Supporting Multiple Isolation Levels in Replicated Environments

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Sistemas Informáticos y Computación

Tesis Doctoral

Presentada por:
José María Bernabé Gisbert

Dirigida por:
Dr. Francisco Daniel Muñoz Escoí

Junio de 2013

Contents

List of Figures

Abstract

Replication is used by databases to implement reliability and provide scalability. However, achieving transparent replication is not an easy task. A replicated database is *transparent* if it can seamlessly replace a standard stand-alone database without requiring any changes to the components of the system. Database replication transparency can be achieved if: (a) replication protocols remain hidden for all other components of the system; and (b) the functionality of a stand-alone database is provided.

The ability to simultaneously execute transactions under different isolation levels is a functionality offered by all stand-alone databases but not by their replicated counterparts. Allowing different isolation levels may improve overall system performance. For example, the TPC-C benchmark specification tolerates execution of some transactions at weaker isolation levels in order to increase throughput of committed transactions. In this thesis, we show how replication protocols can be extended to enable transactions to be executed under different isolation levels.

Resumen

La replicación de bases de datos aporta fiabilidad y escalabilidad aunque hacerlo de forma transparente no es una tarea sencilla. Una base de datos replicada es transparente si puede reemplazar a una base de datos centralizada tradicional sin que sea necesario adaptar el resto de componentes del sistema. La transparencia en bases de datos replicadas puede obtenerse siempre que (a) la gestión de la replicación quede totalmente oculta a dichos componentes y (b) se ofrezca la misma funcionalidad que en una base de datos tradicional.

Para mejorar el rendimiento general del sistema, los gestores de bases de datos centralizadas actuales permiten ejecutar de forma concurrente transacciones bajo distintos niveles de aislamiento. Por ejemplo, la especificación del benchmark TPC-C permite la ejecución de algunas transacciones con niveles de aislamiento débiles. No obstante, este soporte todavía no está disponible en los protocolos de replicación. En esta tesis mostramos cómo estos protocolos pueden ser extendidos para permitir la ejecución de transacciones con distintos niveles de aislamiento.

Resum

La replicació de bases de dades aporta fiabilitat i escalabilitat tot i que fer-ho de forma transparent no és senzill. Una base de dades replicada és transparent en la mesura en la que es puga substituir per una base de dades centralitzada tradicional sense requerir adaptar la resta de components del sistema. La transparència en bases de dades replicades es pot obtindre si (a) la gestió de la replicació queda totalment oculta a la resta de components i (b) s'ofereix la mateixa funcionalitat que en una base de dades tradicional.

Per a millorar el rendiment general del sistema, els gestors de bases de dades centralitzades existents permeten executar de forma concurrent transaccions baix distints nivells d'aïllament. Per exemple, l'especificació del benchmark TPC-C permet l'execució d'algunes transaccions amb nivells d'aïllament dèbils. No obstant això, aquest suport encara no està disponible en els protocols de replicació actuals. En aquesta tesi mostrem com aquests protocols poden ser ampliats per a permetre l'execució de transaccions amb distints nivells d'aïllament.

Aknowledgements

Usually, a complex and large project, such as a thesis is the result of the work of more than one person and this is not an exception. In my case, I have been lucky to receive very valuable help, advice and support from many people I love and admire. Their help and support improved the result in one way or another and even in some cases have been fundamental to get the work done.

Chronologically, this thesis would never have started without Pepe Corell's and Luis Irún-Briz's help and encouragement. Pepe was my second father during probably the worst period of my life until now (I hope). Thanks to him I started working in my current company, the Instituto Tecnológico de Informática (ITI), thirteen years ago where I have written this thesis. Luís, my fellow friend and former workmate here, encouraged me to start researching in general, and to aim for a PhD in particular. From the technical point of view, this work would never be as it is without the reviews, ideas, wise advices, tireless help an unbreakable faith of my thesis supervisor, Francesc Muñoz Escoí. Finally, some of the main ideas in this thesis have been also reviewed and/or discussed with Vaidė Narváez, my fellow friend and former workmate, who also made some valuable advises. All of them are remarkable people either from professional and personal point of views and I really feel in debt with them.

I would also like to thank professor Fernando Pedone for letting me stay for few months at the distributed systems group at the Università della Svizzera Italiana (USI). That stay was an invaluable experience for me and I learned a lot from Fernando, Vaidė and the rest of the fellow mates there. I would also like to thank my workmates and friends here at ITI and all the other researchers I met and had with whom I have had discussions at several conferences and via journals. Special greetings go to Idoia Ruiz, Enrique Armendáriz and the people from the JCSD for the countless discussions on isolation and database replication.

Last but not least, this would never been possible without the help of my family and friends, specially without my mother, father and sister and my dear friend Diana Pallás.

Preface

This thesis is the result of several years of work (from 2006 to 2013) under the supervision of Francesc D. Muñoz Escoí in the Distributed Systems department at the Instituto Tecnológico de Informática. During this time I have participated in the research projects MADIS[1], CONDEP[2] and IDEA[3]. Some intermediate results included in this thesis have been previously published in the following papers:

J. M. Bernabé-Gisbert, R. Salinas-Monteagudo, L. Irún-Briz, and F. D. Muñoz-Escoí. Managing multiple isolation levels in middleware database replication protocols. In 6th International Symposium on Parallel and Distributed Processing with Applications (ISPA), December 2006, volume 4330 of Lecture Notes in Computer Science, pages 511-523 Sorrento (Naples), Italy. Springer. CORE B. Position 538/581 in CiteSeer Venue Impact Factor 2008. Indexed in ISI Proceedings (7 cites according to ISI Web of Knowledge. 24 cites according to Google Scholar).

J. M. Bernabé-Gisbert, J. E. Armendáriz-Íñigo, R. de Juan-Marín, F. D. Muñoz-Escoí. Providing Read Committed Isolation Level in Non-Blocking ROWA Database Replication Protocols. XV Jornadas de Concurrencia y Sistemas Distribuidos (JCSD), June 2007, pages 159-171 Torremolinos (Málaga), Spain. (9 cites according to Google Scholar).

J. M. Bernabé-Gisbert. Providing support for data replication protocols with multiple isolation levels. In On the Move to Meaningful Internet Systems (OTM) Workshops, November 2007, volume 4805 of Lecture Notes in Computer Science, pages 265-274, Vilamoura, Algarve, Portugal. Springer. ISSN 0302-9743. Position 396/581 in CiteSeer Venue Impact Factor 2008. Indexed in ISI Proceedings (2 cites according to Google Scholar).

J. M. Bernabé-Gisbert, F. D. Muñoz-Escoí. Extending Mixed Serialisation Graphs to Replicated Environment. 3rd International Conference on Availability, Reliability and Security (ARES), pages 369-375. March 2008,

[1] Funded by the EU FEDER and the Spanish MCYT, under grant TIC2003-09420-C02.

[2] Funded by the EU FEDER and the Spanish MEC, under grant TIN2006-14738-C02.

[3] Funded by the EU FEDER and the Spanish MICINN, under grant TIN2010-17193

Barcelona, Spain. IEEE-CS Press. CORE B. Position 476/581 in CiteSeer Venue Impact Factor 2008. Indexed in ISI Proceedings (1 cite according to ISI Web of Knowledge. 3 cites according to Google Scholar).

J. M. Bernabé-Gisbert, F. D. Muñoz-Escoí. Supporting multiple isolation levels in database replication. Data & Knowledge Engineering (DKE), September 2012, volume 79-80, pages 1-16. Elsevier. ISSN 0169-023X. Impact factor 2011: 1.422.

J. M. Bernabé-Gisbert, F. D. Muñoz-Escoí. A Compoundable Specification of the Snapshot Isolation Level. XXI Jornadas de Concurrencia y Sistemas Distribuidos (JCSD), June 2013, Donostia-San Sebastián, Spain.

These results have never been published as a part of any other thesis before.

During this period I have also participated in other research works which are not part of this thesis but are related on one way or another with issues addressed in some of the chapters. The results of those works have also been published in conference proceedings and some of them are even referenced in this thesis:

R. Salinas-Monteagudo, J. M. Bernabé-Gisbert, F. D. Muñoz-Escoí, J. E. Armendáriz-Íñigo, J. R. González de Mendívil. SIRC: A Multiple Isolation Level Protocol for Middleware-based Data Replication. 22nd International Symposium on Computer Information Sciences (ISCIS), pages 1-6, November 2007, Ankara, Turkey, IEEE-CS Press. CORE C. Indexed in ISI Proceedings (11 cites according to Google Scholar).

J. M. Bernabé-Gisbert, V. Zuikeviciute, F. D. Muñoz-Escoí, F. Pedone. A Probabilistic Analysis of Snapshot Isolation with Partial Replication. 27th International Symposium on Reliable Distributed Systems (SRDS), pages 249-258 October 2008, Napoli, Italy IEEE-CS Press. CORE A. Position 259/581 in CiteSeer Venue Impact Factor 2008. Indexed in ISI Proceedings (3 cites according to ISI Web of Knowledge. 6 cites according to Google Scholar).

F. D. Muñoz-Escoí, J. M. Bernabé-Gisbert, R. de Juan-Marín, J. E. Armendáriz-Íñigo, J. R. González de Mendívil. Revising 1-Copy Equivalence in Replicated Databases with Snapshot Isolation. 11th International Symposium on Distributed Objects, Middleware and Applications (DOA), Lecture Notes in Computer Science, vol. 5870, pgs. 467-483, November 2009, Vilamoura, Algarve, Portugal, Springer. Position 472/581 in CiteSeer Vanue Impact Factor 2008. Indexed in ISI Proceedings (4 cites according to Google Scholar).

The SRDS paper was made in the framework of a three months stay at the Distributed Systems group at the Università della Svizzera Italiana, directed by Professor Fernando Pedone[4].

[4] Funded by the EU ESF/IMPIVA IMAETB/2007/30

Part I

Introduction

Chapter 1

Introduction

According to the Digital Universe Study published by EMC Corporation in 2011, in 2010 the amount of information digitally generated exceeded the zettabyte, increasing in a factor of 9 in only five years [30]. These data are accessed by people and applications to perform their daily tasks and must be always available to their users. Replication has been traditionally used to provide availability and improve performance in databases. In a replicated database, there are several copies of the data spread over a set of nodes. However, replication complicates data consistency and transactions isolation, two of the four ACID properties assumed for database management systems. The necessary data propagation and the coordination effort to ensure consistency and isolation limits the replication protocol scalability. In the last years, several non-relational alternatives, known as NoSQL systems, appeared showing an excellent performance and scalability at the cost of weakening isolation and consistency [55]. They have been widely used by many internet applications, usually characterized by performing a huge amount of reads when compared with writes. That is the case of search engines. However, those solutions cannot be used by applications with a high portion of writes, depending on relational databases or with strong consistency and isolation requirements for some of their operations.

In this thesis we address the replication of databases efficiently and transparently to applications built to access relational databases. Our proposal is based on adjusting isolation guarantees to transaction requirements. Applications indicate those requirements by using SQL or any of the standard interfaces like Java database connectivity (JDBC) or Open database connectivity (ODBC), as they do when accessing centralised and relational Database Management Systems (DBMS) like Oracle, PostgreSQL, MySQL, SqlServer, etc. That improves performance and scalability since provides strong isolation guarantees only when it is strictly necessary. We also provide the theoretical background to ensure the correctness of our proposals, that is, that all our replication mechanisms manage isolation like existing centralised DBMSs. That will make them adequate to replicate existing centralized databases transparently to applications, users and other system components.

Existing works on relational database replication are mostly oriented to mechanisms with strong isolation guarantees. Few works have proposed replication protocols supporting weak and strong isolation guarantees at the same time. None of them have developed the necessary theoretical background to prove the correctness of their proposals. We not only modify the main replication schemes to provide this support but also include this background to prove the correctness of our proposals. These correctness criteria do not depend on specific techniques and can be applied to existing and future replication protocols.

1.1 Database replication

A database is composed by a set of data items. In a replicated database there are multiple copies of every item. If one of the copies fails the others are still available to users and applications. Every working copy can serve different users concurrently, improving the system performance. To support more accesses replicated systems add more replicas. A transaction is a sequence of read and write operations executed atomically, i.e., all or none are executed. Atomicity is the first of the four ACID properties that must be guaranteed for every transaction being executed in a relational database. Consistency and Isolation are the following two and are explained later. Durability is the last one and indicates that the effects of successful transaction writes must be persisted and cannot be undone even if the system fails.

1.1.1 Consistency

In databases theory, the consistency property states that a transaction must always produce a correct state in the sense of not violating any integrity constraint. Otherwise, the entire transaction effects must be rolled back.

In replicated databases, when a given item copy is modified the update must be eventually propagated to its other replicas. In this context, consistency refers to the guarantees offered by this propagation and not to the ACID semantic consistency. The consistency model provided by a given system is a contract between the system and the applications and states the rules to be followed when updates are propagated and applied. In a strong consistency model the entire system must behave like a centralised database [17]. Thus, once a transaction finishes, all its updates are automatically available to any following transaction. In this thesis the word *consistency* always refers to replica consistency and not to the ACID semantic consistency.

Replica consistency is achieved by propagating item updates to all replicas and applying them in a given order. Transaction updates can be spread one by one or in a single bunch (we will see later that this bunch is known as the transaction write-set), usually propagated once the transaction is going to finish. The guarantees offered by this propagation determine the consistency model of a replicated database. For example, in a strong consistency model, a transaction cannot read an item copy if a finished transaction has modified the same item but its updates have not been applied in this copy yet. Weaker consistency models increase concurrency since they allow different degrees of stale data reads, i.e., they allow to read data even if it has been modified by finished transactions at other nodes.

1.1.2 Isolation

In a database, transactions should ideally be executed in isolation without interfering with each other. However, in practice DBMSs allow some interference in order to increase concurrency and so, improve system response time and throughput. Unfortunately, concurrent execution of transactions may generate anomalies (or phenomena) that must be resolved at the application tier. For example, transaction T_1 reads x data item while T_2, another concurrent transaction, is updating the same data item. Such a situation is defined as *non-repeatable* or *fuzzy read* phenomenon in [6]. If transaction T_1 reads data item x again, it may read a different value. Isolation levels are characterized by the anomalies that are forbidden in the execution of transactions [6].

The strongest isolation level is a serial execution of transactions (i.e., transactions cannot be executed concurrently). Due to performance issues, serialisability is usually the strongest isolation level provided by a commercial DBMS. Serialisability allows concurrency as long as the final result can be considered equivalent to a serial execution. Not all transactions require such strong guarantees. Therefore, commercial DBMSs also support weaker isolation levels. Many DBMSs (e.g., PostgreSQL [48], Oracle [45], and Microsoft SQL Server [42]) use by default the *read committed* [6] (RC) isolation level. This level is much weaker than the serialisable level since it allows non-repeatable reads and phantom reads [6] (a transaction may obtain different results if it executes the same read operation twice). These DBMSs delegate strong isolation levels for sensitive transactions with strong isolation restrictions. For example, a transaction T_1 that withdraws cash from a bank account requires stronger isolation than a transaction T_2 that only retrieves a list of account balances. Notice that transaction T_1 probably first reads the balance to check if there is enough money to withdraw. Meanwhile if another transaction T_3 modifies the same account balance, the application may take incorrect decisions since the account balance changed when T_1 updated the account. However, this anomaly would never be produced by T_2. Therefore, transaction T_2 may be executed under a weaker isolation level.

Although multiple commercial DBMSs agree on using RC by default they do not agree on how the *American National Standards Institute* (ANSI) *serialisable* [27] isolation level should be implemented. Several DBMSs provide a slightly weaker isolation level known as *snapshot isolation* (SI) [6] that is faster but may produce non-serialisable executions. Some commercial DBMSs, such as Microsoft SQL Server [42], allow applications to decide if they want to use serialisable or SI. It is uncommon for applications to mix serialisable and SI transactions; yet this may happen when multiple applications accessing the same database select different isolation levels for transactions with strong isolation requirements. As a result, these DBMSs may face situations in which serialisable, SI and RC transactions need to be executed concurrently.

In a replicated database, some sort of coordination is necessary when concurrent transactions operate over different copies of the same items. A replication protocol correctly manages isolation if the result of executing a set of transactions is equivalent to a serialisable execution of the same set in a centralised DBMS. That is known as one-copy serialisability or 1SR [14]. Surprisingly, only few works studied what happens when weak isolation is used for some transactions and none of them has proposed the necessary theoretical background to prove the correctness in that case. The implementation of this feature in existing replication protocols not only will improve their performance and scalability but also their ability to integrate with applications developed to access relational databases. They can also be used to improve the performance of replication protocols based on group communication systems.

1.1.3 Scalability of replication protocols

Consistency and isolation require data propagation and replica coordination. This process limits the system scalability because it usually involves all replicas in the system and cannot be mitigated just adding more replicas. Notice that, regardless of the number of nodes, all of them must apply the updates and check the isolation guarantees for each transaction. In the 90s, group communication systems were introduced. They provide a set of primitives to propagate messages to groups of nodes and with a given guarantee. For example, a reliable atomic broadcast [18] ensures that all live replicas deliver a given message and that any two messages are delivered in the same order in any two replicas. Replication protocols using group communication protocols improve performance and scalability but, in practice, they are still limited to small clusters.

In the last years, several non-relational alternatives have appeared showing great performance and scalability, specially when most transactions perform only reads, by weakening ACID Isolation and Consistency properties. However, some applications still require strong guarantees for some of their operations. Some others have been developed to access relational databases using standard interfaces and do not integrate well with non-relational and non-standard systems. The main goal of this thesis is to provide correct mechanisms to improve performance and flexibility of replication protocols. Our mechanisms can be used to replicate existing relational databases transparently to applications and users.

1.2 Replication transparency

Replication transparency states that *although the same data item may be replicated at several nodes of the network, the programmer may treat the item as if it were stored as a single item at a single node* [56]. Thus, the replication of a database is *transparent* if it can be implemented and deployed in existing systems without requiring changes to system components or external applications.

Existing replication protocols hide replication management by implementing a standard database access interface such as JDBC. Unfortunately, these solutions do not really implement all the functionality available in stand-alone databases. A typical *database management system* (DBMS) simultaneously executes transactions under different isolation levels, but most replication protocols support a single level, which is usually either serialisable [14] or snapshot isolation [6]. The authors of [16] identified this problem as one of the challenges in database replication. Both TPC-C and TPC-W benchmarks also demonstrate that typical applications need to execute transactions at different isolation levels, mainly for performance reasons.

1.3 Benefits of supporting multiple isolation levels

Weak isolation levels may provide performance and economical benefits.

1.3.1 Performance benefits

Weak isolation levels reduce the complexity of concurrency control management and so may improve the performance of DBMSs. Note that these levels can allow higher degrees of concurrency than the strictest isolation levels [12, 33, 38, 4, 31]. Performance improvements are especially appealing for ROWAA (*read-one, write-all-available*) [14] replication protocols since in these protocols isolation management usually involves all database replicas. In several ROWAA replication solutions, transactions are executed optimistically at a single replica (local replica); and updates are propagated to all replicas before the transaction finishes. Once updates are delivered, a validation step is executed to guarantee that no forbidden phenomena occur due to conflicts with transactions executed in other replicas. The stronger the isolation level, the larger the set of phenomena that must be checked. Therefore, the complexity of the validation step depends on the isolation level. Furthermore, the stronger the isolation level of a transaction, the greater is the probability that the transaction will abort. For example, as we explain later in Chapter 8, serialisable transactions require either the inclusion of the accessed values when updates are broadcast; or the propagation of the local replica validation result. However, this step is unnecessary when weak isolation levels are used. Under high workloads replicas spend resources validating transactions that cannot be mitigated by adding more replicas, since all replicas have to perform the same validation step. If many transactions can be executed under weaker isolation levels such as RC, then network and CPU loads, as well as transaction abort rates, can be reduced significantly. For instance, the performance results presented in [53] show that with the *SIRC replication protocol* and for a given type of update transactions, only 60% of the completion time needed with SI was required when RC isolation was used. Additionally, the abortion rate with RC isolation was 26 times lower than with SI.

In other ROWAA protocols (for example, primary copy replication protocols) all transactions are validated by the same replica (usually known as the primary replica or the central validator). In these cases, the validation gains obtained by weak isolation levels are negligible since no data propagation is demanded to complete the validation step. Note, however, that in a system based on a central validator the performance may degrade under medium and high workloads because the central validator does most of the work and so easily becomes a bottleneck [60]. It also might become a single point of failure. As a result, there is no optimal deployment for the validation tasks, and relaxed isolation levels can always increase concurrency.

1.3.2 Economical benefits

Recently, public clouds have emerged as software as a service (SaaS) hosting platforms based on a pay-as-you-go business model. In those systems, resources are assigned dynamically depending on the application load and the more resources consumed the more have to be paid to the cloud provider. Adjusted consumption may make the difference between a profitable and a non profitable system. In data intensive systems, a wise use of isolation and consistency may reduce memory, CPU and network consumption and, at the end, save money [36].

1.4 Objectives

The main goal of this thesis is to support multiple isolation levels in database replication protocols as a way to improve performance and transparency at the same time. This allows to correctly fit the isolation guarantees to every transaction isolation needs in existing replication protocols as centralised concurrency control mechanisms do. We not only provide the necessary theoretical background but also modify the main replication schemes to include this feature.

We also modify some well known replication protocols with this support.

1.5 Contributions

The main contributions of this thesis are the following:

- **Snapshot Isolation level alternative definition**: this work extends Mixed Serialisation Graphs or MSG, Adya's way to represent dependencies among transactions in executions involving several isolation levels at the same time. The main weaknesses of MSG are the lack of support of Snapshot Isolation level and its stand-alone systems orientation. Snapshot Isolation is not supported because its definition is not based on the same abstraction than the rest of the main isolation levels. Our Snapshot Isolation definition separates the isolation requirements (to execute a transaction over a committed state of the database) from some consistency issues (the freshness of the snapshot observed by the transaction) and it can be easily integrated in Adya's theory.
- **The theoretical background to support multiple isolation levels correctly and transparently**. It can be used to prove the correctness of replication protocols when they support several isolation levels. Notice that *correctness* in this context means to behave like existing commercial DBMS.
- **The upgrade of the principal database replication schemes to support multiple isolation levels**. A lot of replication protocols have been proposed in literature. Several works have categorized them in a few replication schemes which represent the main techniques to replicate relational databases. We modify such mechanisms and prove their correctness. In most cases, only a few changes are necessary to correctly support the main isolation levels. The result can be used as a guide to provide this feature in existing replication protocols. This will allow to adjust isolation guarantees to transaction isolation requirements, improving performance and scalability without affecting existing applications. Those mechanisms have been widely used in commercial centralised systems but have not been successfully exported yet to database replication.
- **Included support of multiple isolation levels in some well known replication protocols** by using the modified schemes.

1.6 Roadmap

Chapter 2 summarizes previous contributions related to this thesis. In Chapter 3 we provide a basis for this work by introducing the assumed system model.

Chapter 4 discusses the existing definitions and why they cannot be used in our case. Chapter 5 introduces our alternative definition of snapshot isolation. Chapter 6 updates Adya's Mixed Serialisation Graphs to contemplate transactions requesting Snapshot Isolation. Chapter 7 identifies the conditions a replication protocol must satisfy to behave like a stand-alone DBMS. Chapter 8 shows a general scheme to upgrade existing replication protocols to simultaneously support multiple isolation levels. Chapter 8.2 shows how the suggested scheme can be applied to some existing ROWAA update-everywhere replication protocols; and example protocols are presented in Chapter 9. Chapter 10 concludes the work. Some lemmas, theorems and proofs are detailed at the Appendices.

Chapter 2

Related work

The results presented here summarize and extend our previous works in [11], [53], [7], [10] and [9]. The problem is identified in [11], later pointed in [16] as one of the existing challenges in database replication. In [11] we also explain how existing replication protocols may be extended to include multiple isolation levels support. It also informally suggests how concrete PL-1, PL-2, PL-2.99 and PL-3 levels [2] can be supported and presents an example protocol. Another protocol is presented in [53], which also includes some empirical results over our distributed middleware MADIS [28, 43]. This paper concludes that supporting weaker isolation levels may reduce the abort rate without impacting the response time. Early results were also presented in [7], which defines the main goals and introduces correctness, equivalence and replication schemes modification as the main challenges of this research. Our work [8] introduces for the first time the idea of extending Mixed Serialisation Graphs [2] as a way to prove replication protocol correctness, even in the presence of several isolation levels. This idea is later extended and formalised in [9] and applied to certification-based replication scheme [60] and an example protocol. This formalisation is completely redefined in this thesis by using the new Snapshot Isolation level definition we present in [9], also revisited in this thesis, which extends the time-precedes order definition used by Adya in [1]. The final conclusions are implemented to all the eager replication schemes identified in [60] and to some existing replication protocols.

One of the main goals of this work is to specify the correctness criteria needed to decide whether a replication protocol is correct when it supports multiple isolation levels. Traditionally, serialisability theory has been used as a correctness criterion for centralised systems [14]. The execution of a set of transactions is considered correct if the result is equivalent to one possible serial execution of the same set of transactions. If the same set is executed in a replicated system, it is considered correct if it is equivalent to one possible execution of the same set of transactions in a serialisable centralised DBMS. This criterion was introduced by *Bernstein et al.* [14] as one-copy serialisability or 1SR. However, serialisability can be expensive and existing DBMSs improve performance by allowing transactions to be executed with weaker isolation guarantees. Most DBMSs support the set of isolation levels defined by ANSI [27]. Unfortunately, ANSI definitions have proven to be ambiguous and incomplete [6] and some systems that apparently provided serialisability were actually providing a slightly weaker isolation level defined as *snapshot isolation* (SI) by *Berenson et al.* [6]. SI is supported today by many commercial DBMSs (Oracle, PostgreSQL, MS SQL Server, etc.) and is widely used by applications. Some works extended Bernstein equivalence theory to one-copy-SI [38, 40] to define under which circumstances a replicated system behaves as a centralised DBMS providing SI.

However, sometimes even SI is too expensive and regular applications use weaker isolation levels for some transactions [57]. This may be reasonable in some cases. Note that some isolation guarantees may be ensured at the application tier. In other cases, the appearance of some phenomena (such as the reading of stale data) in the execution of some transactions is considered a minor problem and can be accepted if there are some performance improvements (e.g., a higher degree of concurrency that enables a faster transaction completion time). Unfortunately, as far as we know, existing one-copy equivalence definitions are not general enough and cannot be used when multiple isolation levels are supported concurrently in a replicated system. This thesis tries to fill this void.

A.J. Bernstein et al. [12] also highlighted the importance of using weak isolation levels to improve the overall system performance. They focus on when the execution of a transaction or a set of transactions is semantically correct. The execution of a set of transactions is semantically correct if none of the database and applications semantic restrictions are violated and the final result reflects the cumulative expected results of all transactions in some serial order. Since the semantic restrictions and the expected results may not be as restrictive as serialisation, semantic correctness is weaker than serialisable. A.J. Bernstein et al. also describe whether a transaction can be executed at a given isolation level without violating any semantic restrictions but they do not explain when the selected isolation level has been successfully ensured and that is the main goal of our work.

Some of our previous works [11, 53] present specific protocols that support several isolation levels simultaneously ([53] includes some empirical results). However, these papers do not present a general solution, nor do they formally demonstrate the correctness of their proposals.

Previous works also pointed out the importance of using weak isolation levels to improve performance and concurrency in replicated databases but they only suggest one separated protocol per isolation level instead of a single one supporting all levels [33]. In [49, 52], the authors present a meta-protocol that can execute several replication protocols at the same time. Before the execution of a transaction, one of the supported replication protocols is selected based on the transaction requirements. This approach is more modular and general than [11, 53] since isolation is only one possible criterion to match a protocol to a given transaction. However, sometimes modifying an existing and implemented protocol is a more straightforward solution than deploying a meta-protocol and developing at least one protocol per supported isolation level. Furthermore, the work in [49, 52] does not prove the correctness of the protocols, since it only discusses the architecture and performance of the meta-protocol. The specifications presented here can be applied to prove the correctness of a given combination of protocols managed by the meta-protocol.

In [31, 4] the authors present two replication protocols supporting *serialisable*, *snapshot isolation* (SI) and *generalised SI* (GSI) [19], a variation of SI more suitable to distributed databases (in GSI, each transaction may start immediately in its local replica by reading from its currently available database snapshot, whilst a strict interpretation of SI semantics requires the latest system-wide database snapshot to be obtained and this might demand an additional synchronisation step at each transaction start). Our work can be used to prove the correctness of these protocols and all their variations.

