not a bottleneck. In contrast, the throughput presented by NODO is extremely low, around 4000 tpm, and its latency is extremely high (Figure 3.7(c)). This drawback can be easily explained by the contention observed in $Q2$ (Figure 3.7(d)).

Unfortunately, with the conservative and optimistic approaches presented above, one may have to choose between latency and fairness. In the NODO, for 3,240 clients, 2,481 transactions wait in $Q2$ around 40 s to start executing (Figure 3.8(a)). In contrast, an optimistic transaction waits 1000 times less and the number of transactions waiting to be applied is very low.

The abort rate is below 1% in both optimistic approaches as there is no contention and the likelihood of conflicts is low in such situations (Figure 3.8(b)). However, to show that the optimistic protocols may not guarantee fairness, we conducted a set of experiments in which one



(a) Time in $Q2$ (b) Abort



(c) Delivery's Abort in DBSM

Figure 3.8: Latency & Abort in DBSM, PGR, and NODO.

requests an explicit table level locking on behalf of the *Delivery* transaction thus mimicking a hotspot. This is a pretty common situation in practice, as application developers may explicitly request locks to improve performance or avoid concurrency anomalies. In this case, the abort rate is around 5% and this fact does not have an observable impact on latency and throughput but almost all Delivery Transactions abort, around 99% (Figure 3.8(c)). In [70], a table level locking is acquired on behalf of the *Delivery* transaction to avoid flooding the network and improve the certification procedure. Although the reason to do so is different, the issue is the same.

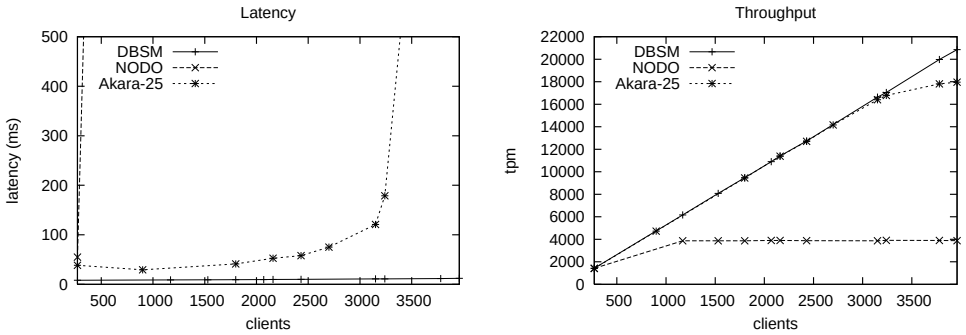
In all the experiments, the time between an optimistic delivery and a final delivery were always below 1 ms, thus excluding $Q1$ from being an issue.

To improve the performance of the conservative approach while at the same time guaranteeing fairness, we used the AKARA protocol. We ran the AKARA protocol varying the number of optimistic transactions that might be concurrently submitted to the database in order to figure out which would be the best value for our environment. This degree of optimistic execution is indicated by a number after the name of the protocol. For instance, AKARA-25 means that 25 optimistic transactions might be concurrently submitted and AKARA-n means that there is no restriction on this number.

Table 3.1 shows that indefinitely increasing the number of optimistic transactions that might be concurrently submitted is not worthwhile. Basically for AKARA-n, latency drastically increases and as a consequence throughput decreases. This occurs because the number of trans-

	Latency (ms)	Throughput (tpm)	Unsuc. rate (%)
AKARA-25	178	16780	2
AKARA-45	480	16474	5
AKARA-n	37255	3954	89
AKARA-25 with Light-Tran	8151	9950	21
AKARA-25 with Active-Tran	109420	1597	21
AKARA-25 with Passive-Tran	295884	625	22

Table 3.1: Analysis of AKARA.



(a) Latency (b) Throughput

Figure 3.9: Performance of DBSM, NODO and AKARA-25.

actions that fails the certification procedure increases. For 3240 clients, more than 89% of the transactions fail the certification procedure (i.e. in-core certification procedure like in PGR, see Section 3.2.3). Furthermore, after failing such transactions are conservatively executed and compete for resources with optimistic transactions that may be executing. Keeping the number of optimistic transactions low however reduces the number of transactions allowed in the database and neither is worth. After varying this number from 5 to 50 in increments of 1, we figured out that the best value for the TPC-C in our environment is 25.

In what follows, we used the DBSM as the representative of the family of optimistic protocols thus omitting the PGR. Although both protocols present similar performance in a LAN, the PGR is not worth in a WAN due to its extra communication step [70].

Figure 3.9 depicts the benefits provided by the AKARA-25. In Figure 3.9(a), we notice that latency in the NODO is extremely high. In contrast, the AKARA-25 starts degenerating after 3,240 clients. For 3,240 clients the latency in the DBSM is about 9 ms, and in the AKARA-25, it is about 178 ms. This increase in latency directly affects throughput as shown in Figure 3.9(b). The NODO presents a steady throughput of 4000 tpm; the AKARA-25, a steady throughput of 18605 tpm after 3,960 clients; and the DBSM increases its throughput almost linearly. The DBSM starts degenerating when the database becomes a bottleneck.

Table 3.1 shows the impact on performance when the maintenance activities are handled by our protocol. These maintenance activities represented by the transactions *Active-Tran* and *Light-Tran* are actively executed and integrated in runs with the *AKARA-25*: *AKARA with Active-Tran* and *AKARA with Light-Tran*, respectively. In order to show the benefits of an active execution in such scenario, we provide a run named *AKARA with Passive-Tran* in which the updates performed by the *Active-Tran* are atomically multicast. The run with the *Passive-Tran* presents a latency higher than that with the *Active-Tran* as the former needs to transfer the updates through the network. However, both approaches have a reduced throughput and high latency when compared with the normal AKARA-25 due to contention caused by a large number of updates.

The run with the *Light-Tran* does not have a large number of updates but its throughput decreases when compared with the AKARA-25 due to failures in the certification procedure. This is caused by the fact that the transaction *Light-Tran* mimics a change on the structure of a table and thus requires an exclusive lock on it.

In a real environment, we expect that maintenance operations occur with a rate lower than 1% and so they should not be a problem as the optimistic execution of other transactions might compensate for the temporary decrease in performance.

3.5 Open Issues

Most benchmarks are modeled as an open or closed system, although, a partly-open system is more accurate for most real scenarios. In a nutshell, new requests are triggered by request completions followed by think time if the system is closed. On the other hand, if the system is open, new requests arrive independently of request completions. The TPC-C is modeled as a closed system [127].

This has a direct impact on the results presented in this work. Open and partly-open systems have a worse degradation in performance due to contention when compared with closed systems: a higher mean response time and reduced throughput. The variability of service demand also has a large impact on the mean response time. This is particularly important when taking into account the *Delivery* transaction which takes about three times longer to execute when compared with other transactions.

Any additional contention introduced by a replication protocol is troublesome for the overall system performance and should be avoided or circumvented whenever possible. Disregarding this key factor leads to the intensification of weakness in the protocols (e.g. queuing and abort rate) and most likely makes them infeasible for most real application scenarios. For those reasons, it is extremely important to evaluate the protocols presented here, in particular AKARA, with a partly-open benchmark in order to figure out whether it would behave as expected.

Another issue is that, although the current implementation of AKARA statically specifies the multiprogramming limit (MPL) by establishing the number of optimistic transactions that can be concurrently executed on a replica, this information could be dynamically defined as in [126]. One might use an adaptive mechanism [90] to determine this value taking into account the idleness of the database and the abort rate due to the optimistic execution. In [48], an adaptive mechanism to control the MPL is proposed. However, in this case, it basically avoids that la-

tency of the conservative protocol increases drastically by reducing or increasing the number of connections or balancing load among replicas. There is no attempt to reduce the time spent in queues.

Regarding the transaction execution model, deciding whether a transaction should be passively or actively executed is a task that can be done automatically or manually. In the former case, AKARA might learn from previous executions of a transaction in order to come up with a decision. Usually, the higher the number of changes the more appropriate is the use of active replication. Furthermore, AKARA might exploit the GORDA API (Chapter 4) to extract information from a database such as the number of changes made by a transaction and whether there are DDL statements or not. The GORDA API might also be used to help in removing non-deterministic information in statements by withdrawing most of the work from the replication middleware.

Finally, it is worth noticing that having conflict classes based on tables eases the classification procedure regardless if it is done automatically or manually. In particular, if the classification is done manually, it is pretty simple to automatically detect labeling mistakes.

3.6 Conclusion

The performance of group-based database replication protocols can be challenged by demanding workloads. Namely, conservatively synchronized protocols overly restrict concurrency, and thus throughput, unless a careful application-specific definition of conflict classes is done. On the other hand, optimistically synchronized protocols make it difficult for long-running and prone to conflicts transactions to commit. Finally, both depend on shipping updated data items, which makes it hard to deal with very large updates or DDL statements. Although all these issues can easily be avoided in benchmarks, they are a significant hurdle to adoption in real scenarios.

In this work we address these issues with the AKARA protocol, which seamlessly combines multiple execution strategies. Experimental evaluation with the TPC-C workload shows that the proposed protocol provides adequate throughput without requiring application-specific tuning of conflict classes. By introducing a small number of transactions with large write-sets or DDL statements in the mix to be actively replicated, one can also see that fairness is ensured and network usage is minimized.

Chapter 4

GAPI: GORDA Architecture and Programming Interface

4.1 Introduction

A key point of the architecture presented in Chapter 2 is the reflector, which, among other things, should have the ability to intercept and modify client requests and results, controlling operation scheduling and influencing the commit order.[1] The purpose of this component is to export a replication-friendly interface to the replicator. In this way, database replication protocols can be implemented independently of the DBMS being used at deployment time, thus promoting the design and implementation of protocols that can be deployed in a wide range of configurations.

The independence between a specific DBMS and the replication protocols is achieved by augmenting the standard database interfaces with additional primitives that provide abstractions reflecting the usual processing stages and transactions (e.g., transaction parsing, optimization, and execution) inside the DBMS engine. Naturally, the implementation details of the replicator vary depending on the specific DBMS instance.

A well-known software engineering approach to build systems with such complex requirements is reflection [78, 87]. By exposing at the interface an abstract representation of the systems' inner functionality, the latter can be inspected and manipulated, thus changing its behavior without loss of encapsulation. Database management systems have long taken advantage of reflection, namely, on the database schema, on triggers, and when exposing the log.

In this thesis, we propose a general-purpose database management system reflection architecture and interface that supports a number of useful extensions while at the same time admitting efficient implementations. The interface described includes also some functionality that is available through standard client interfaces, but which might admit custom implementations with higher performance.

The rest of this chapter is structured as follows. Section 4.2 gives some background about reflection on systems and interfaces. Section 4.3 overviews the architecture and API. Section 4.4 presents use cases. Finally, Section 4.6 offers the conclusions of the chapter.

[1] See Appendix A for a detailed list of requirements.

4.2 Related Work

4.2.1 Reflective Architectures

Logging, debugging, tracing facilities and autonomic functions, such as self-optimization or self-healing, are some examples of important add-ons to database management systems that are today widely available [77]. The computation performed by such plugins is known as a computational reflection, and the systems that provide them are known as reflective systems. Specifically, a reflective system can be defined as a system that can reason about its computation and change it. Reflective architectures ease and smooth the development of systems by encapsulating functionalities that are not directly related to their application domains. This can be done to a certain extent in an ad-hoc manner, by defining hooks in specific points of a system, or with support from a programming language. In both cases, there is a need for providing a reflective architecture where the interaction between a system (i.e., base-level objects) and its reflective counterpart is done by a meta-level object protocol and the reflective computation is performed by meta-level objects. These objects exhibit a meta-level programming interface.

In this thesis, we propose to use hooks into DBMSs to develop a meta-level protocol, along with meta-level objects, which exploit a set of concepts based on a common transaction processing abstraction (e.g., parsing, optimization, execution), although implementations are highly dependent on database management systems. By exposing a common meta-level programming interface, our approach eases the development of a variety of plugins (e.g., replication, query caching, self-optimization). We name it the GORDA Reflective Architecture and Programming Interfaces (GAPI) [41, 56].

4.2.1.1 Dependable Systems

The use of computational reflection is not new in the field of dependable applications, and the first approach is from the early 1990s. The MAUD, GARF, FRIENDS, IRL, and FTS systems briefly introduced below are representative of this approach:

- The MAUD (Meta-level Architecture for Ultra Dependability) uses a high-level language [3] based on the actor model which provides a mathematical framework for concurrent systems. Actors are first-class entities that can make decisions, create other actors, receive and send messages.

- The GARF System [61] is an extension to a Smalltalk Environment based on a set of classes and a runtime environment. It divides computation among data objects, common objects created by the Smalltalk, which handle functional properties of a system, and behavioral objects, which handle crosscutting concerns. Whenever a data object is created, the GARF runtime environment wraps it with a behavioral object, enabling it to intercept invocations to the data object.

- The FRIENDS System (Flexible and Reusable Implementation Environment for your Next Dependable System) [47] uses a specialized meta-level protocol based on Open C++, a

pre-processing extension to C++, which provides special statements to associate an object with a meta-level object.

- Interoperable Replication Logic (IRL) and Fault-Tolerant Service (FTS) provide fault tolerance by using CORBA request portable interceptors to forward requests to proxies that furnish fault-tolerant services by means of replication [49, 88].

In such projects, a reflective architecture, along with object-oriented programming methods, frees developers from details of particular dependability protocols and promotes reusability. In that broad sense, our approach aims at the same goals, given that it encapsulates details on databases by means of the GAPI, thus easing the development of plugins and promoting their reusability among different database vendors. In contrast to previous approaches, GAPI does not rely on extensions to a programming language, nor is it bounded to one.

4.2.1.2 Database Management Systems

Most database management systems provide a reflective mechanism where hooks are defined by means of triggers, notifying meta-level code whenever a relation is updated. Although this native approach might be used to handle some functionalities required by the GAPI, it might generate an unbearable overhead as the life-cycle of the meta-information produced (i.e., write-set) is restricted to the meta-level execution. Furthermore, this native approach does not provide other important requirements to ease the development of add-ons. For instance, by using it, one cannot hold the execution when a transaction commits, thus forbidding the development of add-ons such as synchronous replication protocols that require processing in transactions' contexts.

In [89], reflection is used to introduce self-tuning properties into database management systems. Configuration and performance parameters are reflected into tables, and triggers are used to orchestrate interaction among components. Periodically, a monitor tool that uses a DB2 UDB snapshot API collects performance information and stores it into tables. Right after, a trigger notifies diagnosis functions to decide whether to change configuration information into tables or not. Once a configuration is modified, a trigger notifies the DB2 which applies the changes. In [117], the same idea is presented but without relying on a specific DBMS vendor, thus assuming that most DBMSs provide the means to inspect performance information and change configuration parameters.

The GAPI reflective architecture can also be used to develop self-tuning solutions by inspecting information on query plans and tracking the number of concurrent transactions and throughput. The proposed interface provides only some capabilities to collect such information, but it can be used and easily extended to achieve those purposes.

In [19], a reflective system on a TP Monitor (Transaction Processing Monitor) is described in order to support the development of extended transaction models (e.g., long-lived transactions). The functional aspects such as transaction execution, lock management, and conflict detection are reflected through adapters, which provide meta-level objects and a meta-level programming interface. This interface is different from what the GAPI provides as Roger Barga et al. are concerned with mechanisms that enable, for instance, to joining and splitting transactions, thus

requiring more information on locks and conflicts and different meta-information on transactions. Not only information on read- or write-sets needs to be reflected but also information on types of locks and pending lock requests. This is necessary to transfer locks acquired and to be acquired on behalf of a transaction to another transaction. The conflict detection adapter is used to relax consistency when executing extended transaction, thus providing access to shared tuples which would be blocked by a normal conflict detection mechanism such as those based on the 2PL mechanism.

