UNIVERSIDAD POLITÉCNICA DE VALENCIA

On the Consistency, Characterization, Adaptability and Integrity of Database Replication Systems

by

María Idoia Ruiz Fuertes

under the supervision of
Francisco Daniel Muñoz Escoí

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

to the
Departamento de Sistemas Informáticos y Computación

July, 2011

“It always takes longer than you think even when you take Hofstadter’s Law into account.”

Hofstadter’s Law

Abstract

Since the appearance of the first distributed databases until the current modern replication systems, the research community has proposed multiple protocols to manage data distribution and replication, along with concurrency control algorithms to handle transactions running at every system node. Many protocols are thus available, each one with different features and performance, and guaranteeing different consistency levels. To know which replication protocol is the most appropriate, two aspects must be considered: the required level of consistency and isolation (i.e., the correctness criterion), and the properties of the system (i.e., the scenario), which will determine the achievable performance.

Regarding correctness criteria, one-copy serializability is broadly accepted as the highest level of correctness. However, its definition allows different interpretations regarding replica consistency. In this thesis, we establish a correspondence between memory consistency models, as defined in the scope of distributed shared memory, and possible levels of replica consistency, thus defining new correctness criteria that correspond to the identified interpretations of one-copy serializability.

Once selected the correctness criterion, the achievable performance of a system heavily depends on the scenario, i.e., the sum of both the system environment and the applications running on it. In order for the administrator to select a proper replication protocol, the available protocols must be fully and deeply known. A good description of each candidate is fundamental, but a common ground is mandatory to compare the different options and to estimate their performance in the given scenario. This thesis proposes a precise characterization model that allows us to decompose algorithms into individual interactions between significant system elements, as well as to define some underlying properties, and to associate

each interaction with a specific policy that governs it. We later use this model as basis for a historical study of the evolution of database replication techniques, thus providing an exhaustive survey of the principal existing systems.

Although a specific replication protocol may be the best option for certain scenario, as systems are dynamic and heterogeneous, it is difficult for a single protocol to continuously be the proper choice, as it may degrade or be unable to meet all requirements. In this thesis we propose a metaprotocol that supports several replication protocols which follow different replication techniques and may provide different isolation levels. With this metaprotocol, replication protocols can either work concurrently with the same data or be sequenced for adapting to dynamic environments.

Finally we consider integrity constraints, which are extensively used in databases to define semantic properties of data but are often forgotten in replicated databases. We analyze the potential problems this may involve and provide simple guidelines to extend a protocol so that it notices and properly manages abortions due to integrity violations.

Resumen

Desde la aparición de las primeras bases de datos distribuidas hasta los actuales sistemas de replicación modernos, la comunidad de investigación ha propuesto múltiples protocolos para administrar la distribución y replicación de datos, junto con algoritmos de control de concurrencia para manejar las transacciones en ejecución en todos los nodos del sistema. Muchos protocolos están disponibles, por tanto, cada uno con diferentes características y rendimiento, y garantizando diferentes niveles de coherencia. Para saber qué protocolo de replicación es el más adecuado, dos aspectos deben ser considerados: el nivel necesario de coherencia y aislamiento (es decir, el criterio de corrección), y las propiedades del sistema (es decir, el escenario), que determinará el rendimiento alcanzable.

Con relación a los criterios de corrección, la serialización de una copia es ampliamente aceptada como el más alto nivel de corrección. Sin embargo, su definición permite diferentes interpretaciones en cuanto a la coherencia de réplicas. En esta tesis se establece una correspondencia entre los modelos de coherencia de memoria, tal como se definen en el ámbito de la memoria compartida distribuida, y los posibles niveles de coherencia de réplicas, definiendo así nuevos criterios de corrección que corresponden a las diferentes interpretaciones identificadas sobre la serialización de una copia.

Una vez seleccionado el criterio de corrección, el rendimiento alcanzable por un sistema depende en gran medida del escenario, es decir, de la suma del entorno del sistema y de las aplicaciones que se ejecutan en él. Para que el administrador pueda seleccionar un protocolo de replicación apropiado, los protocolos disponibles deben conocerse plena y profundamente. Una buena descripción de cada candidato es fundamental, pero un marco común es imperativo para comparar

las diferentes opciones y estimar su rendimiento en un escenario dado. Esta tesis propone un modelo de caracterización precisa que nos permite descomponer los algoritmos en interacciones individuales entre los elementos significativos del sistema, así como en algunas propiedades subyacentes, y asociar cada interacción con una política específica que la rige. Más tarde se utiliza este modelo como base para un repaso histórico de la evolución de las técnicas de replicación de bases de datos, proporcionando así un estudio exhaustivo de los principales sistemas existentes.

Aunque un cierto protocolo de replicación puede ser la mejor opción para un escenario determinado, los sistemas son dinámicos y heterogéneos, y por tanto es difícil que un único protocolo sea continuamente la elección correcta, ya que puede degradarse o puede no llegar a satisfacer todas las necesidades. En esta tesis se propone un metaprotocolo que soporta varios protocolos de replicación que siguen diferentes técnicas y pueden proporcionar diferentes niveles de aislamiento. Con este metaprotocolo, los protocolos de replicación puede trabajar simultáneamente con los mismos datos o intercambiarse para adaptarse a entornos dinámicos.

Por último se tienen en cuenta las restricciones de integridad, que son ampliamente utilizadas en bases de datos para definir las propiedades semánticas de los datos, pero son a menudo olvidadas en bases de datos replicadas. Se analizan los posibles problemas que esto puede implicar y se ofrecen pautas sencillas para ampliar un protocolo de forma que identifique y gestione adecuadamente los abortos causados por violaciones de integridad.

Résumé

Depuis l'apparition des premières bases de données distribuées jusqu'aux systèmes de réplication actuels, la communauté des chercheurs a proposé plusieurs protocoles pour gérer la distribution des données et la réplication, ainsi que des algorithmes de contrôle de concurrence pour traiter les opérations en cours d'exécution dans chaque noeud du système. De nombreux protocoles sont donc disponibles, chacun avec différentes caractéristiques et performances, garantissant des niveaux différents de cohérence. Pour savoir quel protocole de réplication est le plus approprié, deux aspects doivent être considérés : le niveau requis de cohérence et isolement (c'est à dire, le critère de correction), et les propriétés du système (le scénario), qui déterminera la performance atteignable.