The correctness theory presented here to prove the correctness of replication protocols mixes *serialisation graphs* and one-copy-equivalence concepts introduced by *Bernstein et al.* [14] with the *mixing theorem* introduced by *Adya* [1] for centralised systems.

Serialisation graphs represent dependencies among transactions during an execution in a DBMS. Since Bernstein's work focuses on the serialisable isolation level, we use Adya's *mixed serialisation graphs* (MSG) and *mixing-correct* definition [1] to support other isolation levels as well. With Adya's MSG a conflict is represented as an edge if it matters to the involved transactions isolation levels. An execution preserves all transactions isolation needs if the associated MSG does not show any cycle. Unfortunately, Adya's mixing theory does not include the SI level and, most importantly, his specifications are not directly applicable to replicated systems, as illustrated by *Lin et al.*'s [40] extensions. In Chapter 6 we extend MSG and *mixing-correct* definitions to examine the SI level. In Chapter 7.1 we make a further extension to support replicated systems.

One-copy-equivalence is needed to decide, from the user point of view, whether the execution of a set of transactions in a replicated system can be considered equivalent to an execution of the same set in a correct stand-alone system. We use our *serialisation graph* extensions to make that comparison. Some other works use similar concepts and methodologies but none support multiple isolation levels. For example, *Lin et al.*'s 1-copy-SI [40] focuses on SI replication protocols. Our paper extends this approach since our correctness criteria can be applied to any replication protocol supporting one, all, or a subset of the main isolation levels.

The protocol SER CBR used in Chapter 9 is a *serialisable* version of existing SI CBR [50] that supports *snapshot isolation.* Both can be considered variations or interpretations of well known *serialisable* and *snapshot isolation* protocols [32, 19, 38, 17]. MUL CBR combines SER CBR and SI CBR to support the four main isolation levels considered in this work (see Chapter 4).

Finally, this work uses the Adya isolation level definitions [1] with minor changes that are explained later in Chapter 4.

Part II

Theoretical conditions

Chapter 3

Background

This chapter introduces the main models, assumptions and concepts further used in the following chapters. First we introduce the general model and the communication model assumed. Next, we focus on database and transactions models and definitions, including histories as a way to represent transactions execution in a system. We also discuss about the concept of concurrency and logical time in a distributed system. Finally, we also introduce some basic graphs theory later referenced in this thesis.

3.1 General model

We assume an asynchronous system composed of a set of nodes $\mathcal{N}$. A system is asynchronous if bounds are not assumed in communication delays and in nodes local clock drifts [26]. Each node in $\mathcal{N}$ has a complete copy of the database managed by a typical stand-alone DBMS locally supporting several isolation levels. Thus, this work focuses on full replication but the main results can be easily applied to partially replicated and distributed databases. Replication is implemented by a middleware [13] deployed on top of the DBMS.

Transactions are issued by clients. A transaction can be initially submitted to any node in the system and which then becomes its *local node*.

Nodes may fail by crashing. However, note that node failures and their recoveries are not the focus of this paper. Database replication protocols should deal with failures, and many papers have provided solutions to this problem, including [29, 34]. The recovery subprotocols that are needed to manage this problem do not have any effect on our specifications nor on the architectural protocol details needed for adequately managing multiple isolation levels. Therefore, no further discussion on failures is given.

3.2 Communication model

Communication takes place by exchanging messages. The middleware has access to a group communication system with an atomic broadcast primitive [26]. An atomic broadcast is a reliable broadcast with a total order property. Every message m propagated includes its sender *sender(m)* and a sequence number *seq(m)*. A reliable broadcast is characterized by two primitives, *broadcast(m)* and *deliver(m)*, and the following properties:

- **Validity**: if a correct node broadcasts a message m, then all correct nodes eventually deliver m.
- **Agreement**: if a correct node delivers a message m, then all correct processes eventually deliver m.
- **Integrity**: for any message m, every correct node delivers m at most once, and only if m was previously broadcast by *sender(m)*.

The **Total order** property is defined as follows: if correct processes N_a and N_b both deliver messages m and *m'*, then N_a delivers m before *m'* if and only if N_b delivers m before *m'*.

3.3 Databases and transactions

A database is a set of items that can be read and written. Updates, inserts and deletes are all treated as writes. Clients (usually applications) read and write database items through transactions. A transaction is a sequence of read and write operations plus an initial start operation and a final commit or abort operation. Operations in a transaction are executed atomically. If the transaction is committed all its writes are persisted in the database. If it is aborted all writes are rolled back.

Operation $w_i(x_i)$ represents transaction T_i's write on item x, being x_i the value written. Operation $r_i(x_j)$ represents T_i's read of the value x_j written by transaction T_j. We represent the set of items read and written by T_i as RS_i and WS_i, respectively. We call a transaction *read-only* if it does not contain any write operations ($WS_i = \emptyset$), and *update* otherwise. A transaction is initiated with a start operation s_i and terminates with a commit operation c_i or an abort operation a_i. Note that c_i and a_i are mutually exclusive, i.e., transactions either commit or abort. If a transaction performs several writes on item x, $w_i(x_{i.n})$ represents the n-th write on item x performed by T_i. If no suffix is present, x_i represents the last value established by T_i. Operation $r_i(x_{j.n})$ indicates a T_i's read of the n-th T_j's write on x and $r_i(x_j)$ T_i's read of T_j's last write on x. Finally, x_0 is the initial value of an item x and o_i represents any operation performed by T_i. Hereafter, we present a formal definition of a transaction:

Definition 1 (Transaction)**.** *Once completely executed, a transaction T_i is a totally ordered set of operations with a binary relation $<$ where:*

- $T_i \subseteq \{r_i(x_j), w_i(x_i) | x \textit{ is a data item}\} \cup \{s_i, a_i, c_i\}$.
- $s_i \in T_i$.
- $c_i \in T_i$ *iff* $a_i \notin T_i$[1]
- *For any T_i's operation o_i, if $o_i \neq s_i$ then $s_i < o_i$.*
- *If $c_i \in T_i$ then, for any operation $o_i \neq c_i$, $o_i < c_i$.*
- *If $a_i \in T_i$ then, for any operation $o_i \neq a_i$, $o_i < a_i$.*
- *For any two T_i's operations o_1 and o_2, $o_1 < o_2$ or $o_2 < o_1$.*

If it is necessary to explicitly refer to a replicated system, a copy of data item x at node N_a is represented by x^a; while T_i^a denotes the subset of transaction T_i operations executed at N_a. The notation of read, write, commit, and abort operations is also extended in the same way. For example, $r_i^a(x_j)$ represents transaction T_i's read operation executed at node N_a over the last update on x performed by transaction T_j on node N_a.

3.4 Execution of transactions

When a set of transactions is executed in the system, the operation execution order is determined by a system scheduler. A history represents how transactions have been ordered during the execution. Given two operations o_1 and o_2, $o_1 <_H o_2$ in a history H if they have been executed in that order and either belong to the same transaction or are conflictive. Operations o_1 and o_2 conflict if they operate over the same item and at least one of them is a write. Thus, two read operations of distinct transactions never conflict and are not directly ordered in H. Formally:

Definition 2 (History)**.** *Given a set of transactions $\mathcal{T} = \{T_1, ..., T_n\}$, a history H is a partially ordered set of the operations in $\mathcal{T}$'s transactions with a binary relation $<_H$ where:*

- *For any transaction $T_i \in \mathcal{T}$ and any operation $o_i \in T_i$, $o_i \in H$.*
- *For any transaction $T_i \in \mathcal{T}$ and any two operations $o_1, o_2 \in T_i$, if $o_1 < o_2 \in T_i$ then $o_1 <_H o_2 \in H$.*

[1] Commit or abort operations would not appear when transactions still in execution are represented but, in this work, only completed transactions are considered.

- *If $r_i(x_j) \in H$ then $w_j(x_j) \in H$, $w_j(x_j) <_H r_i(x_j) \in H$ and $\nexists w_k(x_k)$ such that $w_j(x_j) <_H w_k(x_k) <_H r_i(x_j)$.*
- *For any two conflicting operations $o_1, o_2 \in H$, $o_1 <_H o_2 \in H$ or $o_2 <_H o_1 \in H$.*

For the sake of simplicity and readability we use $<$ instead of $<_H$ except in cases of ambiguity. Hence, $o_i < o_j \in H$ is equivalent to $o_i <_H o_j \in H$.

The committed projection of a history H is the portion of H including only the operations of committed transactions [14]. More formally:

Definition 3 (Committed Projection)**.** *Given a history H over the operations of a transaction set $\mathcal{T}$ and $\mathcal{T}_c \subseteq \mathcal{T}$ the subset of committed transactions, the committed projection of H or $C(H)$ is a history such that:*

- *For any transaction T_i and any operation $o_i \in T_i$, $o_i \in C(H)$ iff $o_i \in H$ and $T_i \in \mathcal{T}_c$*
- *For any transactions $T_i, T_j \in \mathcal{T}$ and any two operations $o_i \in T_i$ and $o_j \in T_j$, $o_i <_H o_j \in C(H)$ iff $o_i, o_j \in C(H)$ and $o_i <_H o_j \in H$.*

In replicated systems, a transaction execution history over the replicated nodes is composed of all the local executions. Given a set of transactions $\mathcal{T}$, $\mathcal{T}^a$ is the subset of $\mathcal{T}$ executed in node N_a. Thus, if $T_i \in \mathcal{T}$, $T_i^a \in \mathcal{T}^a$. Hence, the *replicated history* is defined as follows:

Definition 4 (Replicated history)**.** *Given a set of transactions $\mathcal{T} = \{T_1, ..., T_n\}$ and a set of nodes $\mathcal{N} = \{N_1, ..., N_n\}$, a replicated history H_r is the partially ordered set composed by all the operations of $\mathcal{T}$'s transactions executed in every one of the nodes in $\mathcal{N}$ with a binary relation $<_r$ where:*

- *For every $T_i \in \mathcal{T}$ and every operation $o_i \in T_i$, exists $N_a \in \mathcal{N}$ such that $T_i^a \in \mathcal{T}^a$ and $o_i^a \in T_i^a$ (o_i is executed at least in one node).*
- *For every two conflicting operations o_i, o_j, exists $N_a \in \mathcal{N}$ such that $T_i^a, T_j^a \in \mathcal{T}^a$, $o_i^a \in T_i^a$ and $o_j^a \in T_j^a$ (at least one node detects a conflict).*
- *For every $T_i^a \in \mathcal{T}^a$ and every operation $o_k^a \in T_i^a$: $o_k^a \in H_r$.*
- *For every $T_i^a \in \mathcal{T}$ and every two operations $o_1^a, o_2^a \in T_i^a$: if $o_1^a < o_2^a \in T_i^a$, then $o_1^a <_r o_2^a \in H_r$.*
- *If $r_i^a(x_j) \in H_r$ then $w_j^a(x_j) \in H_r$, $w_j^a(x_j) <_r r_i^a(x_j) \in H_r$ and $\nexists w_k^a(x_k)$ such that $w_j^a(x_j) <_r w_k^a(x_k) <_r r_i^a(x_j)$.*
- *For any two conflicting operations $o_i^a(x), o_j^a(x) \in H_r$ executed in node N_a: $o_i^a(x) <_r o_j^a(x) \in H_r \vee o_j^a(x) <_r o_i^a(x) \in H_r$.*

For the sake of readability we use $o_i^a < o_j^a \in H_r$ instead of $o_i^a <_r o_j^a \in H_r$. We use $o_i \in H_r$ to indicate that at least one node has executed operation o_i. $o_i < o_j \in H_r$ states that the operations are executed in that order in at least one node and there is no other node where operations execute in different order (i.e., $\exists N_a \in \mathcal{N}$ such that $o_i^a < o_j^a \in H_r$ and $\nexists N_b \in \mathcal{N}$ such that $o_j^b < o_i^b \in H_r$). $T_i < T_j \in H_r$ indicates that $o_i < o_j \in H_r$ for any two conflicting operations $o_i \in T_i$ and $o_j \in T_j$. Finally, H_r^a is the subset of H_r including only the operations executed at node N_a and the orderings among those operations.

As with normal histories, the committed projection of a replicated history H_r or $C(H_r)$ is the part of H_r including only the operations of committed transactions.

Although the core of this work does not depend on specific technologies, most of the replicated schemes modified in Chapter 8.2 and the examples shown later in Chapter 9 use ROWAA *update everywhere* replication solutions. In those protocols, the read operations of a transaction only execute at the local node and write operations must be applied in all the database replicas. More formally:

Definition 5 (ROWA transaction)**.** *Given a transaction T_i and a node N_a, T_i^a is a ROWA transaction if:*

- *T_i^a is a subset of T_i .*
- *If $T_i^a \neq \emptyset$ and $c_i \in T_i$, then $c_i^a \in T_i^a$.*
- *If $T_i^a \neq \emptyset$ and $a_i \in T_i$, then $a_i^a \in T_i^a$.*
- *If $r_i(x) \in T_i$, then $r_i^a(x) \in T_i^a$ iff T_i is local to N_a.*
- *If $o_1 < o_2 \in T_i$, and o_1^a, $o_2^a \in T_i^a$, then $o_1^a < o_2^a \in T_i^a$.*
- *If $c_i \in T_i$, then $\forall w_i(x) \in T_i$: $w_i^a(x) \in T_i^a$.*

3.5 Logical time

The scheduler assigns transaction start and end points. They represent when transactions start and finish and determine which committed database state is observed by every transaction when it is started. A committed state includes the last committed values in the database at some moment in time. A *time-precedes order* [1] is defined in the following way:

Definition 6 (Time-precedes order)**.** *Given a history H and E the subset of H including only the start and commit operations of committed transactions, a time-precedes order $<_t$ is a partial order on E such that:*

a) For any transaction T_i committed in H, $s_i <_t c_i$.

b) Given T_i, T_j transactions committed in H, $c_i <_t s_j$ or $s_j <_t c_i$.

Two transactions T_i and T_j are concurrent in H if $s_i <_t c_j$ and $s_j <_t c_i$.

Definition 7 (Conflict-aware time-precedes order). *Given a history H and E the subset of H including only the start and commit operations of committed transactions, a conflict-aware time-precedes order $<_c$ is a partial order on E such that:*

a) $<_c$ is a time-precedes order of E.

b) if $w_i(x_i) <_H r_j(x_i) \in H$ or $w_i(x_i) <_H w_j(x_j) \in H$ then $c_i <_c s_j$.

c) if $r_i(x_k) <_H w_j(x_j) \in H$ then $s_i <_c c_j$.

In a replicated system, multiple nodes may be involved in a transaction T_i execution. To decide which transactions are concurrent, nodes must agree on how start and commit points are ordered. In these environments, the time-precedes order can be generalised in the following way:

Definition 8 (Replicated time-precedes order). *Given a replicated history H_r representing the execution of a set of transactions $\mathcal{T}$ in a replicated system with $\mathcal{N}$ nodes. Given E the subset of H_r including only the start and commit operations of committed transactions, a replicated time-precedes order $<_{tr}$ is a partial order on E such that:*

a) For any node $N_a \in \mathcal{N}$, $<_{tr}$ is a time-precedes order of E^a, where E^a is the subset of E containing only start and commit operations executed in node N_a.

b) For any transaction $T_i \in \mathcal{T}$ committed in H_r and any node $N_a \in \mathcal{N}$ where T_i operations are executed, $s_i^a <_{tr} c_i^a$ in H_r^a. We represent that as $s_i <_{tr} c_i$.

c) For any two transactions $T_i, T_j \in \mathcal{T}$ committed in H_r, if $\mathcal{N}_{ij} \subset \mathcal{N}$ is the subset of nodes where both T_i and T_j are executed, $\forall N_a \in \mathcal{N}_{ij}$ $c_i^a <_{tr} s_j^a$ or $\forall N_a \in \mathcal{N}_{ij}$ $s_j^a <_{tr} c_i^a$. We represent that as $c_i <_{tr} s_j$ or $s_j <_{tr} c_i$.

Two transactions T_i and T_j are concurrent in H_r if $s_i <_{tr} c_j$ and $s_j <_{tr} c_i$.

Definition 9 (Replicated conflict-aware time-precedes order). *Given a replicated history H_r and E the set of start and commit operations of transactions committed in H_r, a replicated conflict-aware time-precedes order $<_{cr}$ is a partial order on E such that:*

a) $<_{cr}$ is a replicated time-precedes order.

b) if $w_i(x_i) <_r r_j(x_i) \in H_r$ or $w_i(x_i) <_r w_j(x_j) \in H_r$ then $c_i <_{cr} s_j$.

c) if $r_i(x_k) <_r w_j(x_j) \in H_r$ then $s_i <_{cr} c_j$.

Given a protocol P, two time-precedes orders are equivalent if they order start and commits points in the same way for any possible execution of P.

In centralised systems, the time-preceding order usually relies on the system local clock to timestamp transactions start point and commit. Thus, $c_i <_t s_j$ if T_i committed before T_j started in the system. However, in a distributed system nodes local clocks may drift in an unbounded amount of time. If two transactions are executed at different nodes local clocks cannot be used to assign timestamps. In those situations, another kind of logical time should be defined to designate transactions starts and commits orderings.

3.6 Graphs Theory

From now on, we introduce some graph theory definitions later used in this work, specially in chapters 4 to 7.

A graph $G = (V, E)$ is a pair of sets such that V is the set of vertices and $E \subseteq \{\{u, v\} | u, v \in V, u \neq v\}$ the set of edges. A pair $\{u, v\}$ is usually written as uv.

G is a *directed graph* if the pairs in E are ordered such that $uv \neq vu$. Every pair is composed by a *source* vertex and a *target* and is represent as $u \longrightarrow v$.

G is a *labeled graph* if a label is assigned to every vertex, edge or both.

A graph $G' = (V', E')$ is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$.

A *walk* P in a graph $G = (V, E)$ is a sequence $v_1 e_1 v_2 e_2 ... e_{n-1} v_n$ of vertices and edges such that $V_P = \{v_1, ..., v_n\} \subset V$, $E_p = \{e_1, ..., e_{n-1}\} \subset E$ and every edge $e_i = v_i v_{i+1}$. v_1 and v_n are the P's end vertices. The rest are *inner* vertices. Similarly, e_1 and e_{n-1} are P's end edges and the rest P's inner edges. Notice that $P = (V_p, E_p)$ is a subgraph of G.

P is a *directed walk* if every edge $e_i = v_i \longrightarrow v_{i+1}$. In that case, v_1 and v_n are the *source* and *target* vertices of P.

A walk $P = (V_p, E_p)$ is a *path* if $u \neq v \ \forall u, v \in V_p$. P is a *directed path* if it is a path and a directed walk.

A directed walk $P = (V_p = \{v_1, ..., v_k\}, E_p)$ of a graph $G = (V, E)$ is a *cycle* if $u \neq v \ \forall u, v \in V_p$ except that $v_1 = v_k$. A cycle P is *chord-free* if $\forall u, v \in V_p$, if $u \longrightarrow v \notin E_p$ then $u \longrightarrow v \notin E$. The cycle of minimun length in G is necessarily chord-free [21].

Given a walk $P = (V_p, E_p)$, $l = |V_p|$ is the number of elements in V_p or the length of P.

A walk $P' = (V_{p'}, E_{p'})$ is a *subwalk* of a walk $P = (V_p, E_p)$ if P' is a subgraph of P. The walk P' is a *directed subwalk*, a *subpath* or a *directed subpath* of P if P is a directed walk, path or directed path.

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are equal if $E_1 = E_2$ and $V_1 = V_2$. If the graph is labeled then the labels must be equal too.

The graph $G1 \cup G2 = (V_1 \cup V_2, E_1 \cup E_2)$. $G1 \cap G2 = (V_1 \cap V_2, E_1 \cap E_2)$. Recall that two labeled edges e_1 and e_2 are equal if they involve the same vertices and show the same label. If e_1 and e_2 are directed, their source and target vertices must be the same too.

Two edges $e_1, e_2 \in E$ are adjacent in graph $G = (V, E)$ if a vertex $v \in V$ is in e_1 and e_2. If e_1 and e_2 are directed, v must be e_1's target and e_2's source. We also say that v is *shared* by e_1 and $e2$.

Chapter 4

Stand-alone systems

Ensuring a strict isolation is costly and, depending on the concrete mechanism being used, this implies many blocked transactions and/or many aborts. To improve performance, commercial DBMSs allow transactions to be executed with weaker isolation levels at the cost of allowing certain types of interferences or phenomena, which must be either managed by the application tier or accepted by the user. An example is the phenomenon known as Write Skew [6]. Assume a database with two items x and y and an integrity constraint requiring that $x + y > 0$. If two transactions T_i and T_j concurrently read $x = 50$ and $y = 50$ and later T_i is allowed to write $x = -10$ and T_j $y = 0$, both will think that the integrity constraint is preserved but it is actually violated in the final state. If a transaction is executed with an isolation level which does not prevent this phenomenon we must be sure that such scenario is managed by the application logic avoiding that kind of concurrent transactions.

In this section we present the isolation level definitions used in the rest of this work.

4.1 Concurrency control mechanisms

In stand-alone systems, isolation is managed by concurrency control protocols that usually rely on locks, versions, or both to manage concurrency. With a lock-based concurrency control [6], transaction operations acquire locks to access data items that block the operations of other transactions until the lock is released. A blocked operation can execute once the lock is released. There are two kinds of locks: read and write; and they can be used for different durations: long and short. Read locks only block write operations while write locks block both reads and writes since read locks can be shared but write locks are exclusive. Long locks are released when the transaction finishes, short locks are released when the operation finishes. The isolation level provided depends on the locks used during transactions execution.

Version-based concurrency control mechanisms store multiple versions per item [14, 58]. A new version of a data item is created in every write operation, but it becomes definitive (i.e., visible to new transactions) only when the transaction commits. In order to commit, a validation test must be applied to abort transactions that violate isolation constraints. The isolation level is determined by phenomena forbidden by this test.

4.2 Isolation levels

Several isolation level classifications have been proposed in the literature [27, 6, 1, 2]. The majority identify possible phenomena that may appear when transactions are executed concurrently and categorise isolation levels depending on the forbidden phenomena. The ANSI/INCITS specification [27] is widely accepted but, as Berenson et al. [6] showed up, it is ambiguous. Berenson et al. [6] refined ANSI definitions and extended the classification with new phenomena and isolation level definitions. Actually, they suggested one of the first definitions of Snapshot Isolation, supported at that moment by some commercial DBMS as Serialisable due to a loose interpretation of ANSI phenomena. They proved that SI allows some non-serialisable executions. Indeed, other papers [23] showed up that there were other anomalies in SI histories. However, Berenson's specification focuses on lock-based concurrency control, ignoring other mechanisms like multi-versioning, widely used to provide Snapshot Isolation.

Due to that fact, Adya [1] presented an alternative specification that is independent of concrete concurrency control mechanisms. Adya used a variation of Bernstein's serialisation graphs to represent histories as graphs showing dependencies among transactions. Phenomena are defined as properties in those graphs.

4.2.1 Direct serialisation graphs (DSG)

A history H's direct serialisation graph or $DSG(H)$ is a labeled directed graph such that V is the set of committed transactions in H and E represents dependencies between transactions based on conflicts in H.

Definition 10 (DSG)**.** *Given a history H, $DSG(H) = (V, E)$ is a labeled directed graph containing one vertex per committed transaction in H and an edge $T_i \longrightarrow T_j$ if one of the following dependencies occurs*[1]*:*

- *T_j directly read-depends on T_i, denoted as $T_i \xrightarrow{wr} T_j$, if $r_j(x_i) \in H$. $T_i \xrightarrow{wr} T_j$ is a read-dependency edge.*

[1] We refer to the definitions given in [2] instead of those presented in [1].

- *T_j directly write-depends on T_i, denoted as $T_i \xrightarrow{ww} T_j$, if $w_i(x_i) <_H w_j(x_j) \in H$ and it does not exist any other operation $w_k(x_k)$ such that $w_i(x_i) <_H w_k(x_k) <_H w_j(x_j) \in H$. $T_i \xrightarrow{ww} T_j$ is a write-dependency edge.*

- *T_j directly anti-depends on T_i, denoted as $T_i \xrightarrow{rw} T_j$, if $r_i(x_m) <_H w_j(x_j) \in H$ and it does not exist any other operation $w_k(x_k)$ such that $r_i(x_m) <_H w_k(x_k) <_H w_j(x_j) \in H$. $T_i \xrightarrow{rw} T_j$ is an anti-dependency edge.*

Notice that $DSG(H) = DSG(C(H))$ since it only considers committed transactions.

We say that T_j directly depends or depends on T_i if T_j directly read- or write-depends on T_i. We also say that T_j anti-depends on T_i if it directly anti-depends on T_i. Similarly, write-dependency edges and read-dependency edges are dependency edges.

As an example of DSG, given the following history (this is the flatten representation of H_1 but remember that a history is a partial order and not a total order):

$$H_1 = w_0(x_0)w_0(y_0)w_0(z_0)c_0r_i(x_0)w_i(x_i)r_i(y_0)c_iw_j(y_j)w_j(x_j)c_j$$

The associated $DSG(H_1)$ is depicted in Figure 4.1.

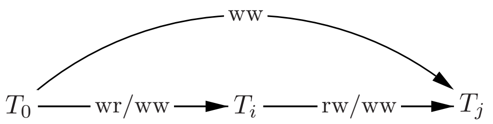

Figure 4.1: DSG of H_1

Adya used DSG to define a set of possible isolation phenomena. The main ones are the following:

- **G0: Write Cycles**: a history H exhibits phenomenon G0 if $DSG(H)$ contains a directed cycle composed only by write-dependency edges.

- **G1a: Aborted Reads**: a history H exhibits phenomenon G1a if it contains an aborted transaction T_i and a committed transaction T_j such that $w_i(x_{i.m}) <_H r_j(x_{i.m}) \in H$.

- **G1b: Intermediate Reads**: a history H exhibits phenomenon G1b if a transaction T_i reads in H a value written by T_j which is not the last write of T_j over the item. Formally, $w_i(x_{i.m}) <_H r_j(x_{i.m}) <_H w_i(x_{i.n}) \in H$ and $c_j \in H$.

- **G1c: Circular Information Flow**: a history H exhibits phenomenon G1c if $DSG(H)$ contains a directed cycle composed only by dependency edges.
- **G2: Anti-dependency Cycles**: a history H exhibits phenomenon G2 if $DSG(H)$ contains a directed cycle containing at least one anti-dependency edge.

Based on the previous phenomena, the following isolation levels are defined:

- **PL-1**: it forbids phenomenon G0 and provides a generalised specification for Read Uncommitted.
- **PL-2**: it forbids phenomena G0, G1a, G1b and G1c and provides a generalised specification for Read Committed.
- **PL-3**: it forbids phenomena G0, G1a, G1b, G1c and G2 and provides a generalised specification for Serialisable.

Instead of focusing on what should be observed in every transaction execution to determine if its isolation requirements have been ensured, Adya's definitions indicate what should happen in an entire history to guarantee a given isolation level to all committed transactions. Thus, if PL-1, PL-2 and PL-3 transactions are executed, we do not know if isolation requirements have been ensured to every transaction but which isolation level is ensured to the whole transaction set execution represented by H. To fill that void, Adya suggested a variation of serialisation graphs named Mixed Serialisation Graphs (MSG). Given a history H and its $DSG(H)$, $MSG(H)$ has all $DSG(H)$ nodes but only those edges representing obligatory dependencies for one of the involved transactions isolation levels. The overall execution is correct if $MSG(H)$ does not have cycles and does not show G1a and G1b phenomena for PL-2 and PL-3 transactions. The obligatory dependencies are the following:

- All direct write-dependencies.
- Direct read-dependencies ending in PL-2 and PL-3 transactions.
- Direct anti-dependencies starting from a PL-3 transaction.

4.3 Snapshot Isolation

Unfortunately, MSGs do not consider transactions requesting the SI level. The reason is that the SI specification proposed by Adya, named PL-SI, is not based on DSGs but on another variation named Start-dependency Serialisation Graphs (SSGs). Given a history H and a time-precedes order $<_t$, $SSG(H, <_t)$ has all $DSG(H)$ nodes and edges plus start-dependency edges:

- T_j **start-depends** on T_i, denoted as $T_i \xrightarrow{s} T_j$, if $c_i <_t s_j$. $T_i \xrightarrow{s} T_j$ is a start-dependency edge.