In [124], a reflection mechanism for database replication is proposed. In contrast to our approach, it assumes that reflection is achieved by wrapping the DBMS server and intercepting requests as they are issued by clients. By choosing beforehand such implementation approach, one can only reflect computation at the first stage (statements), i.e., with a very large granularity. Exposing further details requires rewriting large portions of a DBMS at the wrapper level. As an example, Sequoia [35] has additional parsing and scheduling stages at the middleware level. Theoretically, this proposal can be more generic and usable on closed systems. In practice, this is not always true, since DBMSs usually do not exactly support the same language, and middleware solutions must be customized for a certain system. Despite that, we will see later in this chapter that this approach can introduce a significant overhead to latency of transactions by requiring extra communication steps and/or extra processing of requests.

4.2.2 Design Patterns

The design of a meta-programming interface is based on design patterns that have been useful in the broader context of object-oriented distributed applications. Namely, façade [50], inversion-of-control, and container managed concurrency patterns are used[2]:

- The façade pattern allows inspection of diverse data structures through a common interface. A very well known example is the ResultSet of the JDBC[3] specification, which allows results to be stored in a DBMS native format. The alternative is the potentially expensive conversion to a common format such as XML. The proposed architecture suggests using this for most of the data that is conveyed between processing stages (e.g., parser, optimizer).

- The inversion-of-control pattern eases the deployment of software components. In detail, meta-objects, such as transactions, are exposed to an object container, which is configured with reflection components. The container is then responsible for injecting the required meta-objects into each reflection component during initialization.

- The container-managed concurrency pattern allows the container implementation to schedule event notifications according to its own performance and correctness criteria. For instance, by ensuring that no two transaction commit notifications are issued concurrently, it implicitly exposes a commit order.

[2] http://www.martinfowler.com/articles/injection.html
[3] http://java.sun.com/javase/technologies/database/

4.3 Reflector: Replication-friendly Database Support

In this section, we overview and motivate the GORDA DBMS reflective Architecture and Programming Interface, simply denoted GAPI. The full details about the architecture and interfaces are described in GORDA projects deliverable [56]. In the next sections, we will illustrate GAPI with several use cases and evaluate its overhead in real implementations.

4.3.1 Target Reflection Domain

The GAPI has been designed having in mind the support for database replication applications. Although the use of the GAPI is not limited to this class of applications, replication protocols are quite demanding in terms of functionality that needs to be exposed, and their requirements strongly influenced the design and implementation of our reflective interface.

Previous reflective interfaces for database management systems were mainly targeted at application programmers using the relational model. Their domain is, therefore, the relational model itself. Using this model, one can intercept operations that modify relations by inserting, updating, or deleting tuples, observe the tuples being changed, and then enforce referential integrity by vetoing the operation (all at the meta-level) or by issuing additional relational operations (base-level).

In contrast, there are protocols concerned with details that are not visible in the relational model, such as modifying a statement to remove non-deterministic statements, as those involving `NOW()` and `RANDOM()`. One may be interested in intercepting a particular statement as it is submitted, whose text can be inspected, modified (meta-level), and then re-executed, locally or remotely, within some transactional context (base-level).

Therefore, a more expressive target domain is required. We select an object-oriented concurrent programming environment. Specifically, we use the JAVA platform (but any similar language would also fit our purposes). For those DBMSs that already have support for the JAVA language, the JAVA is a great choice as it eases the interaction between the meta-level and the base-level, as described in Section 4.3.5.

For instance, the fact that a series of activities (e.g., parsing) is taking place on behalf of a transaction is reflected as a transaction object, which can be used to inspect the transaction (e.g., wait for it to commit) or to act on it (e.g., force a rollback). Meta-level code can register to be notified when specific events occur. Thus, when a transaction commits, a notification is issued and contains a reference to the corresponding transaction object (meta-object). Actually, handling notifications is the way that meta-level code dynamically acquires references to meta-objects describing the on-going computation.

4.3.2 Processing Stages

The usefulness of the meta-level interface depends on what is exposed as meta-objects. If a very fine granularity is chosen, the interface cannot be easily mapped to different DBMSs, and the resulting performance overhead is likely to be high. On the other hand, if a very large granularity is chosen, the interface may expose too little to be useful. Therefore, we abstract transaction

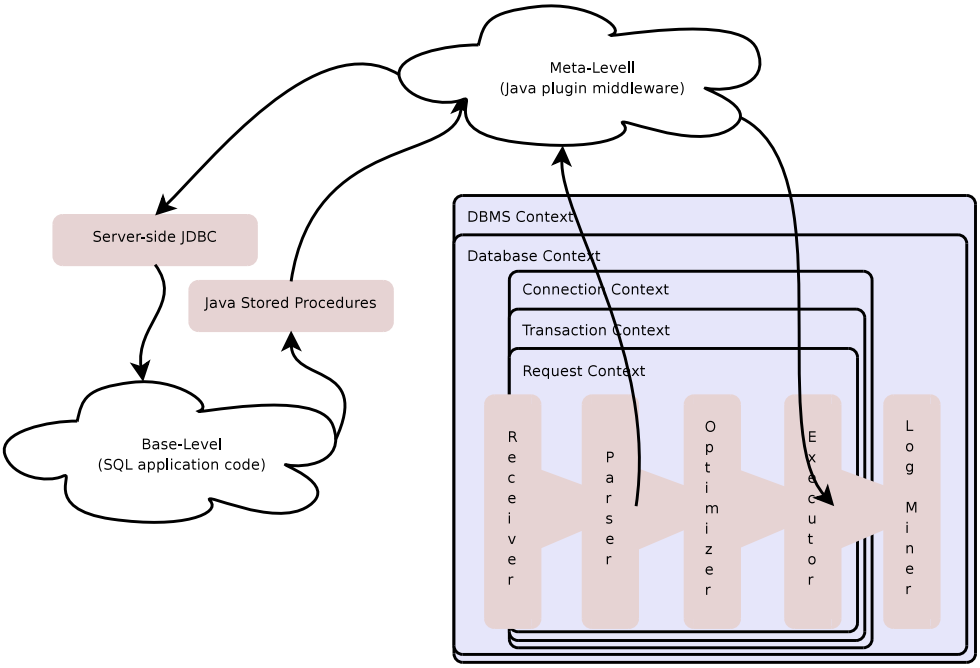


Figure 4.1: Major Meta-level Interfaces: Processing Stages and Contexts.

processing as a pipeline, as described in [51] and illustrated in Figure 4.1. The meta-objects exposed by the GAPI, at each stage of the pipeline processing, are briefly listed below (for further details, please see [56]). The plugin is notified of these meta-objects in the order they are listed.

Receiver Stage receives new statements from the clients. Notifies the reception of a new statement that can be inspected and/or modified at this moment;

Parser Stage parses single statements received, thus producing a parse tree;

Optimizer Stage receives the parse tree and transforms it, according to various optimization criteria, heuristics, and statistics into an execution plan;

Executor Stage executes the plan and produces object sets;

Log Miner Stage deals with mapping from logical objects to physical storage.

In general, one wants to receive a notification at the meta-level whenever computation proceeds from one stage to the next. For instance, when a write-set is produced at the execution stage, a notification is issued such that it can be observed. The interface thus exposes meta-objects for each stage and for data that moves between them.

4.3.3 Processing Contexts

The meta-interface exposed by the processing pipeline is complemented by nested context meta-objects, also shown in Figure 4.1. These show on whose behalf some operation is being performed. In detail, the contexts are the following:

DBMS Context represents the database management system, exposes metadata, and allows notification of life-cycle events;

Database Context represents a database, also exposes metadata, and allows notification of life-cycle events;

Connection Context reflects existing client connections to the DBMS. They can be used to retrieve connection-specific information, such as thread identification or the character set encoding used;

Transaction Context is used to notify events related to a transaction such as its startup, commit, or rollback. Synchronous event handlers available here are the key to synchronous replication protocols;

Request Context is used to ease the manipulation of the requests within a connection and the corresponding transaction.

Usually, events fired by processing stages refer to the directly enclosing context. Each context has, then, a reference to the next enclosing context and can enumerate all enclosed contexts. This allows, for instance, to determine all connections to a DBMS or which is the current active transaction in a specific connection. There are some contexts, however, that do not obey this rule. In particular, a connection may be enclosed by a database or may be allowed to access any available database in a DBMS. In PostgreSQL, a connection is restricted to a single database and thus its transactions. Conversely, in MySQL, a connection may access any available database, wich means that a transaction may need a 2PC [23] to commit its work. Furthermore, it is not possible to determine on behalf of which transaction a specific disk block is being flushed by the log miner stage.

4.3.4 Synchronization

The meta-level code registers event handlers to intercept the flow of data structures within the execution pipeline. An event handler can be set in two different modes: blocking and non-blocking. This is chosen at run time when setting the handler. When a handler is set in blocking mode, the database suspends the current activity (i.e., thread) until both the event handler has returned and the continue or cancel methods have been invoked on the event object. The meta-level code can do it in any order. When a handler is set in non-blocking mode, the database does not wait for the order to continue or cancel its execution. In fact, in this scenario, an exception is thrown if any of these methods are called.

The handlers can be notified concurrently, even if they were registered in blocking mode, but the dependency relations that exist between events in nested contexts must not be disregarded. It is up to the meta-level code to handle synchronization where required. The notification to continue or cancel execution can be issued by a different order from those that were received. This implies that the commit order is determined by the order by which meta-level code orders the execution to continue. The meta-level code, however, must ensure that no concurrent invocations of such method exist within the same database context. This must be done to ensure a global correctness criterion such as 1-Copy Serialiability or 1-Copy Snapshot Isolation.

The validity of a meta-level object depends on its life-cycle, and any information required to survive its boundaries must be copied into the scope of the meta-level language. Although a reference to a meta-level object may be kept during its life period, invocations of its methods are only allowed while it, or an object event that is somehow related to it, is being processed by an event handler.

4.3.5 Base-level and Meta-level Calls

In this architecture, the base-level call is a client call that makes a request by means of an SQL statement, for instance, whereas the meta-level call is a reflection call exposed by a reflective DBMS. Meta-level programing allows to have a clean separation between the base- and meta-level architecture concerns, but it has the advantage that the base- and meta-level calls can be mixed, as there is no inherent difference between base- and meta-objects. This happens also in the proposed interface, albeit with some limitations.

In detail, a direct call to meta-level code can be forced by a plugin programmer by registering it as a native procedure and then using the CALL SQL statement. This causes a call to the meta-level code to be issued from the base-level code within the *Execute* stage. The target procedure can then retrieve a reference to the enclosing *Request* context and thus to all relevant meta-interfaces. The reason for allowing this only from the *Execute* stage is simplicity, as this is inherently supported by any DBMS and does not seem to impact generality.

Meta-level code can callback into base-level in three different situations. The first is within a direct call from base-level to issue statements in an existing enclosing request context. This can be achieved using the JDBC client interface by looking up the “jdbc:default:connection” driver, as is usually done in Java procedures. The second option is to use the enclosing DBMS or database context to open a new base-level connection. The third approach is through a set of interfaces enabled at each stage by means of creating a context and injecting external data into internal structures. The reason for allowing base-level to use the JDBC interface is simplicity and easiness in portability among different DBMSs. This may, however, have an impact on the performance which may be circumvented by the third approach by means of a proprietary and optimized data format, thus avoiding unnecessary data conversations.

The calls between meta-level and base-level are exemplified in the Section 4.4.5. This example shows how to build a caching mechanism using the proposed interface. As it can be seen in Listing B, the `cacheLookup` procedure is stored in the database when it starts (lines 39 to 58) and is called later when a new incoming statement is notified (lines 26 to 38). This exemplifies the direct call to the meta-level code. The stored procedure (lines 12 to 25) uses a database con-

nection to populate the cache in the case that the request is not present, which exemplifies a call from the base-level to the meta-level.

The second issue when considering base-level calls is whether these also get reflected. The proposed option is to disable reflection on a case-by-case basis by invoking an operation on context meta-objects. Therefore, meta-level code can disable reflection for a given request, a transaction, a specific connection, or even an entire database. Actually, this can be used on any context meta-object and thus for performance optimization.

The third issue is how base-level calls issued by meta-level code interact with regular transaction processing regarding concurrency control. Namely, how conflicts that require rollback are resolved, such as in the multi-version concurrency control where the first committer wins or, more generally, when resolving deadlocks. The proposed interface solves this by ensuring that transactions issued by the meta-level do not abort in face of conflicts with regular base-level transactions. Given that a plugin code running at the meta-level has a precise control on which base-level transactions are scheduled, and thus can prevent conflicts among those, it has been sufficient to solve all considered use cases. The simplicity of the solution means that an implementation within the DBMS results in a small set of localized changes.

4.3.6 Exception Handling

DBMSs handle most of the base-level exceptions by aborting the affected transaction and generating an error to the application. The proposed architecture does not change this behavior and the meta-level is notified by an event issued by the transaction context object which allows meta-level to cleanup after an exception has occurred.

Most exceptions within a transaction context that are handled at the meta-level can be resolved by aborting the transaction. However, some event handlers should not raise exceptions to avoid incoherent information on databases or recursive exceptions, namely: while starting up or shutting down a database, while rolling back a transaction, or after committing one. In such points, any unhandled exception will leave the database in a panic mode requiring manual intervention to repair the system. Interactions between the meta-level and base-level are forbidden in such points, and any attempt of doing so puts the database in a panic mode too.

Exceptions from meta-level to base-level calls need additional management. For instance, while a transaction is committing, meta-level code might need to execute additional statements to keep track of custom meta-information on the transaction before proceeding, and this action might cause errors due to deadlock problems or low amount of resources. Such cases are handled as meta-level errors, to avoid disseminating errors inside the database while executing the base-level code.

4.3.7 Attachment, Referenceable, Equality, and Hash Code

A plugin can attach an arbitrary object to each context. This allows context information to be extended as required by each plugin. As an example, when handling an event fired by the first stage of the pipeline, signaling the arrival of a statement in textual format, the plugin gets a reference to the enclosing transaction context. It can then attach additional information to that

context. Later, when handling an event signaling the readiness of parts of the write-set, the plugin follows the reference to the same transaction context to retrieve the information previously placed there.

Any object attached through the proposed interface must implement the `equals` method to allow comparison between a target object and another object of the same type. In addition, every object must implement the `hashCode` method so that if two objects are equal, they have the same hash code. The converse, however, is not necessarily true.

An object representing a context or a phase of the pipeline must implement both methods, and optionally, its implementation must be both `javax.naming .Referenceable` and `java.io.Serializable`, so that the object can be stored in all JNDI naming contexts.

4.4 Case Studies

This section describes how the reflector interface is used to implement state-machine, primary-backup, and certification-based replication protocols and also how it might be used to develop plugins such as tracers and debuggers.

4.4.1 Primary-Backup

Overview In the primary-backup approach to replication, also called passive replication [98], update transactions are executed at a single master site under the control of a local concurrency control mechanism. Updates are then captured and propagated to other sites. Asynchronous primary-backup is the standard replication in most DBMSs and third-party offers. An example is the Slony-I package for PostgreSQL [115].

Implementations of the primary-backup approach differ whether propagation occurs synchronously within the boundaries of the transaction or, most likely, is deferred and done asynchronously. The latter provides optimum performance when synchronous update is not required, as multiple updates can be batched and sent in the background. It also tolerates extended periods of disconnected operation.

Reflector Components Used Synchronous primary-backup replication requires the component that reflects the transaction context to capture the moment a transaction starts executing, commits, or rollbacks at the primary. It also needs the object set provided by the execution stage to extract the write-set of a transaction from the primary and to insert it into the backup replicas.

Replicator Execution The execution of a primary-backup replicator is depicted in Figure 4.2. We start by describing the synchronous variant. It consists of the following steps:

Step 1: Clients send their requests to the primary replica.

Step 2: When a transaction begins, the replicator at the primary is notified, registers information about this event, and allows the primary replica to proceed.