En ce qui concerne aux critères de correction, la sérialisabilité d'une copie est généralement considéré comme le plus haut niveau de correction. Toutefois, sa définition permet des interprétations différentes quant à la cohérence de réplicas. Dans cette thèse, nous établissons une correspondance entre les modèles de cohérence de mémoire, tels que définis dans le champ de la mémoire partagée distribuée, et les niveaux possibles de cohérence de répliques, établissant ainsi de nouveaux critères de correction qui correspondent aux interprétations identifiées de sérialisabilité d'une copie.

Une fois choisi le critère de correction, le rendement possible d'un système dépend en grande partie du scénario, à savoir, la somme de l'environnement du système et des applications s'exécutant dessus. Afin que l'administrateur sélectionne un protocole de réplication approprié, les protocoles disponibles doivent être pleinement et profondément connus. Une bonne description de chaque candidat est fondamental, mais une base commune est impérative pour comparer les différentes options et évaluer leurs rendements dans le scénario donné. Cette

thèse propose un modèle de caractérisation précise qui nous permet de décomposer les algorithmes en les interactions individuelles entre les éléments majeurs du système ainsi qu'en certaines propriétés sous-jacentes et d'associer chaque interaction à une politique spécifique qui la régit. Nous avons ensuite utilisé ce modèle comme base pour une révision historique de l'évolution des techniques de réplication de bases de données, fournissant ainsi une étude exhaustive des principaux systèmes existants.

Même si un protocole de réplication spécifique peut être la meilleure option pour un certain scénario, comme les systèmes sont dynamiques et hétérogènes, il est difficile pour un protocole unique d'être continuellement le bon choix, car il peut se dégrader ou être incapable de satisfaire tous les besoins. Dans cette thèse nous proposons un metaprotocole qui soutient plusieurs protocoles de réplication qui suivent différentes techniques de réplication et peuvent fournir différents niveaux d'isolement. Avec ce metaprotocole, les protocoles de réplication peuvent travailler simultanément avec les mêmes données ou être séquencés pour s'adapter à des environnements dynamiques.

Enfin nous considérons les contraintes d'intégrité, qui sont largement utilisées dans les bases de données pour définir les propriétés sémantiques des données mais sont souvent oubliées dans les bases de données répliquées. Nous analysons les problèmes potentiels de celà et fournisons des lignes directrices simples pour étendre un protocole de sorte qu'il gère correctement les avortements en raison de violations de l'intégrité.

Resumo

Desde o aparecimento das primeiras bases de dados distribuídas até aos actuais sistemas de replicação modernos, a comunidade científica propôs múltiplos protocolos para gerir a distribuição dos dados e a replicação, juntamente com algoritmos de controlo de concorrência para lidar com transações em execução em cada nó do sistema. Muitos protocolos estão, portanto, disponíveis, cada um com diferentes características e desempenho, e garantindo níveis de coerência diferentes. Para saber qual o protocolo de replicação mais adequado, dois aspectos devem ser considerados: o nível de coerência e isolamento (ou seja, o critério de correção), e as propriedades do sistema (ou seja, o cenário), que irá determinar o desempenho alcançável.

Quanto aos critérios de correção, a serializabilidade de uma cópia é amplamente aceite como o mais alto nível de correção. No entanto, sua definição permite diferentes interpretações quanto á coerência de réplicas. Nesta tese, nós estabelecemos uma correspondência entre os modelos de coerência de memória, tal como definidos no âmbito da memória partilhada distribuída e os possíveis níveis de coerência de réplicas, definindo novos critérios de correção, que correspondem às interpretações identificadas da serializabilidade de uma cópia.

Uma vez selecionado o critério de correção, o desempenho possível de um sistema depende muito do cenário concreto, ou seja, a combinação do ambiente do sistema com as aplicações que nele correm. Para que o administrador selecione um protocolo de replicação adequado, os protocolos disponíveis devem ser plena e profundamente conhecidos. Uma boa descrição de cada um dos candidatos é fundamental, mas um terreno comum é imperativo para comparar as diferentes opções e avaliar o seu desempenho no cenário concreto. Esta tese propõe um modelo de caracterização precisa que nos permite decompor os algoritmos em

interações individuais entre os elementos significativos do sistema, bem como em algumas propriedades subjacentes, e associar cada interação com uma política específica que a rege. Mais tarde, usamos este modelo como base para uma revisão histórica da evolução das técnicas de replicação de bases de dados, proporcionando assim um estudo exaustivo dos principais sistemas existentes.

Apesar de um protocolo de replicação específico poder ser a melhor opção para um determinado cenário, como os sistemas são dinâmicos e heterogéneos, é difícil que um único protocolo seja continuamente a escolha adequada, pois pode degradar-se ou ser incapaz de cumprir todos os requisitos. Nesta tese propomos um metaprotocolo que suporta vários protocolos que seguem diferentes técnicas de replicação e podem proporcionar diferentes níveis de isolamento. Com este metaprotocolo, os protocolos de replicação podem trabalhar simultaneamente com os mesmos dados ou ser sequenciados para se adaptar a ambientes dinâmicos.

Finalmente, consideramos as restrições de integridade, que são amplamente utilizadas em bases de dados para definir as propriedades semânticas dos dados, mas muitas vezes são esquecidas em bases de dados replicadas. Analisamos os problemas potenciais que isso pode envolver e fornecemos orientações simples para estender um protocolo para que ele perceba e gere corretamente os abortos devidos a violações de integridade.

Resum

Des de l'aparició de les primeres bases de dades distribuïdes fins als actuals sistemes de replicació moderns, la comunitat d'investigació ha proposat diversos protocols per a administrar la distribució i replicació de dades, juntament amb algorismes de control de concurrència per gestionar les transaccions en execució en tots els nodes del sistema. Molts protocols estan disponibles per tant, cadascun amb diferents característiques i rendiment, i garantint diferents nivells de coherència. Per saber quin protocol de replicació és el més adequat, dos aspectes han de ser considerats: el nivell necessari de coherència i aïllament (és a dir, el criteri de correcció), i les propietats del sistema (és a dir, l'escenari), que determinarà el rendiment assolible.