SSGs consider new phenomena:

- **G-SIa: Interference**: a history H and a time-precedes order $<_t$ exhibit the phenomenon G-SIa if $T_i \xrightarrow{ww} T_j \in SSG(H, <_t)$ or $T_i \xrightarrow{wr} T_j \in SSG(H, <_t)$ but $T_i \xrightarrow{s} T_j \notin SSG(H, <_t)$.
- **G-SIb: Missed Effects**: a history H and a time-precedes order $<_t$ exhibit the phenomenon G-SIb if $SSG(H, <_t)$ contains a directed cycle with exactly one direct anti-dependency edge.

PL-SI isolation level forbids phenomena G0, G1a, G1b, G1c, G-SIa and G-SIb.

4.3.1 An alternative definition for G-SIb

As other works already pointed out [39], PL-SI actually forbids more cycles than those explicitly forbidden by G0, G1c and G-SIb. Thus, we provide an alternative definition which explicitly excludes all graphs representing non PL-SI executions:

Definition 11 (New G-SIb: Missed Effects)**.** *A history H and a time-precedes order $<_t$ exhibit the phenomenon New G-SIb if $SSG(H, <_t)$ contains a directed cycle with at least one direct anti-dependency edge but does not contain two adjacent direct anti-dependency edges.*

Lemma 1 proves that both G-SIb definitions can be indistinctly used in PL-SI definition.

Lemma 1 (G-SIb and New G-SIb are equivalent)**.** *Given a history H and a time-precedes order $<_t$, $SSG(H, <_t)$ forbids G0, G1a, G1b, G1c, G-SIa and GSI-b phenomena iff $SSG(H, <_t)$ forbids G0, G1a, G1b, G1c, G-SIa and New G-SIb phenomena.*

Proof. New G-SIb encompasses more histories than the original G-SIb and, thus, when New G-SIb is proscribed, it is admitting less histories than the original definition. Conversely, we prove that any history H and time-precedes order $<_t$ proscribing G0, G1a, G1b, G1c, G-SIa and G-SIb proscribes also New G-SIb. This implies that both conditions sets are equivalent. By absurd reduction, we assume $SSG(H, <_t)$ is PL-SI and it has a cycle with anti-dependency edges but without two adjacent anti-dependency edges. Since $SSG(H, <_t)$ is PL-SI, it has not any cycle with a single anti-dependency edge and, thus, the cycle has at least two of those edges. Assume that one of them goes from T_i to T_j, $T_i \xrightarrow{rw} T_j$. Thus, $s_i <_t c_j$ because, otherwise, $c_j <_t s_i$ and there is a start-dependency edge from T_j to T_i which closes a cycle with a single anti-dependency edge which violates G-SIb. Since there are not two adjacent anti-dependency edges, there must be an edge from another node T_k to T_i and another one from T_j to T_l, both dependency or start-dependency edges. Thus, by G-SIa and start-dependency definitions, $c_k <_t s_i$ and $c_j <_t s_l$. Since $s_i <_t c_j$, then $c_k <_t s_l$ and this implies that $T_k \xrightarrow{s} T_l \in SSG(H, <_t)$. Consequently, there is a shorter cycle without $T_i \xrightarrow{rw} T_j$ anti-dependency edge. We can iteratively apply the same criterion until getting a cycle with a single anti-dependency edge which contradicts the initial assumption saying that H and $<_t$ avoid the original G-SIb phenomenon. $\square$

Chapter 5

Alternative definition of Snapshot Isolation

In this chapter we present an alternative definition of SI suitable to be included in MSG (see Chapter 6). This definition is equivalent to the original PL-SI definition and forbids the same phenomena than the original Snapshot Isolation level as defined by Berenson et al [6].

A history H is PL-SI if it has been generated by a scheduler $S(E, <_t)$ such that E is the set of start and commit events of committed transactions in H, $<_t$ is a time-precedes order on E and $SSG(H, <_t)$ does not show G0, G1a, G1b, G1c, G-SIa and G-SIb phenomena. Actually, $<_t$ is also a conflict-aware time-precedes order since for any dependency edge $T_i \xrightarrow{ww/wr} T_j$ $c_i <_t s_j$ due to G-SIa and for any edge $T_i \xrightarrow{rw} T_j$ then $s_i <_t c_j$ due to G-SIb (if $c_j <_t s_i$ then a start-dependency appears closing a cycle with a single anti-dependency, which is forbidden by G-SIb). Notice that start-dependency edges based on conflicts can be deduced from $DSG(H)$ itself. If the scheduler generates PL-SI histories and $T_i \xrightarrow{ww/wr} T_j$ then $c_i <_t s_j$. Furthermore, if $T_i \xrightarrow{rw} T_j$ then $s_i <_y c_j$ since, otherwise, there will be a start-dependency from T_i to T_j closing a cycle with a single anti-dependency edge. If $T_i \xrightarrow{ww} T_j \xrightarrow{wr} T_k$ then $c_i <_t s_k$ too. Similarly, if $T_i \xrightarrow{ww} T_j \xrightarrow{rw} T_k \xrightarrow{wr} T_l$ then $c_i <_t s_l$. The orderings of starts and commits of transactions not connected in $DSG(H)$ are meaningless. Then, a scheduler S is PL-SI if a conflict-aware time-precedes order $<_t$ can be defined over the set of start and commit operations of $C(H)$ for every H it generates. This $<_t$ actually represents how the scheduler ordered conflicting transactions. That is the basis of our alternative definition of snapshot isolation. Informally, given a PL-SI' history H:

- The starts and commits of committed transactions in H can be ordered following a conflict-aware time-precedes order $<_c$.

- $SSG(H, <_c)$ is PL-SI.

As we are going to prove now, if that happens PL-SI and PL-SI' can be considered equivalent.

5.1 PL-SI': an alternative definition of PL-SI

G-SIb can be redefined in the following way:

Definition 12 (G-SIb': Missed Effects)**.** *A history H exhibits the phenomenon G-SIb' if $DSG(H)$ contains a directed cycle with at least one direct anti-dependency edge but without two adjacent direct anti-dependency edges.*

This definition is quite similar to G-SI* presented in [39] but our definition gets rid of start-dependency edges and that will help us later to support Snapshot Isolation level in Adya's Mixed Serialisaton Graphs.

Thus, PL-SI' can be defined in the following way:

Definition 13 (PL-SI')**.** *A history H is PL-SI' if it forbids G0, G1a, G1b, G1c and G-SIb'.*

As this definition states, the really important thing about Snapshot Isolation is not when the snapshot is taken considering real time but that the snapshot isolation level observed by transactions is consistent for some time-precedes order, which actually represents the virtual time followed by the system scheduler to order transactions.

5.2 PL-SI' and PL-SI equivalence

PL-SI' and PL-SI do not represent actually the same isolation level, basically because PL-SI' is based on dependencies in H but PL-SI also contemplates dependencies among start and commit operations in E. However, a PL-SI' history H is also PL-SI if it represents a correct PL-SI execution or, in other words, if exists a time-precedes order $<_t$ such that $SSG(H, <_t)$ is PL-SI. Thus, PL-SI' is equivalent to PL-SI if for any PL-SI' history H exists at least one scheduler $S(E, <_t)$ such that $SSG(H, <_t)$ is PL-SI. In this section we prove that any PL-SI' history H can be generated by a PL-SI scheduler S based on a conflict-aware time-precedes order $<_c$.

The inverse assertion is trivially true. If H and $<_c$ are PL-SI then H is PL-SI' since $DSG(H) \subseteq SSG(H, <_c)$. If there is not any cycle with anti-dependencies but without two adjacent anti-dependencies in $SSG(H, <_c)$ then this cycle obviously does not exist in $DSG(H)$. Thus if G-SIb is proscribed in $SSG(H, <_c)$, G-SIb' does not appear in $DSG(H)$.

We split this proof in two complementary parts. First, given a PL-SI' history H and a conflict-aware time-precedes order $<_c$ over the set E of start and commit operations of H's committed transactions, we prove that $SSG(H, <_c)$ is PL-SI. Next, we prove that at least one conflict-aware time-precedes order can be defined over E if H is PL-SI'.

5.2.1 $SSG(H, <_c)$ is PL-SI

Thus, assume a PL-SI' history H and a conflict-aware time-precedes order $<_c$ over the set E of starts and commits in $C(H)$. We prove that $SSG(H, <_c)$ is PL-SI.

Both PL-SI and PL-SI' forbid G0, G1a, G1b and G1c phenomena. Then, H is PL-SI if $SSG(H, <_c)$ does not show G-SIa and G-SIb. Notice that $<_c$ actually represents how SI orders transactions depending on which committed state they observe. If T_j observed something from T_i then a transaction T_i is in T_j snapshot then T_i committed before T_j started. On the contrary, if T_i is not part of T_j snapshot then T_i committed after T_j started. Thus, a scheduler S following a conflict-aware time-precedes order forbids G-SIa b) condition of Definition 7. Lemma 2 proves that $SSG(H, <_c)$ also forbids New G-SIb.

Lemma 2 (H forbids New G-SIb)**.** *Given a PL-SI' history H and a scheduler S ensuring a conflict-aware time-precedes order $<_c$, $SSG(H, <_c)$ does not show New G-SIb phenomenon.*

Proof. By absurd reduction, assume $SSG(H, <_c)$ shows a New G-SIb forbidden cycle C (i.e., a directed cycle with at least one anti-dependency edge but without two adjacent anti-dependency edges). $SSG(H, <_c)$ has all $DSG(H)$ vertices and edges plus the start-dependency edges. Thus, given S the set of start-dependency edges generated from $<_c$ and $DSG(H) = (V, E)$, $SSG(H, <_c) = (V, E \cup S)$. Since H is PL-SI', there are no cycles in $DSG(H)$ composed by dependency and anti-dependency edges where two no adjacent anti-dependency edges appear. Then, C must include at least one start-dependency. Assume $e = T_i \xrightarrow{s} T_j$ is one of those start-dependency edges. Since e is part of a New G-SIb cycle, there is a directed path from T_j to T_i without two adjacent anti-dependency edges. From Lemma 8 we deduce $s_j <_c c_i$ but, since e is a start-dependency edge, $c_i <_c s_j$ also which is a contradiction. Then, $SSG(H, <_c)$ does not show New G-SIb phenomena.

□

Theorem 1. *If H is PL-SI' then $SSG(H, <_c)$ is PL-SI.*

Proof. G0, G1a, G1b, G1c are avoided by definition since H is supposed to be PL-SI'. Thus, we only have to prove that G-SIa and G-SIb are avoided too:

- **G-SIa**: It is trivially avoided by Condition b) of Def. 7.

- **G-SIb**: Lemma 2 proves that New G-SIb never shows up in a PL-SI' history H produced by a scheduler based on a conflict-aware time-precedes order. Lemma 1 shows that a history H avoiding G0, G1a, G1b, G1c, G-SIa and New G-SIb also proscribes G-SIb.

□

Thus, a history H produced by a PL-SI' scheduler S ensuring a conflict-aware time-precedes order $<_c$ is also a PL-SI history.

Now we prove that any history PL-SI' H can be produced by a scheduler ensuring a conflict-aware time-precedes order.

5.2.2 A conflict-aware time-precedes order $<_c$ can be defined over a PL-SI' history H

By absurd reduction, assume a PL-SI' history H such that the set O of start and commit operations of committed transactions in H can never be ordered by a conflict-aware time-precedes order $<_c$. That can only happen if the conflict-aware time-precedes order restrictions based on conflicts necessarily produce a contradiction like $s_i <_c c_j$ and $c_j <_c s_i$.

Given an order $<_t$ of O, $<_t$ is a time-precedes order if it is a partial order, any pair $s_i, c_j \in O$ composed by a start and a commit operations are ordered (i.e., $s_i <_t c_j$ or $c_j <_t s_i$) and $s_i <_t c_j$ if $i = j$. For example, the order $<_r$ such that $\forall s_i, c_j \in O$, $s_i <_r c_j$ is a time-precedes order. A scheduler based on $<_r$ executes all transactions concurrently (no transaction is allowed to commit unless all transactions have started).

In a conflict-aware time-precedes order $<_c$ some extra restrictions are added to force a specific order when a conflict appears among two transactions. Those restrictions are based on the snapshot isolation level definition. If T_i overwrites or reads a value established by T_j then $c_j <_c s_i$. If a value read by T_i is updated by T_j then $s_i <_c c_j$. The start and commit operations of a history H cannot be ordered by a conflict-aware time-precedes order if those restrictions are contradictory in H. For example, if T_i overwrites a value established by T_j and T_i reads a value overwritten by T_j in H then $c_j <_c s_i$ and $s_i <_c c_j$ for any possible conflict-aware time-precedes order $<_c$, which is a contradiction. There are other possible contradictions as well. For example, if $T_i \xrightarrow{ww} T_j$ and $T_j \xrightarrow{wr} T_i$ in $DSG(H)$ then $c_i <_c s_j$ and $c_j <_c s_i$. Since $s_i <_c c_i$ and $s_j <_c c_j$ also then $s_i <_c c_i <_c s_j <_c c_j <_c s_i$ which is a contradiction. We prove that if a contradiction appears then H is not PL-SI'.

First we introduce what a fixed ordering is.

Definition 14 (Fixed ordering)**.** *Given a history H and O the set of starts and commits of transactions in $C(H)$, the ordering of $c_i, s_j \in O$ is fixed if it can be deduced from the conflict-aware time-precedes order restrictions.*

For example, if $T_i \xrightarrow{ww} T_j \in DSG(H)$ then $c_i <_c s_j$ is fixed by Definition 7's condition b). Since $s_i <_c c_i$ and $s_j <_c c_j$ by Definition 7's condition a) and Definition 6 condition a), $s_i <_c s_j$, $c_i <_c c_j$ and $s_i <_c c_j$ are also fixed. If also $T_j \xrightarrow{rw} T_k \in DSG(H)$ then, by Definition 7's condition c), $s_j <_c c_k$ and $s_i <_c c_k$ are fixed too and so on.

It can be proved that the ordering of operations s_i and c_j $(i \neq j)$ is fixed under the presence of certain type of paths in $DSG(H)$. More specifically, Lemma 3 proves that $s_i <_c c_j$ is fixed iff there is in $DSG(H)$ a directed path P from T_i to T_j without two adjacent anti-dependency edges. Furthermore, Lemma 4 shows that $c_i <_c s_j$ is fixed iff P ends are dependency edges.

Assume a PL-SI' history H such that its commit and start operations in O cannot be ordered by any conflict-aware time-precedes order because a contradiction like $s_i <_c c_j$ and $c_j <_c s_i$ arises due to fixed orderings. From Lemmas 3 and 4, we deduce that it must exist a directed path from T_i to T_j without two adjacent anti-dependency edges and another path from T_j to T_i also without two adjacent anti-dependency edges and with dependency edges at the ends. Thus, there is a directed cycle in $DSG(H)$ involving T_i and T_j without two adjacent anti-dependences and, hence, H is not PL-SI', contradicting the initial assumption. Thus, if H is PL-SI' that kind of contradiction never appears and, hence, no contradictions will arise due to fixed orderings. Since the rest of the orderings can be freely established, O can be always ordered by a conflict-aware time-precedes order $<_c$ and $SSG(H, <_c)$ is PL-SI.

Lemma 3 (Fixed ordering 1)**.** *Given a PL-SI' history H and its $DSG(H) = (V, E)$, the set O of start and commit operations of transactions in $C(H)$, $s_i, c_j \in O$ and $<_c$ a conflict-aware time-precedes order, $s_i <_c c_j$ is fixed iff there is a directed path $P = (V_p \subseteq V, E_p \subseteq E)$ from T_i to T_j without two adjacent anti-dependency edges.*

Proof. The proof is split in two:

a) **$s_i <_c c_j$ is fixed if there is a directed path P from T_i to T_j without two adjacent anti-dependency edges in $DSG(H)$.** We prove it by induction over the length n of P.

- **Base case** $(n = 2)$: P is composed by a single edge $e = T_i \longrightarrow T_j$. If e is a dependency edge then $c_i <_c s_j$ by Definition 7's condition b) and, then, $s_i <_c c_i <_c s_j <_c c_j$ from Definition 6 condition a). If e is an anti-dependency edge we get $s_i <_c c_j$ directly by Definition 7's condition c). In both cases $s_i <_c c_j$.
- **Induction hypothesis** $(n < l)$: the lemma holds if P is $n < l$ length.
- **Induction step** $(n = l)$: P length is $n = l > 2$. Then, $P = T_i \longrightarrow T_k \longrightarrow T_j$ where $e = T_k \longrightarrow T_j$ is P's last edge

and P' the subpath from T_i to T_k. P' is a $n-1$ length directed path without two adjacent anti-dependency edges and, thus, from the induction hypothesis $s_i <_c c_k$. Depending on e's type two possibilities arise:

- *e is a dependency edge.* In that case, by Definition 7's condition b) we get $c_k <_c s_j$. Then, $s_i <_c c_k <_c s_j$. By Definition 6 condition a), $s_i <_c s_j <_c c_j$.
- *e is an anti-dependency edge.* By Definition 7's condition c) we get $s_k <_c c_j$. Since P length $n > 2$ then the length of P' is at least two. Then, $P' = T_i \xrightarrow{P''} T_{k-1} \xrightarrow{e'} T_k$. e' is a dependency edge since P does not has two adjacent anti-dependency edges. Thus, $c_{k-1} <_c s_k$ and, then, $c_{k-1} <_c s_k <_k c_j$. If P'' is length 1 (the only two edges are e and e') then $T_{k-1} = T_i$. Since $s_i <_c c_i$ (from Definition 6 condition a)), $s_i <_c c_i <_c s_k <_c c_j$. If P'''s length is greater than 1, by the induction hypothesis $s_i <_c c_{k-1}$ since P'' is also a subpath of P and, hence, it is a directed path without two adjacent anti-dependency edges of length $< l$. In that case $s_i <_c c_{k-1} <_c s_k <_c c_j$. In both cases we get $s_i <_c c_j$.

b) **There is a directed path P from T_i to T_j without two adjacent anti-dependency edges in $DSG(H)$ if $s_i <_c c_j$ is fixed**. Since Definition 7 only directly fixes conflicting operations which produce an edge in $DSG(H)$, if $s_i <_c c_j$ then there must be a walk P connecting T_i and T_j. We do the rest of the proof by induction over the length n of P.

- **Base case** ($n = 2$): since P is composed by a single edge e, no two adjacent anti-dependency edges may exist in P. As $s_i <_c c_j$ then, by Definition 7 conditions, e must be either a dependency edge or an anti-dependency edge starting from T_i and ending in T_j. Thus P is a directed path from T_i to T_j.
- **Induction hypothesis** ($n < l$): the lemma holds if P is $n < l$ length.
- **Induction step** ($n = l$): P's length is $n = l > 2$. Take any edge $e \in E_p$ such that $P = P'T_k \longrightarrow T_l P''$, P' and P'' are subwalks of P in $DSG(H)$, P' starts with edge T_i and P'' ends with edge T_j and $e = T_k \longrightarrow T_l$. P' and P'' are of length $< l$ and at least one of them is of length > 1 since $n > 2$. e must be of one of the following types:
  - *$e = T_k \overset{ww/wr}{\longrightarrow} T_l$ is a dependency edge.* By Definition 7's condition b) we get $c_k <_c s_l$. Since $s_i <_c c_j$ is fixed due to P, $s_i <_c c_k <_c s_l <_c c_j$ and, hence, $s_i <_c c_k$ and $s_l <_c c_j$ are fixed. From the induction hypothesis, P' and P'' are

directed paths without two consecutive anti-dependency edges. P' goes from T_i to T_k and P'' from T_l to T_j. Since the connecting edge e is not an anti-dependency edge, the resulting concatenated walk P is also a directed path without two consecutive anti-dependency edges.

– $e = T_k \xrightarrow{rw} T_l$ *is an anti-dependency edge.* By Definition 7's condition c) $s_k <_c c_l$ is fixed. Since we assume $s_i <_c c_j$ is also fixed due to P, $s_i <_c s_k$ and $c_l <_c c_j$. Since $s_k <_c c_k$ and $s_l <_c c_l$ (from Definition 6 condition a)), $s_i <_c c_k$ and $s_l <_c c_j$. If P' and P'' are of length > 2, by the induction hypothesis, P' and P'' are directed paths without two adjacent anti-dependency edges, the first goes from T_i to T_k and the second one from T_l to T_j. Assume $e' = T_{k-1} \longrightarrow T_k$ the edge just before e and $e'' = T_l \longrightarrow T_{l+1}$ the edge just after e. Since $s_k <_c c_l$ then $s_i <_c o_{k-1} <_c s_k <_s c_l <_s o_{l+1} <_c c_j$ where o_{k-1} and o_{l+1} are either the start or commit operation of T_{k-1} and T_{l+1}. Since Definition 7's condition only explicitly orders starts with commits for a given edge, never starts with starts or commits with commits, o_{k-1} must be commit and o_{l+1} must be a start. Then, e' and e'' are necessarily dependency edges. Thus, the path composed by P', e and P'' is a directed path from T_i to T_j without two adjacent anti-dependency edges. The cases when P' and P'' are of length 1 can be similarly deduced. In the first case $T_i = T_k$ and P'' is a directed path from T_l to T_j without two anti-dependency edges. Thus, if $e'' = T_l \longrightarrow T_{l+1}$ is the edge after e and the first edge of P'', $s_i <_c c_l <_c o_{l+1} <_c c_j$, o_{l+1} must be start, e'' is a dependency edge and, thus, the concatenation of e and P'' is a directed path from T_i to T_j without two adjacent anti-dependency edges. If P'' is of length 1 then P' is a directed path from T_i to T_k without two adjacent anti-dependency edges. If $e' = T_{k-1} \longrightarrow T_k$ is the edge just before e and the last edge of P' then $s_i <_c o_{k-1} <_c s_k <_c c_j$ and, thus, o_{k-1} is a commit, e' is a dependency edge and, then, the concatenation of P' and e is a directed path from T_i to T_j without two adjacent anti-dependency edges. In any case, the resulting path is a directed path from T_i to T_j without adjacent anti-dependency edges.

□

Lemma 4 (Fixed ordering 2)**.** *Given a PL-SI' history H and its $DSG(H) = (V, E)$, the set O of start and commit operations of transactions in $C(H)$, $c_i, s_j \in O$ and $<_c$ a conflict-aware time-precedes order, $c_i <_c s_j$ is fixed iff there is a* *directed path $P = (V_p \subseteq V, E_p \subseteq E)$ from T_i to T_j without two adjacent anti-dependency edges and dependency edges in the ends.*

Proof. The proof is split in two:

a) $c_i <_c s_j$ **is fixed if there is a directed path P from T_i to T_j without two adjacent anti-dependency edges in $DSG(H)$ and dependency edges on the ends**. We prove it by induction over the length n of P.

- **Base case** $(n = 2)$: P is composed by a single edge $e = T_i \xrightarrow{ww} T_j$. Since e is a dependency edge then $c_i <_c s_j$ by Definition 7's condition b).
- **Induction hypothesis** $(n < l)$: the lemma holds if P is $n < l$ length.
- **Induction step** $(n = l)$: P is length $n > 2$ and the ends are dependency edges. If $n = 3$ then $P = T_i \xrightarrow{ww/wr} T_k \xrightarrow{ww/wr} T_j$. By Definition 7's condition b) we get $c_i <_c s_k$ and $c_k <_c s_j$. Since by Definition 6's condition a) $s_k <_c c_k$, $c_i <_c s_j$. If $n > 3$ then there is a subpath P' of P such that $P = T_i \xrightarrow{e} T_k \xrightarrow{P'} T_l \xrightarrow{e'} T_j$ where e and e' are P's ends and, hence, dependency edges. Since P' is a directed path without adjacent anti-dependency edges, from Lemma 3 $s_k <_c c_l$.Since e and e' are dependency edges and, by Definition 7's condition b), $c_i <_c s_k$ and $c_k <_c s_j$. Then, $c_i <_c s_k <_c c_l <_c s_l$. As a result, $c_i <_c s_j$ is fixed.

b) **There is a directed path p from T_i to T_j without two adjacent anti-dependency edges in $DSG(H)$ and dependency edges on the ends if $c_i <_c s_j$ is fixed**. From Definition 6's condition a) $s_i <_c c_i$, $s_j <_c c_j$ and, hence, $s_i <_c c_j$. Then, there is a directed path P from T_i to T_j without adjacent anti-dependency edges. We prove by induction over the length n of P that P ends are dependency edges.

- **Base case** $(n = 2)$: P is composed by a single edge $e = T_i \longrightarrow T_j$. Since $c_i <_c s_j$, by Definition 7's condition b), e only can be a dependency edge.
- **Induction hypothesis** $(n < l)$: the lemma holds if P is $n < l$ length.
- **Induction step** $(n = l)$: P is length $n > 2$. Since $c_i < s_j$ due to a conflict-aware time-precedes order and $n > 2$, necessarily $c_i < o_k < o_l < s_j$. If $n = 3$ then $T_k = T_l$. Since only dependencies order commits with other operation in a conflict-aware time-precedes order then $o_k = s_k$ and $o_l = c_k$

and $P = T_i \overset{ww/wr}{\longrightarrow} T_k \overset{ww/wr}{\longrightarrow} T_j$. If $n > 3$ then $T_k \neq T_l$. Anyway, conflict-aware time-precedes order conditions only order commits with starts and, hence, $o_k = s_k$ and $o_l = c_l$, so, $P = eP'e'$ where $e = T_i \overset{ww/wr}{\longrightarrow} T_k$ and $e' = T_k \overset{ww/wr}{\longrightarrow} T_j$. Since $s_k < c_l$ then P' is a directed path from T_k to T_l without two consecutive anti-dependency edges. Since e and e' are dependencies, P is a directed path without two consecutive anti-dependency edges which starts and ends with dependency edges.

□

5.3 PL-SI' and SI equivalence

PL-SI' correction can be also proved by showing that PL-SI' and SI forbid and tolerate the same phenomena. Instead of using Adya's phenomena definitions, now we use the original definitions presented by Berenson et al [6]. This time we don't do a formal correctness proof but just informally prove that histories representing the different phenomena tolerated and forbidden by SI, including the read-only transaction anomaly detected by Fekete et al [23], are also respectively tolerated and forbidden by PL-SI'. We do not contemplate histories with reads of intermediate or aborted values since they are explicitly prohibited by PL-SI' (G1a and G1b).

Berenson et al detected that two possible interpretations of ANSI phenomena definitions are possible and concluded that loose interpretations of those definitions should be used to truly support a serialisable isolation level. Actually, they proved that some DBMS (like Oracle) were not providing serialisable as they claimed but a slighty relaxed isolation level named Snapshot Isolation level or SI. However, that loose interpretation also proscribes some serialisable executions and Berenson serialisable is stricter than it should be [1]. For example the history $w_1(x_1)r_2(x_1)c_1c_2$ violates Berenson's Dirty Read but it is a serialisable history. That is one of the reasons why our work is based on Adya phenomena definitions and not on Berenson's ones. Thus, instead of focusing on Berenson's phenomena definitions, we focus on the histories Berenson and Fekete used to explain SI forbidden and allowed phenomena and show that PL-SI' forbids and allows the same things than SI.

Write-Skew phenomenon The *Write-Skew phenomenon* is allowed by SI [6] and appears when transactions T_1 and T_2 read items x and y concurrently and, later, T_1 updates x and T_2 y. The history is non-serialisable because T_1 and T_2 never see each other updates.

$$r_1(x_0)r_1(y_0)r_2(x_0)r_2(y_0)w_1(x_1)w_2(y_2)c_1c_2$$

As Figure 5.1 shows up this history generates a cycle with two adjacent anti-dependencies and, thus, it is not PL-3 (serialisable) but it is PL-SI'.

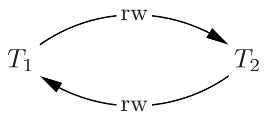

Figure 5.1: Write Skew example DSG

Read-Only Transaction Anomaly The *Read-Only Transaction Anomaly* is also allowed by SI [23] and represents a variation of the classical bank account example. Assume x and y are two empty accounts ($x = 0, y = 0$) and an application logic rule stating that there is a withdrawal penalty if $x + y$ goes below 0. Now, assume a transaction T_1 reads x and deposits 20$ on y, T_2 reads x and y and subtracts 10$ from x and transaction T_3 reads x and y and prints the result to the costumer. Depending on how those transactions are scheduled, with SI that execution may result in the following history:

$$r_2(x_0)r_2(y_0)r_1(x_0)w_1(y_1)c_1r_3(x_0)r_3(y_1)c_3w_2(x_2)c_2$$

This execution is not serialisable since transaction T_2 does not observe T_1 update but overwrites T_3 reads which observes T_1 updates at the same time. Figure 5.2 shows the resulting DSG cycle.