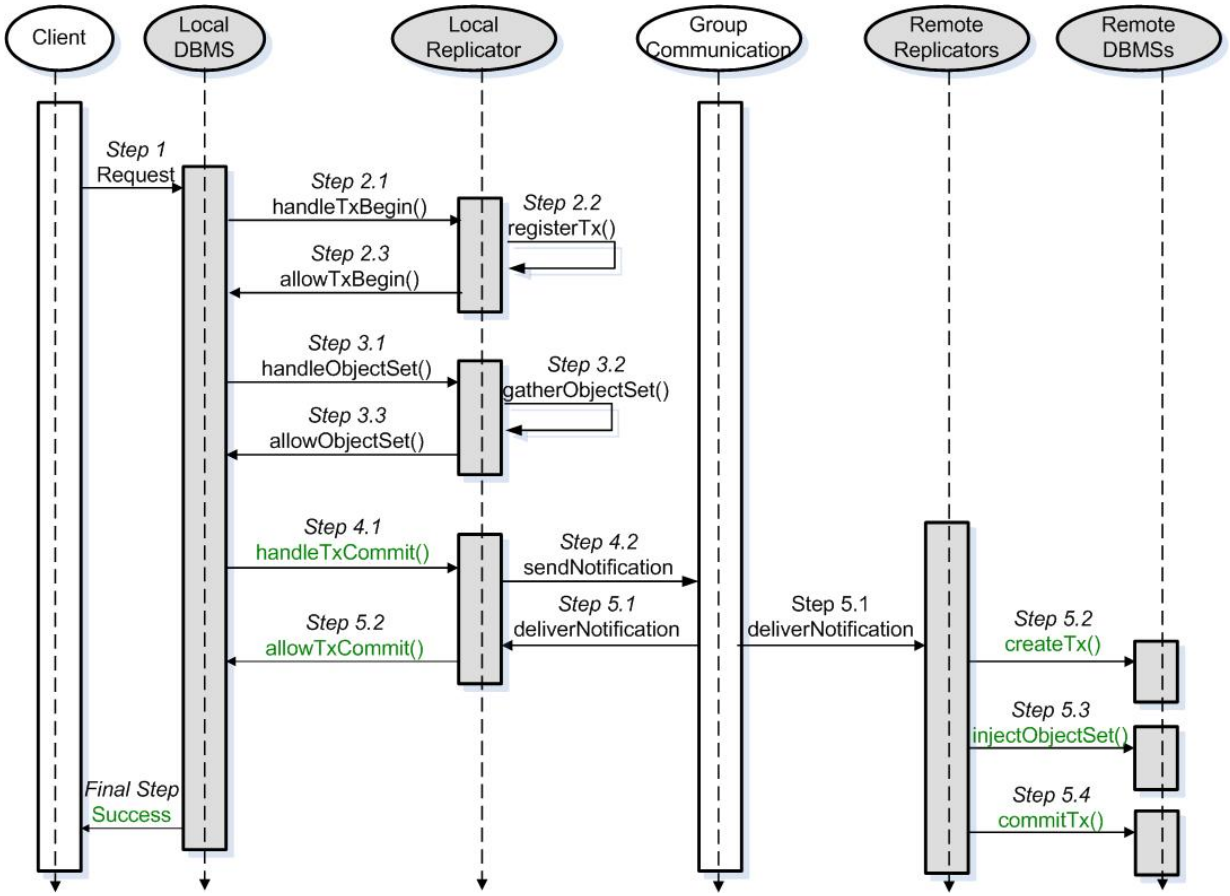


Figure 4.2: Primary-backup Replication using GAPI.

Step 3: Right after processing a SQL command, the database notifies the replicator through the execution stage component, sending an object set. Generally, the object set provides an interface to iterate on a statement's result set (e.g., write-set). Specifically, in this case, it is used to retrieve the statement's updates which are immediately stored in an in-memory structure with all other updates from the same transaction context.

Step 4: When a transaction is ready to commit, the transaction context component notifies the replicator of the primary. The replicator atomically broadcasts the gathered updates to all backup replicas (this broadcast should be *uniform* [34]).

Step 5: The write-set is delivered at all replicas. On the primary, the replicator allows the transaction to commit. On the backups, the replicator injects the changes into the DBMS.

Final Step: After the transaction execution, the primary replies to the client.

An asynchronous variant of the algorithm can be achieved by postponing Step 4 (and, consequently, Step 5) for a tunable amount of time.

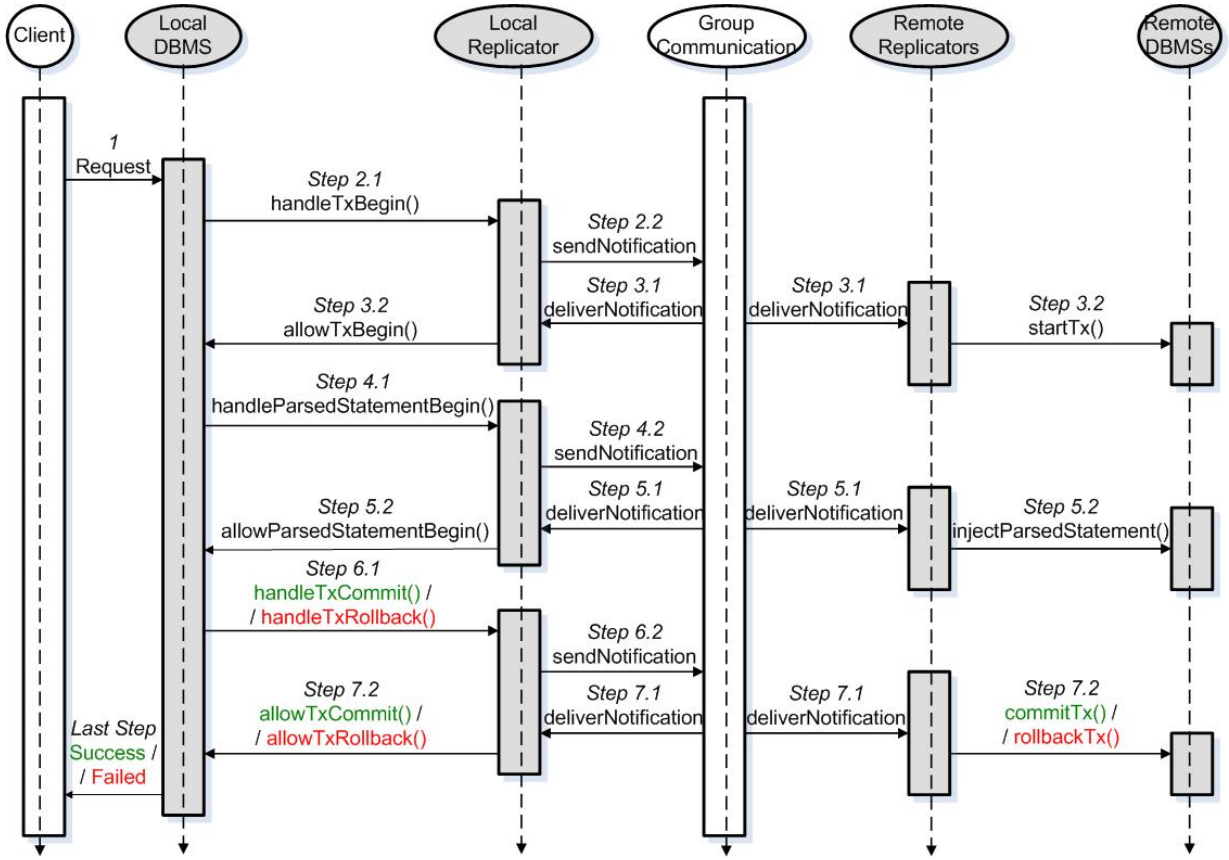


Figure 4.3: State-machine Replication using GAPI.

4.4.2 State-machine

Overview The state-machine approach, also called active replication [98], is a decentralized replication technique. Consistency is achieved by starting all replicas with the same initial state and, subsequently, receiving and processing the same sequence of client requests. Examples of this approach are provided by the Sequoia [35] and PGCluster [94] middleware packages.

Reflector Components Used The state-machine replication requires the use of the transaction context and parsing stage components. On one hand, the transaction component is used to capture the moment a transaction starts executing, commits, or rollbacks at one replica. On the other hand, the parsing stage component is used to capture and start the execution of transactional statements.

Replicator Execution The execution of a state-machine replicator is depicted in Figure 4.3. It consists of the following steps:

Step 1: Clients send their requests to one of the replicas. This replica is called the *delegate* replica.

Step 2: Using the transaction component, the replicator at the delegate replica is notified of the beginning of a transaction. The replicator uses an atomic multicast to propagate this notification to all other replicas.

Step 3: All replicators deliver the notification by the same order. The transaction is started in the remote replicas and resumed in the delegate replica.

Step 4: The transaction is executed at the delegate replica. Every time a new command starts, the replicator is notified through the parsing stage component of the reflector interface. Then, the replicator verifies if its parsed statement does not have any expression or function (e.g., *now()*) that might lead to non-deterministic executions. If so, it changes the parsed statement to remove the non-determinism. The resulting (potentially altered) parsed statement is atomically multicast to all replicators.

Step 5: The parsed statement is delivered at all replicators. Replicators must implement a deterministic scheduler: Each replicator must ensure that no two concurrent conflicting parsed statements are handed to the underlying DBMS. If such conflict exists, the parsed statement is kept on hold. Otherwise, it is handed to the DBMS at all replicas through the parsing stage component. It is worth noting two aspects related to this strategy. First, with this approach, deadlocks may happen, and the replicator should resolve them. Second, should a statement be used instead of a parsed statement, then the replicator would also need to parse it to extract information on tables.

Further steps: Steps 4 and 5 above are repeated.

Step 6: Using the transaction context component, the replicator at the delegate replica is notified when the transaction is about to commit or rollback. This notification is atomically multicast to all replicators.

Step 7: Upon receiving a commit or rollback notification, remote replicas execute the proper command, and the delegate replica allows it to proceed.

Final step: Once the processing is completed, the delegate replica replies to the client.

4.4.3 Certification-based Approaches

Overview Certification-based approaches operate by letting transactions execute optimistically in a single replica and, at commit time, run a coordinated certification procedure to enforce global consistency. Typically, global coordination is achieved with the help of an atomic multicast that establishes a global total order among concurrent transactions [76, 84, 104, 146].

Multiple variants of the certification-based approach have been proposed; we present here an example based on the Database State Machine.

Reflector Components Used Given its similarity to the Primary-Backup approach, the Certification-based replication requires the use of the same components, explicitly the transaction context and execution stage components.

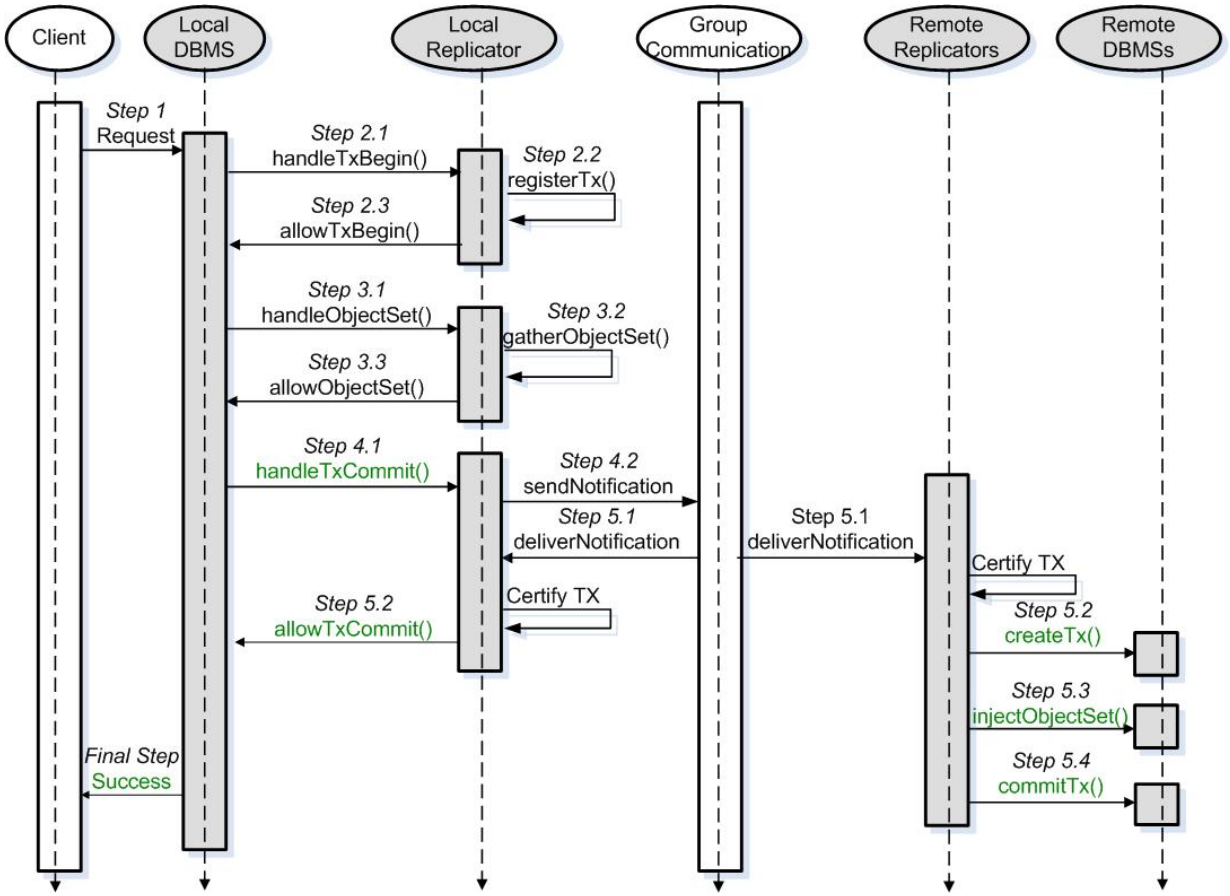


Figure 4.4: Certification-based Replication using GAPI.

Replicator Execution The execution of a certification-based replicator is depicted in Figure 4.4. It consists of the following steps:

Step 1-4: Same as in the Primary-Backup solution presented before.

Step 5: Upon receiving the write-set, each replica certifies the transaction and decides its outcome: commit or abort. If it is an abort, the delegate replica, through the transaction context component, cancels the commit and the remote replicas discard it. If it is a commit, the delegate replica allows it to continue, and the remote replicas inject updates into the DBMS.

Final Step: The delegate replica returns the response to the client.

4.4.4 Satellite Database

Overview Recently, the database community has been studying queries that do not have an exact match over database objects, introducing the notion of best-matching database objects (e.g.,

top-k or skyline queries) [26]. Most DBMSs do not provide such feature, and the architecture and programming interface proposed in this thesis might be used to extend them, thus allowing researches and developers to easily build and test different prototypes.

Such queries require statements with different syntaxes. For instance, a skyline query might look like this:

```
SELECT ... FROM ... WHERE ...
GROUP BY ... HAVING ... ORDER BY ...
SKYLINE OF c1 [MIN|MAX] {, cn [MIN|MAX]}
```

Thus, having the ability to intercept queries with a syntax that is not known at a delegate database is what is needed to start developing a plugin and transparently answer such requests. One might build a plugin locally or simply redirect the queries to another database (i.e., a satellite database [113]). Roughly speaking, a satellite database is a database that might be used to provide a functionality that would be difficult to develop on a delegate database.

To avoid changing the life-cycle of a database, the original statement is replaced by a dummy query, i.e., one that does not retrieve information and does not have impact on performance. Before replying to a client, the empty object set is replaced by the answer returned by the satellite database.

This can be extended to different stages of the pipeline, allowing us to develop features such as caches, debuggers, tracers, and optimizers. The more information provided by the stages, the greater the number of additional features that can be developed. However, it is worth noticing that update statements that are sent to satellite databases require that the plugin uses a 2-PC or 3-PC to cope with failures.

Reflector Components Used Satellite database requires the use of the request component which is employed to capture queries that should be parsed and handled by a plugin and to replace the client's result.

Plugin Execution The execution of a plug in is depicted in Figure 4.5. It consists of the following steps:

Step 1: Clients send their requests to a delegate replica.