Pel que fa als criteris de correcció, la serialització d'una còpia és àmpliament acceptada com el més alt nivell de correcció. No obstant això, la seua definició permet interpretacions diferents pel que fa a la coherència de rèpliques. En aquesta tesi, s'estableix una correspondència entre els models de coherència de memòria, tal com es defineixen en l'àmbit de la memòria compartida distribuïda, i els possibles nivells de coherència de rèpliques, definint així nous criteris de correcció que corresponen a les interpretacions identificades de serialització d'una còpia.

Una vegada seleccionat el criteri de correcció, el rendiment que podria obtenir un sistema depèn en gran mesura de l'escenari, és a dir, la suma de tots dos l'entorn del sistema i les aplicacions que s'executen en ell. Perquè l'administrador puga seleccionar un protocol de replicació apropiat, els protocols disponibles han de ser plenament i profunda coneguts. Una bona descripció de cada candidat és fonamental, però un marc en comú és imperatiu per a comparar les diferents opcions i estimar el seu rendiment en l'escenari donat. Aquesta tesi proposa un

model de caracterització precisa que ens permet descompondre els algorismes en les interaccions individuals entre els elements significatius dels sistemes, així com en algunes propietats subjacents, i associar cada interacció amb una política específica que ha de regir-la. Més tard, utilitzem aquest model com a base per a un repàs històric de l'evolució de les tècniques de replicació de bases de dades, proporcionant així un estudi exhaustiu dels principals sistemes existents.

Tot i que un protocol de replicació específic pot ser la millor opció per a un escenari concret, com els sistemes són dinàmics i heterogenis, és difícil que un protocol únic siga contínuament l'elecció correcta, ja que es pot degradar o pot no satisfer totes les necessitats. En aquesta tesi es proposa un metaprotocol que suporta diversos protocols de replicació que segueixen diferents tècniques de replicació i poden proporcionar diferents nivells d'aïllament. Amb aquest metaprotocol, els protocols de replicació poden treballar simultàniament amb les mateixes dades o ser seqüenciats per adaptar-se a entorns dinàmics.

Finalment es tenen en compte les restriccions d'integritat, que són àmpliament utilitzades en bases de dades per definir les propietats semàntiques de les dades, però són sovint oblidades en bases de dades replicades. S'analitzen els possibles problemes que això pot implicar i s'ofereixen pautes senzilles per ampliar un protocol perquè gestione adequadament els avortaments causats per violacions d'integritat.

Acknowledgments

First of all I want to express my gratitude to my advisor, Francisco D. Muñoz-Escoí, for his continuous support and invaluable guidance during these years since he offered me a position at his group. His professionalism, sensibleness and responsibility have been a role model for me.

I also would like to thank André Schiper and Rui Oliveira for their warm welcome into their respective research groups at the École Polytechnique Fédérale de Lausanne and the Universidade do Minho. Those stays gave me the opportunity to consider the topic of my work from other interesting and enriching points of view.

For their insightful remarks, I would like to thank many anonymous reviewers that contributed to greatly improve the quality of the papers I coauthored. I also want to give Simin Nadjm-Tehrani, Esther Pacitti, and Marta Patiño-Martínez special acknowledgment for their dedication to reviewing the draft of this thesis.

I will keep friendly and warm memories of my lab mates at the Instituto Tecnológico de Informática, as well as of those from the Laboratoire de Systèmes Répartis and the Grupo de Sistemas Distribuídos.

Last but not least, my gratitude goes to the nearest and dearest people in my life, my parents and Rodrigo, for their love, patience and support.

Preface

This thesis integrates several lines of work of my PhD research done under the supervision of Francisco D. Muñoz-Escoí at the *Instituto Tecnológico de Informática* of Valencia, Spain, from 2004 to 2011. During this period, I was involved in three National Research Projects devoted to database replication and distributed systems: MADIS[1], CONDEP[2] and IDEA.[3] The last four years I held an FPI fellowship[4] granted by the Spanish *Ministerio de Educación y Ciencia*.

Some results presented in this thesis were previously published in conference proceedings:

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 *Proceedings (Part II) of the OTM Confederated International Workshops and Posters*, volume 4806 of *Lecture Notes in Computer Science (LNCS)*, pages 1052–1061, Vilamoura (Portugal), November 2007. Springer-Verlag.

M. I. Ruiz-Fuertes, F. D. Muñoz-Escoí, and H. Decker. Integrity Constraint Management in Certification-Based Replication Protocols. In *XVI Jornadas de Concurrencia y Sistemas Distribuidos (JCSD)*, pages 217–229, Albacete (Spain), June 2008.

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

[1]Founded by the EU FEDER and the Spanish MCYT, under grant TIC2003-09420-C02.
[2]Founded by the EU FEDER and the Spanish MEC, under grant TIN2006-14738-C02.
[3]Founded by the EU FEDER and the Spanish MICINN, under grant TIN2010-17193.
[4]*Formación de Personal Investigador*, grant BES-2007-17362.

Replication Protocols. In *Proceedings of the OTM Confederated International Workshops and Posters*, volume 5333 of *Lecture Notes in Computer Science (LNCS)*, pages 924–933, Monterrey (Mexico), November 2008. Springer-Verlag.

F. D. Muñoz-Escoí, M. I. Ruiz-Fuertes, H. Decker, J. E. Armendáriz-Iñigo, and J. R. González de Mendívil. Extending Middleware Protocols for Database Replication with Integrity Support. In *Proceedings (Part I) of the OTM Confederated International Conferences*, volume 5331 of *Lecture Notes in Computer Science (LNCS)*, pages 607–624, Monterrey (Mexico), November 2008. Springer-Verlag.