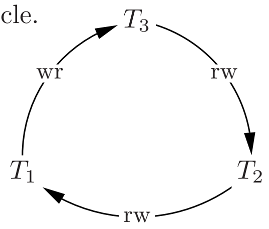

Figure 5.2: Read-only transaction anomaly example DSG

As expected, the DSG shows a PL-SI' legal cycle and, hence, the resulting execution is also PL-SI' but not PL-3.

Lost Update SI forbids the *Lost Update* anomaly [6]. This phenomenon appears when a transaction T_1 reads an item x, another transaction T_2 overwrites the same item and finally T_1 updates x again based on the previous value, the one it has read. The execution is represented in the following history:

$$r_1(x_0)w_2(x_2)c_2w_1(x_1)$$

As Figure 5.3 shows, these lost updates are neither allowed by PL-SI' because of the cycle with a single anti-dependency edge. Notice that this is neither allowed by PL-3 but it is allowed by PL-2 (read committed).

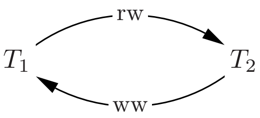

Figure 5.3: Lost Update example DSG

Dirty Read and Dirty Write phenomena *Dirty read* and *Dirty Write* phenomena, both forbidden by SI, are already forbidden by PL-2 as Adya already proved.

Read Skew *Read Skew* phenomena, also proscribed by SI, is also forbidden by PL-SI' as the following example shows. Read skew appears when a transaction T_1 reads x, another transaction T_2 updates x and y and, finally, T_1 reads y. T_1 has seen an inconsistent state of the database. The resulting history is the following:

$$r_1(x_0)w_2(x_2)w_2(y_2)c_2r_1(y_2)c_1$$

As Figure 5.4 shows, that history DSG shows a cycle with a single anti-dependency and, thus, it does not represent a PL-SI' execution. As expected, it neither is PL-3 but it is PL-2.

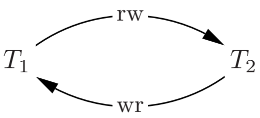

Figure 5.4: Read Skew example DSG

Non-Repeatable or Fuzzy Read Another phenomenon also forbidden by SI is the *Non-Repeatable or Fuzzy Read*. This phenomenon appears when a transaction T_1 reads an item x, this item is updated by T_2 which commits and finally T_1 reads the same item again but observing a different value. The resulting DSG is identical to the one in Figure 5.3 and, thus, it does not represent a PL-SI' history.

Thus, all anomalies allowed by SI are also allowed by PL-SI' and all anomalies forbidden by SI are not allowed by PL-SI'. Thus, both isolation levels are equivalent.

5.4 About the time-precedes order in Snapshot Isolation

As we previously concluded in Section 5.3, any PL-SI' history H is also the result of executing a set of transactions in a PL-SI scheduler. PL-SI schedulers also ensure a conflict-aware time-precedes ordering of transactions start and commit operations. This ordering reflects how these operations have been virtually ordered by the concurrency control mechanism which might or might not be similar to the order expected by applications and users or to the order we obtain considering the real time.

Thus, given the PL-SI' history H representing the execution of a set of transactions, its set of start and commit operations can be ordered in three possible ways:

- $<_s$ represents the ordering observed by one possible PL-SI scheduler equivalent to H execution.
- $<_u$ represents the ordering observed by the user or application.
- $<_r$ represents the real time ordering.

For example, assume three transactions $T_i = w_i(x_i)c_i$, $T_j = r_j(x)r_j(y)c_j$ and $T_k = w_k(y_k)c_k$. Assume that $r_j(x_0) <_h w_i(x_i) \in H$ and $r_j(y_0) <_h w_k(x_y)$, considering x_0 and y_0 both items values before any of those transactions started their executions. If we assume H is PL-SI' then $s_j < c_i$ and $s_j <_s c_k$. The user thinks that $c_i < s_j$ and $s_j < c_k$ because he or she executed T_i first, T_k before receiving T_i response and T_j after receiving T_i response but before observing T_k result. However, in the real time $c_i < s_j$ and $c_k < s_j$.

The previous example might not make sense in a centralised database since, in that case, the last state is always available. However, that may happen in a replicated or distributed database. For example, T_i executed at node N_i, T_j at node N_j and T_k at node N_k. When T_j started its execution at N_j T_i and T_j already committed at N_i and N_k but not at N_j due to some sort of delay. However, the user sent T_j when N_i already sent T_i's result back to the user. In any case the execution is Snapshot. However, the closer $<_s$ is to $<_u$ and $<_r$ the more transparent will be our replicated database (i.e., the more similar will be to a centralised system). Actually, this similarity is closely related to the replica consistency level provided by a replicated scheduler. Consistency among replicas is outside the scope of this thesis but it might be an interesting research field in further works.

Chapter 6

Extended mixed serialisation graph (EMSG)

As we have seen in Chapter 4, Adya introduced MSG to analyse isolation correctness when multiple isolation levels come concurrently into play. The following example explains why DSGs cannot be used in such cases:

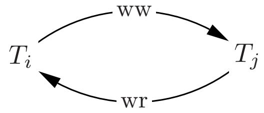

Figure 6.1: Example graph

The DSG in Figure 6.1 depicts the dependencies between transactions T_i and T_j. The execution is correct if both T_i and T_j requested PL-1; if T_i and T_j requested PL-2 the given execution is incorrect since it shows G1c phenomena (see Section 4.2). If transaction T_i requested PL-1 and transaction T_j requested PL-2, the given DSG is insufficient to detect if the given execution history is correct. EMSGs are instead used to define correct execution histories if different isolation levels are used for different transactions.

MSGs represent dependencies among PL-1, PL-2 and PL-3 transactions in histories. Given a history H and its $DSG(H) = (V, E)$, $MSG(H) = (V, \{e | e \in E \textit{ and e is obligatory}\})$ is a labeled directed graph. Which edges are *obligatory* depend on the isolation levels of their source and target transactions. All direct write-dependencies are considered obligatory since they matter to all isolation levels (all levels forbid G0 phenomenon). Only direct read-dependencies edges ending in a PL-2 or PL-3 transaction are considered obligatory since only those levels are concerned on what they read (PL-1 transactions are allowed to see dirty values). Finally, only direct anti-dependencies starting with a PL-3 transaction are considered obligatory since only transactions requesting this isolation level are worried about who is overwriting the values they read (that's why PL-3 forbids G2).

Unfortunately, MSGs do not contemplate the snapshot isolation level because it is defined using SSG instead of DSG. In this section, we take profit of our SI definition, PL-SI', to extend MSG with full support to executions including SI transactions:

Definition 15 (EMSG)**.** *Given a history H and its $DSG(H) = (V, E)$, $EMSG(H) = (V, \{e | e \in E$ and a is obligatory$\})$ is a labeled directed graph. An edge a is obligatory if either:*

- *e is a direct write-dependency edge, or*
- *e is a direct read-dependency edge and e's target is executed under PL-2, PL-3 or PL-SI', or*
- *e is a direct anti-dependency edge and e's source is executed under PL-SI' or PL-3.*

The extension is based on the PL-SI' definition given in Chapter 5. Notice that PL-SI' forbids cycles including direct write-dependencies, direct read-dependencies and even in some cases including also anti-dependencies. PL-SI' transactions are concerned about what they read and overwrite (only values coming from their snapshot) and about who overwrites their reads (only transactions not included in their snapshot) and, hence, all types of dependencies matter to PL-SI' transactions.

Adya's mixing-correct [1] definition can also be extended to identify valid histories involving PL-1, PL-2, PL-SI' and PL-3 transactions:

Definition 16 (Valid history)**.** *A history H is valid if:*

(a) PL-2, PL-3 or PL-SI' transactions do not read aborted or intermediate values.

(b) $EMSG(H)$ does not contain any directed cycle unless it contains two adjacent anti-dependency edges sharing a PL-SI' transaction.

The main difference between Adya's definition and ours is that we support PL-SI' transactions and, thus, allow the presence of cycles with two adjacent anti-dependency edges as long as they are connected by a PL-SI' transaction, like in PL-SI' definition. Theorem 7 formally proves that a valid H history correctly manages Snapshot Isolation level (see page 112 in Chapter A). Fekete and other authors reached to similar conclusions [21, 39]. Fekete detected which phenomena should be avoided to provide serialisable executions when transactions only request serialisable or Snapshot Isolation. The execution is serialisable if its serialisation graph does not show cycles and the only possible cycles in a scheduler supporting serialisable and SI transactions are those with two adjacent anti-dependencies pivoting on a PL-SI transaction. Notice that we reach exactly to the same conclusion, a valid scheduler supporting PL-3 and PL-SI' only allows cycles including two adjacent anti-dependencies sharing a PL-SI' transaction. Note also that we are not interested on providing serialisable executions in the presence of PL-SI' transactions but under wich circumstances every transaction involved in a given execution gets the isolation level it requests. Thus, unless all transactions request PL-3, the final result might not be serialisable. Furthermore, Fekete work is not based on Adya's DSGs but on Bernstein serialisation graphs. However his conclusions are equivalent to ours. Lin et al work was only focused on Snapshot Isolation and cannot be applied to histories observing transactions with different isolation needs. Unlike our G-SIb' phenomena in Definition 13 (see Chapter 5 in page 41), their G-SIb* phenomenon still keeps start-dependency edges and cannot be applied in our case but they also point out that cycles with two consecutive anti-dependency edges must be avoided in histories including only SI transactions.

Thus, the execution history presented in Figure 6.1 is valid if transaction T_i is executed under PL-1 and transaction T_j is executed under PL-2. In that case the edge $T_j \xrightarrow{wr} T_j$ is not obligatory and the EMSG does not show any cycle. However, the history is invalid if T_i requested PL-2 and T_j requested PL-1 since, in that case, both edges are obligatory and there is a cycle in the resulting EMSG.

Definition 16 does not necessarily mean that concurrency control protocols should search for cycles in EMSGs. It only points out *what* should be considered to detect valid execution histories and not *how* that should be done. Actually, as we previously explained in Chapter 4, existing DBMSs supporting serialisable and snapshot use multi-version and lock-based concurrency control to ensure every transaction isolation needs.

Unfortunately, EMSGs and Definition 16 cannot be used in replicated environments. In those systems there is one transaction execution history per node and so Definition 16 cannot be applied directly because it does not account for dependencies between nodes. For example, two conflicting transactions can execute at two distinct nodes in different order but the local graphs at each node could still satisfy the conditions of Definition 16. This case is studied in Chapter 7.

6.1 Strict histories

Regular centralised DBMS (PostgreSQL, MySql, SqlServer, Oracle, etc.) disallow some correct executions to avoid some undesirable effects from the recoverability point of view [14]. The DBMS recovery system is in charge of ensuring that the database behaves as if only the committed transactions effects are present in the database. If all transactions commit the recovery is rather easy. However, if transactions may abort the system must deal with some tricky executions. For example, the PL-3 isolation level definition as exposed by Adya allows something like this:

$$w_i(x_i)r_j(y_0)r_j(x_i)w_j(y_j)r_k(y_j)c_ic_jc_k$$

This execution is serialisable since it is equivalent to the serial execution

$$w_i(x_i)r_i(y_0)c_ir_j(x_i)w_j(y_j)c_jr_k(y_j)c_k.$$

However, from the recoverability point of view, if T_i aborts then T_j and T_k should be aborted too. To prevent cascading aborts, centralised DBMS usually avoid PL-2, PL-SI' and PL-3 transactions to read values written by uncommitted transactions (PL-1 allow dirty reads). Lock-based systems use long write locks and read locks (long for PL-3, short for PL-2) to block reads until writes are committed. In version-based systems reads get the last committed version of items. In both cases the value obtained comes from a committed transaction at the time the read is performed.

Similar problems may arise when multiple and concurrent writes are performed over the same item. In the general case, when a transaction updates an item x and aborts, the original image of x is restored. However, in executions like the following one the recovery is not that easy:

$$w_i(x_i)w_j(x_j)c_ic_j.$$

If T_i aborts the value to be restored should be x_j. If T_j also aborts the previous value was x_i but T_i has been aborted too. Thus, the original value has been lost. To overcome that effect, again DBMS's do not allow transactions to overwrite values written by non-committed transactions. Lock-based systems use long locks for writes to not allow concurrent updates. Alternatively, in version-based systems new versions are established at commit time. Thus, when a transaction commits its updates overwrite the last committed version. In both cases the DBMS does not allow a transaction to overwrite uncommitted values.

In [14], the authors defined an execution as *strict* if avoids both problems by delaying reads and writes (i.e., using a lock-based approach). We generalise that terminology in the following way:

Definition 17 (Strict history). *A history H is strict if the following conditions hold:*

1. *If $w_i(x_i) < r_j(x_i) \in H$, $c_i \in H$ and $T_i \xrightarrow{wr} T_j \in EMSG(H)$ then $w_i(x_i) < c_i < r_j(x_i) \in H$.*
2. *If $w_i(x_i) < w_j(x_j) \in H$ and $c_i \in H$ then $w_i(x_i) < c_i < w_j(x_j) \in H$.*
3. *If $w_i(x_i) < w_j(x_j) \in H$ and $a_i \in H$ then $w_i(x_i) < a_i < w_j(x_j) \in H$.*

This definition fits with lock-based and version-based concurrency control mechanisms. The first one will delay operations by using locks while the second one will update at commit time the last committed version. Note that PL-1 transactions are allowed to see uncommitted value. Notice only reads from PL-2, PL-SI' and PL-3 transactions are expected to see committed values in strict histories since PL-1 transactions are allowed to see dirty reads.

Chapter 7

Extending EMSG to replicated environments

In this chapter we extend EMSG to evaluate isolation correctness in replicated histories. Firstly, we extend EMSGs to model replicated executions; then, in Section 7.2 we define when a replicated and a stand-alone histories are considered equivalent; independently of the isolation level used by transactions. Finally, in Section 7.3 we identify the conditions under which the transaction history produced by a replication protocol is correct.

7.1 Extending EMSG to replicated systems

A replicated system is composed by a set of nodes N. When a transaction is processed its operations might be executed at several nodes in N. Although EMSGs can be used to separately model and evaluate the execution of each node, they cannot represent a transaction execution history of the whole replicated system. For example, assume two transactions T_i and T_j update the same data item x at nodes N_a and N_b but in a different order. The EMSGs representing the two executions are depicted in Figure 7.1.

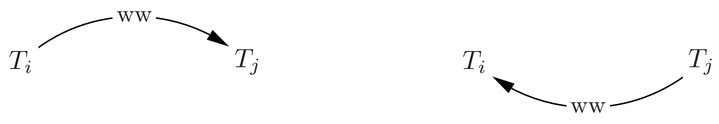

Figure 7.1: Example: N_a and N_b EMSGs

Taken separately each EMSG presents a valid execution history, but unfortunately, the global execution is invalid. The cycle is depicted in Figure 7.2.

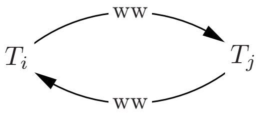

Figure 7.2: Example: global execution

Some authors have proposed to combine serialisation graphs to represent executions in replicated systems. *Lin et al.* [40] defined *union serialisation graphs* (USG) to present 1-copy-SI, a set of conditions to ensure valid SI-only executions. Previously, *Bernstein et al.* [14] used a similar approach to define 1-copy-serialisability. Those solutions are oriented to single isolation-level executions and are not suitable to represent the execution in Figure 7.2 if T_i and T_j have different isolation requirements.

What we suggest is to extend EMSG by combining local EMSGs into a single graph — a *replicated mixed serialisation graph* or RMSG – to admit executions involving transactions with different isolation levels.

Definition 18 (Replicated mixed serialisation graph (RMSG))**.** *Given a replicated history H_r over a set of nodes $N = \{n_0, ..., n_k\}$ and a set of transactions $\mathcal{T} = \{T_0, ..., T_n\}$, $RMSG(H_r) = \bigcup_{a=0}^{k} EMSG(H_r^a)$, being H_r^a the local execution at node N_a and $EMSG(H_r^a)$ its extended mixed serialisation graph.*

Then, for instance, $T_i \xrightarrow{ww} T_j \in RMSG(H_r)$ if $T_i \xrightarrow{ww} T_j$ is in at least one local EMSG.

Now, we refine the latter definition stating when a RMSG is well-formed.

Definition 19 (Well-formed RMSG)**.** *Given a replicated history H_r over a set of transactions $\mathcal{T}$ and the set of local histories $\mathcal{H}$, $RMSG(H_r)$ is well-formed if the following holds:*

(a) Every local history $H_r^a \in \mathcal{H}$ satisfies Def. 16.

(b) $RMSG(H_r)$ does not contain any directed cycle unless it includes two adjacent anti-dependency edges sharing a PL-SI' transaction.

7.2 Equivalence between replicated and stand-alone histories

To the best of our knowledge, all research on equivalence between transaction histories is based on a single isolation level. Conflict and view equivalences were defined for protocols that guarantee serialisability [14]; *SI-equivalence* was proposed in [38]. Hereafter we propose a new definition that is suitable for transaction executions where multiple isolation levels are supported.

Informally, a replicated history is equivalent to a stand-alone history if all the following conditions are held:

- *Uniform writes*: all replicas and the stand-alone system see the same sequence of updates on each database item (however, note that this does not imply that every replica sees exactly the same sequence of database states since updates on different items may be served in different orders in different replicas),
- *Uniform reads*: a read operation obtains the same value in the replica histories and in the stand-alone history,
- *Uniform isolation management*: all replicas and the stand-alone system implement isolation levels in the same way.

The above concepts are formalised as follows:

Definition 20 (Equivalence definition)**.** *A replicated history H_r and a stand-alone history H are equivalent if:*

C1: H and H_r execute the same set of transactions $\mathcal{T}$ and $C(H) = C(H_r)$.

C2: $r_i(x_j) \in C(H)$ iff $r_i(x_j) \in C(H_r)$.

C3: $w_i(x) < w_j(x) \in C(H)$ iff $w_i(x) < w_j(x) \in C(H_r)$.

If H_r and H execute the same set $\mathcal{T}$ of transactions, H_r is equivalent to H iff $C(H_r)$ is equivalent to $C(H)$ since C2 and C3 only consider committed transactions.

C1, C2 and C3 adapt *Bernstein et al.* [14] view-equivalence definition to our context. Actually, our definition is a bit stricter since it expects the same sequence of states in both histories instead of only the same final state. C1 ensures that H and H_r are over the same sets of transactions and commit the same subset. C2 guarantees that every committed transaction read sees the same value in H_r and in H. C3 ensures that both H and H_r see the same sequence of states on every database item.

7.3 Replication protocol correctness

Traditionally, correctness in replication has been associated to 1SR. An execution of a set of transactions in a replicated system is correct if it is equivalent to a serial execution of the same set in a centralised system. This correctness criterion ensures ACID properties to transactions and a reasonable replica consistency level (the exact level depends on the concrete replication strategies used by the replication protocol [51, 62]). However, for performance reasons nowadays commercial DBMSs allow applications to weaken the isolation guarantees for some transactions, usually because some of them are provided by the application logic or because the effects of some phenomena are a smaller problem than the performance impact of avoiding them. Thus, for us a replicated system correctly executes a set of transactions if the result is one-copy equivalent to an execution of the same set in one of those centralised DBMS, considering that transactions may have different isolation needs. Now we will state such a correctness definition based on the equivalence concept introduced previously. Notice that we are not addressing the issue of whether an application can request a weaker isolation level for a given transaction (some other works have dealt with that issue [12]). What we try to define is under which circumstances from the external user point of view a replicated system is behaving like a one-copy system supporting several isolation levels. Thus, we are addressing replication transparency.

Definition 21 defines when a replicated history can be considered valid.

Definition 21 (Valid replicated history)**.** *A replicated history H_r is considered valid if RMSG(H_r) is well-formed.*

Definition 22 states when a valid replicated history H_r should be considered correct.

Definition 22 (Correct replicated history)**.** *A valid replicated history H_r is correct if there is an equivalent valid history H.*

Theorem 2 says that all valid replicated histories are correct.

Theorem 2 (Correctness)**.** *Every valid replicated history H_r has an equivalent valid history H. Therefore, every valid H_r is correct.*

A correctness proof and a set of interesting properties can be found in the appendices.

Part III

Replication protocols

Chapter 8

Supporting multiple isolation levels in replication protocols

Theorem 2 meets one of the main goals of this work by showing when a replication protocol correctly manages isolation when transactions with different isolation requirements are executed concurrently. This part focuses on adding this support to existing replication protocols.

8.1 Protocols classification

There have been several attempts in the past to classify replication protocols [24, 25, 61, 59, 60, 17]. In this thesis we focus on the classification presented in [60].

A replication protocol is composed by a subset of the following atomic steps:

- **Submit**: the user submits a new transaction to a replica, wich becomes its local replica.
- **Forward**: local replica propagates at least the transaction updates. The data sent and its format depend on the concrete replication scheme and transaction. It can contain the set of updated values (write-set or WS), the set of read values (read-set or RS) or even the transaction itself. Notice that we have slightly varied this step original use. In [60] the Forward step represents the entire transaction propagation and is used only in the active replication scheme. We use it in all schemes to represent when the transaction operations are propagated without expecting any specific

format (transaction, WS, RS) and even assuming that reads might not be propagated at all.

- **Processing**: transaction operations are executed on the local database copy.
- **Certification**: the transaction is certified against concurrent transactions to ensure the isolation guarantees.
- **Termination**: replica starts an agreement protocol to consistently decide the transaction's fate.
- **Update**: once received and certified, updates are applied in the local replicas.
- **End**: transaction result is sent back to the user.

Submit, *Forward*, *Processing* and *End* steps are mandatory but the others may appear depending on the concrete replication scheme used. In [60] the authors identify the following replication schemes:

- **Active Replication**: a transaction is initially sent by the user to any of the nodes in the system, which becomes its local replica. The transaction is immediately forwarded to be deterministically processed by all replicas. All replicas send the result back to the user to prevent failures.
- **Certification-Based Replication**: transaction is processed in the local replica and, before committing, its RS and WS are forwarded to be deterministically certified by all replicas. If it passes the certification step, the WS is applied and the transaction committed. Otherwise, it is aborted. The local replica sends the result back to the user.
- **Weak Voting Replication**: similar to Certification-Based Replication but certification is performed only in the local replica and the result propagated in a termination step.
- **Primary Copy Replication**: all transactions are executed by the same replica (primary copy) and the updates forwarded to the other replicas (secondary or backup copies). The primary copy also sends the result back to the user. This one is the simplest replication technique since isolation is fully managed by the primary copy local DBMS.

Lazy Replication it is not considered here since it might violate some basic ACID properties [17, 60].

The replication schemes as presented in [60] were supposed to provide serialisable (PL-3) isolation level. To support other isolation the following steps should be extended:

- **Forward**: transaction isolation level should be included as a part of the propagation message.
- **Processing**: when transaction operations are executed in the replica local copy of the database, it should be provided locally the same isolation level requested globally.
- **Certification**: the certification process should contemplate the isolation level requested by the transaction.
- **Termination**: the termination strategy should also consider the transaction isolation level requested.

From now on, we suggest how the previous replication schemes can be extended to correctly support PL-1, PL-2, PL-SI and PL-3 transactions concurrently. We also prove the correctness of those extensions by showing that any possible execution will ensure Theorem 2.

8.2 Conflict Resolution

In this chapter we describe in detail the main eager serialisable replication schemes (active, certification-based, weak-voting and primary copy) and present modified versions to support several isolation levels. Changes are highlighted to stress the slight differences between both versions. New schemes correctness is also proved by using Theorem 2.

Schemes pretend to show the big picture of managing multiple isolation levels with the main eager replication techniques and that is why they do not go into implementation details. They have been constructed to be simple and easy to understand. Actually, some of the existing protocols include some optimizations which derive in subtle variations of the scheme behaviour. Although those variations have not been included in the schemes due to simplicity, they are commented in the explanations, as footnotes or in the complete example protocols detailed in Chapter 9.

The algorithms presented hereafter assume that every node database local copy is managed by a DBMS which ensures correct and strict executions (see Definition 17 at page 57). If $w_i(x_i) < w_j(x_j)$ then $c_i < w_j(x_j)$ and if $w_i(x_i) < r_j(x_i)$, T_i is PL-2, PL-SI' or PL-3 and T_i commits then $c_i < r_j(x_i)$. So, given a history H, if $T_i \xrightarrow{wr} T_j \in EMSG(H)$ or $T_i \xrightarrow{ww} T_j \in EMSG(H)$ then $c_i < c_j \in H$.

Assuming an item x in the database with a value x_0, a side effect of most centralised DBMS is that, if $r_i(x_0)w_j(x_j)$, T_i requests PL-3 and commits then $r_i(x_0)c_iw_j(x_j)$. The reason depends on the technique used. In lock-based systems, PL-3 is ensured by using long locks for reads and writes [6, 42]. That forbids some phenomena like non-repeatable read (i.e., executions like $r_i(x_0)w_j(x_j)r_i(x_j)$) since ensures that never an item x is overwritten if it has been read or written by an uncommitted transaction. In version-based systems, to ensure PL-3 reads are treated as writes by using `SELECT FOR UPDATE` sentences[22, 45] or similar mechanisms [48, 44]. Summarizing, locks and versions ensure that if $r_i(x_0)w_j(x_j)$ and T_i commits then $r_i(x_0)c_iw_j(x_j)$. Thus, if $T_i \xrightarrow{rw} T_j$ then $c_i < c_j$.

Once a transaction reaches the update step to be committed in a given node, it may be aborted if an integrity constraint is violated. Existing standalone DBMS check integrity constrains during the execution and, hence, during the processing step. In this case that will not affect the following schemes since the transaction is aborted in the local replica before propagation in the case of certification, weak-voting and primary-copy schemes and deterministically in all replicas in the active replication scheme. However, some protocols proposed in literature [37, 15, 41] suggest to perform those checks at commit time. If that happens, transactions may abort once they have been validated by replicas and are going to be persisted in the local DBMS [50, 40]. That would be easily supported in the schemes presented since all nodes will deterministically abort or commit the transaction but it will make the proofs slightly more complex. In order to make them more readable, we assume that once a scheme decides to commit a transaction during the validation step then this transaction is never aborted by the local DBMS during the update state.

Finally, some schemes are based on a virtual clock to calculate start and commit timestamps for transactions executed in the system. When a reliable total order broadcast primitive is used, those clocks are usually implemented by counting committed or validated update transactions (counting committed transactions can be used only if transactions commit in delivery order). In this section we do not use any specific implementation but we do assume a transformation function *clock* with the following properties. Given a transaction T_i and any possible replicated history H_r generated by the scheme:

- All nodes involved in T_i's execution share the same value for $clock(s_i)$ and $clock(c_i)$. $clock(s_i)$ is usually calculated at the local replica and propagated to the rest of the nodes while $clock(c_i)$ is deterministically calculated at every node.
- *clock* is a time-precedes order over $C(H_r)$ and, hence:
  - for any transaction T_i, $clock(s_i) < clock(c_i)$ and
  - given T_i, T_j transactions committed in H_r, $clock(c_i) < clock(s_j)$ or $clock(s_j) < clock(c_i)$.
- Given T_i and T_j two committed and conflicting transactions, if $c_i < c_j \in H_r$ then $clock(c_i) < clock(c_j)$.

To simplify the notation, we define as $st_i = clock(s_i)$ the start-timestamp of T_i and $ct_i = clock(c_i)$ as the commit-timestamp.

8.2.1 Active replication scheme

An active replication protocol propagates every transaction to all replicas using an atomic broadcast. Once delivered, transactions are deterministically executed to ensure consistency.

In this context, *deterministically* means that all replicas resolve conflicts in the same way. Expressed in terms of dependencies, if two conflicting operations o_1 and o_2 are executed in replicas N_a and N_b and produce edges e_a in $EMSG(H_a)$ and e_b in $EMSG(H_b)$, then $e_a = e_b$. If a protocol propagates all transactions and executes them deterministically in all replicas, then $EMSG(N_a) = EMSG(N_b)$. Two possible mechanisms to achieve determinism are to request all necessary locks in an atomic step before any transaction operation is executed [35] and to propagate every operation separately [20].