Step 2: Using the Request Component, the plugin is notified and the request is parsed. If a token such as *skyline* is found, a satellite database is contacted to handle the request. Otherwise, nothing is done.

Step 3: The request is replaced by a dummy statement, and the database is allowed to continue.

Step 4: Upon processing the dummy statement, the plugin is notified and waits for an answer from the satellite database. The use of a group communication infra-structure is not a requirement, and a point-to-point communication might be used. By using a group communication infra-structure, however, one might also provide fault tolerance by diversity [52].

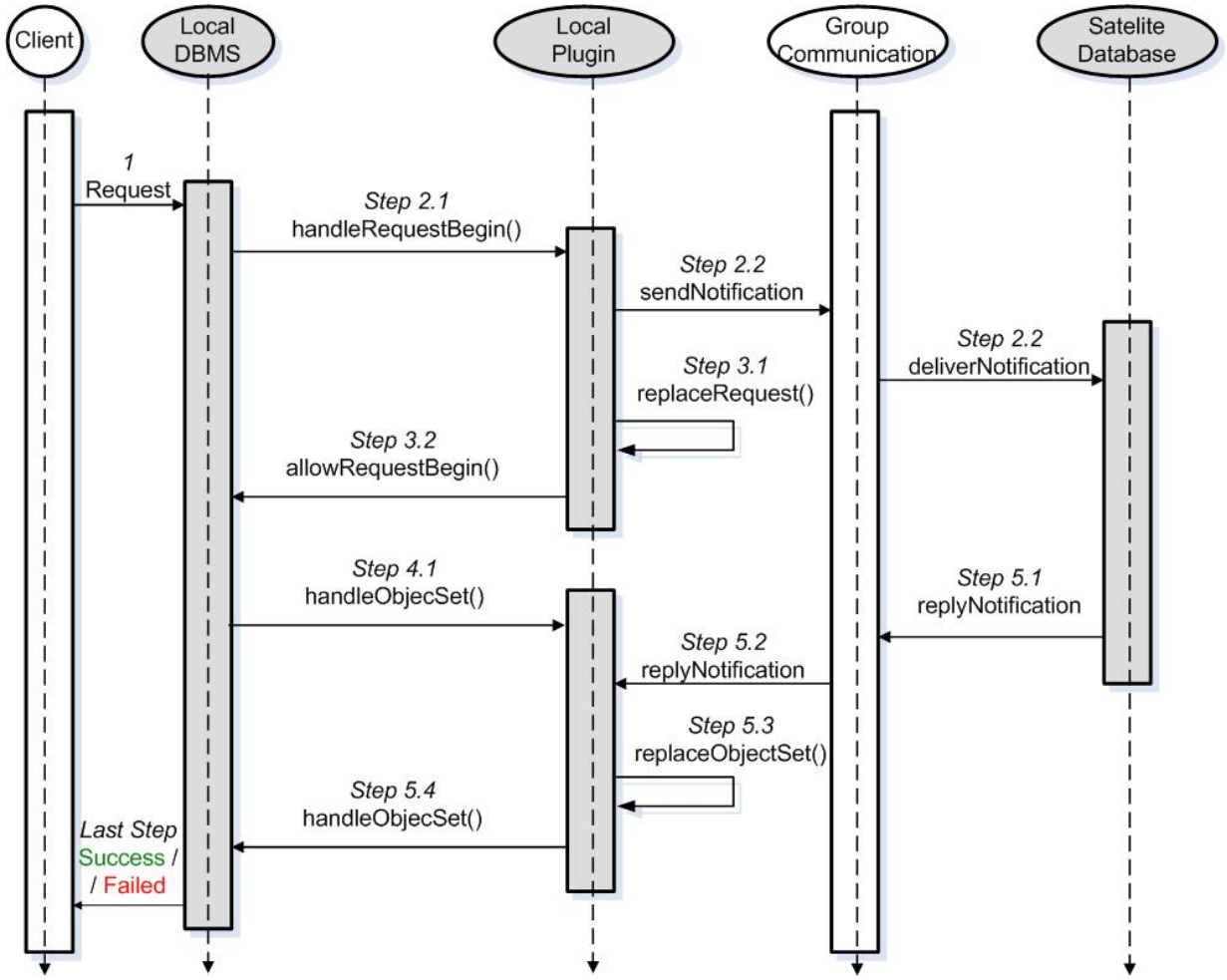


Figure 4.5: Satellite Databases and Plugins using GAPI.

Step 5: When such answer is received, the client's result set of the original statement is replaced by the result set from the satellite database.

Final Step: The delegate replica returns the response to the client.

4.4.5 Database Caching

Overview Database caching is an important technique to improve system performance and scalability, increasing throughput and reducing latency, by offloading database management systems. It is particularly suited for applications that have a high number of read operations when compared with the number of writes. Multi-tier environments, where a middle-tier application server accesses a database backend, might also take advantage of this technique by caching intermediate results, thus reducing communication steps and database usage [83].

In both cases, the GAPI might be used to create a plugin to intercept statements and results.

Using the intercepted information, cache entries may be created and invalidated. For instance, one might capture statements and information provided by the parser and optimizer. Specifically, the parser should be used to easily identify a statement as a query or an update and the optimizer to provide information on the cost to process the statement. When identified as a query, a result set might be used to populate the cache. This would be done only when a threshold based on the cost provided by the optimizer was reached. When identified as an update, the cache should be invalidated, automatically refreshed, or tagged as invalid. Tags might be used to identify, for instance, the number of changed items since the last refresh, thus allowing a query to specify the number of stale information that it tolerates. In other words, such tags might be used to postpone invalidation by relaxing the consistency criterion provided by the database, thus further improving performance and scalability [63].

Reflector Components Used The Query Caching plugin requires the database context to capture the moment when a database is started and the transaction context to define transactional boundaries. It also needs statements provided by the receiver stage and write-sets extracted by the executor stage. The execution of the Query Cache plugin consists of the following steps:

Plugin Execution

Step 1: The database is started, and a procedure is registered to make cache lookups and execute statements that are not in the cache (lines 45 to 55);

Step 2: When a statement is received, the plugin is notified and executes the previously stored procedure that verifies if the cache contains the required statement. If the statement is a read operation and the cache contains the statement, the results are returned to the client. Otherwise, the statement is executed, the results are added to the cache and then returned to the client (lines 26 to 36 and the procedure in lines 12 to 25);

Step 3: When a request is executed and changes information, the plugin is notified and uses the write-set to invalidate possible entries in the cache that contain obsolete information (lines 59 to 67). It is worth noting that this should be done only when (and if) a transaction commits. This feature and the code of cache invalidation are omitted for simplification.

As shown, the GAPI can be used not only as a monitoring API but also as an API to act and accommodate the behavior of the DBMS to the system needs. This example shows how transactions can be delayed to avoid concurrency in the system, thus avoiding conflicts in the lock manager based on previous executions.

4.5 Evaluation

The GAPI has been implemented on four different systems, namely, SleepyCat, PostgreSQL, Apache Derby, and Sequoia. These systems illustrate the effort required to implement the GAPI

using different apporaches. In this section, we provide a brief description of each of these implementations, including information about the number of lines of code required to implement GAPI on each architecture.

We also evaluate the performance of the SleepyCat GAPI implementation and compare different approaches to database reflection, namely the in-core implementation of the GAPI interface and the database wrapper.

4.5.1 Implementation Effort

SleepyCat 3.3.2

Berkley DB Java Edition (JE), or simply Sleepycat, is an embedded open-source object-based database engine completely written in Java [100]. It provides a persistent storage through a log-based filesystem [123] and an in-memory cache through a concurrent B^+Tree.

Implementing the proposed architecture in Sleepycat was easy as this storage engine is written in Java and is well-documented.

Implementation Effort The size of SleepyCat is 145,242 lines of code. In order to implement the GAPI interface on SleepyCat, 21 files were changed by inserting 3,298 lines and deleting 2,430 lines. Particularly, the GAPI is decoulpled from the core and represents 20 files with approximately 4,119 lines.

PostgreSQL 8.1

PostgreSQL 8.1 [60] is a fully featured database management system distributed under an open-source license. Written in C, it has been ported to multiple operating systems and is included in most Linux distributions as well as in recent versions of Solaris. Commercial support and numerous third-party add-ons are available from multiple vendors. Since version 7.0, it provides a multi-version concurrency control mechanism supporting snapshot isolation.

A challenge in implementing the proposed architecture in Postgres is the mismatch between its concurrency model and the multi-threaded meta-level runtime. PostgreSQL 8.1, as all previous versions, uses multiple single-threaded operating system processes for concurrency. This is masked by using the existing PL/J binding to Java, which uses a single standalone Java virtual machine and inter-process communication. This imposes an inter-process remote procedure call overhead on all communication between base and meta-level.

Therefore, the prototype implementation of the GORDA interface in PostgreSQL 8.1 uses a hybrid approach. Instead of directly patching the reflector interface on the server, key functionality is added to existing client interfaces and as loadable modules. The proposed meta-level interface is then built on these modules. The two layer approach avoids introducing a large number of additional dependencies in the PostgreSQL code, most notably on the Java virtual machine. As an example, transaction events are obtained by implementing triggers on transaction begin and end. A loadable module is then provided to route such events to meta-objects in the external PL/J server.

Implementation Effort The size of PostgreSQL is 667,586 lines of code; the PL/J package adds 7,574 lines of C code and 16.331 of JAVA code. To implement the GAPI interface on Postgres, 21 files were changed by inserting 569 lines and deleting 152 lines; 1,346 lines of C code were added in new files; and 11512 lines of Java code were added in new files.

Apache Derby 10.2

Apache Derby 10.2 [132] is a fully featured database management system with a small footprint that uses locking to provide serializability. It can either be embedded in applications or run as a standalone server. It was developed by the Apache Software Foundation and distributed under an open-source license; it is also distributed as IBM Cloudscape and in the Sun JDK 1.6 as JavaDB.

The GAPI prototype implementation takes advantage of Derby being natively implemented in Java to load meta-level components within the same JVM and thus closely coupled with the base-level components. Furthermore, Derby uses a different thread to service each client connection, thus making it possible that notifications to the meta-level are done by the same thread and thus reduced to a method invocation, which has negligible overhead. This is therefore the preferred implementation scenario.

Implementation Effort The total size of the Apache Derby engine is 514,941 lines of code. To implement the GAPI interface, 29 files were changed by inserting 1,250 lines and deleting 25 lines; in total, 9,464 lines of code were added in new files.

Sequoia 3.0

Sequoia [35] is a middleware package for database clustering, built as a server wrapper. It is primarily targeted at obtaining replication or partitioning by configuring the controller with multiple backends as well as improving availability by using several interconnected controllers.

Nevertheless, when configured with a single controller and a single backend, Sequoia provides a state-of-the-art JDBC interceptor. It works by creating a virtual database at the middleware level, which re-implements part of the abstract transaction processing pipeline and delegates the rest to the backend database.

The current prototype exposes all context objects and the parsing and execution objects, as well as calling from meta-level to base-level with a separate connection. It does not allow calling from base-level to meta-level, as execution runs in a separate process. It can, however, be implemented by directly intercepting such statements at the parsing stage. It does also not prevent base-level operations from interfering with meta-level operations, and this cannot be implemented, as described in the previous sections, as one does not modify the backend DBMS. It is, however, possible to the clustering scheduler already present in Sequoia to avoid concurrently scheduling base-level and meta-level operations to the backend, thus precluding conflicts.

Implementation Effort The size of the generic portion of Sequoia is 137,238 lines, which includes the controller and the JDBC driver; additional 29,373 lines implement plugable replication and partitioning strategies, that are not used by GAPI. To implement the GAPI interface on Sequoia, 7 files were changed by inserting 180 lines and deleting 23 lines, and 8,625 lines of code were added in new files.

Notes on the GAPI Implementation Effort

The effort required to implement a subset of the GAPI interface can roughly be estimated by the amount of lines changed in the original source tree as well as the amount of new code added. The numbers presented in the previous sections show that it is possible to implement the GAPI interface in three different architectures (SeelpyCat, PostgreSQL, and Apache Derby), with consistently low intrusion in the original source code. This translates in low effort both when implementing it and also when maintaining the code when the DBMS server evolves.

Note also that a significant part of the additional code is shared, namely in the definition of the interfaces (6,144 lines). There is also a firm belief that most of the rest of the code could also be shared, as it performs the same container and notification support functionality. This has not happened as each implementation was developed independently and concurrently.

Finally, it is interesting to note that the amount of code involved in developing a state-of-the-art server-wrapper is in the same order of magnitude as a fully-featured database (i.e., hundreds of thousands of code). In comparison, implementing the GAPI involves 100 times less effort as measured in lines of code.

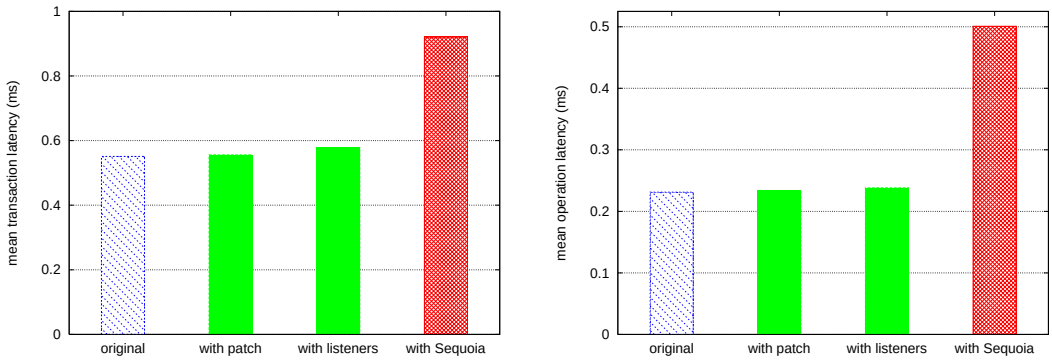
4.5.2 Performance

In this section, we evaluate the performance of one prototype implementation of the proposed interface in Apache Derby, and compare different approaches to database reflection, namely the in-core implementation of the GAPI interface and the database wrapper. The purpose of the evaluation is to assess the overhead introduced by an in-core implementation and compare this overhead with other solutions that are based on a DBMS wrapper. It is important to evaluate also the overhead of the introduced changes when not in use, which, if not negligible, is a major obstacle to the adoption of the proposed architecture.

We use the workload generated by the Poleposition benchmark. The Poleposition [114] benchmark is a framework to build benchmark tests. The tests create a small database in the DBMS. The size of transactions can change with the number of operations that are defined for each transaction. The results are measured in the client, and it measures the latency of transactions in milliseconds. In these tests, we measured the latency of update operations, with 1 and 100 operations on each transaction.

The following scenarios were tested: (i) *Unmodified DBMS* is the original DBMS, without any modification, serving as the baseline; (ii) *DBMS with patch* is the modified DBSM, as described in the previous section, but without any meta-level objects and thus with all reflection disabled. Ideally, this does not introduce any performance overhead; (iii) *DBMS with listeners* is the modified DBMS with listeners registered for transactional events, statements, and object sets. This means that each transaction generates at several events for each transaction; and (iv) *DBSM with Sequoia* is the unmodified DBMS with the Sequoia database wrapper but without doing any reflection.

The results are presented in Figure 4.6. Figure 4.6(a) shows the mean transaction latency of one transaction with a single operation. As it can be seen, when no meta-level objects are configured, the overhead introduced by the patches is negligible. We can observe the same



(a) Transactions. (b) Operations.

Figure 4.6: Performance Results using GAPI.

behavior when we add a plugin that listens to events. It is worth noting that one of the events is the notification of the object set produced by the transactions. An extra processing is done inside the DBMS to collect the object set, but it is also negligible. As we can see in the final test, the impact of adding a DBMS wrapper is noticeable as this causes an extra communication step and extra processing to parse incoming statements. Figure 4.6(b) depicts the mean operation latency and was measured by making 100 operations per transaction. The expected behavior of this test is to have some of the latency caused by the DBMS wrapper masked by the low number of commits. Note that the overhead caused by the wrapper is very significant.