M. I. Ruiz-Fuertes and F. D. Muñoz-Escoí. Study of a Metaprotocol for Database Replication Adaptability. In *XVII Jornadas de Concurrencia y Sistemas Distribuidos (JCSD)*, pages 125–138, Sagunto, Valencia (Spain), June 2009.

M. I. Ruiz-Fuertes and F. D. Muñoz-Escoí. Performance Evaluation of a Metaprotocol for Database Replication Adaptability. In *Proceedings of the 28th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 32–38, Niagara Falls, New York (USA), September 2009. IEEE-CS Press.

M. I. Ruiz-Fuertes, J. M. Bernabé-Gisbert, R. de Juan-Marín, and F. D. Muñoz-Escoí. 1-Copy Equivalence: Atomic Commit versus Atomic Broadcast in Termination Protocols. In *XVIII Jornadas de Concurrencia y Sistemas Distribuidos (JCSD)*, pages 19–29, Núria, Gerona (Spain), June 2010.

Contents

Figures

Tables

Chapter 1

Introduction

Database systems have always been focus of special attention, as they constitute an element of primordial interest at any organization, due to the necessity of both ensuring data consistency and offering a good performance regarding availability, as well as guaranteeing the correct running of the system even when some component fails.

Replication is the common solution to achieve high availability and fault tolerance. To provide a database system with these properties, replicas –copies– of the data are stored at different nodes. This way, distributed and replicated databases appeared, as a replacement of the existing centralized, stand-alone systems which had as main drawbacks their fragility (a single point of failure) and poor performance.

In a replicated database, updates made to any of the copies of a data item must be broadcast to the rest of replicas in order to maintain consistency. This process is managed by a replication or consistency protocol. Since the appearance of the first distributed databases until the current modern replication systems, the research community has proposed multiple protocols to manage data distribution and replication, along with concurrency control algorithms to handle transactions running at every system node. Therefore, many protocols are available, each one with different features and different performance, and guaranteeing a different consistency level. The question that raises immediately is: which replication protocol is the most appropriate? Two aspects must be considered to answer such a question: on one hand, the required level of isolation and consistency; on

the other hand, the properties of the system that will determine the achievable performance.

The required consistency level depends on the needs of the application and the nature of the data. Intuitively, a replicated system with n copies of the data, an n-copy system, should behave as a non-replicated, one-copy system. This is known as the one-copy equivalence (1CE) property. Besides replica consistency, a certain isolation level must be enforced to meet the requirements of the application. The combination of both aspects defines the correctness criterion that must be guaranteed by the replication system. In early replicated distributed database systems, correctness was usually defined as one-copy serializability: a database spread over multiple sites must behave as a unique node, and the result of concurrently executing transactions must be equivalent to that of a serial execution of the same set of transactions. In order to ensure such guarantees, concurrency was managed by distributed locking, and atomic commit protocols controlled transaction termination. Such systems, however, suffered from low performance and poor scalability due to the high cost of the protocols in use. Research focused then on improving performance and scalability while maintaining the correctness criterion of one-copy serializability. Atomic broadcast was proposed as a better alternative to atomic commit protocols, and transactions were locally executed before being broadcast to the rest of replicas. Outstanding performance improvements were achieved. However, enforced guarantees were subtly but significantly modified. This thesis carefully analyzes both solutions and states the different guarantees they provide by establishing a correspondence with memory consistency models as defined in the scope of distributed shared memory. This first contribution allows the proposal of new correctness criteria that will be used throughout the thesis.

Once a certain correctness criterion is selected, the achievable performance of a system heavily depends on the scenario, i.e., the sum of both the system environment and the applications running on it. Many different characteristics should be considered when selecting the replication protocol. Thus, regarding the environment, two important factors are the network latency (fast or slow connections) and the workload (low, high or with peaks of activity). When analyzing the applications, factors as the conflict rate (low, if most of the transactions are read-only, or high, if most of the transactions are updates), the access pattern (hot spots in the database, high locality of data) or the transaction length (short, long) should be considered in order to decide the best protocol for each scenario.

For example, in a system with slow connections, the network may act as a bottleneck. This forces the administrator to select a replication protocol that minimizes the number of messages sent. At least one message per transaction is needed (a constant number of messages higher than 1 may be also bearable), leading to the selection of a constant interaction protocol. On the other hand, if the network is fast, there is no need to save messages and any (reasonable) number of them can be sent, e.g., depending on the transaction length. In this case, a linear interaction protocol can be more interesting. In general, depending on the characteristics of the environment, the administrator can choose one replication protocol as the most suitable.

Considering the applications running in the system, an important factor is the transaction length. The probability of abortion of a transaction is proportional to the probability of overlapping with other transactions in the system, which, in turn, is proportional to the length of transactions. This way, an application with long transactions will present a high probability of overlapping and, thus, of abortions. To abort a long transaction implies that all the invested time was wasted and all the performed work must be rolled back. For this reason, in long-transaction applications it is preferable to use a pessimistic replication that avoids abortions at the end of transactions by preventing concurrency between potentially conflicting transactions. As a consequence of reducing concurrency, the throughput of the system decreases. On the other hand, in a short-transaction application, the probability of overlapping is low and, even if transactions overlap and some conflicts appear, aborting and retrying transactions is bearable due to their short duration. In this case, an optimistic replication, which allows transactions to execute concurrently and solves conflicts by aborting the necessary transactions at the end, achieves lower response times and higher throughput. In general, depending on the requirements of the client application, the administrator can choose one replication protocol as the most suitable.

However, in order for the administrator to make such election, the available protocols must be fully and deeply known. A good description of each candidate is fundamental, but a common ground is mandatory to compare the different options and to estimate their performance in the given scenario. The second contribution of this thesis is to provide such common ground in the form of a precise characterization model for replication protocols. This model, based on policies and strategies, allows us to decompose algorithms into individual interactions between significant system elements, as well as to define some underlying properties, and to associate each interaction with a specific policy that governs it. Thus,

a replication system can be described as a combination of policies. This common framework allows an easy understanding and comparison between protocols and is later used as basis for a historical study of the evolution of database replication techniques, thus providing, as third contribution, an exhaustive survey of the principal existing systems, where more than 50 different systems are characterized in detail following our model.