A general PL-3 replication schema is depicted here:

1. *Submit*: a replica N_i receives a transaction T_i from client C_i.
2. *Forward*: T_i is atomically broadcast to all replicas.
3. *Processing*: when T_i is delivered at a replica N_a:
   - T_i operations are executed deterministically.
4. *End*: once T_i operations have been completely executed in N_a:
   - T_i is committed.
   - If $N_i = N_a$, the transaction result is sent back to C_i.

Some active protocols propagate only update operations [54]. For the sake of simplicity and according to [60], we assume all transactions are propagated and all operations are executed by all replicas in the system. This approach may be used to provide fault tolerance even in the presence of byzantine errors [47].

The previous schema represents active replication as presented in [60] and supports only PL-3 isolated transactions. Because it is active replication, all transactions are executed in all replicas which limits the system scale-out capabilities [24]. Since all transactions require PL-3 isolation, all conflicts among operations of distinct transactions are treated as obligatory edges and hence make replicas EMSG and the global RMSG more dense. Determinism implies ordering among operations in obligatory edges and ordering can either be ensured using locks (operations wait until all previous ordered operations are executed), which decreases concurrency and increases execution time, or optimism (operations are optimistically executed without wait and ordering violations are solved aborting affected transactions), which increases abort rate. Allowing weaker isolation levels does not solve the intrinsic problem of executing everything everywhere (i.e., scale-out limit) but reduces locking/aborting rate and the number of ordering restrictions to be checked or managed.

Active replication scheme does not need to be changed to support multiple isolation levels. We only must be sure that every replica executes every transaction deterministically with the specified isolation level. That is automatically achieved if every transaction explicitly establishes its isolation level with a `SET TRANSACTION ISOLATION LEVEL` sentence or all nodes use the same level by default. In other cases, the protocol might have to include the isolation level in every transaction propagation message and be sure that it is notified to the local DBMS when transactions are going to be locally executed.

Since all replicas execute deterministically and in the same order the same set of transactions and local DBMS are supposed to be valid and strict, the resulting replicated executions are trivially correct. Notice that $EMSG(H^a) = RMSG(H_r)$ for any node N_a. More formally:

Theorem 3 (Active scheme correctness)**.** *Any possible history H_r produced by an active-based protocol is correct.*

Proof. Assume an active protocol P executes a set of transactions $\mathcal{T}$ in a system composed by $\mathcal{N}$ nodes. H_r represents the global execution produced in the system and $\mathcal{H}$ is the set of local histories representing the executions produced in every node in $\mathcal{N}$. Since active protocols execute all transactions entirely and deterministically in all replicas, for any two histories $H^a, H^b \in \mathcal{H}$, $EMSG(H_a) = EMSG(H_b)$. From Definition 18, $RMSG(H_r) = \bigcup_{H_a \in \mathcal{H}} EMSG(H_a)$. Thus, $\forall H^a \in \mathcal{H}$, $RMSG(H_r) = EMSG(H_a)$. Since nodes are supposed to produce correct local histories, H_r should be correct also, i.e., $RMSG(H_r)$ is well-formed as we prove below:

- **Every local history $H_a \in \mathcal{H}$ satisfies Def. 16**. Active replication is based on the assumption that every local DBMS produces correct local histories. Thus, this condition is ensured by definition of active replication.
- $RMSG(H_r)$ **does not contain any forbidden cycle**. Since every node is assumed to produce correct local histories, EMSGs do not have forbidden cycles. Since RMSG has exactly the same edges and vertices than any EMSG, it will not have any cycle forbidden either.

□

8.2.2 Certification-based replication scheme

In a certification-based replication protocol, a transaction is initially executed in its local replica. Once it requests to commit, the write-set is gathered and broadcast to the entire set of replicas. When it is delivered in a replica, passes a certification step to check if conflicting transactions have been executed elsewhere. If it succeeds, the updates are applied and the transaction committed. Otherwise, the write-set is discarded and the transaction aborted. During the certification step, no communication is performed with other nodes. Read-only transactions are entirely executed and committed in the local replica without being broadcast. The original scheme for PL-3 transactions is depicted here:

1. *Submit*: when a replica N_i receives a transaction T_i from a client C_i:

   - T_i is started at the underlying DBMS with PL-3 isolation level.

2. *Processing*: when replica N_i is ready to execute T_i:

   - T_i's operations are executed until commit is requested[1]. A read operation over item x will observe at least the last update of x performed by a transaction T_j with the biggest commit timestamp such that $ct_j < st_i$.

3. *Forward*: when C_i requests to commit T_i:

   - T_i's writes and reads are collected into WS_i and RS_i.
   - If $WS_i = \emptyset$, T_i is committed.
   - If $WS_i \neq \emptyset$, WS_i and RS_i are atomically broadcast to all replicas.

4. *Validation*: when WS_i is delivered at a replica N_a:

   - WS_i is discarded if there is a committed transaction T_j such that $RS_i \cap WS_j \neq \emptyset$ and $st_i < ct_j$.

5. *Update*: if T_i terminates the validation step at N_a:

   - If $N_i = N_a$, T_i is aborted if WS_i is discarded during the validation step. Otherwise, it is committed, the updates are persisted and become available at N_a to other transactions executed at this node. Notice that local transaction writes are persisted in committing order, not in execution order. Fortunately, all known concurrency control mechanisms ensure such a restriction.
   - If T_i passes the validation step and $N_i \neq N_a$, WS_i is applied. The implementation of the protocol must ensure that WS_i is not aborted by the replica's DBMS.

6. *End*: if $N_a = N_i$, T_i's result is sent back to C_i.

[1] Some protocols [32] delay writes until the termination step.

The previous scheme contemplates only PL-3 transactions. Once a transaction commits in a given replica, the scheme assumes that its updates are available to any transaction executed at that replica.

From now on, we modify the *Submit*, *Forward* and *Validation* steps to also support PL-1, PL-2 and SI isolation levels. We also show how Theorem 2 can be used to prove the correctness of the new scheme.

Multiple Isolation Levels Certification-Based Scheme or MCBS:

1. *Submit*: when a replica N_i receives a transaction T_i **and its isolation level** from a client C_i:
   - T_i is started at the underlying DBMS **with the requested isolation level**.
2. *Processing*: when replica N_i is ready to execute T_i:
   - T_i's operations are executed until commit is requested[2]. **In PL-2 and PL-3**, a read operation over item x will observe at least the last update of x performed by a transaction T_j with the biggest commit timestamp such that $ct_j < st_i$. **For PL-1 transactions, read dependencies are not obligatory and so there are no restrictions on reads. If T_i is PL-SI', T_i's snapshot includes the updates performed by the last available transactions** or, in other words, if T_i sees or overwrites x_k version of item x then $ct_k < st_i$. For any newer version x_l, $st_i < ct_l$.
3. *Forward*: when C_i requests to commit T_i:
   - T_i's writes are collected into WS_i. **For PL-3 transactions**, reads are also gathered into RS_i.
   - If $WS_i = \emptyset$, T_i is committed.
   - If $WS_i \neq \emptyset$, WS_i, RS_i **(only for PL-3 transactions)** and T_i**'s isolation level** are atomically broadcast to all replicas.
4. *Validation*: when WS_i is delivered at a replica N_a:
   - **If T_i is PL-3**, WS_i is discarded if there is a committed transaction T_j such that $RS_i \cap WS_j \neq \emptyset$ and $st_i < ct_j$.
   - **If T_i is PL-SI', WS_i is discarded if there is a committed transaction T_j such that $WS_i \cap WS_j \neq \emptyset$ and $st_i < ct_j$.**
   - **PL-1 and PL-2 do not need to be validated**.
5. *Update*: if T_i terminates the validation step in N_a:

[2] Some protocols [32] delay writes until the termination step.

  - If $N_i = N_a$, T_i is aborted if WS_i is discarded during the validation step. Otherwise, it is committed, the updates are persisted and become available to other transactions executed at N_a. Notice that local transaction writes are persisted in committing order, not in execution order. Fortunately, all known concurrency control mechanisms ensure such a restriction.
  - If T_i passes the validation step and $N_i \neq N_a$, WS_i is applied. The implementation of the protocol must ensure that WS_i is not aborted by the replica's DBMS.

6. *End*: if $N_a = N_i$, T_i's result is sent back to C_i.

Notice that validation, update and end steps are executed as a single atomic step. To increase the performance of the system, some existing protocols [32, 38, 5] execute validation and termination separately, allowing validation even if the previous write-sets are not yet applied.

Given a PL-SI' transaction T_i, the algorithm also assumes that T_i's snapshot includes any transaction T_j such that $ct_j < st_i$. That actually means that T_i's snapshot gets the last committed state existing at T_i's local replica. Existing centralised DBMS fit that restriction if the number of validated or committed transactions at the local replica is used to calculate the start timestamp.

Below we first prove that all replicas apply write-sets in the same order. Then, we show that the conditions of Definition 21 are satisfied.

Lemma 5 (MCBS applies write-sets in the same order)**.** *Given a replicated history H_r over a set of transactions $\mathcal{T}$ and a set of nodes $\mathcal{N}$.*

- *For any two replicas $N_a, N_b \in \mathcal{N}$ that apply WS_i and WS_j, they apply WS_i and WS_j in the same order.*
- *If a replica $N_a \in \mathcal{N}$ applies WS_i then every other replica $N_b \in \mathcal{N}$ applies WS_i.*

Proof. Write-sets are propagated using an atomic broadcast and, hence, are delivered to all replicas in the same order. Validation, update and end are executed in a single atomic step and so all the write-sets are applied in order of their delivery, and this fact proves the first part of Lemma 7.

Assume WS_i is going to be validated. We prove by induction over the number n of previously committed update transactions that the second part of the Lemma holds once WS_i is applied.

- **Base case** ($n = 0$): no update transaction has been previously delivered and committed and, thus, T_i commits at N_a and N_b.

- **Induction hypothesis** ($n < l$): the lemma holds when T_i is applied if less than l update transactions committed.

- **Induction step** ($n = l$): l update transactions committed previously. From the induction hypothesis, when the last update transaction committed the lemma was not violated and, thus, all N_a and N_b applied the same set $\mathcal{U}$ of update transactions and in the same order. When T_i is applied the behaviour of MCBS varies depending on the isolation level:

  – *T_i is PL-1 or PL-2*: If T_i requests a PL-1 or PL-2 isolation level it is directly validated and applied in both replicas.

  – *T_i is SI*: if T_i is PL-SI', since both nodes observe the same set $\mathcal{U}$ of previously committed transactions when T_i is going to be validated, the validation result will be different in both replicas if there is a committed transaction $T_j \in \mathcal{U}$ for which $WS_i \cap WS_j \neq \emptyset$ and $st_i < ct_j$ in N_a; but $ct_j < st_i$ at N_b. Since we assume both nodes share the same values for st_i and ct_j T_i's will get the same validation result.

  – *T_i is PL-3*: the same proof can be applied for PL-3 transactions. The only difference is that RS_i is used instead of WS_i. However, in both cases, it is compared against previously applied write-sets.

Therefore, T_i never obtains a different validation result at N_a and N_b and this proves the Lemma's second assertion. □

Theorem 4 (MCBS protocols are correct)**.** *Any history H_r produced by a MCBS protocol is correct.*

Proof. H_r is correct if $RMSG(H_r)$ is well-formed (see Def. 21 and Theorem 2). To begin with, we prove each condition of Definition 19 separately:

- **Every local history $H_a \in \mathcal{H}$ satisfies Def. 16**. MCBS is based on the assumption that every local DBMS produces correct local histories. Thus, this condition is ensured by definition of MCBS.

- $RMSG(H_r)$ **does not contain any forbidden cycle**. Only the cycles with two consecutive anti-dependency edges sharing a PL-SI' transaction are allowed. $RMSG(H_r)$ only includes obligatory edges and an edge e is obligatory if:

  – $e = T_i \xrightarrow{ww} T_j$: since we assume strict histories and update transactions are applied in the same order at all nodes, $c_i < c_j \in H_r$ and $ct_i < ct_j$. If T_j is PL-SI' then also $ct_i < st_j$ since otherwise is aborted in the validation step.

- $e = T_i \xrightarrow{wr} T_j$ and T_j is PL-2, PL-SI' or PL-3: since we only allow strict histories then $c_i < c_j$ in T_j's local node. The total order broadcast ensures that $c_i < c_j$ in the other nodes if T_j is an update transaction and, otherwise, T_j is executed only at the local node. Consequently, $c_i < c_j \in H_r$ and, hence, $ct_i < ct_j$. If T_j is PL-SI' MCBS also assumes that every read gets the value written by the transaction T_i with biggest commit timestamp such that $ct_i < st_j$.
- $T_i \xrightarrow{rw} T_j$ and T_i is PL-SI' or PL-3: if T_i is PL-SI' and reads a value x_k from its snapshot, $ct_k < st_i$ and there is no other transaction T_l in that snapshot which updates x and $ct_k < ct_l$. Then, $st_i < ct_j$. If T_i is PL-3 then, by assumption, $c_i < c_j$ (see this chapter introduction) in T_i's local node. Recall that, if T_i is an update transaction and is propagated, the total order delivery and the atomicity in validation and update steps ensure that all nodes commit the writesets in the same order. Thus, $c_i < c_j \in H_r$ and the virtual clock ensures that $ct_i < ct_j$.

By absurd reduction, assume there is a cycle $C = \{V, E\} = T_1 e_1 T_2 e_2 ... e_n T_1 \in RMSG(H_r)$ without two consecutive anti-dependency edges sharing a PL-SI' transaction. We differentiate two possible situations:

- C does not include any anti-dependency edge starting with a PL-SI' transaction. In that case, for any edge $e = T_i \longrightarrow T_j \in E$, $ct_i < ct_j$ and, hence, for every vertex $T_i \in V$, $ct_i < ct_i$ which is a contradiction.
- C includes anti-dependencies starting with a PL-SI' transaction. Assume $e = T_i \xrightarrow{rw} T_j \in E$ is one of those anti-dependency edges. Then $st_i < ct_j$. Since C does not contain two consecutive anti-dependencies sharing a PL-SI transaction, the previous edge e' in C is a dependency edge. If $e' = T_k \xrightarrow{ww/wr} T_i$ then $ct_k < st_i$ because T_i is PL-SI. Then, $ct_k < ct_j$. For any other edge $T_a \longrightarrow T_b \in C$, $ct_a < ct_b$ and, thus, excluding PL-SI' transactions starting an anti-dependency edge in C, $ct_i < ct_i$ for any transaction $T_i \in C$, which is a contradiction. More formally, for every path $P \subseteq C$ such that $P = T_i e'_p T_p^{si} e_p T_j$, T_p^{si} is a PL-SI' transaction and e_p is an anti-dependency edge, we say that P is a virtual path of C, e'_p is the dependency edge in P, e_p is the anti-dependency edge. Since $ct_k < ct_j$, $e_P^v = T_i \longrightarrow T_j$ is the virtual start-dependency edge of P. Given $\mathcal{P}$ the set of virtual paths in C, we define $E_p = \bigcup_{P \in \mathcal{P}} e_P^v$, $E_{SI} = \bigcup_{P \in \mathcal{P}} e_p \cup \bigcup_{P \in \mathcal{P}} e'_p$ and $V_{SI} = \bigcup_{P \in \mathcal{P}} e_P^{si}$. We can define a new cycle $C_v = \{V_v = V - V_{SI}, E_v = (E - E_{SI}) \cup E_p\}$. Notice that for any edge $T_i \xrightarrow{v} T_j \in E_v$, $ct_k < ct_j$ and, then, for every vertex $T_i \in C_v$, $ct_i < ct_i$ which is a contradiction.

Concluding, $RMSG(H_r)$ does not include cycles without two consecutive anti-dependency edges sharing a PL-SI' transaction.

As a result, every replicated history H_r generated by a MCBS protocol is valid. Hence, Theorem 2 is applicable to H_r and the valid replicated history is correct. □

8.2.3 Weak voting replication scheme

As with a certification-based scheme, every transaction is initially executed in its local replica. However, once it requests to commit, a weak voting protocol only propagates the write-set. When it is delivered, the local replica validates and broadcasts the result. The other replicas hold the write-set until the validation result message arrives. At that moment, the write-set is committed or discarded depending on the result type (commit or abort). The original scheme for PL-3 transactions is outlined here:

1. *Submit*: when a replica N_i receives a transaction T_i from a client C_i:
   - T_i is started at the underlying DBMS with PL-3 isolation level.
2. *Processing*: when replica N_i is ready to execute T_i:
   - T_i operations are executed until commit is requested[3]. A read operation over item x will observe at least the last update of x performed by a transaction T_j with biggest commit timestamp such that $ct_j < st_i$.
3. *Forward*: when C_i requests to commit T_i:
   - T_i's writes and reads are collected into WS_i and RS_i.
   - If $WS_i = \emptyset$, T_i is committed.
   - If $WS_i \neq \emptyset$, WS_i is atomically broadcast to all replicas (notice that RS_i is not propagated).
4. *Validation*: when WS_i is delivered at a replica N_a:
   - If $N_a = N_i$ and an abort message has been previously sent for T_i, WS_i is discarded. Otherwise the validation continues.
   - If $N_a = N_i$ and T_i has not been previously aborted, WS_i is discarded if a transaction T_j has been previously validated, $RS_i \cap WS_j \neq \emptyset$ and $st_i < ct_j$. The result is propagated.
5. *Termination*: if T_i terminates the validation step in N_a:
   - Wait for the validation result delivery.

[3] As with certification-based protocols, some protocols delay writes until the termination step.

6. *Update*: when T_i's validation result is delivered at N_a:
    - If $N_i = N_a$, T_i is aborted if WS_i is discarded during the validation step. Otherwise, it is committed, the updates are persisted and become available at N_a to other transactions executed at this node. Notice that local transaction writes are persisted in committing order, not in execution order. Fortunately, all known concurrency control mechanisms ensure such a restriction.
    - If T_i passes the validation step and $N_i \neq N_a$, WS_i is applied. The implementation of the protocol must ensure that WS_i is not aborted by the replica's DBMS. If T_i is aborted then WS_i is discarded.
7. *End*: if $N_a = N_i$, T_i's result is sent back to C_i.

As suggested in Chapter 8, we modify the *Submit*, *Forward* and *Validation* steps of the original scheme to also support PL-1, PL-2 and SI isolation levels. We also show how Theorem 2 can be used to prove the correctness of the new scheme which we call Multiple Weak Voting Scheme or MWVS:

1. *Submit*: when a replica N_i receives a transaction T_i **and its isolation level** from a client C_i:
    - T_i is started at the underlying DBMS **with the requested isolation level**.
2. *Processing*: when replica N_i is ready to execute T_i:
    - T_i operations are executed until commit is requested. For **PL-3 and PL-2 transactions**, a read operation over item x will observe at least the last update of x performed by a transaction T_j with biggest commit timestamp such that $ct_j < st_i$. **For PL-1 transactions, read dependencies are not obligatory and so there are no restrictions on reads. If T_i is PL-SI', T_i's snapshot includes the updates performed by the last available transactions** or, in other words, if T_i's sees or overwrites x_k version of item x then $ct_k < st_i$. For any newer version x_l, $st_i < ct_l$.
3. *Forward*: when C_i requests to commit T_i:
    - T_i's writes and, **only for PL-3 transactions**, also reads are collected into WS_i and RS_i.
    - If $WS_i = \emptyset$, T_i is committed.
    - If $WS_i \neq \emptyset$, WS_i and T_i **isolation level** are atomically broadcast to all replicas (notice that RS_i is not propagated).
4. *Validation*: when WS_i is delivered at a replica N_a:

- If $N_a = N_i$ and an abort message has been previously sent for T_i, WS_i is discarded. Otherwise the validation continues.
- If T_i **is PL-3**, $N_a = N_i$ and T_i has not been previously aborted, WS_i is discarded if a transaction T_j has been previously validated, $RS_i \cap WS_j \neq \emptyset$ and $st_i < ct_j$. The result is propagated.
- **If T_i is SI, WS_i is discarded if there is a committed transaction T_j, that has already been validated, $WS_i \cap WS_j \neq \emptyset$ and** $st_i < ct_j$.
- **PL-1 and PL-2 do not need to be validated**.

5. *Termination*: if T_i terminates the validation step in N_a:

   - **If T_i is PL-3**, wait for the validation result delivery.

6. *Update*: when T_i's validation result is available at N_a:

   - If $N_i = N_a$, T_i is aborted if WS_i is discarded during the validation step. Otherwise, it is committed, the updates persisted and become available at other transactions executed at N_a. Notice that local transaction writes are persisted in committing order, not in execution order. Fortunately, all known concurrency control mechanisms ensure such a restriction.
   - If T_i passes the validation step and $N_i \neq N_a$, WS_i is applied. The implementation of the protocol must ensure that WS_i is not aborted by the replica's DBMS. If T_i is aborted then WS_i is discarded.

7. *End*: if $N_a = N_i$, T_i's result is sent back to C_i.

Like in MCBS, Validation, Termination, Update and End steps are executed as a single atomic step.

The correctness proof of MWVS follows a similar approach to the one for MCBS. We first prove that all replicas apply write-sets in the same order, and then we show that the conditions of Definition 21 are satisfied.

Lemma 6 (MWVS applies write-sets in the same order)**.** *Given a replicated history H_r over a set of transactions $\mathcal{T}$ and a set of nodes $\mathcal{N}$.*

- *for any two replicas $N_a, N_b \in \mathcal{N}$ that apply WS_i and WS_j, they apply WS_i and WS_j in the same order.*
- *if a replica $N_a \in \mathcal{N}$ applies WS_i then every other replica $N_b \in \mathcal{N}$ applies WS_i.*

Proof. The correctness proof is almost identical to the proof of Lemma 7. The only difference is related on how PL-3 transactions are validated in both schemes and affects the correctness of second part of the Lemma. In this case, the validation is performed only at T_i's and the result propagated to the rest of the replicas. Thus, the termination result is trivially the same in all of them. □

Theorem 5 (MWVS protocols are correct)**.** *Any possible history H_r that can be produced by a MWVS protocol is correct.*

Proof. The proof is identical to the correctnes proof of Theorem 4. □

8.2.4 Primary copy replication scheme

In a pure primary copy replication protocol, all transactions are completely executed by the same local replica. Before committing, updates are propagated to the rest of the replicas, known as secondary or backup, using a reliable broadcast and applied in the same order to preserve consistency. Since the sender is always the primary replica, order can be easily ensured by numbering update messages and taking care replicas apply them in order.

Unfortunately, the performance is limited by the primary replica and, hence, primary copy protocols do not scale well. To improve scalability, most protocols execute read-only transactions at backup nodes while the primary replica focuses on update transactions. This approach may work for small/medium read-only intensive systems with short update transactions but not in the general case [60].

A general PL-3 primary copy replication schema is depicted here:

1. *Submit*: when a replica N_i receives a transaction T_i from a client C_i:
    - If N_i is not the primary replica N_p and T_i is an update transaction, N_i redirects C_i's request to N_p which becomes T_i's local node.
2. *Processing*: when replica N_i is ready to execute T_i:
    - T_i operations are executed until commit is requested. A read operation over item x will observe at least the last update of x performed by a transaction T_j with biggest commit timestamp such that $ct_j < st_i$.
3. *Forward*: when C_i requests to commit T_i:
    - T_i writes are collected into WS_i.
    - If $WS_i = \emptyset$, T_i is committed.
    - If $WS_i \neq \emptyset$, WS_i is forwarded using a reliable FIFO broadcast.
4. *Update*: when WS_i is delivered at N_a.

- If $N_a = N_p$, T_i is committed.
- If $N_a \neq N_p$, WS_i is applied. The implementation of the protocol must ensure that WS_i is not aborted by the replica's DBMS.

5. *End*: if $N_a = N_i$ the result is sent back to C_i.

It is straightforward to provide multiple isolation levels in these protocols since concurrency is fully handled by the primary replica. As with active replication, the original scheme can be used without modifications if all replicas share the same default isolation level and a `SET TRANSACTION ISOLATION LEVEL` sentence is used when transactions have other isolation requirements. Otherwise, the replication protocol only extra concern is to communicate to local replica the isolation level requested by the client for every transaction. For update transactions the primary node will be the local replica but read-only transactions might be executed by any replica.

PL-SI' transactions are assumed to see the last snapshot available considering the system virtual clock. Thus, if $T_i \xrightarrow{ww/wr} T_j$ and T_j is PL-SI' then $ct_i < st_j$. If $T_i \xrightarrow{rw} T_j$ then $st_i < ct_j$.

In the primary copy scheme, isolation management is almost entirely delegated to the primary replica concurrency protocol. Assuming N_p is the primary replica and H^p the resulting EMSG when a set of transactions $\mathcal{T}$ is executed, H^p is supposed to be valid and strict. Since secondary replicas only execute read-only transactions and writesets in delivery order, the resulting H_r will be also correct. That is described in detail in Theorem 6 proof but, first, we will prove that for any two update transactions, its writes are applied in the same order in all replicas.

Lemma 7 (Primary Copy Scheme applies write-sets in the same order)**.** *Given a replicated history H_r over a set of transactions $\mathcal{T}$ and a set of nodes $\mathcal{N}$.*

- *for any two replicas $N_a, N_b \in \mathcal{N}$ that apply WS_i and WS_j, they apply WS_i and WS_j in the same order.*
- *if a replica $N_a \in \mathcal{N}$ applies WS_i then every other replica $N_b \in \mathcal{N}$ applies WS_i.*

Proof. All update transactions are executed at the primary replica N_p, which is supposed to be valid and strict. Once a transaction T_i is going to commit, WS_i is propagated using a FIFO reliable broadcast and applied at all secondary nodes in the same order. Thus, all nodes apply updates also in delivery order. Concluding, all nodes apply the same updates and in the same order.

□

Theorem 6 (Theorem 2 conditions are satisfied)**.** *Any possible history H_r produced by a Primary copy based protocol is correct.*

Proof. H_r is correct if $RMSG(H_r)$ is well-formed (see Def. 21 and Theorem 2). To begin with, we prove each condition of Definition 19 separately:

- **Every local history $H_a \in \mathcal{H}$ satisfies Def. 16**. Primary copy is based on the assumption that every local DBMS produces correct local histories. Thus, this condition is ensured by definition of the primary copy scheme.
- $RMSG(H_r)$ **does not contain any forbidden cycle**. Only the cycles with two consecutive anti-dependency edges sharing a PL-SI' transaction are allowed. We first prove that $c_i < c_j$ for any edge $e \in RMSG(H_r)$ unless e is an anti-dependency edge starting with a PL-SI' transaction. Next we prove that $RMSG(H_r)$ is correct. $RMSG(H_r)$ only includes obligatory edges. An edge e is obligatory if:
  - $e = T_i \xrightarrow{ww} T_j$: since we assume strict local histories and writes are applied in the same order at all replicas (see Lemma 7), $c_i < c_j$ at all replicas and, hence, $c_i < c_j \in H_r$ and $ct_i < ct_j$. If T_i is PL-SI' then the protocol also ensures that $ct_i < st_j$.
  - $e = T_i \xrightarrow{wr} T_j$ and T_j is PL-2, PL-SI' or PL-3. Since we assume strict local histories, $c_i < c_j$ at T_j's local node. If T_j is a read-only transaction then it is executed only at T_j, $c_i < c_j \in H_r$ and $ct_i < ct_j$. If it is an update-transaction then all nodes execute the writes in the same order $c_i < c_j \in H_r$ anyway and $ct_i < ct_j$. If T_j requested PL-SI' then the replication protocol ensures $ct_i < st_j$.
  - $e = T_i \xrightarrow{rw} T_j$ and T_i is PL-SI' or PL-3. Since we assume strict histories, if T_i is PL-3 then $c_i < c_j$ in T_i's local node. As with read-dependencies, if T_i is read only then $ct_i < ct_j \in H_r$ since T_i is executed only at its local node. If it is an update-transaction then $c_i < c_j \in H_r$ because all nodes execute and commit the same writesets in the same order and, hence, $ct_i < ct_j$ anyway. However, if T_i is PL-SI' we only can say that $s_i < c_j$ at T_i's local node. If T_i is read-only then the protocol ensures that $st_i < ct_j$.