4.6 Conclusions

The development of new DBMS plugins for different purposes, such as replication and clustering, require more functionality than the one currently provided by DBMSs' transactional APIs. Previously suggested solutions to meet these demands, such as patching the database kernel or building complex wrappers, require a large development effort in supporting code, cause performance overhead, and reduce the portability of middleware. In this thesis, we advocate for the use of a reflective architecture and interface to expose the relevant information about transaction processing in a useful way, namely allowing it to be observed and modified by external plugins.

We have shown the usefulness of the approach by illustrating how the interface can by applied to different settings, such a replication and query caching, among others. We have also shown that the approach is viable and cost-effective, by describing its instantiation on four different and representative architectures, namely the Apache Derby, PostgreSQL, SleepyCat, and the Sequoia server wrapper. We measured the overhead introduced by the in-core implementation on Apache Derby and compared it with a middleware solution. These prototypes are published as open source, can be downloaded from the GORDA project's home page, examined, and benchmarked. A modular replication framework that builds on the proposed architecture and thus runs on PostgreSQL, Apache Derby, SleepyCat, or any DBMS wrapped by Sequoia, is also available there.

The architecture presented here still has some limitations, that should be addressed by future work. It needs to be extended to perform the composition of multiple independente meta-level plugins, for instance, how to configure a DBMS to simultaneously use a self-management plugin and a replication plugin. Again, previous work on reflective systems might provide a direction [7]. Another open issue is the adequacy of the proposed architecture to non-classical database architectures, namely, how to match it with a column-oriented DBMS, such as MonetDB [68], or with an inherently clustered DBMS, such as Oracle RAC [118].

Chapter 5

Designing and Integrating Components

5.1 Introduction

GAPI provides replication-friendly database support and is a key component in designing database replication protocols for heterogeneous clusters. Specifically, it provides the means to capture write- and read-sets and control the life-cycle of a transaction. Developing a database replication protocol, however, requires not only a replication-friendly interface but also the means to coordinate database replicas in order to avoid integrity issues, disseminate changes, and cope with failures.

These additional requirements are fulfilled by the distributed replicator and the communication infra-structure as outlined in Chapter 2. In this chapter, we discuss the obstacles of designing and integrating such components and how we have overcome them to provide a set of building blocks for group-based replication protocols.

The distributed replicator aims to disseminate updates and enforce database integrity by coordinating the interaction among the replicated databases. This component relies on GAPI to capture updates, inject them into remote replicas, and control a transaction life-cycle and on the communication infra-structure for communication and replica membership control. The communication infra-structure exploits the group communication primitives to facilitate the development of replication protocols by transparently providing the means to reliably disseminate the write- and read-sets for all available replicas, to set a serial order for transactions, and to cope with failures amongst other features.

The availability of GAPI in different DBMSs makes it possible to build heterogeneous clusters where clients can redirect specific statements to a replica to take advantage of features only accessible there [113]. In a simple case, read-only transactions would always be sent to replicas with Snapshot Isolation to avoid blocking update concurrent transactions. This flexibility, however, reveals the need to discuss how 1-SR can be achieved with databases that have different consistency criteria and the fact that there is a lack of a clear definition on the read-set, which is a key factor to guarantee 1-SR on the set of group-based replication protocols discussed in this thesis. Furthermore, when transactions from the same client are handled by different replicas, causal consistency which is weaker than 1-SR may be violated [2, 99].

The rest of this chapter is organized as follows. Section 5.2 presents the replicator, refined to handle group-based replication protocols, and describes the communication infra-structure. Section 5.3 discusses how 1-SR can be achieved when DBMSs have different consistency criteria and focuses on the definition and extraction of the read-set. Section 5.4 discusses why causal consistency may be violated when transactions from the same client are handled by different replicas and proposes a solution to this issue. Finally, Section 5.5 provides the conclusions of the chapter.

5.2 Pluggable Replication Protocols

5.2.1 Group-based Replicator

The replicator is a distributed component responsible for coordinating the interaction among all DBMS replicas in order to enforce the consistency of the replicated database. It directly interfaces with the reflector and relies on the GCS module for all communication and replica membership control, as shown in Figure 5.1.

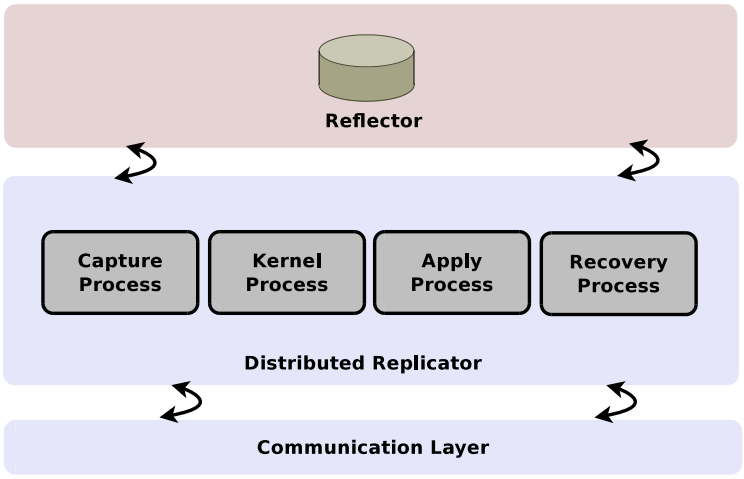


Figure 5.1: Replicator Architecture

It is within the replicator that the replica consistency protocols are implemented. The module is built around four process abstractions that are able to express most, if not all, database replication protocols. These are the Capture, Kernel, Apply, and Recovery processes and are described next.

Capture process The capture process is the main consumer of the reflector events. It receives events from the DBMS, converts them to appropriate events within the replicator, and notifies the other processes. In particular, it receives a transaction begin request and registers the current transaction context. For instance, for update transactions, the capture process may instruct the reflector to receive the read- and write-sets of the transactions when the commit request is

performed. Using this information, it may construct an internal transaction event that carries the transaction identification along with the corresponding read- and write-sets. It then notifies the kernel process which, in turn, is responsible for distributing the transaction data and enforcing a consistency criterion.

To improve performance of the capture process, we should consider zero-copy policies. In other words, this process should avoid copying information before sending it to database replicas. This could be done, for instance, pointing directly to tuple identifications, i.e., disk block and position in the block. Once a transaction is started and a tuple is updated on its behalf, it is guaranteed that the identification remains the same while the transaction is active. Storing a reference and accessing attributes upon commit to disseminate a transaction avoids copies and improves performance. This is true, however, when there is enough space in cache to keep blocks accessed by a transaction in it, until commit time; otherwise this approach leads to additional access to storage, thus dramatically harming performance.

Kernel process This process implements the core of the replica consistency protocol. In general, it handles the replication of local transactions by distributing relevant data and determining their global commit order. Additionally, it handles incoming data from remotely executed transactions. The local outcome of every transaction is ultimately decided by the kernel process, to ensure a target global consistency criterion. To execute its task, the kernel process exchanges notifications with the capture and apply processes, and interfaces directly with the GCS component.[1]

Apply process The apply process is responsible for efficiently injecting incoming transaction updates into the local database through the reflector component. To achieve optimum performance, this implies executing multiple apply transactions concurrently and, when possible, batching updates to reduce the number of transactions. It needs, however, to ensure that the agreed serialization order is maintained.

Recovery process The recovery process intervenes whenever a replica joins or rejoins the group. It is responsible for exporting the database state when acting as a donor or bringing the local replica up-to-date if recovering.

Both the recovery and the kernel modules cooperate closely with the GCS module. To allow the integration of the new replica into the group, the kernel module is required to temporarily block any outgoing messages until the complete recovery of the new replica is performed by the recovery process.

5.2.2 GCS: Communication and Coordination Support

All database replica consistency protocols require communication and coordination support. Some of the more relevant abstractions to support database replication are: reliable multicast

[1]The kernel process encompasses both the distribution and coordination processes discussed in Chapter 2.

(disseminates updates among the replicas), total order (defines a global serial order for transactions), and group membership (manages the set of currently active replicas in the system).

A software package that offers this sort of communication and coordination support is typically bundled in a package called a *Group Communication Toolkit*. Since the pioneering work initiated two decades ago with Isis [25], many other toolkits have been developed. Appia [92], Spread [12], and JGroups [18] are, among others, some of the group communication toolkits in use today. Therefore, group communication is a mature technology that greatly eases the development of practical database replication systems.

At the same time, group communication is still a hot research topic, as performance improvements and wider applicability are sought [106–108, 130, 140]. Furthermore, group communication is clearly an area where there is no one solution that fits all application scenarios. For instance, just to offer total order multicast, dozens of different algorithms have been proposed [43], each outperforming the others for a specific setting: There are protocols that perform better for heavily loaded replicas in switched local area networks [62], others for burst traffic in LANs [71], others for heterogeneous wide-area networks [121], etc.

Therefore, having a clear interface between the replication protocols and the GCS has multiple practical advantages. To start with, it allows us to tune the communication support (for instance, by selecting the most appropriate total order protocol) without affecting the replication protocol. Furthermore, given that different group communication toolkits implement different protocols, it should be possible to re-use the same replication protocols with different group communication toolkits.

To address these problems, one defines a generic interface to group communication services, called *Group Communication Service for Java*, or simply jGCS [1], that may be used to wrap multiple toolkits.

To fulfill the promise of scalable replication under demanding workload, the group communication toolkit needs to provide two important features: (*i*) optimistic uniform total order and (*ii*) primary partition support.

Optimistic uniform total order The notion of optimistic total order was first proposed in the context of local-area broadcast networks [105]. In many of these networks, the spontaneous order of message reception is the same in all processes. Moreover, in sequencer-based total order protocols, the total order is usually determined by the spontaneous order of message reception in the sequencer process. Following these two observations, a process may estimate the final total order of messages based on its local receiving order and, therefore, provide an optimistic delivery as soon as a message is received from the network. With this optimistic delivery, the application can make some progress. For example, a database replication protocol can apply the changes in the local database without committing them. The commit procedure can only be made when the final order is known and if it matches the optimistic order. If the probability of the optimistic order matching the final order is very high, the latency window of the protocol is reduced and the system gains in performance.

Unfortunately, spontaneous total order does not occur in wide-area networks. The long latency in wide-area links causes different processes to receive the same message at different points

in time. To circumvent this issue, [122, 130] operate by introducing artificial delays in the message reception to compensate for the differences in the network delays.

When configured to use this protocol, the group communication toolkit delivers the original message as soon as it is received (network order). Notifications about optimistic total order and final uniform total order are later delivered, indicating that progress can be made regarding a particular message.[2]

Primary partition support Partitions in the replica group may happen due to failures in the cluster (network, switching hardware, etc.). In asynchronous systems, virtual partitions (indistinguishable from physical partitions) may happen due to unexpected delays. A partitionable group membership service allows multiple concurrent views of the group, each corresponding to a different partition, to co-exist and evolve in parallel [16, 44]. In the context of database replication, this is often undesirable as it may lead to different replicas processing and committing conflicting updates in an uncoordinated form. A partition in the group membership can then easily lead to the *split-brain* phenomenon: The state in different replicas diverges and is no longer consistent. In contrast, a primary-partition group membership service maintains a single agreed-upon view of the group at any given time, delivering a totally ordered sequence of views (processes that become disconnected from the primary partition block or are forced to crash and later rejoin the system).

In our approach, primary partitions are defined by majority quorums. The initial composition of the primary partition is defined at configuration time, using standard management interfaces. The system remains alive as long as a majority of the previous primary partition remains reachable [20, 73]. The dynamic update of the primary partition is coordinated and has to be committed by the majority of members of the previous primary. This is deterministic and ensures that only one partition exists at a time. Using this mechanism, a replica that belongs to a primary partition can move to a non-primary partition when a view changes. In this case, the replication protocol only gets notified that the group has blocked and does not receive any view as it has not reintegrated in a primary partition.

5.3 Transaction's Read-set and 1-SR

The strictest correctness criterion to handle concurrent transactions in centralized databases is known as Serializability (SR) and has been accepted for a long time [23]. Most databases that offer SR do so by means of the strict two-phase locking mechanism which grants simultaneous access to objects (e.g., data items, pages, tables, etc.) for non-conflicting operations and blocks conflicting operations otherwise [51]. Intuitively, two operations conflict if they are issued by two distinct and concurrent transactions, access the same resource, and at least one of them is a write operation.

In a distributed scenario, the strictest correctness criterion to handle concurrent transactions

[2] We have not fully exploited the optimistic approach to avoid introducing complexity in the code and just used it to unpack messages and prepare them to be processed when the final order has been decided.

is One-Copy Serializability (1-SR) [23], which states that a concurrent execution of a set of transactions should be equivalent to a serial execution on a single database.

Conservative protocols (e.g., NODO [111]) achieve 1-SR by avoiding that two possibly conflicting transactions are concurrently executed. In optimistic protocols (e.g., PGR [76] and DBMS [104]), integrity problems arising from concurrent transactions executing at different replicas are filtered by a certification procedure that checks for conflicts (i.e., read-set & write-set) and aborts those transactions that do not preserve database integrity. AKARA, in its conservative step, is like NODO and, in its optimistic step, like both PGR and DBMS.

The lack of a clear definition on the read-set is troublesome as an ad-hoc extraction might commit transactions that were supposed to abort, resulting in integrity issues. Note that this is not about implementation details but rather about a key point lacking a precise specification.

In what follows, we show what is the read-set and how to extract it, but first, we present a simple model that is used throughout this section.

5.3.1 Model

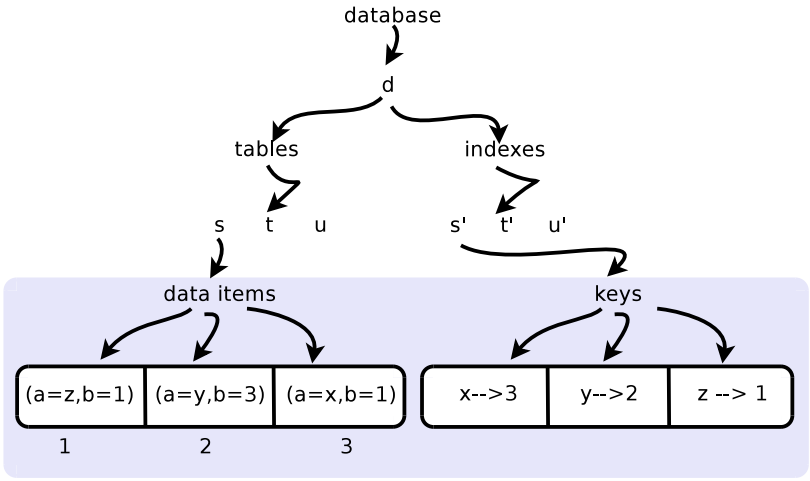


Figure 5.2: Hierarchical Path.

A relational database $db = \{s_1, \ldots, s_n\}$ is a set of relations (i.e., tables) in the form of $R \subseteq D_1 \times \ldots \times D_n$, where a domain is an arbitrary, non-empty set, finite or countably infinite. Each element $(a_1, a_2, \ldots, a_n)$ of a table s is a data item, and each a is an attribute (i.e., column), such that $a_n \in D_n$. We use $r.a$ to denote a column a of a table r, or simply a when there is no ambiguity. $s\prime_n$ denotes a unique-index (i.e., B-Tree Index [95]) over a table s built on a column a_n.

On behalf of a transaction t, one may execute: a query operation $t.Qr$ on s represented by $t.Qr \leftarrow (s.a_1 = x \ldots \wedge s.a_n = y)$, where $x \in D_1$ and $y \in D_n$; a change operation, either a delete or an update on s represented by $t.Up \leftarrow (s.a_1 = x \ldots \wedge s.a_n = y)$, where $x \in D_1$ and

$y \in D_n$; and an insert operation on s represented by $t.In \leftarrow (s.a_1 = x \ldots, s.a_n = y)$, where $x \in D_1$ and $y \in D_n$.