Even provided with a framework to compare replication techniques, the common approach followed by the designers of database replication systems is to analyze their general scenario *once* and accordingly elect a *single* replication protocol that will *permanently* remain in their systems. Such combination of a single analysis and a permanent decision over a unique protocol constitutes an approach that is perfect if everything is static and homogeneous. But systems are dynamic: environments evolve changing their characteristics, which can drastically decrease the performance of the elected protocol. This is the first target of an adaptable system: as the environment changes, the replication system should adapt to the new situation. But dynamism in the environment is not the only problem: applications are also dynamic and undergo updates which may modify or increase the requirements of the application; and systems are heterogeneous and multiple applications or procedures can concurrently access the same database. For example, when a long- and a short-transaction applications execute concurrently in the system, long and short transactions must be completed. If optimistically managed, long transactions may always abort due to conflicts with faster short ones. If pessimistically replicated, short transactions may last much longer. This is the second target of an adaptable system: when facing changing and different (even opposite) concurrent requirements, the replication system should adapt to meet all of them as much as possible.

This way, the common static, single-protocol approach lacks flexibility for changing scenarios or when dealing with heterogeneous client application requirements. In those cases, the initially chosen protocol degrades or cannot meet all requirements. The reason for this is that dynamic and heterogeneous scenarios require a highly adaptable replication. The fourth contribution of this thesis is the proposal of a metaprotocol that supports several replication protocols which may follow different replication techniques or provide different isolation levels. Within this metaprotocol, the replication protocols can either work concurrently with the same data or be sequenced for adapting to dynamic environments. Experimental results demonstrate the low overhead of the metaprotocol and measure the influence of protocol concurrency on system performance.

The last contribution of this thesis deals with integrity constraints, an important aspect which is widely forgotten in replicated databases. Integrity constraints are extensively used in databases to define semantic properties of data. If a transaction violates any of such constraints, the database management system (DBMS) unmercifully aborts it. In a replicated database, this abortion may happen after the middleware replication protocol decided (and assumed) the commitment of the transaction. While integrity is guaranteed by the DBMS, it is possible that the protocol does not notice the unexpected abortion, which can lead protocols to lose their liveness and make mistakes when processing subsequent transactions. In this thesis, we argue that a replication protocol must recognize abortions due to integrity violations, and react appropriately.

1.1 Contributions and Outline of this Thesis

Basic concepts and definitions are presented in chapter 2. Remaining chapters are devoted to each of the contributions of this thesis, which are summarized here:

- New correctness criteria that explicitly state the enforced isolation level and the guaranteed replica consistency (Chapter 3).
- A characterization model to describe in detail and compare database replication systems (Chapter 4).
- An exhaustive analysis of database replication systems proposed since the origin of this research branch, with a complete description according to the previous model and the specification of the exact correctness criteria guaranteed (Chapter 5).
- A metaprotocol that provides database replication systems with the required adaptability by allowing the activation and deactivation, as well as the concurrent execution, of different replication protocols, defined accordingly to the previous model (Chapter 6).
- A methodology for supporting integrity constraints in middleware database replication protocols (Chapter 7).

After the thesis conclusions in Chapter 8, Appendix A discusses about the correctness of the metaprotocol, whose complete pseudocode is included in Appendix B.

Chapter 2

Concepts and Definitions

All necessary background concepts are defined in this chapter.

Server, clients and failures A distributed (and possibly replicated) database is a distributed system composed of database *servers*, also called *nodes* or *sites*, $N_1, N_2, \ldots, N_n$, which store data, and *clients* or *users*[1] that contact these servers to access the data. Servers only communicate by message passing, as they do not have shared memory. Consequently, they neither have a global clock. Depending on the *failure model* assumed, failures are considered at different degrees. In the *crash-stop* model, when servers fail they permanently stop all their processing. In this case, a site is *correct* if it never crashes, otherwise it is *faulty*. The *crash-recovery* model allows failed servers to eventually recover after a crash. A more complex failure model, *Byzantine failures*, assumes that sites and their environment can behave in an arbitrary way.

Distribution and replication From the point of view of the client, a *database* is a collection of logical data items. Each logical data item is physically stored at the servers. If the set of database items is partitioned and distributed among the sites, the system constitutes a *purely distributed database*. If some replication is introduced, so that different physical copies of the same logical database item are stored at different sites, the system is a *replicated database*. Replication is

[1] We employ the term *user* to specifically refer to a human agent.

managed by a *replication protocol*. The number of physical copies of data item x is the *degree of replication* of x. Depending on the degree of replication of the data items and on the number of system nodes, a complete copy of the database may be stored at each site. This is called *full replication*. When not all sites store the complete set of data items, but only a subset, the replication is *partial*. Each of the copies of a given data item is called a *replica*. In full replication, the term replica is also used to refer to any server, as they contain a copy of all data items. When the system is fully replicated, we also denote sites as $R_1, R_2, \ldots, R_n$, to highlight that they are replicas.

Advantages of replication Replication is a common solution to achieve *availability*: by storing multiple copies, clients can operate even if some sites have failed. Moreover, and despite the drawback of having to update all copies of each data item, replication can also improve performance, as clients are more likely to access a copy close by. For these reasons, replication is mostly preferred over pure distribution. Even initial systems, which were fundamentally distributed, introduced replication at some degree.

Local databases and transactions A local *Database Management System*, or DBMS for short, executes at each site and is responsible for the control of the data. The DBMS allows clients to access the data for reading and writing, through the use of *transactions*. Transactions are sequences of read and write operations (e.g., sequences of SQL sentences) followed by a commit or an abort operation, and maintain the ACID properties [55]: atomicity, consistency, isolation and durability. When a transaction *commits*, all its data changes are persistently applied. If it *aborts*, however, all its changes are undone. A transaction is called a *query* or a *read-only transaction* if it does not contain any write operation; otherwise it is called an *update transaction*. The set of logical items a transaction reads is called the *readset*. Similarly, the *writeset* is the set of logical items written by a transaction, and usually it also includes the updated or inserted values. The *resultset* is compound by the results that will be returned to the client.