By contradiction, assume $RMSG(H_r)$ shows a cycle without two consecutive anti-dependency edges sharing a PL-SI' transaction. Then, for any edge $e = T_i \xrightarrow{rw} T_j$ such that T_i is PL-SI', the previous edge in the cycle is a dependency edge, say $e' = T_k \xrightarrow{ww/wr} T_i$, and $ct_k < st_i < ct_j$. For any other edge $e'' = T_n \longrightarrow T_{n+1}$ in the cycle $ct_n < ct_{n+1}$, since it is not an anti-dependency starting with a PL-SI' transaction. Then, if there is such a cycle we will get a contradiction $ct_n < ct_n$ and, thus, those cycles do not appear in $RMSG(H_r)$.

□

Chapter 9

Examples

In this chapter we show how the changes suggested in Chapter 8.2 can be applied to some existing protocols. We focus on weak voting and certification-based algorithms because in active and primary copy protocols isolation is fully managed by local DBMS and do not need any extra changes except including the isolation level in the transaction propagation message.

9.1 SER-CBR

In this section we show how the changes suggested in Section 8.2.2 can be applied to an existing certification-based protocol. We have taken SER CBR, a variation of the SI CBR protocol from [50] (also an adaptation from [33, 46]) which supports only PL-3, and extended it to also support SI, PL-2 and PL-1. We call the new protocol *multiple isolation level certification-based replication protocol* or MUL CBR. SER CBR and SI CBR are combinations and variations of other well known protocols [32, 19, 38, 17] and the changes suggested in this chapter can be easily exported to those protocols.

SER CBR (see Figure 9.1) needs transaction read-sets in order to validate transactions (see its `certify` method). The first difference revealed by MUL CBR (see Figure 9.2) is that it includes transaction isolation level as a part of the broadcast message (line 14). The second difference is that read-set propagation is used only for PL-3 transactions (lines 15-16). In contrast to SER CBR, the `certify` method of MUL CBR validates transactions depending on their isolation level. PL-2 and PL-1 transactions do not require validation: atomic broadcast together with local DBMS concurrency control are sufficient to forbid the phenomena. PL-SI' transactions only require checks for conflicts between writes. Thus, reads are only involved in PL-3 transactions validation and, hence, only in those cases the readset should be propagated. Notice that only a few changes are required to the original protocol.

```
Init at N$_a$
    count ← 0
Upon t received at N$_a$
    t.start ← count
    Execute t.
On t commit request at N$_a$
    ws.data ← wset(t)
    ws.start ← t.start
    ws.local ← $N_a$
    if (ws.data = ∅) then
        t.commit ← count
        send(c,COMMIT)
    else
        rs.data ← rset(t)
        TO-bcast(N, ⟨rs, ws⟩)
certify(rs, ws, wslist$_a$)
    for (old_ws ∈ wslist$_a$) do
        if (ws.start ⟨ old_ws.commit) and
        (rs.data ∩ old_ws.data ≠ ∅) and
        (ws.local ≠ old_ws.local) then
            return ABORT
    return COMMIT
Upon ⟨rs, ws⟩ reception at N$_a$
    mutex.lock
    status$_t$ ← certify(rs, ws, wslist$_a$)
    if (status$_t$ = COMMIT) then
        count ← count +1
        ws.commit ← count
        if (ws.local ≠ $N_a$) then
            DB.apply(ws)
        status$_t$ ← DB.commit(t)
        if (status$_t$ = COMMIT) then
            append(wslist$_a$, ws)
    else DB.abort(t)
    mutex.unlock
    if (ws.local = $N_a$) then
        send(c, status$_t$)
```

Figure 9.1: SER CBR certification-based protocol.

MUL CBR fits perfectly with the MCBS scheme in Chapter 8.2.2 (page 70). There is only one significant variation related on how conflicts between writesets and local unvalidated transactions are resolved. This protocol assumes that a certified writeset might be aborted by the local DBMS after certification due to some integrity constraint. This process is assumed to be deterministic and, thus, all local DBMS make the same decision about a given writeset. That may increase the number of aborted transactions since some certified transactions may abort. Then, RMSG histories may show less edges compared to the original scheme but the scheme correctness can be applied anyway in this case.

To show that MUL CBR is correct we only need to prove that the virtual clock based on counting committed updated transactions is consistent with the assumptions made at the start of Chapter 8.2. Unfortunatelly, the partial order associated to the timestamp calculation method used ni MUL CBR is not a time-precedes order since, for example, $st_i = ct_i$ if T_i is a read-only transaction and no other transaction commits in T_i's local replica while T_i is being executed. However, it is possible to define the following transformation function f such that the resulting transformed timestamps met with the assumptions made in Chapter 8.2.

Definition 23 (Transformation funcion)**.** *Given a transaction T_i, the transformation function f is defined as follows:*

$$f(st_i) = st_i + 0.1$$

```
Init at N$_a$
  count ← 0
Upon t received at N$_a$
  t.start ← count
  Execute t.
On t commit request at N$_a$
  ws.data ← wset(t)
  ws.start ← t.start
  ws.local ← $N_a$
  if (ws.data = $\emptyset$) then
    t.commit ← count
    send(c,COMMIT)
  else
    ws.level ← t.level
    if (t.level = PL-3)
      rs.data ← rset(t)
    else
      rs.data ← 0
    TO-bcast(N, ⟨rs, ws⟩)
certify(rs, ws, wslist$_a$)
  for (old_ws ∈ wslist$_a$) do
    if (level = PL-3) and
      (ws.start ⟨ old_ws.commit) and
      (rs.data ∩ old_ws.data ≠ $\emptyset$) and
      (ws.local ≠ old_ws.local) then
        return abort
    else if (level = SI) and
      (ws.start ⟨ old_ws.commit) and
      (ws.data ∩ old_ws.data ≠ $\emptyset$) then
      return abort
  return commit
Upon ⟨rs, ws⟩ reception at N$_a$
  mutex.lock
  status$_t$ ← certify(rs, ws, wslist$_a$)
  if (status$_t$ = commit) then
    count ← count +1
    ws.commit ← count
    if (ws.local ≠ $N_a$) then
      DB.apply(ws)
    status$_t$ ← DB.commit(t)
    if (status$_t$ = COMMIT) then
      append(wslist$_a$, ws)
  else DB.abort(t)
  mutex.unlock
  if (ws.local = $N_a$) then
    send(c, status$_t$)
```

Figure 9.2: MUL certification-based protocol.

$$f(ct_i) = \begin{cases} ct_i + 0.2 & \textit{if } T_i \textit{ is read-only} \\ ct_i & \textit{if } T_i \textit{ is update} \end{cases}$$

We prove now that, given a history MUL CBR H_r over a set $\mathcal{T}$ of transactions, H_r fulfils every of the Chapter 8.2's assumptions:

- *All nodes share the same start and commit timestamps*: in this protocol every transaction start timestamp is calculated at the local replica and propagated to the rest of the nodes. Thus, all of them share this value. The commit timestamp is calculated after the transaction is certified and is going to commit. The `certify` function is deterministic if update transactions are validated in the same order at all replicas and that is ensured by the reliable total-order primitive used to propagate transaction writesets (and reads for PL-3 transactions). Hence, all nodes get the same commit timestamp for every transaction. The transformation function is also deterministic and, hence, all nodes share the same values.

- $f(st_i) < f(ct_i)$ *for any* $T_i \in \mathcal{T}$ *committed in* H_r: the timestamp calculation is based on counting committed transactions. The transaction commit timestamp (lines 11 and 37) is calculated after the start timestamp (line 5) and, thus, $st_i \leq ct_i$. If the transaction is an update transaction, the commit timestamp is calculated in line 37 after increasing the local timestamp counter and, hence, in that case $st_i < ct_i$. The transformation funcion adds 0.1 to all transactions start timestamp and 0.2 only to read-only transactions commit timestamps. Then, $f(st_i) < f(ct_i)$ regardless of the transaction type.

- *Given* $T_i, T_j \in \mathcal{T}$ *two committed transactions in* H_r, $f(st_i) < f(ct_j)$ *or* $f(ct_j) < f(st_i)$. The protocol assigns a start and commit timestamp to any committed transaction and, hence, either $st_i < ct_j$, $ct_j < st_i$ or $st_i = ct_j$. In the first case $f(st_i) < f(ct_j)$ since the function f only adds decimals to the original timestamps which are always natural integers. Similarly, $f(ct_j) < f(st_i)$ in the second case. In the third case f adds 0.1 to st_i and, either, 0 (if T_j is an update transaction) or 0.2 (if T_j is read-only) to ct_j. Then, either, $f(ct_j) < f(st_i)$ or $f(st_i) < f(ct_j)$. So, $f(st_i) < f(ct_j)$ or $f(ct_j) < f(st_i)$.

- *Given two conflicting and committed transactions* T_i *and* T_j, *if* $c_i < c_j \in H_r$ *then* $f(ct_i) < f(ct_j)$. Notice that two transactions conflict if both operate over the same item and at least one of them updates it. If T_j is an update transaction then $ct_i < ct_j$ since ct_i and ct_j are the number of committed transactions when T_i and T_j commit, including their own commit if they are update transactions, and T_j is an update transaction and commits later. Since f only adds decimals and the commit timestamps are natural numbers, $f(ct_i) < f(ct_j)$. If T_i is an update transaction but T_j is read-only then $ct_i \leq ct_j$, $f(ct_i) = ct_i$ and $f(ct_j) = ct_j + 0.2$. Hence, $f(ct_i) < f(ct_j)$ anyway.

Given a PL-SI' transaction T_i, this protocol also ensures that T_i's snapshot will include the updates of any other transaction T_j whose $f(ct_j) < f(st_i)$. If $f(ct_j) < f(st_i)$ and T_j is an update transaction then necessarily $ct_j \leq st_i$ since ct_j and st_j are natural numbers, $f(ct_i) = ct_i$ and $f(st_i) = st_i + 0.1$. The protocol assumes that T_i observes the last available snapshot at the local replica when T_i starts and, hence, if $ct_j \leq st_i$ then, necessarily, T_j committed before T_i started and, hence, is part of T_i's snapshot. If $f(st_i) < f(ct_j)$ and T_j is an update transaction then, necessarily $st_i < ct_j$ and, hence, T_j committed after T_i started and is not included in its snapshot.

9.2 Blocking SER-D

In this section we present a blocking version of Kemme's weak-voting SER-D protocol [32] and extend it to support other isolation levels. We call it *blocking* because validation, update and end are executed as a single atomic step instead of separated and a transaction will get blocked before starting its validation step if any other transaction has been previously delivered and has not finished yet. As the original SER-D, the blocking version is based on the deferred updates technique [3] to simplify the validation process. Deferred update protocols execute only read operations during the processing step and delay writes until transactions reach the validation or update steps. The serialisable version of the blocking SER-D is presented in Figure 9.3.

When a transaction is delivered at its local replica only reads are executed. At commit time, the deferred writes are gathered in a writeset and propagated to all nodes in the system using a total-ordered multicast. Once a writeset WS_i is delivered in a node, the validation process starts aborting all active conflicting transactions which have not been validated yet. Since those transactions have not yet executed their writes, only conflicts involving local transaction reads and the writeset writes are checked. If a transaction is aborted after propagating its writeset, an abort message is also sent to all replicas. If a writeset is successfully applied and an abort message has not been previously sent for that transaction, a commit message is propagated instead. Notice that only a reliable message is necessary to propagate abort and commit messages.

Blocking SER-D does not explicitly define any logical time but relies on the lock-based concurrency control to prevent isolation anomalies. Instead of aborting a transaction in its validation step, with this protocol a transaction T_i is aborted in its local node if a concurrent writeset is validated first and modifies items read by T_i. With this early detection technique most transactions may be aborted before propagating their writesets which improves the aborted transaction response time and makes the protocol more network-effective than the original scheme. If T_i is aborted after propagating its writeset, an abort message must be also disseminated.

Blocking SER-D is highly inefficient since, once a transaction T_i starts its vali-

```
Upon t received at $N_a$
   mutex.lock
   t.state $\leftarrow$ LOCAL
   t.local $\leftarrow$ $N_a$
   Execute t reads
   mutex.unlock
On t commit request at $N_a$
   mutex.lock
   ws.data $\leftarrow$ wset(t)
   ws.t $\leftarrow$ t
   if (ws.data $\neq \emptyset$) then
      t.state $\leftarrow$ SENT
      TO-bcast(N,$\langle$ws$\rangle$)
   else
      t.state $\leftarrow$ COMMITTED
      DB.commit(t)
   mutex.unlock
getConflicts(ws)
   conflicts $\leftarrow \emptyset$
   active $\leftarrow$ DB.getActiveTransactions()
   for (t in active) do
      if (rset(t) $\cap$ ws.data $\neq \emptyset$)
         append(conflicts,t)
   return conflicts
Upon $\langle$ws$\rangle$ reception at $N_a$
   mutex.lock
   if (ws.t.state $\neq$ ABORTED) then
      conflicting$\leftarrow$ DB.getConflicts(ws)
      for (t in conflicting) do
         if (t.state = LOCAL) then
            DB.abort(t)
            t.state $\leftarrow$ ABORTED
            send(C, ABORT)
         if (t.state = SENT) then
            R-bcast(N,$\langle$abort,t$\rangle$)
            t.state $\leftarrow$ ABORTED
      DB.requestLocks(ws)
      DB.apply(ws)
      ws.t.state $\leftarrow$ WRITE
      if (ws.t.local = $N_a$) then
         R-bcast(N,$\langle$commit,ws.t$\rangle$)
      msg $\leftarrow$ waitForConfirmationMessage()
      if (msg.status=COMMIT) then
         ws.t.state $\leftarrow$ COMMITTED
         DB.commit(ws.t)
      else if (msg.status=ABORT) then
         ws.t.state $\leftarrow$ ABORTED
         DB.abort(ws.t)
      if (ws.t.local = $N_a$) then
         send(c,msg.status)
   mutex.unlock
```

Figure 9.3: Blocking SER-D weak-voting based protocol.

dation in a given node, the next transactions delivered at that node must wait until T_i is committed or aborted. However, with this protocol all writesets are validated, applied and committed (or aborted) in delivery order and that is a big advantage to support Snapshot Isolation.

As we see in Figure 9.4 and was also pointed out in Section 8.2.3, the final voting step is only necessary for PL-3 transactions since only that isolation level requires to validate reads which are not propagated. Notice also that a explicit logical time based on counting validated writesets is used to certify PL-SI' transactions. Since all steps among validation and end are applied in a single atomic step, we guarantee that any PL-SI' T_i transaction gets a snapshot including the updates of all transactions with a commit-timestamp smaller or equal to T_i's start timestamp. If that is not ensured, the lost-update phenomenon [6], forbidden by Snapshot Isolation, and other inconsistencies may appear during the execution of a PL-SI' transaction. As we show later with MUL-D and has been also studied in other works [38], a variation of the original SER-D supporting Snapshot Isolation does not ensure this unless PL-SI' transactions get blocked until any pending validated PL-3 transaction commit or aborts, specially if it writes something the PL-SI' transaction reads.

As MWVS, the blocking version of MUL-D ensures that all nodes commit the same writesets in the same order. Notice that the certification process and the timestamp rely on the validation order, which is the same at all nodes, PL-3 transactions fate is exclusively decided by their local node and PL-1 and PL-2 transactions are always committed by the protocol. Hence, MUL-D correctness can be proved similarly to the MWVS scheme if we use the transformation function presented in Definition 23.

9.3 Non-blocking SER-D

This protocol shows how the original SER-D protocol [32] can be modified to support other isolation levels as well. Unlike Blocking SER-D, the non-blocking version updates and commits transactions in two separated steps such that two non-conflictive transactions can be executed in any order at two different replicas. This protocol is also based on the deferred writes optimisation which consists in delaying the execution of write operations until the end of the validation step to simplify the conflict resolution.

As the blocking version, non-blocking SER-D takes advantage of local lock-based DBMSs. Instead of aborting the transaction being validated, in the validation step a writeset will abort conflicting transactions not yet validated and will get blocked by those waiting for the confirmation message. Locks ensure that never two conflicting transactions writesets are applied in different order at different replicas.

Once WS_i is validated at its local node, a commit message is propagated unless an abort message for this transaction was previously sent. Notice that abort

```
Init at $N_a$
   $count_a \leftarrow 0$
   $wslist_a \leftarrow \emptyset$
Upon t received at $N_a$
   mutex.lock
   t.start $\leftarrow$ $count_a$
   t.state $\leftarrow$ LOCAL
   t.local $\leftarrow$ $N_a$
   Execute t reads
   mutex.unlock
On t commit request at $N_a$
   mutex.lock
   ws.data $\leftarrow$ wset(t)
   ws.t $\leftarrow$ t
   if (ws.data $\neq \emptyset$) then
      t.state $\leftarrow$ SENT
      TO-bcast(N,$\langle$ws$\rangle$)
   else
      t.state $\leftarrow$ COMMITTED
      DB.commit(t)
   mutex.unlock
getConflicts(ws)
   conflicts $\leftarrow \emptyset$
   active $\leftarrow$ DB.getActiveTransactions()
   for (t in active) do
      if (t.level = PL-3) and
         (rset(t) $\cap$ ws $\neq \emptyset$) then
         append(conflicts,t)
   return conflicts
certify(ws, $wslist_a$)
   if (ws.t.level = PL-SI) then
      for (old_ws $\in$ $wslist_a$) do
         if (ws.start < old_ws.commit) and
            (old_ws.data $\cup$ ws.data $\neq \emptyset$) then
            return ABORT
   return COMMIT
Upon $\langle$ws$\rangle$ reception at $N_a$
   mutex.lock
   if (ws.t.state $\neq$ ABORTED) then
      $count_a \leftarrow count_a$+1
      ws.t.commit $\leftarrow$ $count_a$
      conflicting$\leftarrow$ DB.getConflicts(ws, $wslist_a$)
      for (t in conflicting) do
         if (t.state = LOCAL) then
            DB.abort(t)
            t.state $\leftarrow$ ABORTED
            send(c, ABORT)
         if (t.state = SENT) then
            R-bcast(N,$\langle$abort,t$\rangle$)
            t.state $\leftarrow$ ABORTED
      DB.requestLocks(ws)
      $status_t \leftarrow$ certify(ws,$wslist_a$)
      if ($status_t$ = ABORT) then
         if (ws.t.local = $N_a$) then
            DB.abort(ws.t)
            send(c, $status_t$)
      else
         DB.apply(ws.t)
         if (ws.t.level $\neq$ PL-3) then
            DB.commit(ws.t)
            append($wslist_a$,ws)
            ws.t.state $\leftarrow$ COMMITTED
            send(c, $status_t$)
         else
            ws.t.state $\leftarrow$ WRITE
            if (ws.t.local = $N_a$) then
               R-bcast(N,$\langle$commit,ws.t$\rangle$)
         msg $\leftarrow$ waitForConfirmationMessage()
         if (msg.status = COMMIT) then
            ws.t.state $\leftarrow$ COMMITTED
            DB.commit(ws.t)
            append($wslist_a$,ws)
            if (ws.t.local = $N_a$) then
               send(c, COMMIT)
         else
            ws.t.state $\leftarrow$ ABORTED
            DB.abort(ws.t)
            if (ws.t.local = $N_a$) then
               send(c, ABORT)
   mutex.unlock
```

Figure 9.4: Blocking MUL-D weak-voting based protocols.

```
Upon t received at $N_a$
  mutex.lock
  t.state ← LOCAL
  t.local ← $N_a$
  Execute t reads
  mutex.unlock
On t commit request at $N_a$
  mutex.lock
  ws.local ← $N_a$
  ws.data ← wset(t)
  ws.t ← t
  if (ws.data $\neq \emptyset$) then
    t.state ← SENT
    TO-bcast(N,⟨ws⟩)
  else
    t.state ← COMMITTED
    DB.commit(t)
  mutex.unlock
Upon ⟨ws⟩ reception at $N_a$
  mutex.lock
  if (ws.t.state ≠ ABORTED) then
    conflicting← DB.getConflicts(ws)
    for (t in conflicting) do
      if (t.state = LOCAL) then
        DB.abort(t)
        t.state ← ABORTED
        send(C, ABORT)
      if (t.state = SENT) then
        t.state ← ABORTED
        R-bcast(N,⟨abort,t⟩)
    DB.requestLocks(ws)
    DB.apply(ws)
    ws.t.state ← WRITE
    if (ws.t.local = $N_a$) then
      ws.t.state ← COMMITTED
      R-bcast(N,⟨commit,ws.t⟩)
  mutex.unlock
Upon ⟨commit,t⟩ reception at $N_a$
  mutex.lock
  DB.commit(t)
  if (t.local = $N_a$) then
    send(c,COMMIT)
  mutex.unlock
Upon ⟨abort,t⟩ reception $N_a$
  mutex.lock
  DB.abort(t)
  if (t.local = $N_a$) then
    send(c,ABORT)
  mutex.unlock
getConflicts(ws)
  conflicts ← $\emptyset$
  active ← DB.getActiveTransactions()
  for (t in active) do
    if (rset(t) $\cap$ ws.data $\neq \emptyset$)
      append(conflicts,t)
  return conflicts
```

Figure 9.5: Non-blocking SER-D weak-voting protocol.

and commit messages are sent using only a reliable multicast. WS_i commits once the commit message is delivered. If an abort message arrives instead, the writeset is aborted.

As Figure 9.6 shows up, the multiple isolation level version of the SER-D, named as Non-blocking MUL-D or simply MUL-D, has many differences with the original protocol. MUL-D mixes SER-D with SI-D, the Snapshot Isolation version also presented by Kemme. SER-D and SI-D differ in many aspects. One of them is how writesets are validated. As SI-D, with MUL-D a PL-SI' writeset is aborted if there is a write-dependency with any concurrent transaction previously validated. Whether two transactions are considered concurrent depend on their start and commit timestamps which are calculated by counting the number of validated writesets (`count` in the algorithm) when a transaction starts in the system (`t.start`) and when enters the validation step (`t.commit`) respectively. The absence of a final voting round is another big difference among SI-D and SER-D and that is also reflected in MUL-D. Thus, in MUL-D, all nodes can decide every transaction's fate by themselves without waiting for the local node's decision unless for PL-3 transactions, which still need a final voting step. So, from that point of view, SI-D is a symmetric protocol (like certification-based ones) while SER-D is asymmetric due to this final voting round and that's why MUL-D acts like a weak-voting protocol only for PL-3 transactions and as a certification-based protocol when transactions request any other isolation level.

To ensure the protocol correctness, it assumes that every PL-SI' transaction T_i must get the snapshot including the updates of any transaction T_j such that $ct_j \leq st_i$. As with original SI-D, that is ensured by using write locks even for PL-SI' transactions. Those locks are taken during validation and released once the transaction finishes committing or aborting. A solution based on counting timestamps on commit time instead than in validation process can not be used since PL-3 commit and abort messages are sent using only a reliable broadcast and are not totally ordered. Hence, their delivery order may vary among nodes and transaction may get a different commit timestamp at different nodes.

PL-2 and PL-1 transactions are never aborted during the validation step. Notice that writes are deferred and their locks requested in delivery order which ensures the absence of cycles composed only by write-dependency edges. Local DBMSs also ensure that only committed values are seen by PL-2 transactions which also avoids all G1 phenomena. Thus, no extra validation is needed.

Since the protocol is based on locks, PL-3 and PL-2 reads will get blocked by the local node DBMS if any other transaction is holding a write lock on the same item. Since write locks are requested in an atomic step during validation, those transactions will always see the update performed by the last validated transaction by the time the read is performed, as soon as that transaction finally commits. Hence, if a PL-2 or PL-3 transaction T_i reads a value x_j established by T_j then $ct_j \leq ct_i$.

The correctness proof is almost identical to the MWVS one. Actually, the main difference is that, in this case, non-conflicting PL-3 transactions may abort in

```
Init at $N_a$
  count$_a$ $\leftarrow$ 0
  wslist$_a$ $\leftarrow \emptyset$

Upon t received at N$_a$
  mutex.lock
  t.start $\leftarrow$ count$_a$
  t.state $\leftarrow$ LOCAL
  t.local $\leftarrow$ N$_a$
  Execute t reads
  mutex.unlock

On t commit request at N$_a$
  mutex.lock
  ws.local $\leftarrow$ N$_a$
  ws.data $\leftarrow$ wset(t)
  ws.t $\leftarrow$ t
  if (ws.data $\neq \emptyset$) then
    t.state $\leftarrow$ SENT
    TO-bcast(N,$\langle$ws$\rangle$)
  else
    t.state $\leftarrow$ COMMITTED
    DB.commit(t)
  mutex.unlock

Upon $\langle$ws$\rangle$ reception at N$_a$
  mutex.lock
  if (ws.t.state $\neq$ ABORTED) then
    count$_a$ $\leftarrow$ count$_a$+1
    ws.t.commit $\leftarrow$ count$_a$
    conflicting$\leftarrow$ DB.getConflicts(ws, wslist$_a$)
    for (t in conflicting) do
      if (t.state = LOCAL) then
        DB.abort(t)
        t.state $\leftarrow$ ABORTED
        send(c, ABORT)
      if (t.state = SENT) then
        R-bcast(N,$\langle$abort,t$\rangle$)
        t.state $\leftarrow$ ABORTED
    DB.requestLocks(ws)
    status$_t$ $\leftarrow$ certify(ws,wslist$_a$)
    if (status$_t$ = ABORT) then
      if (ws.local = N$_a$) then
        DB.abort(ws.t)
        send(c, status$_t$)
      ws.t.state $\leftarrow$ ABORTED
    else
      DB.apply(ws)
      if (ws.t.level $\neq$ PL-3) then
        DB.commit(ws)
        append(wslist$_a$,ws)
        ws.t.state $\leftarrow$ COMMITTED
        send(c, status$_t$)
      else
        ws.state $\leftarrow$ WRITE
        if (ws.local = N$_a$) then
          ws.t.state $\leftarrow$ COMMITTED
          R-bcast(N,$\langle$commit,ws.t$\rangle$)
  mutex.unlock

Upon $\langle$commit,t$\rangle$ reception at N$_a$
  mutex.lock
  DB.commit(t)
  append(wslist$_a$,ws)
  if (t.local = N$_a$) then
    send(c, COMMIT)
  mutex.unlock

Upon $\langle$abort,t$\rangle$ reception N$_a$
  mutex.lock
  DB.abort(t)
  if (t.local = N$_a$) then
    send(c, ABORT)
  mutex.unlock

getConflicts(ws)
  conflicts $\leftarrow \emptyset$
  active $\leftarrow$ DB.getActiveTransactions()
  for (t in active) do
    if (t.level = PL-3) and
      (rset(t) $\cap$ ws.data $\neq \emptyset$) then
  return conflicts
      append(conflicts,t)

certify(ws, wslist$_a$)
  if (ws.t.level = PL-SI) then
    for (old_ws $\in$ wslist$_a$) do
      if (ws.start < old_ws.commit) and
        (old_ws.data $\cap$ ws.data) then
        return ABORT
  return COMMIT
```

Figure 9.6: Non-blocking MUL-D weak-voting protocol.

different order which messes up Lemma 6 and first part of Theorem 4, the one in which we explain how commit and start timestamps are ordered for every type RMSG obligatory edge. Thus, we now prove that Non-blocking MUL-D orders starts and commits exactly in the same way MWVS does and, hence, the second part of the theorem can be safely applied to finally prove the protocol correctness. Even read-only transactions commit-timestamps are never calculated in the protocol, we assume it is the number of validated transactions when the transaction commits at its local replica. As with MUL CBR and MUL-D, we also use the transformation function f from Definition 23.

Given a history H_r, an edge e is obligatory in $RMSG(H_r)$ if:

- $e = T_i \xrightarrow{ww} T_j$: since the protocol validates writesets in delivery order and in a single atomic step, the commit-timestamp is calculated during that step which is atomic and the write locks are requested at the end of validation, T_j's writeset has been delivered after T_i's one and, hence, $ct_i < ct_j$ and $f(ct_i) < f(ct_j)$. Recall that the function f only adds decimals to the timestamps which are natural numbers.