Figure 5.2 depicts a database d with tables s, t, u and their respective indexes $s\prime$, $t\prime$, $u\prime$. In particular, table s has two columns a and b, a non-unique index $s\prime$ over s built on column a and data items $(a = z, b = 1)$ $(a = y, b = 3)$ $(a = x, b = 1)$. In $s\prime$, $x \rightarrow 3$ indicates that the data item whose a equals x is the third data item in table s and so on.

5.3.2 Locks

Locks are a well-known mechanism to ensure logical consistency of a database and may be obtained at different granularities (e.g., keys, data items, pages, tables, etc.) and in different modes (e.g., Exlusive, Shared, etc.) and be held for different durations (e.g., instant duration and commit duration) [95–97].

Most mainstream DBMSs support a fine-granularity locking at the key level, although there are solutions that have a coarse-granulatiry such as Berkley DB [39] that provides page locking and MySQL's Storage Memory [38] that provides table locking.

Read operations acquire locks in either S (Shared) or IS (Intention Shared) mode while update operations acquire locks in either X (eXclusive) or IX (Intention Exclusive) mode. Different transactions may hold locks on the same object if they are compatible as indicated by the check marks ('$\surd$') in the lock mode compatibility matrix:

	S	X	IS	IX	SIX
S	$\surd$		$\surd$		
X					
IS	$\surd$		$\surd$	$\surd$	$\surd$
IX			$\surd$	$\surd$	
SIX			$\surd$		

DBMSs hold information structured hierarchically, as depicted in Figure 5.2, and any operation goes through this hierarchical structure before retrieving or updating a data item. To ensure database consistency, locks are acquired on the higher-level objects while traversing the hierarchical structure too. The intention mode locks give the privilege of requesting the corresponding intention or non-intention mode locks on lower-level objects. On the other hand, the non-intention mode locks implicitly grant locks of the corresponding mode on lower-level objects. For example, SIX on a table implicitly grants S on all the data items of that table, and allows X to be requested on the data items.

To access a specific data item or a set of data items, no matter how complex a statement (i.e., update, a delete, or a query) is, a DBMS scans either a table (*table scan*) where the data items are inserted or an index (*index scan*) on that table. While scanning either the table or the index, each data item is checked against a predicate extracted from the statement. The decision on what type of scan should be used takes into account the availability of an index and the size of the table, among other factors and is delegated to the optimizer which tries to choose the fastest execution path [51]. Unfortunately, a table scan may have a severe impact on performance, by reducing

the degree of concurrency, and further implications on the certification procedure, as we shall discuss. Note that an insert updates a table and every index on that table.

5.3.2.1 Serializability

Serializability theory has been studied for quite some time, and there is an extensive amount of literature on it [23]. It defines the correct criterion to execute concurrent transactions on a centralized database and states that such execution should be equivalent to a serial execution of the same set of transactions [23]. Clearly, this idea eases application development as programmers do not need to be concerned about database integrity issues that might result from concurrent executions. Mainstream databases provide SR by means of a strict two-phase locking mechanism [23].

Queries, Updates, and Deletes - Table Scan If a table scan is used, the table has either an X or S lock mode acquired on it. For instance, the operation $t.Up1 \leftarrow (a = x \wedge b = 1)$ acquires an IX lock mode on d and an X lock mode on s. On the other hand, $t.Qr2 \leftarrow (a = x \wedge b = 1)$ acquires an IS lock mode on d and an S lock mode on s. Clearly, concurrent transactions cannot change any data item in s while such locks are held, thus severely harming performance. This is done to block concurrent transactions that can *potentially* disturb t's execution and ensure the equivalence to a serial execution.

Queries, Updates, and Deletes - Index Scan If an index scan is used, either an S or X lock mode is acquired on data items.[3] The exact set of data items on which locks are acquired is determined by the set of data items that satisfy the predicate in an operation along with the *last visited data item* while traversing an index. Generally, a database engine stops scanning an index when it finds an exact match for a predicate or when the current position does not satisfy it. Assuming the database presented in Figure 5.2, let us take a look at the following examples:

- $t.Qr3 \leftarrow (a = x)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = x, b = 1)$;
- $t.Qr4 \leftarrow (a < x)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = x, b = 1)$;
- $t.Qr5 \leftarrow (a \leq x)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = x, b = 1)$;
- $t.Qr6 \leftarrow (a = z)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = z, b = 1)$;

[3] Although there are differences between the key and data item locking, we omit them without harming correctness and use data item locking and key locking as synonymous in order to ease the presentation. Furthermore, we assume that a delete operation logically erases a data item. For further details, see [96].

- $t.Qr7 \leftarrow (a > z)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on a special data item named eof, as there is no subsequent data item after $(a = z, b = 1)$;

- $t.Qr8 \leftarrow (a \geq z)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on both data items $(a = z, b = 1)$ and eof;

- $t.Qr9 \leftarrow (a = yy)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = z, b = 1)$;

- $t.Qr10 \leftarrow (a > yy)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = z, b = 1)$ and eof;

- $t.Qr11 \leftarrow (a \geq yy)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = z, b = 1)$ and eof;

- $t.Qr12 \leftarrow (a < yy)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = x, b = 1)$, $(a = y, b = 3)$, and $(a = z, b = 1)$;

- $t.Qr13 \leftarrow (a \leq yy)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = x, b = 1)$, $(a = y, b = 3)$, and $(a = z, b = 1)$.

Clearly, if $t.Qr9$ is executed on behalf of t, any concurrent transaction cannot insert either $(a = yyy, b = 1)$, $(a = p, b = 1)$, or $(a = q, b = 1)$. The additional lock is a mechanism to block concurrent transactions that can *potentially* disturb t's execution and to ensure the equivalence to a serial execution.

Inserts If there is no index on a table where a data item is about to be inserted, an X mode lock is acquired on the table. Otherwise, when a key is about to be inserted into an index, an instant IX mode lock is acquired for the duration of the operation on the data item that corresponds to the next key or on the eof. Finally, an IX mode lock is acquired on the new data item to guarantee that concurrent inserts do not conflict with it. For instance, the operation $t.In14 \leftarrow (a = w, b = 1)$ acquires an IX lock mode on d, an IX lock on s, an instant IX mode lock on $(a = x, b = 1)$, and an IX mode lock on $(a = w, b = 1)$.

5.3.2.2 Snapshot Isolation

Snapshot Isolation was formally presented in [22]. In this isolation level, a transaction executes on a database snapshot, which means that the transaction sees its own changes and those made by transactions that were committed before it started executing. Changes made by concurrent transactions, however, are not visible to it. Consequently, reads are never blocked, and write operations are managed by locks on data items along with the first-committer-wins rule. This means that a transaction trying to update a data item must abort whenever a concurrent transaction updates the same data item and commits first.

Updates and Deletes - Table Scan It acquires an IX lock mode on d, an IX lock mode on s, and an X lock mode on the changed data item.

Updates and Deletes - Index Scan It acquires an IX lock mode on d, an IX lock mode on s, and an X lock mode on the changed data item.

Inserts It acquires an IX lock mode on d, an IX lock mode on s, and an X lock mode on the inserted data item.

Assuming the database presented in Figure 5.2, let us take a look at the following examples:

- $t.Up1 \leftarrow (a = x \wedge b = 1)$ acquires an IX lock mode on d, an IX lock mode on s, and an X lock mode on data item $(a = x, b = 1)$.
- There is no lock acquired for $t.Qr2$ - $t.Qr13$.
- $t.In14 \leftarrow (a = w, b = 1)$ acquires an IX lock mode on d, an IX lock on s and an IX mode lock on $(a = w, b = 1)$.

5.3.3 Read-set, Write-set, and Write-values

Both the write-sets and read-sets are employed to detect conflicts among concurrent transactions. In particular, the term *write-set* has been used throughout this thesis to designate changes on a database (i.e., data items) that are extracted, propagated to remote replicas, and applied on them. In the previous sections, however, we have discussed that update (i.e., insert, delete, and update) operations acquire locks on data objects which are not necessarily modified and cause additional conflicts among concurrent transactions. There is therefore a clear distinction between conflicts and changed data items.

The changed data items are what we call hereafter *write-values* and can be easily extracted, for example, by means of a trigger. On the other hand, the write-set is about conflicts, and this includes the instant IX mode locks acquired while processing an insert operation. Thus, taking into account the information presented on locks, we have what follows for **serializability**:

$$\text{write-set} \supseteq \text{write-values}$$

For **snapshot isolation**, as locks are only acquired on changed data items, we have what follows:

$$\text{write-set} = \text{write-values}$$

Therefore we can define the write-set as the set of *potential conflicts* generated due to delete, insert, and update operations.

If we consider that to change a data item, it must be first read and artificially assume that an insert obeys the same rule, the expressions above can be augmented as follows for both **serializability** and **snapshot isolation**:

$$\text{read-set} \supseteq \text{write-set} \supseteq \text{write-values}$$

Therefore we can define the read-set as the set of *potential conflicts* generated due to delete, insert, update, and query operations.

In what follows, we describe how the read- and write-sets can be extracted when the database provides either **serializability** or **snapshot isolation**. The idea consists of augmenting the table and index scan iterators [51]. In particular for the index scan, we extract the expression used to filter data and change the right end-point if the *last visited data item* while traversing the index is not included there.[4]

5.3.3.1 Serializability

For a system that locally provides serializability and the statements presented before, the read-set would have:

- $t.Up1 \leftarrow (a = x \wedge b = 1)$ acquires an IX lock mode on d and an X lock mode on s. The read-set would have $-\infty < a \leq \infty$;
- $t.Qr2 \leftarrow (a = x \wedge b = 1)$ acquires an IS lock mode on d and an S lock mode on s. The read-set would have $-\infty < a \leq \infty$;
- $t.Qr3 \leftarrow (a = x)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = x, b = 1)$. The read-set would have $x \leq a \leq x$;
- $t.Qr4 \leftarrow (a < x)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = x, b = 1)$. The read-set would have $-\infty < a \leq x$;
- $t.Qr5 \leftarrow (a \leq x)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = x, b = 1)$. The read-set would have $-\infty < a \leq x$;
- $t.Qr6 \leftarrow (a = z)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = z, b = 3)$. The read-set would have $z \leq a \leq z$;
- $t.Qr7 \leftarrow (a > z)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on a special data item named eof, as there is no subsequent data item after $(a = z, b = 3)$. The read-set would have $z < a < \infty$;
- $t.Qr8 \leftarrow (a \geq z)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on both data items $(a = z, b = 3)$ and eof. The read-set would have $z \leq a < \infty$;
- $t.Qr9 \leftarrow (a = yy)$ acquires an IS mode lock on d, an IS mode lock on s and an S mode lock on data item $(a = z, b = 3)$. The read-set would have $yy \leq a \leq zz$;

[4]Note that by this definition, concurrent inserts that do not conflict locally will be aborted by the certification procedure. These spurious aborts are acceptable as they do not harm correctness. This is done to ease the presentation and avoid cluttering the text with unnecessary details.

- $t.Qr10 \leftarrow (a > yy)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = z, b = 1)$ and eof. The read-set would have $yy < a < \infty$;
- $t.Qr11 \leftarrow (a \geq yy)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = z, b = 1)$ and eof. The read-set would have $yy \leq a < \infty$;
- $t.Qr12 \leftarrow (a < yy)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = x, b = 1)$, $(a = y, b = 3)$, and $(a = z, b = 1)$. The read-set would have $-\infty \leq a \leq zz$;
- $t.Qr13 \leftarrow (a \leq yy)$ acquires an IS mode lock on d, an IS mode lock on s, and an S mode lock on data item $(a = x, b = 1)$, $(a = y, b = 3)$, and $(a = z, b = 1)$. The read-set would have $-\infty \leq a \leq zz$;
- $t.In14 \leftarrow (a = w, b = 1)$ acquires an IX lock mode on d, an IX lock on s, an instant IX mode lock on $(a = x, b = 1)$, and an IX mode lock on $(a = w, b = 1)$. The read-set would have $w \leq a \leq x$.

5.3.3.2 Snapshot Isolation

For a system that locally provides snapshot and the statements presented before, the read-set would have:

- $t.Up1 \leftarrow (a = x \wedge b = 1)$ acquires an IX lock mode on d, an IX lock mode on s, and an X lock mode on data item $(a = x, b = 1)$. The read-set would have $-\infty < a \leq \infty$;
- $t.Qr2 \leftarrow (a = x \wedge b = 1)$ acquires no lock. The read-set would have $-\infty < a \leq \infty$;
- $t.Qr3 \leftarrow (a = x)$ acquires acquires no lock. The read-set would have $x \leq a \leq x$;
- $t.Qr4 \leftarrow (a < x)$ acquires no lock. The read-set would have $-\infty < a \leq x$;
- $t.Qr5 \leftarrow (a \leq x)$ acquires no lock. The read-set would have $-\infty < a \leq x$;
- $t.Qr6 \leftarrow (a = z)$ acquires no lock. The read-set would have $z \leq a \leq z$;
- $t.Qr7 \leftarrow (a > z)$ acquires no lock. The read-set would have $z < a < \infty$;
- $t.Qr8 \leftarrow (a \geq z)$ acquires no lock. The read-set would have $z \leq a < \infty$;
- $t.Qr9 \leftarrow (a = yy)$ acquires no lock. The read-set would have $yy \leq a \leq zz$;
- $t.Qr10 \leftarrow (a > yy)$ acquires no lock. The read-set would have $yy < a < \infty$;
- $t.Qr11 \leftarrow (a \geq yy)$ acquires no lock. The read-set would have $yy \leq a < \infty$;
- $t.Qr12 \leftarrow (a < yy)$ acquires no lock. The read-set would have $-\infty \leq a \leq zz$;
- $t.Qr13 \leftarrow (a \leq yy)$ acquires no lock. The read-set would have $-\infty \leq a \leq zz$;

- $t.In14 \leftarrow (a = w, b = 1)$ acquires an IX lock mode on d, an IX lock on s, and an IX mode lock on $(a = w, b = 1)$. The read-set would have $w \leq a \leq w$.

5.3.4 Open Issues

The way to mix SI and SR and have a serializable execution as an outcome by either changing the application's code or combining transactions with different isolation levels, i.e., SI and SR, is described in [8]

In [30], it is investigated how to achieve 1-SR with SI. That study provides a tool to analyze the transactional log and determine which transactions (i.e., applications' code) must be changed. Although transactions may be submitted to any replica, there is a master determining a global consistency criterion, which resembles a centralized execution as studied in [8].

A modification to a concurrency control algorithm to take advantage of the SI while at the same time providing SR as an outcome is presented in [31]. There are other works based on serialization graph testing [23, 24]. However, according to [31], the space and the overhead to maintain them might be prohibitive.

Although the idea of providing SR on databases with SI has been discussed in different contexts, such ideas cannot be directly applied to our problem as they either assume a centralized database or replica or require changes to the application's code. In contrast, our proposal does not require a centralized replica or changes to the application's code and can be used in heterogenous environments where the databases provide either SR or SI. However, to facilitate the presentation and avoid cluttering the text with unecessary details.

It is worth noting that users may override the locking mechanism by explicitly defining locking hints, also known as advisory locks or user locks. Clearly, this affects the read-set and as such must be extracted and taken into consideration in the certification procedure. Finally, one needs to manage concurrent transactions, executing on different replicas, inserting duplicate values into unique indexes.

Mainstream databases (e.g., MS SQL-Server) have a lock scaling mechanism that avoids managing a large collection of locks, usually locks on data items. Thus, according to a defined threshold, this mechanism grabs a lock on a coarse grain, such as a page or a file, which encompasses the previous locks, and then releases the finer locks. However, pages or files are physical objects that most likely do not have valid semantics across different replicas. The extraction procedure must regard these locks at a logical level (e.g., regarding the page lock as a table lock). The same idea must be applied on databases in which pages are their finest grain.