Transactions may execute concurrently in a DBMS. In this case, the concurrency control of the DBMS establishes which executions of concurrent transactions are correct or legal.

Workload, delegate and remote nodes The database workload is composed of transactions, $T_1, T_2, \ldots$ Transactions from the same client session are submitted sequentially, but may be addressed to different servers, either by the client itself, by a load balancer, or by another component or server that redirects the request to another server. If the request is finally addressed to only one server, this server is called the *delegate* for that transaction and it is responsible for its execution. The rest of the system nodes are called *remote* nodes for that transaction. When the database is purely distributed (partitions of the data stored at different nodes) or partial replication is used, if the contacted server does not store all the items the transaction needs to access, other servers can be requested to execute different portions of the transaction, which is then called a *distributed transaction*. In this case, the concepts of delegate and remote nodes are no longer applicable: the client contacts to a server where a root transaction is started, and accesses to data stored in other server involve the creation of a subtransaction in that other node, which is also called *cohort*. This way, each operation may involve a communication with another node in the system. If accessed items are replicated, subtransactions must be executed in all copies. To coordinate the local subtransactions executed at each participating site, all accesses are managed by a distributed concurrency control. In order to commit these distributed transactions, atomic commit protocols, explained below, are used, acting the root transaction as coordinator.

Interactive execution vs. service request A transaction can be submitted for execution either operation by operation, or in a single message. In the former case, called *interactive transaction*, the client submits an operation and waits for its results before sending the next operation. The latter case, called *service request*, is a call to a procedure stored in the database. When the transaction is completed, the transaction outcome is sent to the client. In the case of interactive transactions, this outcome is a commit or abort confirmation. For service requests, the outcome also includes the results of the request.

Server interaction and writeset application In the presence of replication, updates from a committed transaction must be propagated to other copies of the affected data items. This is achieved through a mechanism of *server interaction* that sends all write operations to the appropriate sites. This propagation can be made on a per operation basis, distributing writes immediately to other nodes in a *linear interaction* [138] approach; or deferring communication until transaction

end, in a *constant interaction* approach. The former case requires more messages (usually one per write operation) than the latter (one message per transaction). The *deferred update* approach [130] is a constant interaction approach, where, once a transaction finishes its operations, its writeset is sent to the appropriate remote nodes, which will then *apply the writeset*, i.e. perform its updates, in their local database copy. Depending on when this propagation and application of updates is made, systems can be *eager* or *lazy* [52]. Eager replication ensures that every node applies the updates inside the transaction boundaries, i.e., before the results are sent back to the client, so that all replicas are updated before such a response is sent. On the other hand, lazy replication algorithms asynchronously propagate updates to other nodes after the transaction commits in its delegate node. An hybrid approach ensures that all replicas have received the updates and the delegate has committed them when the results are sent to the client. Remote nodes will later apply such updates.

Update propagation: ROWA and ROWAA With either type of server interaction, writes are propagated and applied in remote copies of the affected data items. A basic approach is *Read One Write All* [15], where write operations are required to update all copies so that read operations only need to access one of the replicas. In case of a site failure, it may be impossible to write all replicas and thus the processing must stop. As this is not desirable, the common approach is *Read One Write All Available* (ROWAA) [15]. According to ROWAA, each write operation over data item x is applied at every *available* copy of x, i.e., replicas stored at sites that have not failed. Failed sites are ignored until they *recover* from their failure. Whenever a site recovers from a failure, all its copies must be brought up-to-date before the node can serve read operations.

Update propagation: quorums Another approach for write propagation is the use of *quorums*. A quorum is the minimum number of replicas that is required for completing an operation. Each transaction operation must then be successfully executed in a quorum of replicas for being considered successful: read operations are required to access a read quorum of replicas before returning the read value to the client, while write operations must update a write quorum of replicas. In a system of size N, sizes for the read quorum, R, and the write quorum, W, are defined in such a way that they guarantee any required property. For example, with $R = 1$ and $W = N$, the previously presented ROWA approach is obtained. A more common quorum configuration ensures that both $W + W$ and $R + W$ are

greater than N, so that each write quorum has at least one replica in common with every read quorum and every write quorum, thus ensuring access to the last updated value and also detecting conflicting transactions. Including a majority of the nodes into each quorum allows the system to avoid system partitioning and consequent divergence.

Conflicts Two transactions *conflict* if they have conflicting operations. Two operations conflict if they are issued by different transactions, access the same data item and at least one of the operations is a write. Conflicts among transactions should be treated somehow, ensuring that the conflicting operations are executed at the same order at every replica. There are two main approaches for treating conflicts. A system is *pessimistic* or *conservative* if it avoids conflicts by establishing some locks, mutexes or other barriers over items accessed by a transaction, so that they cannot be concurrently accessed by other transactions. On the other hand, a system is *optimistic* if it lets transactions freely access items, resolving possible conflicts only when they appear or at the end of the transaction, during termination.

Replica consistency: atomic and sequential Applying the writesets and ensuring the same order for conflicting operations are necessary actions for maintaining the required level of *replica consistency*. Replica consistency measures the synchronization among the copies of the same data item, i.e., the state of replicas with regard to each other. Different levels of replica consistency may be enforced, depending on the needs of the clients. *Atomic replica consistency* is achieved when all the copies of a data item *appear* to change atomically during transaction execution. From the user's point of view, there is only one copy of the data item, and it is updated as soon as the transaction commits. Atomicity is the highest level of replica consistency. *Sequential consistency* relaxes the synchronization level in order to achieve greater performance. With sequential consistency, all the copies of a data item are updated following the same sequence of values. Intuitively, in a sequentially consistent system updates take some time to arrive to all the copies. A deeper discussion about replica consistency is provided in Chapter 3.

Serializable isolation Transactions are executed under some isolation level, which defines the visibility among operations of different concurrent transactions.

The highest isolation level is the *serializable* level, which guarantees a completely isolated execution of transactions, as if they were serially performed, one after the other. Changes made by transaction T are only visible to other transactions after the commitment of T. On the other hand, if T aborts, then its changes are never seen by any other transaction.