- $e = T_i \xrightarrow{wr} T_j$ and T_j is PL-2, PL-SI' or PL-3: since we assume a lock-based correct DBMS, PL-2, PL-3 and PL-SI' transactions only observe committed values. Furthermore, reads are executing at the beginning in the local replica and writes at the end of the validation step, after propagation. Hence, if T_j observes a T_i update necessarily T_i has been validated before T_j and $ct_i \leq ct_j$. If T_j is an update transaction then also $ct_i < ct_j$ and $f(ct_i) < f(ct_j)$ since the timestamp is increased before assigning the commit-timestamp to every validated transaction. If T_j requested PL-SI' then the protocol itself assumes that it will see the updates performed by transactions with a commit-timestamp lower or equal to T_j's start-timestap. Then, in that case, $ct_i \leq st_j$. Since T_i is an update transaction, $f(ct_i) = ct_i$ and $f(st_j) = st_j + 0.1$. Thus, $f(ct_i) < f(st_j)$.

- $e = T_i \xrightarrow{rw} T_j$ and T_i is PL-SI' or PL-3: since the protocol assumes that a PL-SI' transaction observes the updates of all transactions validated before the PL-SI' transaction started, if T_i is PL-SI' then $st_i < ct_j$ and, hence, $f(st_i) < f(ct_j)$ because the timestamps are natural numbers and f only adds decimals. If T_j is PL-3 then WS_j will abort T_i if WS_j is validated while T_i is active but before it is validated. Hence, either T_i is validated first or T_i starts after T_j is validated. In the first case, $ct_i < ct_j$ and, then, $f(ct_i) < f(ct_j)$. In the second case, T_i reads will get blocked by T_j writes and the anti-dependency will never happen. Concluding, $f(ct_i) < f(ct_j)$.

The rest of the correctness proof is identical to the last part of Theorem 4.

9.4 Conclusions

It seems that certification-based protocols are easier to adapt to support multiple isolation levels and, most important, it can be extended in a more optimized way when PL-SI' transactions come into play. Weak-voting protocols need to block at least some PL-SI' transactions to avoid inconsistencies and that may have an important impact in their performance. However, weak-voting may be an interesting alternative when most transactions request PL-3 isolation level since readsets do not need to be propagated. Furthermore, weak-voting is specially appealing to properly support integrity constraints and we also suggest to use that kind of protocols when those constraints are an important issue.

Part IV

Conclusion

Chapter 10

Conclusions

Nowadays, most information systems are distributed and have high availability requirements. Usually, the applications used on top of such a distributed system are unaware of the underlying architecture. Hence, it is important that database replication protocols provide the same functionality of a stand-alone DBMS without compromising any of the replication advantages, i.e., availability and scalability. Unfortunately, achieving true transparent replication is not a trivial task.

We have addressed the problem of supporting multiple isolation levels in existing ROWAA replication protocols. The authors of [16] identified this problem as one of the challenges in database replication. Most existing replication solutions support a single isolation level which is generally one of the strictest: serialisability or snapshot isolation.

In this Ph.D. Thesis we have identified the conditions under which replication protocols may manage multiple isolation levels transparently and proven that protocols that satisfy these conditions are correct. To this end, we suggest a new Snapshot Isolation level definition which fits better with the dependency graphs used as a base representation of transaction conflicts [1]. This definition unifies the base SI definition and the generalised version [6] suggested in other works [1, 19]. We have then modified the popular ROWAA-based replication scheme to support different isolation levels. As an example, we have further demonstrated how these extensions can be applied to specific protocols. The majority of the replication solutions under consideration require only minor changes to support multiple isolation levels, which may result in an improved degree of concurrency and minor transaction completion times for those transactions that can be executed in a relaxed isolation level.

Our model is general enough to be applied to any database replication protocol. As a possible future work we plan to use this model in order to prove the correctness of several existing metaprotocols (e.g., [49]) that concurrently support several replication protocols, each being able to support a different isolation level.

Part V

Appendices

Appendix A

Appendices

The aim of these appendices is to provide the necessary basis for proving the correctness of Theorem 2 (given in Chapter A.5). To this end, Section A.1 introduces a Lemma referenced during Theorem 1 correctness proof in Chapter 5 (page 41). Section A.2 proves that valid histories H correctly manage Snapshot Isolation. Section A.3 proves that valid replicated histories also correctly manage SI transactions. Section A.4 presents several properties derived from Def. 21 that will provide lemmas needed in Section A.5, where Theorem 2 correctness is detailed.

A.1 Chapter 5 lemmas

Lemma 8 (Start and commit orderings by $<_c$ in PL-SI' histories)**.** *Given a PL-SI' history H produced by a scheduler based on a conflict-aware time-precedes order $<_c$, $s_i <_c c_j$ if*

a) A path P in $SSG(H)$ connects nodes T_i and T_j and

b) P is a directed path from T_i to T_j and

c) P does not contain two consecutive anti-dependency edges.

Proof. The proof is separated in two complementary parts. The first one focuses on paths without anti-dependency edges and the second one covers paths with at least one anti-dependency edge:

a) P **is composed only by start and dependency edges**: then trivially $s_i <_c c_j$. Given $p = T_i \overset{s/ww/wr}{\longrightarrow} T_0 \overset{...}{\longrightarrow} T_m \overset{s/ww/wr}{\longrightarrow} T_j$, by $<_c$ first condition $c_i <_c s_0$, ..., $c_m <_c s_j$. Since $<_c$ is a time-precedes order, $s_k <_c c_k$ for any node T_k, $s_i <_c c_i <_c s_0 <_c ... <_c c_m <_c s_j <_c c_j$ and, thus, $s_i <_c c_j$.

b) P **has at least one anti-dependency edge**: we prove $s_i <_c c_j$ by induction over the length of P, n:

- **Base case** ($n = 2$): P contains a single edge e which necessarily is an anti-dependency. By $<_c$ third condition, $s_i <_c c_j$.
- **Induction hypothesis** ($n < l$): the lemma holds if P is $n < l$ length.
- **Induction step** ($n = l$): imagine $e = T_k \xrightarrow{rw} T_{k+1}$ is one of the anti-dependency edges of P. Thus, by $<_c$ third condition, $s_k <_c c_{k+1}$. The previous and the next edges $T_{k-1} \longrightarrow T_k$ and $T_{k+1} \longrightarrow T_{k+2}$ are start or dependency edges because there aren't two consecutive anti-dependency edges. Thus, by $<_c$ second condition, $c_{k-1} <_c s_k$ and $c_{k+1} <_c s_{k+2}$. The paths from T_i to T_{k-1} and from T_{k+2} to T_j are less than $l - 1$ and, by the induction hypothesis, $s_i <_c c_{k-1}$ and $s_{k+2} <_c c_j$. Thus, $s_i <_c c_{k-1} <_c s_k <_c c_{k+1} <_c s_{k+2} <_c c_j$. Exactly the same proof can be used when e is the first or last edge in P. In this case, the next or previous edge must be a dependency and the induction hypothesis is applied to the subpath from this edge to T_j if e is the first edge or to T_i if it is the last one.

□

A.2 Snapshot correctness in valid histories

Adya restates the Snapshot Isolation definition by introducing two properties which must be ensured in any SI history [1]. He calls those properties Snapshot Read and Snapshot Write. Snapshot Read says that all reads performed by a transaction T_i occur at its start point. Snapshot Write is equivalent to first-committer wins rule which states that T_i cannot commit if it modifies the same item that T_j, both transactions are concurrent and T_j commits first. We slightly modify those properties to be suitable in executions when not all transactions request Snapshot Isolation. Later we proof that any valid history H fulfils Snapshot Read and Snapshot Write for all involved committed PL-SI' transactions.

Definition 24 (Snapshot Read)**.** *Being H a history representing an execution of a set of transactions $\mathcal{T}$, H has the Snapshot Read property if a time-precedes order $<_t$ is defined over H such that for every PL-SI' transaction T_i and any other transaction T_j:*

(a) If $T_j \xrightarrow{wr} T_i \in DSG(H)$ then $c_j <_t s_i$.

(b) *If* $T_i \xrightarrow{rw} T_j \in DSG(H)$ *then* $s_i <_t c_j$.

Definition 25 (Snapshot Write)**.** *Being H a history representing an execution of a set of transactions $\mathcal{T}$, H has the Snapshot Write property if a time-precedes order $<_t$ is defined over H such that for every PL-SI' transaction T_i and any other transaction T_j:*

(a) *If* $T_j \xrightarrow{ww} T_i \in DSG(H)$ *then* $c_j <_t s_i$.

(b) *If* $T_i \xrightarrow{ww} T_j \in DSG(H)$ *then* $c_i <_t c_j$.

Unlike Adya's Snapshot Write property, we cannot assure in condition (b) that $c_i <_t s_j$ because T_j may request any possible isolation level. If it requests SI then condition (a) ensures that $c_i <_t s_j$ as Adya states.

Before proving that any valid history H correctly manages SI we first need to introduce the concept of fixed ordering and some lemmas.

Definition 26 (SI Fixed Ordering)**.** *Given a history H, the set O of start and commit operations of transactions committed in H and a time-precedes order $<_t$ such that it fulfils Snapshot Read and Snapshot Write properties, $o_i <_t o_j$ is SI fixed in $<_t$ if their ordering can be deduced from Snapshot Read and Snapshot Write conditions.*

The following lemmas introduce some cases where two operations in O are fixed.

Lemma 9 (SI start and commit ordering 1)**.** *Given a valid history H and its $DSG(H) = (V, E)$, the set O of start and commit operations of transactions committed in H, $s_i, c_j \in O$ and $<_t$ a time-precedes order of O which fulfils Snapshot Read and Snapshot Write properties, $s_i <_t c_j$ is fixed iff there is a directed path $P = (V_p \subseteq V, E_p \subseteq E)$ from T_i to T_j without two adjacent anti-dependency edges and composed only by the kind of edges contemplated in Snapshot Read and Snapshot Write properties (dependency-edges ending in a PL-SI' transaction and anti-dependencies and write-dependency edges starting with a PL-SI' transaction).*

Proof. This proof is very similar to Lemma 3 proof. Snapshot Read and Snapshot Write conditions only directly order start and commit operations if they are connected by a write-dependency edge in $DSG(H)$ involving a PL-SI' transaction, a read-dependency edge ending in a PL-SI' transaction or an anti-dependency starting in a PL-SI' transaction. Thus, if $s_i <_t c_j$ there must be a path P connecting T_i and T_j and composed only by those kind of edges. Similarly to Lemma 3, we prove that this path P fulfils the conditions described in the lemma. We split the proof in two parts:

a) $s_i <_c c_j$ **is fixed if there is a directed path P which fulfils Lemma 9 conditions**. We prove it by induction over the length n of P.

- **Base case** ($n = 2$): P is composed by a single edge $e = T_i \longrightarrow T_j$. If e is a dependency edge ending with a PL-SI' transaction then $c_i <_t s_j$ by conditions (a) of Snapshot Read and Snapshot Write. Since $<_t$ is a time-precedes order (See Definition 6 condition (a)), $s_i <_t c_i$, $s_j <_t c_j$ and, hence, $s_i <_t c_i <_t s_j <_t c_j$. If e is an anti-dependency edge we get $s_i <_t c_j$ directly from Snapshot Read condition (b). Finally, if e is a write-dependency edge starting with a PL-SI' transaction we get $c_i <_t c_j$. Again, since $s_i <_t c_i$ from Definition 6 condition (a), $s_i <_t c_i <_t c_j$. In all cases $s_i <_t c_j$.
- **Induction hypothesis** ($n < l$): the lemma holds if P is $n < l$ length.
- **Induction step** ($n = l$): P length is $n = l > 2$. Then, $P = T_i \longrightarrow T_k \longrightarrow T_j$ where $e = T_k \longrightarrow T_j$ is P's last edge and P' the subpath from T_i to T_k. P' is a $l-1$ length directed path without two adjacent anti-dependency edges and, thus, from the induction hypothesis $s_i <_t c_k$. Depending on e's type the following possibilities arise:
  - *e is a read-dependency edge.* In that case, by Snapshot Read condition (b) we get $c_k <_t s_j$. Then, $s_i <_c c_k <_c s_j$. By Definition 6 condition (a), $s_i <_c s_j <_c c_j$.
  - *e is a write-dependency edge.* If e finishes with a PL-SI' transaction then, by Snapshot Write condition (a), $c_k <_t s_j$ and, like with read-dependencies, $s_i <_t c_k <_t s_j <_t c_j$. If e starts with a PL-SI' transaction then $c_k <_t c_j$ and, hence, $s_i <_t c_k <_t c_j$.
  - *e is an anti-dependency edge.* By Snapshot Read condition (b) we get $s_k <_c c_j$. Since P length $n > 2$ then the length of P' is at least two and $P' = T_i \xrightarrow{P''} T_{k-1} \xrightarrow{e'} T_k$. e' is a dependency edge ending with a PL-SI' transaction since P has not two adjacent anti-dependency edges (T_k is PL-SI' because we assume P only contains anti-dependencies starting with PL-SI' transactions and e is an anti-dependency). Thus, $c_{k-1} <_c s_k$ and, then, $c_{k-1} <_c s_k <_k c_j$. If P'' is length 1 (the only two edges are e and e') then $T_{k-1} = T_i$. Since $s_i <_c c_i$ (from Definition 6 condition (a)), $s_i <_c c_i <_c s_k <_c c_j$. If P'''s length is greater than 1, by the induction hypothesis $s_i <_c c_{k-1}$ since P'' is also a subpath of P and, hence, it is a directed path without two adjacent anti-dependency edges of length $< l$. In that case $s_i <_c c_{k-1} <_c s_k <_c c_j$. In both cases we get $s_i <_c c_j$.

b) **There is a directed path P from T_i to T_j composed only by dependencies ending with a PL-SI' transaction and write-**

dependencies and anti-dependencies starting with a PL-SI' transaction and without two adjacent anti-dependency edges in $DSG(H)$ if $s_i <_c c_j$ is fixed. As we previously said, Snapshot Read and Snapshot Write only fix s_i and c_j if T_i and T_j are connected by a path $P \in DSG(H)$ composed by such kind of edges. We prove by induction that P is also a directed path and fits with all the lemma conditions.

- **Base case** ($n = 2$): since P is composed by a single edge e, no two adjacent anti-dependency edges may exist in P. As $s_i <_c c_j$ then, by Snapshot Read and Snapshot Write conditions e must be a dependency edge or an anti-dependency edge starting from T_i and ending in T_j. If $e = T_i \xrightarrow{ww} T_j$ then $c_i <_t s_j$ if T_j is PL-SI' or $c_i <_t c_j$ if T_i is PL-SI' by Snapshot Write conditions (a) and (b). In the first case, $s_i <_t c_i <_t s_j <_t c_j$ by Definition 6 condition (a). In the second case we get $s_i <_t c_i <_t c_j$. Similarly, if $e = T_i \xrightarrow{wr} T_j$ then $c_i <_t s_j$ from Snapshot Read condition (a) and $s_i <_t c_j$ by Definition 6 condition (a). Finally, if $e = T_i \xrightarrow{rw} T_j$ then directly $s_i <_t c_j$ by Snapshot Read condition (b). If e goes from T_j to T_i we can never deduce $s_i <_t c_j$ from Snapshot Read, Snapshot Write and time-precedes order conditions. Thus P is a directed path from T_i to T_j.
- **Induction hypothesis** ($n < l$): the lemma holds if P is $n < l$ length.
- **Induction step** ($n = l$): P's length is $n = l > 2$. Take any edge $e \in E_p$ such that $P = P'T_k \longrightarrow T_lP''$, P' and P'' are subwalks of P in $DSG(H)$, P' starts with edge T_i and P'' ends with edge T_j and $e = T_k \longrightarrow T_l$. P' and P'' are of length $< l$ and at least one of them is of length > 1 since $n > 2$. e must be of one of the following types:
  - *e is a read-dependency edge*: T_l must be PL-SI' since only those read-dependencies fix operations due to Snapshot Read condition (a). Then, $c_k <_c s_l$. Since $s_i <_c c_j$ is fixed due to P, $s_i <_c c_k <_c s_l <_c c_j$ and, hence, $s_i <_c c_k$ and $s_l <_c c_j$ are fixed. From the induction hypothesis, P' and P'' are directed paths and fulfil this lemma conditions. P' goes from T_i to T_k and P'' from T_l to T_j. Since the connecting edge e is not an anti-dependency edge, the resulting concatenated walk P is also a directed path without two consecutive anti-dependency edges.
  - *e is a write-dependency edge*: in this case at least T_k or T_l are PL-SI' since only in those cases operations of different transactions are fixed in Snapshot Read and Snapshot Write. If T_l is PL-SI' then $c_k <_t s_l$ and the rest

of the proof is identical to the read-dependency case. If T_l is not PL-SI' then T_k is PL-SI' and, hence, $c_k <_t c_l$ and $s_i <_t c_k <_t c_l <_t c_j$. By the induction hypothesis, since $s_i <_t c_k$ then P' subpath fulfils the lemma conditions. Since $c_l <_t c_j$ and $s_l <_t c_l$ due to Definition 6 condition (a), $s_l <_t c_j$ and hence P'' is a directed edge from T_l to T_j and so it is $P = P'eP''$.

– *e is an anti-dependency edge*: since P only includes anti-dependencies starting with a PL-SI' transaction, $s_k <_t c_l$ by Snapshot Read condition (b). Since we assume $s_i <_t c_j$ is also fixed due to P, $s_i <_t s_k$ and $c_l <_t c_j$. Since $s_k <_t c_k$ and $s_l <_t c_l$ (from Definition 6 condition (a)), $s_i <_t c_k$ and $s_l <_t c_j$. If P' and P'' are of length > 2, by the induction hypothesis, P' and P'' are directed paths without two adjacent anti-dependency edges, the first goes from T_i to T_k and the second one from T_l to T_j. Assume $e' = T_{k-1} \longrightarrow T_k$ the edge just before e and $e'' = T_l \longrightarrow T_{l+1}$ the edge just after e. Since $s_k <_t c_l$ then $s_i <_t o_{k-1} <_t s_k <_s c_l <_s o_{l+1} <_t c_j$ where o_{k-1} and o_{l+1} are either the start or commit operation of T_{k-1} and T_{l+1}. Since Snapshot Read and Snapshot Write conditions only explicitly order starts with commits or commits with commits for a given edge but never starts with starts, o_{k-1} must be c_{k-1} and e' is a dependency edge since only Snapshot Read and Snapshot Write conditions (a) order those operations that way. If $o_{l+1} = s_{l+1}$ then e is a dependency edge ending in a PL-SI' transaction. If $o_{l+1} = c_{l+1}$ then T_l is PL-SI' and e'' is a write-dependency edge. Thus, the path composed by P', e and P'' is a directed path from T_i to T_j without two adjacent anti-dependency edges. The cases when P' or P'' are of length 1 can be similarly deduced but taking into account that $T_l = T_j$ or $T_i = T_k$ respectively.

□

Lemma 10 (SI start an commit ordering 2)**.** *Given a valid history H and its $DSG(H) = (V, E)$, the set O of start and commit operations of transactions committed in H, $s_i, c_j \in O$ and $<_t$ a time-precedes order of O which fulfils Snapshot Read and Snapshot Write properties, $c_i <_t c_j$ is fixed iff there is a directed path $P = (V_p \subseteq V, E_p \subseteq E)$ from T_i to T_j without two adjacent anti-dependency edges and composed only by the kind of edges contemplated in Snapshot Read and Snapshot Write properties (dependency-edges ending in a PL-SI' transaction and anti-dependencies and write-dependency edges starting with a PL-SI' transaction) and P starts with a dependency edge.*

Proof. a) $c_i <_c c_j$ **is fixed if there is a directed path P which**

fulfils Lemma 10 conditions. We prove it by induction over the length n of P.

- **Base case** ($n = 2$): P is composed by a single edge $e = T_i \longrightarrow T_j$ which is a dependency edge. If T_j is PL-SI' then, by Snapshot Read condition (a) and Snapshot Write condition (a), $c_i <_t s_j$. Since $s_j <_t c_j$ by Definition 6 condition (a), $c_i <_t c_j$. If e is a write-dependency starting with a PL-SI' transaction then, by Snapshot Write condition (b), $c_i <_t c_j$ directly.
- **Induction hypothesis** ($n < l$): the lemma holds if P is $n < l$ length.
- **Induction step** ($n = l$): P length is $n = l > 2$. Then, $P = eP'$ where $e = T_i \longrightarrow T_k$ is P's first edge, which by assumption is a dependency edge, and P' the subpath from T_k to T_j. P and P' fulfil Lemma 9 conditions and, hence, $s_k <_t c_j$. If T_k is a PL-SI' transaction then, by Snapshot Read condition (a) and Snapshot Write condition (a), $c_i <_t s_k$ and, hence, $c_i <_t c_j$. Otherwise, e is a write-dependency starting with a PL-SI' transaction and T_k is not PL-SI'. Hence $c_i <_t c_k$ and the first edge in P' must be a dependency edge since, by assumption, P only contains anti-dependencies starting with a PL-SI' transaction and T_k is not PL-SI'. By the induction hypothesis, $c_k <_t c_j$ and $c_i <_t c_j$.

b) **There is a directed path P fulfilling Lemma 10 conditions if $c_i <_c c_j$ is fixed**. As we previously said, Snapshot Read and Snapshot Write only directly fix c_i and c_j if T_i and T_j are connected by a path $P \in DSG(H)$. We prove by induction that P fits with all the expected conditions.

- **Base case** ($n = 2$): since P is composed by a single edge e, no two adjacent anti-dependency edges may exist in P. As $c_i <_t c_j$ then, by Snapshot Read and Snapshot Write conditions (a) and (b) e must be a dependency edge starting from T_i and ending in T_j. If T_j is PL-SI' then, by Snapshot Read condition (a) and Snapshot Write condition (a), $c_i <_t s_j$ and, by Definition 6 condition (a), $c_i <_t s_j <_t c_j$. If e is a write-dependency and T_i is PL-SI' then, by Snapshot Write condition (b) $c_i <_t c_j$.
- **Induction hypothesis** ($n < l$): the lemma holds if P is $n < l$ length.
- **Induction step** ($n = l$): P's length is $n = l > 2$, $P = eP'$ and $c_i <_t c_j$ due to P. Since $s_i <_t c_i$ by Definition 6 condition (a), $s_i <_t c_i <_t c_j$ and P fulfils Lemma 9. Hence, $e = T_i \longrightarrow T_k$ and P' goes from T_k to T_j. Since P' also fulfils Lemma 9 conditions, $s_k <_t c_j$. If the first edge of P' is a dependency edge then also $c_k <_t c_j$ by the induction hypothesis. Since $P = eP'$ and

$c_i <_t c_j$ due to P, $c_i <_t s_k <_t c_j$ or $c_i <_t c_k <_t c_j$ if T_k is a dependency edge. Then, by Snapshot Read condition (a) and Snapshot Write conditions (a) and (b), necessarily e must be a dependency edge ending in a PL-SI' transaction or a write-dependency starting with a PL-SI' transaction. Note that if e is an anti-dependency then $c_i <_t s_k$ or $c_i <_t c_k$ are never fixed.

□

Lemma 11 (SI start an commit ordering 3)**.** *Given a valid history H and its $DSG(H) = (V, E)$, the set O of start and commit operations of transactions committed in H, $s_i, c_j \in O$ and $<_t$ a time-precedes order of O which fulfils Snapshot Read and Snapshot Write properties, $s_i <_t s_j$ is fixed iff there is a directed path $P = (V_p \subseteq V, E_p \subseteq E)$ from T_i to T_j without two adjacent anti-dependency edges and composed only by the kind of edges contemplated in Snapshot Read and Snapshot Write properties (dependency-edges ending in a PL-SI' transaction and anti-dependencies and write-dependency edges starting with a PL-SI' transaction) and P ends with a dependency edge ending with a PL-SI' transaction.*

Proof. We can proof this Lemma in a similar way we done for Lemma 10.

a) $s_i <_c s_j$ **is fixed if there is a directed path** P **which fulfils Lemma 11 conditions**. We prove it by induction over the length n of P.

- **Base case** ($n = 2$): P is composed by a single edge $e = T_i \longrightarrow T_j$ which is a dependency edge ending in a PL-SI' transaction. By Snapshot Read condition (a) and Snapshot Write condition (a), $c_i <_t s_j$. Since $s_i <_t c_i$ by 6 condition (a), $s_i <_t s_j$.
- **Induction hypothesis** ($n < l$): the lemma holds if P is $n < l$ length.
- **Induction step** ($n = l$): P length is $n = l > 2$. Then, $P = P'e$ where $e = T_k \longrightarrow T_j$ is P's last edge and P' the subpath from T_i to T_k. P and P' fulfil Lemma 9 conditions and, hence, $s_i <_t c_k$. By assumption e is a dependency edge and T_j is PL-SI' and, by Snapshot Read condition (a) and Snapshot Write condition (a), $c_k <_t s_j$. Hence, $s_i <_t s_j$.

b) **There is a directed path** P **fulfiling Lemma 11 conditions if** $s_i <_c s_j$ **is fixed**. Snapshot Read and Snapshot Write only directly fixes s_i and s_j if T_i and T_j are connected by a path $P \in DSG(H)$. We prove by induction that P fits with all the expected conditions.

- **Base case** ($n = 2$): since P is composed by a single edge e, no two adjacent anti-dependency edges may exist in P. As

$s_i <_t s_j$ then, by Snapshot Read condition (a) and Snapshot Write conditions (a), e must be a dependency edge starting from T_i and ending in T_j and T_j is PL-SI'. In that case, $c_i <_t s_j$ and, from 6 condition (a), $s_i <_t c_i$ and $s_i <_t s_j$. In any other case $s_i <_t s_j$ cannot be fixed with a single edge.

- **Induction hypothesis** ($n < l$): the lemma holds if P is $n < l$ length.
- **Induction step** ($n = l$): P's length is $n = l > 2$, $P = P'e$ and $s_i <_t s_j$ due to P. Since $s_j <_t c_j$ by Definition 6 condition (a), $s_i <_t s_j <_t c_j$ and P fulfils Lemma 9. Hence, $e = T_k \longrightarrow T_i$ and P' goes from T_i to T_k. Since P' also fulfils Lemma 9 conditions, $s_i <_t c_k$. Since $P = P'e$ and $s_i <_t s_j$ due to P, $s_i <_t c_k <_t s_j$ or, by induction hypothesis, $s_i <_t s_k <_t s_j$ if T_k is a dependency edge ending in a PL-SI' transaction. In the first case e must be a dependency edge and T_j is a PL-SI' transaction since, otherwise, $c_k <_t s_j$ can never be deduced from Snapshot Read and Snapshot Write conditions. In the second case, again by induction hypothesis e is a dependency edge ending with a PL-SI' transaction.

□

Lemma 12 (SI start an commit ordering 4)**.** *Given a valid history H and its $DSG(H) = (V, E)$, the set O of start and commit operations of transactions committed in H, $s_i, c_j \in O$ and $<_t$ a time-precedes order of O such that it fulfils Snapshot Read and Snapshot Write conditions, $c_i <_t s_j$ is fixed iff there is a directed path $P = (V_p \subseteq V, E_p \subseteq E)$ from T_i to T_j without two adjacent anti-dependency edges and composed only by the kind of edges contemplated in Snapshot Read and Snapshot Write properties (dependency-edges ending in a PL-SI' transaction and anti-dependencies and write-dependency edges starting with a PL-SI' transaction), P starts and finishes with a dependency edge and the last vertex in the path is PL-SI'.*

Proof. This proof is a combinations of previous Lemmas.

If $c_i <_t s_j$ is fixed then, since $s_i <_t c_i$ and $s_j <_t c_j$ from Definition 6 condition (a), $c_i <_t c_j$, $s_i <_t s_j$ and $s_i <_t c_j$. Then, exists a directed path P which fulfils Lemmas 9, 10 and 11 conditions, starts with a dependency edge and ends with a dependency edge finishing in a PL-SI' transaction which proves one of the directions of the Lemma.

If we assume exists a path P from T_i to T_j which fulfils this Lemma conditions then P also fulfils Lemma 9, 10 and 11 conditions. Then, $s_i <_t c_j$, $c_i <_t c_j$ and $s_i <_t s_j$. Since the last edge $e = T_k \longrightarrow T_j$ in P is a dependency edge and T_j is PL-SI', $c_k <_t s_j$. If $T_i = T_k$ (P length is 2) then this is it. Otherwise, from Lemma 10 $c_i <_t c_j$ due to P and, hence, $c_i <_t c_k <_t s_j <_t c_j$ and $c_i <_t s_j$.

□

Notice that edges considered in Snapshot Read and Snapshot Write properties are actually obligatory edges in EMSG. Thus, Lemmas 9, 10, 11 and 12 are also true if we use EMSG instead of DSG.