When we tested our idea in the PostgreSQL and SleepyCat, the extraction did not have a significant impact on performance. Furthermore, building heterogeneous clusters with DBMSs that provide different consistency criteria is feasible as long as we extract the write- and read-set, as described in this section. In fact, guaranteeing 1-SR on clusters with databases that provide SI is easier because there is a large variety of locking algorithms with subtle details to provide SR.

5.4 Reading Your Writes

The group-based protocols adopt the read-one/write-all-available approach for availability and scalability purposes [70, 76, 104, 111]. It offers minimal overhead for read operations and tends to outperform any other quorum settings [109]. To further improve performance, read-only transactions are not handled by the replication protocol and are not subject to any global synchronization. From a data-centric perspective, the intuition supporting the uncoordinated handling of read-only transactions is that as long as replica control is done on the transactions' boundaries, reordering a read-only transaction does not impair the Serializability of the execution. However, this reordering cannot be applied in general as it might easily contradict the users' local, observable, order of events and violate causal consistency which is subsumed by stronger consistency criteria such as SR and SI [2].

Causal consistency means that any two causally related write operations must be seen by all server replicas in the same order. As demonstrated by Brzeziński et al. [29], causal consistency requires four basic session guarantees to be preserved: read your writes, monotonic reads, monotonic writes, and writes follow reads. From these, the first two are endangered by the uncoordinated handling of read-only transactions. The read your writes condition ensures that a read operation is executed by a database replica that has performed all writes previously issued by the requesting client. Monotonic reads ensures that read operations are executed by database replicas that have performed all writes seen by previous reads of the requesting client.

The group-based protocols presented in this paper may fail to guarantee both conditions whenever transactions from the same client are handled by different replicas.

Notice that the above phenomenon is of the sole responsibility of the replica control protocol and is independent of the centralized consistency criteria of the replicas. Since the problem may only occur when transactions from the same client are handled by different replicas, a general workaround would be to simply have all requests of a client be sent to the same replica. This could be done either by the replication protocol itself or by delegating it to a load-balancing layer that would preserve client/replica affinity.

Unfortunately, the assumption of client/replica affinity is increasingly difficult to ensure in the typical usage scenario for database clusters, which are multi-tier systems. In contrast with traditional systems, in which each the user maintains a private session and a private database connection, it is now common that clients connect to an application server which maintains a pool of database connections and dynamically dispatches requests on behalf of multiple clients. Additionally, a caching layer maintained within the application server might be used by multiple clients. Finally, the same end-user might even use multiple concurrent connections to the application server that cannot be easily tracked to a single entity. It is, therefore, unfeasible to assume that connections to different replicas are unrelated.

In [99], we have proposed to handle read-only transactions and update transactions in the same way.

5.5 Conclusion

This chapter proposed building blocks to provide full-fledged group-based replication protocols and described the challenges in designing and implementing them. In particular, we have used a generic interface to group communication services, titled jGCS, to wrap multiple group communication toolkits and to be able to exploit different group communicaton algorithms without changing the replication protocols.

We have also shown how to build heterogeneous clusters where clients can exploit features only available at a replica and still guarantee 1-SR. The key to this achievement was a clear definition of the read-set and its extraction.

Chapter 6

Conclusion

Shared-nothing clusters have been proposed as a cost-effective approach for fault-tolerance and scalability. Unfortunately, mainstream protocols lead to poor scalability with traditional on-line transaction processing (OLTP) workloads or introduce a single point of failure, thus endangering availability.

To circumvent these problems, several group-based replication protocols have been proposed. Unfortunately, even these protocols have drawbacks that are an obstacle to their use in practice. The lack of native interfaces on database engines for providing access to features that are required for the development, evaluation, and deployment of these protocols has also been a major issue.

6.1 Research Assessment

This thesis has tackled these problems and has three main achievements that are summarized as follows.

AKARA The performance of group-based database replication protocols can be challenged by demanding workloads. Namely, conservatively synchronized protocols overly restrict concurrency, and thus throughput, unless a careful application-specific definition of conflict classes is done. On the other hand, optimistically synchronized protocols make it difficult to commit long-running and prone to conflicts transactions. Finally, both depend on shipping updated data items, which makes it hard to deal with very large updates or DDL statements. Although all these issues can easily be avoided in benchmarks, they represent a significant hurdles to adoption in real scenarios.

In this thesis, we have addressed these issues with the AKARA protocol, which seamlessly combines multiple execution strategies. Experimental evaluation with the TPC-C workload have shown that the proposed protocol provides adequate throughput without requiring application-specific tuning of conflict classes. By introducing a small number of transactions with large write sets or DDL statements in the mix to be actively replicated, one can see that fairness is ensured and network usage is minimized.

GAPI The development of new DBMS plugins for different purposes, such as replication and clustering, require more functionality than is currently provided by transactional APIs such

as xDBC, stored procedures, and triggers. Previously suggested solutions to meet the demand for greater functionality, such as patching the database kernel or building complex wrappers, require a large development effort in supporting code, cause performance overhead, and reduce the portability of the middleware.

This thesis has advocated the use of a reflective architecture and interface to expose the relevant information about transaction processing in a useful way, namely allowing it to be observed and modified by external plugins. We have shown the usefulness of the approach by illustrating how the interface can by applied to different settings, such as replication, query caching, among others. We have also shown that the approach is viable and cost-effective, by describing its instantiation on four different and representative architectures, namely the Apache Derby, PostgreSQL, Sleepycat, and the Sequoia server wrapper. We have measured the overhead introduced by the in-core implementation on Apache Derby and compared it with a middleware solution. These prototypes are published as open-source and can be downloaded from the GORDA project's home page, examined, and benchmarked.

Designing and integrating components Assembling several components to provide a set of building blocks to develop full-fledged replication protocols is not an easy task and as such must overcome several obstacles.

In this thesis, we have used a generic interface to group communication services, titled jGCS, to wrap multiple group communication toolkits and be able to exploit different group communication algorithms without changing the replication protocols.

We have shown how to build heterogeneous clusters where clients can exploit features only available on a replica and still guarantee 1-SR. The key to this achievement was a clear definition of the read-set and its extraction.

6.2 Future Directions and Open Issues

There are some points that were not analyzed in detail in this work. In what follows, we briefly present and discuss such points and suggest future work.

AKARA Most benchmarks are modeled as an open or closed system, however, a partly open system is more accurate for most real scenarios. In particular, the TPC-C is modeled as a closed system, where new requests (e.g., transactions) are only triggered by completions of previous requests, following a think time [127]. Open and partly open systems have a greater degradation in performance due to contention when compared with closed systems: a higher average response time and reduced throughput. The variability of service demand also has a major impact on the mean response time. For those reasons, it is extremely important to evaluate the protocols presented here, in particular AKARA, with a partly open benchmark to figure out whether it would behave as expected or not.

Although the current implementation of AKARA statically specifies the multiprogramming limit (MPL) by establishing the number of optimistic transactions that can be concurrently executed on a replica, this information could be dynamically defined as in [126]. One might use an adaptive mechanism [90] to determine this value, taking into account the idleness of the database and the abort rate due to the optimistic execution.

Deciding whether a transaction should be passively or actively executed is a task that might be done automatically or manually. In the former case, AKARA might learn from previous executions of a transaction to come up with a decision. Usually, the higher the number of changes the more appropriate is the use of an active replication. Furthermore, AKARA might exploit the GAPI [41] to extract information from a database such as the number of changes made by a transaction and whether there are DDL statements or not. The GAPI might also be used to help in removing non-deterministic information in statements by withdrawing most of the work from the replication middleware.

GAPI The architecture presented still has some limitations that should be addressed by future work. It needs to be extended to perform the composition of multiple independent plugins. For instance, it needs to find a way to configure a DBMS to use a self-management plugin and a replication plugin at the same time. Previous work on reflective systems might provide a direction in solving this problem [7]. Another open issue is the adequacy of the proposed architecture to non-classical database architectures, namely, how to match it with a column-oriented DBMS, such as MonetDB [68], or with an inherently clustered DBMS, such as Oracle RAC [118].

Designing and integrating components The read-set is composed of the predicates used in table and index scans. Obviously, the more restrictive the predicate the lesser the number of conflicts. The database engine, in particular the optimizer, has an important role in reducing such conflicts [96] as it is responsible for generating an execution plan and thus deciding on the type of scan and predicate to be used. In this thesis, we do not evaluate the trade-off between performance and choosing a slower plan to reduce the number of conflicts. This not an easy task and requires in-depth knowledge of the optimizer along with an adaptive mechanism as long-running transactions have a direct impact on the overall performance and on the number of aborts.

Bibliography

[1] Jgcs. http://jgcs.sf.net.

[2] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, MIT, 1999.

[3] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A Linguistic Framework for Dynamic Composition of Dependability Protocols. In *IFIP DCCA*, 1992.

[4] D. Agrawal and A. E. Abbadi. The generalized tree quorum protocol: an efficient approach for managing replicated data. *ACM TODS*, 17:689–717, 1992.

[5] R. Agrawal, M. J. Carey, and M. Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM TODS*, 12:609 – 654, 1987.

[6] F. Akal, C. Türker, H. Schek, Y. Breitbart, T. Grabs, and L. Veen. Fine-grained replication and scheduling with freshness and correctness guarantees. In *VLDB Conference*, 2005.

[7] M. Aksit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In *ECOOP*, 1992.

[8] M. Alomari, M. Cahill, A. Fekete, and U. Rohm. Serializable Executions with Snapshot Isolation: Modifying Application Code or Mixing Isolation Levels. In *DASFAA*, 2008.

[9] C. Amaza, A. Cox, and W. Zwaenepoel. Conflict-Aware Scheduling for Dynamic Content Applications. In *USENIX USITS*, 2003.

[10] Y. Amir, C. Danilov, M. M. Amir, J. Stanton, and C. Tutu. On the performance of consistent wide-area database replication. Technical report, John Hopkins University, 2002.

[11] Y. Amir, C. Danilov, M. M. Amir, J. Stanton, and C. Tutu. Pratical wide-area database replication. Technical report, John Hopkins University, 2002.

[12] Y. Amir, C. Danilov, and J. Stanton. A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication. In *IEEE DSN*, 2000.

[13] Y. Amir and C. Tutu. From Total Order to Database Replication. In *IEEE ICDCS*, page 494, 2002.

[14] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, consistency, and practicality: are these mutually exclusive? In *ACM SIGMOD*, 1998.

[15] N. S. Arora. Oracle Streams for Ner Real Time Asynchronous Replication. In *Workshop on Database Replication collocated at VLDB Conference*, 2005.

[16] O. Babaoglu, R. Davoli, and A. Montresor. Group Membership and View Synchrony in Partitionable Asynchronous Distributed Systems: Specifications. *ACM SIGOPS*, 31:11–22, 1997.

[17] R. Baldoni, S. Cimmino, C. Marchetti, and A. Termini. Performance Analisys of Java Group Toolkits: a Case Study. In *FIDJI*, 2002.

[18] B. Ban. Design and Implementation of a Reliable Group Communication Toolkit for Java. http://www.cs.cornell.edu/home/bba/Coots.ps.gz, 1998.

[19] R. Barga and C. Pu. A Practical and Modular Implementation of Extended Transaction Models. In *VLDB Conference*, 1995.

[20] A. Bartoli and O. Babaoglu. Selecting a "Primary Partition" in Partitionable Asynchronous Distributed Systems. In *IEEE SRDS*, 1997.

[21] R. G. Bello, K. Dias, A. Downing, J. J. F. Jr., W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized Views in Oracle. In *VLDB Conference*, 1998.

[22] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *ACM SIGMOD*, 1995.

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

[24] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM TODS*, 8:465–483, 1983.

[25] K. Birman and R. van Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, 1993.

[26] S. Borzsonyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *IEEE ICDE*, 2001.

[27] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databates. *SIGMOD Rec.*, 28:97–108, 1999.

[28] H. Breitwieser and M. Leszak. A distributed transaction processing protocol based on majority consensus. In *ACM SIGOPS*, 1982.

[29] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. From Session Causality to Causal Consistency. In *PDP Euromicro*, 2004.

[30] M. Cahill, A. Fekete, and U. Rohm. Towards serializable replication with snapshot isolation. In *PhD Workshop collocated at VLDB Conference*, 2007.

[31] M. Cahill, U. Röhm, and A. Fekete. Serializable isolation for snapshot databases. In *ACM SIGMOD*, 2008.

[32] E. Cecchet. C-JDBC Horizontal Scalability design. Technical report, ObjectWeb, 2004.

[33] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC:Flexible Database Clustering Middleware. In *USENIX Annual Technical Conference*, 2004.

[34] G. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A comprehensive Study. *ACM CSUR*, 2001.

[35] Continuent. Sequoia 4.x. http://sequoia.continuent.org, 2008.

[36] T. H. Cormen, C. E. Leierson, and R. L. Rivest. *Introduction to Algorithms*. Mc Graw Hill, 1990.

[37] I. Corporation. A Practical Guide to DB2 UDB Data Replication V8. http://www.redbooks.ibm.com/abstracts/sg246828.html, 2008.

[38] M. Corporation. MySQL On-line Documentation. http://dev.mysql.com, 2008.

[39] O. Corporation. Oracle. http://www.oracle.com/, 2008.

[40] S. Corporation. Veritas volume replicator. http://www.symantec.com/business/volume-replicator, 2008.

[41] A. Correia, J. Orlando, L. Rodrigues, N. Carvalho, R. Oliveira, and S. Guedes. Gorda: An Open Architecture for Database Replication. In *IEEE NCA*, 2007.

[42] A. Correia, J. Pereira, and R. C. Oliveira. AKARA: A Flexible Clustering Protocol for Demanding Transactional Workloads. In *DOA*, 2008.

[43] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM CSUR*, 36, 2004.

[44] D. Dolev, D. Malki, and R. Strong. A framework for partitionable membership service. In *ACM PODC*, 1996.

[45] D. L. Eager and K. C. Sevcik. Achieving robustness in distributed database systems. *ACM TODS*, 8:354–381, 1983.

[46] Emic. Emic Application Cluster (EAC). http://www.emicnetworks.com, 2003.

[47] J. Fabre and T. Perennou. A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach. *ACM TOCS*, 47:78–95, 1998.

[48] J. M. Franco, M. P. M. R. J. Peri and, and B. Kemme. Adaptive middleware for data replication. In *USENIX International Conference on Middleware*, 2004.

[49] R. Friedman and E. Hadad. Client-side Enhancements using Portable Interceptors. In *WORDS*, 2001.

[50] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Desing Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[51] H. Garcia-Mollina, J. D. Ullman, and J. Widom. *Database Systems The Complete Book*. Prentice Hall, 2002.

[52] I. Gashi, P. Popov, and L. Strigini. Fault Tolerance via Diversity for Off-the-Shelf Products: A Study with SQL Database Servers. *IEEE TDSC*, 4:280–294, 2007.

[53] D. K. Gifford. Weighted voting for replicated data. In *ACM SOSP*, 1979.

[54] J. Goldstein and P. Larson. Optimizing Queries using Materialized Views: A Practical, Scalable Solution. In *ACM SIGMOD*, 2001.

[55] N. Goodman, D. Skeen, A. Chan, U. Dayal, S. Fox, and D. Ries. A recovery algorithm for a distributed database system. In *ACM SIGMOD*, 1983.

[56] GORDA. *Preliminary GORDA Architecture and Interfaces*, October 2005.

[57] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM CSUR*, 25:73–169, 1993.

[58] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *ACM SIGMOD*, 1996.

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

[60] P. G. D. Group. PostgreSQL. http://www.postgresql.org, 2003.

[61] R. Guerraoui, B. Garbinato, and K. Mazouni. Garf: A Tool for Programming Reliable Distributed Applications. *IEEE IPDPS*, 5:32–39, 1997.

[62] R. Guerraoui, D. Kostic, R. Levy, and V. Quema. A High Throughput Atomic Storage Algorithm. In *IEEE ICDCS*, 2007.