Read committed isolation A more relaxed level is the *read committed* isolation. Under this isolation, data read by a transaction T was written by an already committed transaction, but it is not prevented from being modified again by other concurrent transactions. Thus, these data may have already changed when T commits. For a complete discussion about isolation levels, please refer to Berenson et al. [9].

Correctness criterion The combination of the replica consistency level and the transaction isolation level guaranteed by a system is called the *correctness criterion* of the database replication system. Chapter 3 proposes new correctness criteria.

Concurrency control: locks Local isolation at a DBMS is enforced by the use of a local *concurrency control* mechanism. Concurrency control manages the operations that run in a database at the same time. There are two main options for concurrency control: locks and multiversion systems. In a lock-based system, each data item has a *lock* that regulates the accesses to the item. Operations over that item must previously obtain the corresponding lock. There are *shared* and *exclusive* locks. Shared locks are commonly used for read operations. Several transactions can obtain shared read locks and read the same item at the same time. Write operations require an exclusive lock. No other operation, shared or exclusive, is allowed over an item protected with an exclusive lock.

Concurrency control: multiversion In multiversion systems, on the other hand, simultaneous accesses to the same data item are resolved by using multiple versions of the item. Versions can be created and deleted but the value they represent is immutable: updates to a data item create a new version of the item. Although there are several versions for each item, only one is the latest: the value written by the last committed transaction that updated the item. More

recent versions correspond to transactions which are still on execution and thus these versions are not visible to other transactions. Versions generated by aborted transactions are never visible to other transactions. When a transaction starts, a timestamp or a transaction ID is assigned to it. Versions written by a transaction T are marked with the ID of T. With this timestamp information, the multiversion system can determine, for each transaction, which state or snapshot (i.e., which version of each data item) of the database it must read. Thus, a transaction that started at a particular instant t_0 has access, for each data item, to the version of that item which was the latest at time t_0, i.e., which was written by the committed transaction with the highest ID which is smaller than the ID of the reading transaction. As versions are immutable, there is no need to manage locks for read operations. This way, multiversion concurrency control lead to the appearance of a new transaction isolation level called *snapshot isolation*.[2] Some mechanism is usually needed to delete obsolete versions.

Snapshot isolation In snapshot isolation [9], transactions get a start timestamp and a snapshot of the database when they start. Transactions are never blocked attempting a read. Write operations are also reflected in the snapshot of the transaction, so that it can access the updated versions afterwards. On the other hand, updates by other transactions active after the transaction start are invisible to the transaction. When the transaction is ready to commit, a commit timestamp is assigned to it. A transaction T_1 successfully commits only if no other transaction T_2 with a commit timestamp in the interval between the start and the commit timestamps of T_1 wrote data that T_1 also wrote. Otherwise, T_1 will abort. This feature is called the *first-committer-wins* rule.

Two phase locking Serializability can be achieved by a basic *two-phase locking* (2PL) [15] protocol, where each transaction may be divided into two phases: a *growing* phase during which it obtains locks, and a *shrinking* phase during which it releases locks. Once a lock is released, no new locks can be obtained. 2PL forces all pairs of conflicting operations of two transactions to be executed in the same order and so it achieves serializability. However, using 2PL a *deadlock* may appear. This situation arises when two transactions wait for each other to release a lock. Deadlocks must be detected and one of the transactions aborted in order to remove the deadlock.

[2] Read committed isolation is also possible with a multiversion concurrency control.

Strong strict 2PL In order to avoid deadlocks and to provide other desirable properties,[3] a variant of 2PL is commonly used: the *strong strict 2PL* (presented by Bernstein et al. [15] as *strict 2PL* but refined later). In strong strict 2PL, all the locks obtained by a transaction are only released after transaction termination.

Atomic commit protocol: 2PC To ensure consistent termination of distributed transactions, database systems have traditionally resorted to an atomic commit protocol, where each transaction participant starts by voting yes or no and each site reaches the same decision about the outcome of the current transaction: commit or abort. A widely used atomic commit protocol is the two-phase commit protocol (2PC) [50, 77], which involves two rounds of messages for reaching a consensus on the termination of each transaction. 2PC can be centralized or decentralized. In the centralized approach, the coordinator first sends a message to the rest of nodes, with information about the ending transaction. Each server must then reply to the coordinator whether it agrees or not to commit the transaction. If all replies are positive, the coordinator sends a commit message and waits for acknowledgments from all the nodes. If any of the replies was negative, an abort message is sent in the second phase. The decentralized version is similar but with any server starting the process and with responses to every other server. 2PC is a blocking protocol when failures occur.

Atomic commit protocol: 3PC To support failures, a non-blocking atomic commit protocol (NB-AC) [15, 120] must be used. In these protocols, each participant reaches a decision despite the failure of other participants. A NB-AC protocol fulfills the following properties. (a) Agreement: no two participants decide different outcomes. (b) Termination: every correct participant eventually decides. (c) Validity: if a participant decides commit, then all participants have voted yes. (d) Non triviality: if all participants vote yes, and no participant fails, then every correct participant eventually decides commit. The three-phase commit protocol (3PC) [120] adds an intermediate phase to 2PC to become a non-blocking process. This new (second) phase involves sending a *precommit* message when all nodes have agreed to commit the transaction. After all servers sent their acknowledgments to this *precommit* message, the final commit message is sent. Note that when a transaction needs to abort, such a fact is identified at the end of the first phase. On the other hand, agreement on the commitment

[3] Recoverability, cascade abort avoidance and strictness [15], and also commit ordering [109].

is reached at the second phase and the commit is completed in the third phase. Thus, failures in the first phase lead to transaction abortion whilst failures in the second or third ones do not block the protocol nor prevent transaction commitment. The drawback of 3PC is its higher cost due to the extra round of messages. To solve this, Jiménez-Peris et al. proposed another NB-AC protocol that exhibits the same latency as 2PC [61].

Atomic commit protocol: Paxos Commit The Paxos Commit algorithm [51] runs a Paxos consensus algorithm on the commit/abort decision of each participant to obtain an atomic commit protocol. The result is a complete, decentralized and non-blocking algorithm which is proven to satisfy a clearly stated correctness condition (that of the Paxos algorithm [38, 74, 76, 78]).