Theorem 7 (Snapshot Correctness)**.** *Being H a valid history representing an execution of a set of transactions $\mathcal{T}$, a time-precedes order $<_t$ is defined over H such that both fulfil Snapshot Read and Snapshot Write properties.*

Proof. This proof is very similar to 1 proof. By absurd reduction, we assume that the set O of start and commit operations of committed transactions in H cannot be ordered by a time-precedes order $<_t$ such that Snapshot Read and Snapshot Write conditions are fulfilled. That ordering is impossible only if a contradiction may appear when O is scheduled following $<_t$. In other words, if it is possible to have $o_i \in T_i$ and $o_j \in T_j$ $(o_i \neq o_j)$ such that $o_i, o_j \in O$, $o_i <_t o_j$ and $o_j <_t o_i$.

If $o_i = s_i$ and $o_j = c_j$ then $s_i <_t c_j$ and $c_j <_t s_i$. From Lemma 9 there is a directed path $P_1 \in EMSG(H)$ from T_i to T_j without two anti-dependency edges. From Lemma 12 there is another directed path $P_2 \in EMSG(H)$ from T_j to T_i also without two consecutive anti-dependency edges and the first and last edges are dependences. Then, there is a directed cycle $C = P_1P_2$ without two consecutive anti-dependency edges and H is not valid, which contradicts the initial assumption. The same happens if $o_i = c_i$ and $o_j = s_j$.

If $o_i = s_i$ and $o_j = s_j$ then, from Lemma 11 there is a directed path $P_1 \in EMSG(H)$ from T_i to T_j without two consecutive anti-dependency edges and the last edge is a dependency edge. From the same Lemma, there is a similar directed path $P_2 \in EMSG(H)$ from T_j to T_i. Since the last edge of every path is a dependency, the directed cycle $C = P_1P_2$ does not have two consecutive anti-dependency edges and, hence, H is not valid which contradicts the initial assumption.

If $o_i = c_i$ and $o_j = c_j$ then, from Lemma 11, there is a directed path $P_1 \in EMSG(H)$ from T_i to T_j without two anti-dependencies which starts with a dependency and there is also another similar directed path $P_2 \in EMSG(H)$ from T_j to T_i. Since the first edge of both paths is a dependency, the directed cycle $C = P_1P_2$ does no show two consecutive anti-dependency edges and H is not valid, which contradicts the initial assumption.

Resuming, in any possible case, if H is valid then it fulfils the Snapshot Read and Snapshot Write properties for any PL-SI' transaction. □

A.3 Snapshot correctness in valid replicated histories

The Snapshot Read and Snapshot Write properties defined in Section A.2 can be also used in replicated environments by just using RMSGs instead of DSGs or EMSGs.

Definition 27 (Replicated Snapshot Read)**.** *Being H_r a replicated history representing an execution of a set of transactions $\mathcal{T}$ over a system composed by a set of nodes $\mathcal{N}$, H_r has the Replicated Snapshot Read property if a time-precedes order $<_t$ is defined over H_r such that for every SI transaction T_i and any other transaction T_j:*

(a) If $T_j \xrightarrow{wr} T_i \in RMSG(H_r)$ then $c_j <_t s_i$.

(b) If $T_i \xrightarrow{rw} T_j \in RMSG(H_r)$ then $s_i <_t c_j$.

Definition 28 (Replicated Snapshot Write)**.** *Being H_r a replicated history representing an execution of a set of transactions $\mathcal{T}$ over a system composed by a set of nodes $\mathcal{N}$, H_r has the Replicated Snapshot Write property if a time-precedes order $<_t$ is defined over H_r such that for every SI transaction T_i and any other transaction T_j:*

(a) If $T_j \xrightarrow{ww} T_i \in RMSG(H_r)$ then $c_j <_t s_i$.

(b) If $T_i \xrightarrow{ww} T_j \in RMSG(H_r)$ then $c_i <_t c_j$.

Theorem 8 (Replicated Snapshot Correctness)**.** *Being H_r a valid replicated history representing an execution of a set of transactions $\mathcal{T}$ over a system composed by a set of nodes $\mathcal{N}$, a time-precedes order $<_t$ is defined over the set of start and commit operations of H_r's committed transactions such that both fulfil Replicated Snapshot Read and Replicated Snapshot Write properties.*

Proof. Exactly the same methodology followed to prove Theorem 7 correctness in the previous section can be also used here to prove that every valid replicated history H_r correctly manages Snapshot Isolation. Furthermore, Theorem 2 also proves that valid replicated histories are correct in the general case and a specific proof for Snapshot Isolation is not necessary.

Actually, since H_r is valid, local histories are also valid and local time-precedes orders fulfil Snapshot Read and Snapshot Write properties for PL-SI' transactions. Since $RMSG(H_r)$ is just the combination of all local $EMSG(H_a)$ edges, an edge considered in Replicated Snapshot Read and Write properties appears also in one or another local $EMSG$. Then, $<_t$ must include all orderings fixed by the local time-precedes orders. If $o_i <_t^a o_j$ is fixed in node N_a, $o_i <_t o_j$ must be also fixed in H_r. Replicated Snapshot Read and Write properties will be fulfilled only if a contradiction does not show up when fixings coming from different nodes are mixed up in $<_t$. It can be proved that if that happens then a cycle forbidden by valid histories appears, which is a contradiction and proves the theorem. The formal proof is not included here since it is very similar to previous proofs. Informally, we prove that, if $o_i <_t o_j$ is fixed in H_r due to an edge e and $o_j <_t o_i$ is also fixed due to a combination of edges in $RMSG(H_r)$, this combination of edges is actually a directed path P such that, combined with e, conforms a forbidden cycle. Assume $o_j <_t o_i$ is fixed due to a sequence of fixed orderings $o_j <_t o_0 <_t ... <_t o_n <_t o_i$ such that $o_k <_t o_{k+1}$ is locally fixed in at least one node N_k. Every $o_k \in \{s_k, c_k\}$. From Lemmas 9, 10, 11 and 12 there is a directed path $P_k \in EMSG(H^k)$ from T_k to T_{k+1} without two consecutive anti-dependency edges sharing a PL-SI' transaction and this path appears also in $RMSG(H_r)$. If $o_k = c_k$ then the first edge of P_k is a dependency edge. If $o_{k+1} = s_k$ then the last edge is also a dependency edge. Having that in mind, the concatenation of all those paths in $RMSG(H_r)$ makes a directed path P with the same properties. P combined with e makes a cycle without two consecutive anti-dependency edges sharing a PL-SI' transaction. For example, if $s_i <_t c_j$ due to e then e is an anti-dependency edge. Since $c_j <_t s_i$ due to P, P starts and ends with dependency edges and the combination with e makes a forbidden cycle. If $c_i <_t s_j$ then e is a dependency edge then the combination with P is forbidden anyway. The same happens if $c_i <_t c_j$ due to e (notice that $s_i <_t s_j$ cannot be directly fixed due to a Snapshot Read or Write condition). □

A.4 Global order of conflicting operations

The following lemma states that given any pair of conflicting operations o_i, o_j, if $\exists N_a$ for which $o_i^a < o_j^a \in H^a$ then $o_i < o_j \in H_r$.

Lemma 13 (Global order of conflicting operations)**.** *Given a valid replicated history H_r over a set of transactions $\mathcal{T}$, for any two conflicting operations o_i, o_j of committed transactions $T_i, T_j \in \mathcal{T}$, $o_i < o_j \in H_r \vee o_j < o_i \in H_r$.*

Proof. Recall that $o_i < o_j \in H_r$ if $\exists N_a$ for which $o_i^a < o_j^a \in H^a$ but $\nexists N_b$ for which $o_j^b < o_i^b \in H^b$. If o_i and o_j meet only in one replica then their order in H_r is determined by how are they ordered in that replica.

If both are executed in more than one node and both operations are writes, then they are executed in all of them in the same order because otherwise there would be a cycle in the $RMSG$. By absurd reduction, assume $w_i^a(x_i) < w_j^a(x_j) \in H^a$ but $w_j^b(x_j) < w_i^b(x_i) \in H^b$. Write-dependencies are always obligatory edges, so, $T_i \xrightarrow{ww} T_j \in EMSG(H^a)$ and $T_j \xrightarrow{ww} T_i \in EMSG(H^b)$. Since $RMSG(H_r)$ is the union of all local $EMSG$, from H^a we obtain $T_i \xrightarrow{ww} T_j \in RMSG(H_r)$ and from H^b we obtain $T_j \xrightarrow{ww} T_i \in RMSG(H_r)$, which close the cycle and $RMSG(H_r)$ is not valid, which contradicts the initial assumption.

If one operation is a read, then it logically reads the same value in all replicas[1]. If the read operation is $r_i(x_j)$, by the history definition (see Definition 2) $w_i^a(x_j^a) < r_i^a(x_j^a)$ in any replica N_a where that read is executed. Assume $r_i(x_j)$ and $w_k(x_k)$ both executed at N_a and N_b. We prove by absurd reduction that they are always ordered in the same direction. Thus, assume $r_i^a(x_j) < w_k^a(x_k) \in H_r^a$ but $w_k^b(x_k) < r_i^b(x_j) \in H_r^b$. Thus, $w_j^a(x_j) < r_i^a(x_j) < w_k^a(x_k^a) \in H_r^a$ but $w_k^b(x_k) < w_j^b(x_j) < r_i^b(x_j^b) \in H_r^b$. Then, there is a cycle in $RMSG(H_r)$ composed by two dependency edges involving T_i and T_k but $RMSG(H_r)$ is supposed to be valid which is a contradiction. Then, in any case both operations are executed in the same order in all replicas.

□

A.5 Correctness proof of Theorem 2

Hereafter, we prove the correctness of Theorem 2; i.e. if H_r is valid (see Definition 21), then there is a valid and equivalent H. To this end, we firstly suggest a methodology to extract a stand-alone history H from the RMSG of a replicated history H_r. We then show that H is equivalent to H_r and fulfils the conditions of Definition 16. Theorem 2 is therefore proven.

Definition 29 (Replicated Projection)**.** *Given a replicated history H_r representing the execution of a set of transactions $\mathcal{T}$ in set of nodes $\mathcal{N}$, the replicated projection $P(H_r)$ is a partially ordered set of the operations in $\mathcal{T}$ with a binary relation $<_p$ where:*

1. *If operation $o \in H_r$ then $o \in P(H_r)$.*
2. *$\forall T_i \in \mathcal{T}$ and $\forall o_1, o_2 \in T_i$, if $o_1 < o_2 \in T_i$ then $o_1 <_p o_2 \in P(H_r)$.*
3. *$\forall o_i, o_j \in H_r$, if $o_i <_r o_j \in H_r$ then $o_i <_p o_j \in P(H_r)$.*

Notice that $P(H_r)$ might not order any pair of conflicting operations. For example, given $w_i(x_i)$ and $w_j(x_j)$ two conflicting operations executed in H_r, if they have been executed in different order in two of the nodes in $\mathcal{N}$ then $w_i(x_i) \not<_r w_j(x_j)$ and $w_j(x_j) \not<_r w_i(x_i)$. Recall that $w_i(x_i) <_r w_j(x_j)$ if $\exists N_a \in \mathcal{N}$ such that $w_i(x_i) <_a w_j(x_j)$ and $\nexists N_b \in \mathcal{N}$ such that $w_j(x_j) <_b w_i(x_i)$.

Definition 30 (Complete projection)**.** *Given a replicated history H_r, its replicated projection $P(H_r)$ is complete if it is a history (i.e., it fits with Definition 2).*

As we are going to prove now, if H_r is valid then $P(H_r)$ is complete, valid and equivalent to H_r. Then, if H_r is valid then it is also correct.

[1] If reads are executed in more than one replica we assume that the logical value read is the one accepted by the client. Usually protocols fit with this either by taking a ROWA approach or by using deterministic or consensus algorithms

Lemma 14 (Valid replicated projection)**.** *If H_r is valid then $P(H_r)$ is complete.*

Proof. Assume a valid replicated history H_r representing the execution of a set of transactions $\mathcal{T}$ in a set of nodes $\mathcal{N}$. $P(H_r)$ is complete if:

- *For any transaction $T_i \in \mathcal{T}$ and any operation $o_i \in T_i$, $o_i \in P(H_r)$*: $o_i \in P(H_r)$ if $o_i \in H_r$. $o_i \in H_r$ if o_i is executed at least in one of the nodes in $\mathcal{N}$. Since H_r is a replicated history, all operations in $\mathcal{T}$ are executed at least in one node. Hence, $P(H_r)$ also includes all operations in $\mathcal{T}$.

- *For any transaction $T_i \in \mathcal{T}$ and any two operations $o_1, o_2 \in T_i$, if $o_1 < o_2 \in T_i$ then $o_1 <_p o_2 \in P(H_r)$.* This is ensured by Definition 29 condition 2.

- *If $r_i(x_j) \in P(H_r)$ then $w_j(x_j) \in P(H_r)$ and $w_j(x_j) <_p r_i(x_j) \in P(H_r)$.* If $r_i(x_j) \in P(H_r)$ then $r_i(x_j) \in H_r$. Then, $r_i(x_j)$ has been executed at least in one node. Since H_r is valid, all nodes local histories are also valid. For all H^a, if $r_i(x_j) \in H^a$ then $w_j(x_j) \in H^a$ and $w_j(x_j) < r_i(x_j) \in H^a$. Since both operations conflict, from Lemma 13 $w_j(x_j) < r_i(x_j) \in H_r$. From Definition 29 condition 3, $w_j(x_j) <_p r_i(x_j) \in P(H_r)$.

- *For any two conflicting operations $o_1, o_2 \in P(H_r)$, $o_1 <_p o_2 \in P(H_r)$ or $o_2 <_p o_1 \in P(H_r)$.* From Lemma 13, $o_1 < o_2 \in H_r$ or $o_2 < o_1 \in H_r$. From projection definition condition 3, $o_1 <_p o_2 \in P(H_r)$ or $o_2 <_p o_1 \in P(H_r)$.

□

Thus, if H_r is valid then $P(H_r)$ is complete. Now we prove that H_r and $P(H_r)$ are equivalent.

Lemma 15 (A replicated history and its replicated projection are equivalent)**.** *Given a valid replicated history H_r and its replicated projection $P(H_r)$ constructed as shown in Def. 29, H_r and $P(H_r)$ are equivalent.*

Proof. This can be proven by showing that all equivalence conditions are held.

C1: *$P(H_r)$ and H_r execute the same set of transactions $\mathcal{T}$ and commit the same subset $\mathcal{T}_c \in \mathcal{T}$*: by construction of $P(H_r)$ (step 1), both execute exactly the same operations, including commits.

C2: *For any committed transaction $T_i \in \mathcal{T}_c$, $r_i(x_j) \in P(H_r)$ iff $r_i(x_j) \in H_r$.* By definition, every operation executed in H_r appears also in $P(H_r)$. Furthermore, a read executed in several nodes in a replicated system is assumed to see the same values in all of them. Hence, the value read will be always x_j.

C3: *For every two transactions* $T_i, T_j \in \mathcal{T}_c$, $w_i(x) < w_j(x) \in P(H_r)$ *iff* $w_i(x) < w_j(x) \in H_r$: from Lemma 13, if $w_i(x_i) < w_j(x_j)$ in one node then $w_i(x_i) < w_j(x_j) \in H_r$ and, hence $w_i(x_i) <_p w_j(x_j) \in P(H_r)$. Since H_r and $P(H_r)$ see the same operations, both also see the same states sequence.

□

Therefore, we have proved that given any valid H_r, its replicated projection $P(H_r)$ is an equivalent stand-alone history. However, is $P(H_r)$ valid?

Lemma 16 (A replicated history projection is valid)**.** *Given a valid replicated history* H_r *and its replicated projection* $P(H_r)$ *constructed as shown in Def. 29,* $P(H_r)$ *is valid.*

Proof. It is valid if the conditions (a) and (b) of Def. 16 are held:

(a) *PL-2, PL-3 or PL-SI' transactions do not read aborted or intermediate values*: that condition is trivially held since reads obtain the same value in $P(H_r)$ and H_r which is supposed to be valid and, hence, reads never obtain aborted or intermediate values.

(b) $EMSG(P(H_r))$ *does not contain any directed cycle unless it contains two adjacent edges sharing a PL-SI' transaction*: the form in which $P(H_r)$ is constructed ensures that it contains all H_r dependencies and hence, $EMSG(P(H_r))$ has all $RMSG(H_r)$ edges. However, is it possible for $P(H_r)$ to show dependencies not present in H_r? The answer is no since H and H_r are both over the same set of operations, Lemma 13 proves that any pair of conflicting operations are ordered in H_r and Replicated Projection condition 4 (Definition 29) forces the same ordering in $P(H_r)$. Then, $EMSG(H_r)$ and $RMSG(H_r)$ have the same vertices and edges. Since $RMSG(H_r)$ does not show those cycles, neither appears in $EMSG(P(H_r))$.

□

Therefore, given any valid replicated history H_r, we have proven that it is equivalent to its replicated projection $P(H_r)$ which is also a stand-alone valid history. As a result, H_r is correct and Theorem 2 is proven.

Bibliography

[1] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* PhD thesis, Massachusetts Institute of Technology, March 1999.

[2] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *IEEE Intnl. Conf. on Data Engineering*, pages 67–78, San Diego, CA, USA, March 2000.

[3] Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, and Ioana Stanoi. Exploiting atomic broadcast in replicated databases (extended abstract). In *3rd Intnl. Euro-Par Conf.*, volume 1300 of *Lect. Notes Comput. Sc.*, pages 496–503, Passau, Germany, 1997. Springer.

[4] José Enrique Armendáriz-Iñigo, J. R. Juárez-Rodríguez, José Ramón González de Mendívil, Hendrik Decker, and Francesc D. Muñoz-Escoí. *k*-bound GSI: a flexible database replication protocol. In Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo, editors, *SAC*, pages 556–560. ACM, 2007.

[5] J.E. Armendáriz-Iñigo, J.R. Juárez-Rodríguez, J.R. González de Mendívil, J.R. Garitagoitia, L. Irún-Briz, and F.D. Muñoz-Escoí. A formal characterization of SI-based ROWA replication protocols. *Data & Knowledge Engineering*, 70(1):21 – 34, 2011.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Intnl. Conf. on Manag. of Data (SIGMOD)*, pages 1–10, San José, CA, USA, May 1995. ACM Press.

[7] J. M. Bernabé-Gisbert. Providing support for data replication protocols with multiple isolation levels. In *OTM 2007 Workshops*, Vilamoura, Algarve, Portugal, November 2007. Springer.

[8] J. M. Bernabé-Gisbert and F. D. Muñoz. Extending mixed serialisation graphs to replicated environments. In *3rd Intnl. Conf. on Avail., Reliab. and Security (ARES)*, pages 369–375, Barcelona, Spain, March 2008. IEEE-CS Press.

[9] J. M. Bernabé-Gisbert and F. D. Muñoz-Escoí. A compoundable specification of the snapshot isolation level. Technical Report ITI-SIDI-2012/007, Instituto Tecnológico de Informática, June 2012.

[10] J. M. Bernabé-Gisbert and F. D. Muñoz-Escoí. Supporting multiple isolation levels in replicated environments. *Data And Knowledge Engineering*, 79 - 80(0):1 – 16, 2012.

[11] J. M. Bernabé-Gisbert, R. Salinas-Monteagudo, L. Irún-Briz, and F. D. Muñoz-Escoí. Managing multiple isolation levels in middleware database replication protocols. In *6th Intnl. Symp. on Paral. and Distrib. Proces. and Appl. (ISPA)*, volume 4330 of *Lect. Notes Comput. Sc.*, pages 511–523, Sorrento (Naples), Italy, December 2006. Springer.

[12] Arthur J. Bernstein, Philip M. Lewis, and Shiyong Lu. Semantic conditions for correctness at different isolation levels. In *ICDE*, pages 57–66, 2000.

[13] Philip A. Bernstein. Middleware: a model for distributed system services. *Commun. ACM*, 39:86–98, February 1996.

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

[15] Stephanie Cammarata, Prasadram Ramachandra, and Darrell Shane. Extending a relational database with deferred referential integrity checking and intelligent joins. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, SIGMOD '89, pages 88–97, New York, NY, USA, 1989. ACM.

[16] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: The gaps between theory and practice. In *Intnl. Conf. on Manag. of Data (SIGMOD)*, Vancouver, Canada, June 2008. ACM Press.

[17] Bernardette Charron-Bost, Fernando Pedone, and André Schiper. *Replication: Theory and Practice.* Springer, 2010.

[18] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33:427–469, December 2001.

[19] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication providing generalized snapshot isolation. In *24th Intnl. Symp. on Reliab. Distrib. Syst. (SRDS)*, pages 73–84, Orlando, FL, USA, October 2005. IEEE-CS Press.

[20] Javier Esparza Peidro, Francesc D. Muñoz-Escoí, Luis Irún-Briz, and José M. Bernabéu-Aubán. RJDBC: A simple database replication engine. In *ICEIS (1)*, pages 587–590, 2004.

[21] Alan Fekete. Allocating isolation levels to transactions. In *24th Intnl. Symp. Princ. Database Syst. (PODS)*, pages 206–215, New York, NY, USA, 2005. ACM Press.

[22] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30:492–528, June 2005.

[23] Alan Fekete, Elizabeth J. O'Neil, and Patrick E. O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Record*, 33(3):12–14, 2004.

[24] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Intnl. Conf. on Manag. Data (SIGMOD)*, pages 173–182, Montreal, Quebec, Canada, June 1996. ACM Press.

[25] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, 1997.

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

[27] INCITS. *ANSI/INCITS 135-1992 (R1998).- Information Systems - Database Language - SQL*. InterNational Committee for Information Technology Standards, 1101 K Street NW, Suite 610, Washington, DC 20005, USA, January 1992.

[28] Luis Irún-Briz, Hendrik Decker, Rubén de Juan-Marín, Francisco Castro-Company, José Enrique Armendáriz-Iñigo, and Francesc D. Muñoz-Escoí. MADIS: A slim middleware for database replication. *Lect. Notes Comput. Sc.*, 3648:349–359, August 2005.

[29] Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Gustavo Alonso. Non-intrusive, parallel recovery of replicated data. In *21st Intnl. Symp. on Reliab. Distrib. Syst. (SRDS)*, pages 150–159, Osaka, Japan, October 2002. IEEE-CS Press.

[30] John Gantz and David Reinsel. The digital universe study: Extracting value from chaos. http://www.emc.com/leadership/programs/digital-universe.htm, June 2011.

[31] J. R. Juárez-Rodríguez, J. E. Armendáriz-Iñigo, J. R. González de Mendívil, F. D. Muñoz-Escoí, and J. R. Garitagoitia. Weak voting database replication protocols providing different isolation levels. In *7th Intnl. Conf. on New Techn. of Distrib. Syst. (NOTERE)*, pages 261–268, Marrakesh, Morocco, June 2007.

[32] B. Kemme. *Database Replication for Clusters of Workstations*. PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 2000.

[33] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. Database Syst.*, 25(3):333–379, September 2000.

[34] Bettina Kemme, Alberto Bartoli, and Özalp Babaoglu. Online reconfiguration in replicated databases based on group communication. In *Intntl. Conf. on Depend. Syst. and Netw. (DSN)*, pages 117–130, Göteborg, Sweden, July 2001. IEEE-CS Press.

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

[36] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009.

[37] Gilles M. E. Lafue. Semantic integrity dependencies and delayed integrity checking. In *Eigth International Conference on Very Large Data Bases, September 8-10, 1982, Mexico City, Mexico, Proceedings*, pages 292–299. Morgan Kaufmann, 1982.

[38] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Intnl. Conf. on Manag. of Data (SIGMOD)*, pages 419–430, 2005.

[39] Yi Lin, Bettina Kemme, Ricardo Jiménez-Peris, Marta Patiño Martínez, and José Enrique Armendáriz-Iñigo. Snapshot isolation and integrity constraints in replicated databases. *ACM Trans. Database Syst.*, 34(2):11:1–11:49, July 2009.

[40] Yi Lin, Bettina Kemme, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and José Enrique Armendáriz-Iñigo. Snapshot isolation and integrity constraints in replicated databases. *ACM Trans. Database Syst.*, 34(2), 2009.

[41] François Llirbat, Eric Simon, and Dimitri Tombroff. Using versions in update transactions: Application to integrity checking. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB*, pages 96–105. Morgan Kaufmann, 1997.

[42] Microsoft Corp. URL: http://www.microsoft.com/sqlserver/en/us/default.aspx, May 2011.

[43] Francesc D. Muñoz, J. Pla, María Idoia Ruiz, Luis Irún, Hendrik Decker, José Enrique Armendáriz, and J. R. González de Mendívil. Managing transaction conflicts in middleware-based database replication architectures. In *Intnl. Symp. Reliab. Distrib. Syst. (SRDS)*, Leeds, UK, October 2006. IEEE-CS Press.

[44] Oracle Corp. MySQL, developer zone. URL: http://dev.mysql.com/, September 2011.

[45] Oracle Corp. Oracle Database 11*g*. URL: http://www.oracle.com/us/products/database/index.html, September 2011.

[46] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distrib. and Paral. Databases*, 14(1):71–98, 2003.

[47] Frank M. Pittelli and Hector Garcia-Molina. Reliable scheduling in a TMR database system. *ACM Trans. Comput. Syst.*, 7(1):25–60, January 1989.

[48] PostgreSQL Global Development Group. PostgreSQL, the world's most advanced open source database. URL: http://www.postgresql.org/, September 2011.

[49] M. I. Ruiz-Fuertes, R. de Juan-Marín, J. Pla-Civera, F. Castro-Company, and F. D. Muñoz-Escoí. A metaprotocol outline for database replication adaptability. In *2nd Intnl. Wshop. on Reliab. in Decentr. Distrib. Syst. (RDDS)*, pages 1052–1061, Vilamoura, Portugal, November 2007. Springer.

[50] M. I. Ruiz-Fuertes, F. D. Muñoz-Escoí, H. Decker, J. E. Armendáriz-Iñigo, and J. R. González de Mendívil. Integrity dangers in certification-based replication protocols. In *3rd Intnl. Wshop. on Reliab. in Decentr. Distrib. Syst. (RDDS)*, pages 924–933, Monterrey, Mexico, November 2008. Springer.

[51] M. Idoia Ruiz-Fuertes. *On the Consistency, Characterization, Adaptability and Integrity of Database Replication Systems.* PhD thesis, Universidad Politécnica de Valencia, September 2011.

[52] María Idoia Ruiz-Fuertes and Francesc D. Muñoz-Escoí. Performance evaluation of a metaprotocol for database replication adaptability. In *SRDS*, pages 32–38. IEEE, 2009.

[53] R. Salinas-Monteagudo, J. M. Bernabé-Gisbert, F. D. Muñoz-Escoí, J. E. Armendáriz-Iñigo, and J. R. González de Mendívil. SIRC: A multiple isolation level protocol for middleware-based data replication. In *22nd Intnl. Symp. on Comput. Inf. Sc. (ISCIS)*, Ankara, Turkey, November 2007. IEEE-CS Press.

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

[55] Michael Stonebraker and Rick Cattell. 10 rules for scalable performance in 'simple operation' datastores. *Commun. ACM*, 54(6):72–80, June 2011.

[56] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, and Bruce G. Lindsay. Transactions and consistency in distributed database systems. *ACM Trans. Database Syst.*, 7:323–342, September 1982.

[57] Transaction Processing Performance Council (TPC). TPC benchmark C. Standard Specification, 2005.

[58] S. Verma, M. L. Mcauliffe, S. Listgarten, S. Haldar, and C. K. Hoang. *Patent 7243088: Database management system with efficient version control.* Oracle Intnl. Corp., July 2007.

[59] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*, pages 464 –474, 2000.

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

[61] Matthias Wiesmann, André Schiper, Fernando Pedone, Bettina Kemme, and Gustavo Alonso. Database replication techniques: A three parameter classification. In *19th Intnl. Symp. on Reliab. Distrib. Syst. (SRDS)*, pages 206–215, Nürnberg, Germany, 2000. IEEE-CS Press.

[62] Vaidė Zuikevičiūtė and Fernando Pedone. Correctness criteria for database replication: Theoretical and practical aspects. In *Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems:*, OTM '08, pages 639–656, Berlin, Heidelberg, 2008. Springer-Verlag.