[63] H. Guo, P. Larson, and R. Ramkrishnan. Caching with "God Enough", Concurrency, Consistency, and Completeness. In *VLDB Conference*, 2005.

[64] A. Gupta and I. S. Mumick, editors. *Materialized Views Techniques, Implementations, and Applications*. MIT Press, 1999.

[65] V. Hadzilacos and S. Toueg. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Technical report, Cornell University, 1994.

[66] A. Y. Halevy. Answering Queries using Views: A Survey. *VLDB Journal*, 10:270–294, 2001.

[67] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. In *ACM TOPLAS*, 1990.

[68] M. Ivanova, N. Nes, R. Goncalves, and M. L. Kersten. MonetDB/SQL Meets SkyServer: the Challenges of a Scientific Database. In *SSDBM*, 2007.

[69] R. Jiménez-Peris and M. Patiño-Martínez. Replica Control. In *Encyclopedia of Database Systems*. Springer US, 2009.

[70] A. C. Jr., A. Sousa, L. Soares, J. Pereira, F. Moura, and R. Oliveira. Group-based Replication of On-line Transaction Processing Servers. In *LADC*, 2005.

[71] M. Kaashoek and A. Tanenbaum. Group Communication in the Amoeba Distributed Operating System. In *IEEE ICDCS*, 1991.

[72] V. Kathuria, R. Dhamankar, and H. Kodavalla. Transaction Isolation and Lazy Commit. In *IEEE ICDE*, 2007.

[73] I. Keidar and D. Dolev. *Dependable Network Computing*, chapter Totally ordered broadcast in the face of network partitions. Kluwer Academic, 2000.

[74] B. Kemme and G. Alonso. A Suite of Database Replication Protocols Based on Group Communication Primitives. In *IEEE ICDCS*, 1998.

[75] B. Kemme and G. Alonso. A New Approach to Developing and Implementing Eager Database Replication Protocols. In *ACM TODS*, 2000.

[76] B. Kemme and G. Alonso. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *VLDB Conference*, 2000.

[77] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36:41–50, 2003.

[78] G. Kiczales. Towards a New Model of Abstraction in Software Engineering. In *IMSA Workshop on Reflection and Meta-level Architectures*, 1992.

[79] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM CSUR*, 32:422–469, 2000.

[80] N. Krivokapić, A. Kemper, and E. Gudes. Deadlock Detection in Distributed Database Dystems: A New Algorithm and A Comparative Performance Analysis. *VLDB Journal*, 1999.

[81] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM TODS*, 6:213 – 226, 1981.

[82] L. Lamport. Time, clocks, and the ordering of events in a distributed. *CACM*, 21:558 – 565, 1978.

[83] P. Larson, J. Goldstein, and J. Zhou. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In *IEEE ICDCS*, 2004.

[84] Y. Lin, B. Kemme, R. J. Peris, and M. P. Martínez. Middleware based Data Replication providing Snapshot Isolation. In *ACM SIGMOD*, 2005.

[85] Linbit. Drdb. http://www.drbd.org/, 2008.

[86] P. V. M. T. Özsu. *Principles of Distributed Database Systems*. Prentice Hall International, 1999.

[87] P. Maes. Concepts and Experiments in Computational Reflection. In *OOPSLA*, 1987.

[88] C. Marchetti. CORBA Request Portable Interceptors: A Performance Analysis. In *DOA*, 2001.

[89] P. Martin, W. Powley, and D. Benoit. Using Reflection to Introduce Self-Tuning Technology into DBMSs. In *IDEAS*, 2004.

[90] M. Matos, J. A. Correia, J. Pereira, and R. Oliveira. Serpentine: adaptive middleware for complex heterogeneous distributed systems. In *ACM SAC*, 2008.

[91] Microsoft. Microsoft SQL Server. http://www.microsoft.com/sqlserver, 2010.

[92] H. Miranda, A. Pinto, and L. Rodrigues. Appia: a flexible protocol kernel supporting multiple coordinated channels. In *IEEE ICDCS*, 2001.

[93] T. Mishima and H. Nakamura. Pangea: An Eager Database Replication Middleware guaranteeing Snapshot Isolation without Modification of Database Servers. In *VLDB Conference*, 2009.

[94] A. Mitani. Pg clusters. http://www.pgcluster.org/, 2008.

[95] C. Mohan. Aries/IM: An efficient and High Concurrency Index Managemet Method Using Write-Ahead Logging. In *ACM SIGMOD*, 1992.

[96] C. Mohan. Concurrency Control and Recovery Methods for B+-Tree Indexes: ARIES/KVL and ARIES/IM. Technical report, IBM Research Division, 1994.

[97] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, 17, 1992.

[98] S. Mullender. *Distributed Systems*. Addison-Wesley Publishing Co., 1993.

[99] R. Oliveira, J. Pereira, A. C. Jr, and E. Archibald. Revisiting 1-Copy Equivalence in Clustered Databases. In *ACM SAC*, 2006.

[100] Oracle. Oracle Berkeley DB Java Edition. http://www.oracle.com/database/berkeley-db/je/index.html, 2008.

[101] F. Pedone and S. Frølund. Pronto: High availability for standard off-the-shelf databases. *J. Parallel Distrib. Comput.*, 68:150–164, 2008.

[102] F. Pedone, R. Guerraoui, and A. Schiper. Transaction Reordering in Replicated Databases. In *IEEE SRDS*, 1997.

[103] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting Atomic Broadcast in Replicated Databases. In *Euro-par*, 1998.

[104] F. Pedone, R. Guerraoui, and A. Schiper. The Database State Machine Approach. In *Journal of Distributed and Parallel Databases and Technology*, 2003.

[105] F. Pedone and A. Schiper. Optimistic Atomic Broadcast. In *DISC*, 1998.

[106] F. Pedone and A. Schiper. Handling message semantics with Generic Broadcast protocols. *Distributed Computing*, 15:97–107, 2002.

[107] J. Pereira, L. Rodrigues, M. Monteiro, R. Oliveira, and M. Kermarrec. NeEM: Network-friendly Epidemic Multicast. In *IEEE SRDS*, 2003.

[108] J. Pereira, L. Rodrigues, and R. Oliveira. Semantically Reliable Multicast: Definition, Implementation and Performance Evaluation. *IEEE TC*, 52:150–165, 2003.

[109] R. J. Peris, M. P. Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM TODS*, 28:257–294, 2003.

[110] R. J. Peris, M. P. Martínez, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *DISC*, 2000.

[111] R. J. Peris, M. P. Martínez, B. Kemme, and G. Alonso. Improving the Scalability of Fault-Tolerant Database Clusters. In *IEEE ICDCS*, 2002.

[112] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/Rule based Query Rewrite Optimization in Starburst. In *ACM SIGMOD*, 1992.

[113] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *USENIX International Conference on Middleware*, 2004.

[114] J. Plewe. Poleposition. http://www.polepos.org/, 2008.

[115] PostgreSQL. Slony-I. http://www.slony.info/, 2008.

[116] D. Powell, M. Chéréque, and D. Drackley. Fault-tolerance in Delta-4*. *ACM SIGOPS*, 25:122–125, 1998.

[117] W. Powley and P. Martin. A Reflective Database-Oriented Framework for Autonomic Managers. In *ICAS*, 2006.

[118] A. Pruscino. Oracle RAC: Architecture and Performance. In *ACM SIGMOD*, 2003.

[119] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled Query Execution Engine using JVM. In *IEEE ICDE*, 2006.

[120] L. Rodrigues and N. Carvalho. Supporting Linearizability Semantics in Replicated Databases. In *IEEE NCA*, 2008.

[121] L. Rodrigues, H. Fonseca, and P. Veríssimo. Totally Ordered Multicast in Large-Scale Systems. In *IEEE ICDCS*, 1996.

[122] L. Rodrigues, J. Mocito, and N. Carvalho. From Spontaneous Total Order to Uniform Total Order: different degrees of optimistic delivery. In *ACM SAC*, 2006.

[123] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM TOCS*, 10:1–15, 1992.

[124] J. Salas, R. J. Peris, M. P. Martínez, and B. Kemme. Lightweight reflection for middleware-based database replication. In *IEEE SRDS*, pages 377–390, Washington, DC, USA, 2006. IEEE Computer Society.

[125] F. Schneider. Replication Management using the State-Machine Approach. In *Distributed Systems*. ACM Press/Addison-Wesley Publishing Co., 1993.

[126] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wiernam. How to determine a good multi-programming level for external scheduling. In *IEEE ICDE*, 2006.

[127] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open Versus Closed: A Cautionary Tale. In *NSDI*, 2006.

[128] X. D. A. S. N. Sergent. Semi-passive replication. In *IEEE SRDS*, 1998.

[129] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial Replication in the Database State Machine. In *IEEE NCA*, 2001.

[130] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic Total Order in Wide Area Networks. In *IEEE SRDS*, 2002.

[131] A. Sousa, J. Pereira, L. Soares, A. C. Jr., L. Rocha, R. Oliveira, and F. Moura. Testing the Dependability and Performance of GCS-Based Database Replication Protocols. In *IEEE DSN*, 2005.

[132] I. Sun Microsystems. Derby. http://db.apache.org/derby/.

[133] Sybase. Data Replication Server Software. http://www.sybase.com/, 2008.

[134] L. Tan, M. Kerby, and J. Senicka. Veritas Performance brief - Remote Mirroring using VxVM for Metropolitan Area Disaster Recovery. Technical report, Symantec Corporation, 2004.

[135] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM TODS*, 4:180–209, 1979.

[136] A. Thomasian. Concurrency Control: Methods, Performance, and Analysis. *ACM CSUR*, 30:70–119, 1998.

[137] I. L. Traiger, J. Gray, C. A. Galtieri, and B. G. Lindsay. Transactions and consistency in distributed database systems. *ACM TODS*, 7:323–342, 1982.

[138] Transaction Processing Performance Council (TPC). TPC benchmark C Standard Specification Revision 5.0, 2001.

[139] M. Tumma. *Oracle Streams High Speed Replication and Data Sharing*. Rampant, 2004.

[140] P. Vicente and L. Rodrigues. An Indulgent Uniform Total Order Algorithm with Optimistic Delivery. In *IEEE SRDS*, 2002.

[141] G. Vossen and G. Weikum. *Transactional Information Systems: Theory, Algorithms, And The Practice Of Concurrency Control And Recovery*. Morgan Kaufmann, 2002.

[142] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database Replication Techniques: A Three Parameter Classification. In *IEEE SRDS*, 2000.

[143] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding Replication in Databases and Distributed Systems. In *IEEE ICDCS*, 2000.

[144] M. Wiesmann and A. Schiper. Beyond 1-Safety and 2-Safety for replicated databases: Group-Safety. In *EDBT*, 2004.

[145] L. Wong, N. S. Arora, L. Gao, T. Hoang, and J. Wu. Oracle Streams: a High Performance Implementation for Near Real Time Asynchronous Replication. In *IEEE ICDE*, 2009.

[146] S. Wu and B. Kemme. Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation. In *IEEE ICDE*, 2005.

Appendix A

Requirements for Replication-friendly Databases

A.1 Requirements

To achieve modularity without losing performance, the replicator must have access to a set of features provided by the DBMSs:

Lifecycle Events Mechanisms to observe and control the life-cycle of a DBMS site, namely, when a site is started, recover from logs and put on-line to clients. This is required for proper recovery in coordination with remote sites, as a local log might have to be complemented by a remote log or even a state-transfer.

Object and Transaction Meta-Information Mechanisms to record and retrieve protocol-specific meta-information associated both with database objects and with transactions. For instance, global object identifiers or timestamps are needed by multiple protocols.

Statement Inspection Interception of statements submitted by clients either in a textual format or structured as parsed tree. This is required to disseminate statements to replicas, as well, as in most circumstances, to handle DDL statements.

Statement Modification Modification or cancellation of statements, either in a textual format or structured as parsed tree. This is required to remove non-deterministic operations, to cope with partitioning and with incompatible SQL dialects in heterogeneous environments.

Write-set Extraction Capturing updates done to a database in a format that can be transferred and applied remotely.

Read-set Extraction Capturing "identifiers of objects read" to achieve strict consistencies such as 1-Copy Serializability [23] (1-SR). This is required when eager update-anywhere protocols are designed.

Efficient Write-set Injection Although write-sets can always be applied using a regular client interface, it is hard to satisfy both correctness requirements to apply updates in a pre-defined order with performance. This often requires that updates are combined and scheduled beforehand to be applied in parallel.

Transactional Events Observing transactional events such as transaction begin, rollback, or commit. For instance, it can ease the implementation of efficient parallel update application by allowing a predictable commit order to be established. It also can be used to put the transaction on hold while its changes are validated.

Predictable Deadlock Handling Deterministic deadlock resolution mechanism, that can be controlled by middleware. This is required to ensure that validated transactions are not aborted by locally executing transactions to resolve deadlocks.

Result-Set Injection Replace result-sets returned to clients. This is required to reconcile results from multiple database fragments when statements are shipped to remote replicas to improve either efficiency or fault-tolerance by diversity.

Runtime Model A uniform runtime model for portable replication middleware components. Mainly, this is concerned with concurrency model which is different in each DBMS kernel. This is relevant as replication components can be installed in the database server itself and thus benefit from tight coupling with transaction execution kernel.

Configuration Storage Replication protocols usually need also to keep meta-information that is updated only outside user transactions. This is not an issue unless the choice is the target database server itself and is to be used during recovery. This requires an extra step in the server life-cycle that exposes such information while user databases are still off-line being recovered.

Appendix B

Code for the Query Cache Plugin

```
public class QueryCache implements StatementExecutionListener,
  DatabaseStartupListener,ObjectSetWriteListener {
  private static RequestProcessor reqProc =  null;

  public QueryCache(DatabaseProcessor dbProc, RequestProcessor
     reqProc,
  ReceiverStage stmtProc, ExecutorStage objProc) {
    this.reqProc = reqProc;
      dbProc.setDatabaseStartupListener(this, true);
      stmtProc.setStatementExecutionListener(this, true);
      objProc.setObjectSetWriteListener(this, true);
  }
  public static ResultSet cacheLookup(String reqId, String
     query)
   throws SQLException {
      Connection c =
        DriverManager.getConnection("jdbc:default:connection");
      Transaction tx = reqProc.getRequest(reqId).getTransaction
         ();
      Object cached = cache.get(query);
      if (cached == null) {
         java.sql.Statement s = c.createStatement();
         ResultSet rs = s.executeQuery(query);
         cache.put(tx, query, rs);
      }
      c.close();
      return (new ResultSet[]{cached};);
  }
  public void handleStatementExecution(Statement st)
```

```
    throws SQLException {
      switch (st.getState()) {
         case Statement.PIPELINE_PROCESSING:
            if(st.getStatement().startsWith("select")) {
             st.setStatement("CALL cacheLookup('"
                 + st.getRequest().getId() + "', '"
                 + st.getStatement() + "')");
            }
            st.continueExecution();
          break;
      }
    }
    public void handleDatabaseStartup(Database db)
       throws SQLException {
       switch (db.getContextState()) {
          case Database.DATABASE_STARTING:
             db.continueExecution();
           break;
          case Database.DATABASE_UP:
             DataSource ds = db.getDataSource();
             Connection con = ds.getConnection();
             java.sql.Statement st = con.createStatement();
             st.execute("CREATE PROCEDURE "
              + "cacheLookup(reqid VARCHAR(10),"
              + "query VARCHAR(100)) "
              + "EXTERNAL NAME 'gorda.demo.QueryCache.
                 cacheLookup'");
             st.close();
             con.close();
             db.continueExecution();
           break;
       }
    }
    public void handleObjectSetWrite(ObjectSet objSet)
     throws SQLException {
       switch (objSet.getState()) {
         case ObjectSet.PIPELINE_PROCESSING:
            cache.invalidate(objSet);
       break;
       }
       objSet.continueExecution();
   }
}
```

```
abstract class Cache {
  void put(Transaction tx, String query, ResultSet rs);
  String get(String statement);                                    71
  void invalidate(ObjectSet ws);
}
```