Group communication systems and atomic broadcast The communication among system components is based on message passing. A *Group Communication System* [28], or GCS for short, is commonly used to accomplish communication tasks among servers, by choosing the communication primitive (point to point messages, multicasts, broadcasts) with the appropriate guarantees (e.g., uniform guarantees will be commonly necessary when failures must be tolerated). *Atomic broadcast* (abcast for short) is a group communication abstraction defined by the primitives $broadcast(m)$ and $deliver(m)$. Abcast satisfies the following properties [54]. (a) Validity: if a correct site broadcasts a message m, then it eventually delivers m. (b) Agreement: if a correct site delivers a message m, then every correct site eventually delivers m. (c) Integrity: for every message m, every site delivers m at most once, and only if m was previously broadcast. (d) Total Order: if two correct sites deliver two messages m and m', then they do so in the same order. Due to this last property, atomic broadcast is also known as total order broadcast.

Optimistic abcast Two optimistic variants of abcast are the *optimistic atomic broadcast* [103] and the more aggressive *atomic broadcast with optimistic delivery* [70], which allow processes to deliver messages faster, in certain cases. They exploit the spontaneous total order message reception: with high probability, messages broadcast in a local area network are received totally ordered. An atomic broadcast with optimistic delivery is defined by three primitives. First, $TO\text{-}broadcast(m)$ broadcasts the message m to all nodes in the system. Then,

opt-deliver(m) delivers a message m optimistically to the application once it is received from the network, in a *tentative order*. Finally, *TO-deliver*(m) delivers m to the application in the *definitive order*, which is a total order. The following properties are satisfied. (a) Termination: if a site TO-broadcasts m, then every site eventually opt-delivers m and TO-delivers m. (b) Global agreement: if a site opt-delivers m (TO-delivers m) then every site eventually opt-delivers m (TO-delivers m). (c) Local agreement: if a site opt-delivers m then it eventually TO-delivers m. (d) Global order: if two sites N_i and N_j TO-deliver two messages m and m', then N_i TO-delivers m before it TO-delivers m' if and only if N_j TO-delivers m before it TO-delivers m'. (e) Local order: a site first opt-delivers m and then TO-delivers m. With such an optimistic delivery, the coordination phase of the atomic broadcast algorithm is overlapped with the processing of messages. This optimistic processing of messages must be only undone when the definitive total order mismatches the tentative one.

Active and passive replication In a system with *active* replication, the same request is processed by every node. As opposed to this, each request in *passive* replication is processed by only one node, which later transfers the updates to the rest of servers. Depending on the node that can process a request, the next two server architectures can be distinguished.

Server architecture: primary-backup The *server architecture* defines where transactions are executed in the first place. Common server architectures are primary-backup and update-everywhere. In a *primary-backup* system, a specific node –called *primary copy* or *master copy*– is associated to each data item. Any update to that item must be first sent to the primary copy, i.e., the primary copy is the delegate server for any update transaction over that item. The rest of the servers are called *backups* and serve only queries over that item. One possible setting is to select a single server as the primary copy for all database items, although this can cause a bottleneck.

Server architecture: update-everywhere In an *update-everywhere* system, every node is able to serve client requests for updating any data item, so that it is possible that two concurrent updates arrive at different copies of the same data item. In order to avoid inconsistencies, usually some mechanism is used to

decide which update will be successful, aborting one of the transactions. This mechanism is the *transaction termination* protocol.

Transaction termination: voting, weak voting and non-voting termination Whenever a transaction ends, the transaction termination protocol is run to decide the outcome of the transaction (*validation*) and, in case of deciding to commit, take the necessary actions to guarantee transaction durability. Two main approaches can be distinguished: voting and non-voting termination. In a *voting termination*, an extra round of messages is required to coordinate the different sites, as in 2PC. *Weak voting* is a special case of voting termination, where only one node decides the outcome of the transaction and sends its decision to the rest of nodes. In a *non-voting* termination, all sites are able to autonomously and deterministically decide whether to commit or to abort the transaction. In this case, this symmetrical validation process is also called *certification*.[4]

Validation and, thus, certification are usually based on conflicts. The ending transaction T is checked for conflicts with concurrent and already validated (respectively, certified) transactions. If conflicts are found, validation (certification) fails and T is said to be negatively validated (negatively certified) and it is aborted. Otherwise, validation (certification) succeeds and T is said to be validated (certified), successfully validated (successfully certified) or positively validated (positively certified), and it has to be committed in all affected nodes. When validation or certification are not based on conflicts, the validation (certification) succeeds or is positive if the decision taken over T is to commit it. Otherwise, the validation (certification) fails or is negative.

System model Aspects such as server architecture, server interaction and transaction termination are part of the *system model*, which defines the way in which a system operates.

Server layers Inside a server, different layers can be identified: the network or communication layer, at the bottom of the stack; the data layer; the replication

[4]We will use the term validation to generically refer to the process of deciding the outcome of a transaction. We will use the term certification to specifically refer to the validation process performed by each node in an independent, deterministic and symmetrical manner. Other authors, however, consider both terms as synonyms.

layer; and the application layer, on the top. These general layers may appear merged together at different systems. E.g., in a system that manages replication by embedding the necessary code into the DBMS internals, the data and the replication layer are merged. On the other hand, we say a replication system is *based on middleware* [11] when it gathers all replication mechanisms in a replication layer, i.e., a software layer placed between the instance of the database management system and the applications accessing the data. This provides an independence among system components which leads to a high portability (e.g., to migrate the middleware into an environment based on a different DBMS).

Chapter 3

Correctness Criteria for One-Copy Serializable Database Replication Systems

In this chapter, we provide a formal analysis of one-copy serializability, by comparing two types of database replication systems that guarantee such a correctness criterion. This analysis shows that different replica consistency levels are accepted by the criterion, therefore rendering its original definition incomplete regarding consistency. We then propose new correctness criteria where the exact level of accepted replica consistency is defined.

3.1 Introduction

Distributed and replicated database systems appeared in order to provide a higher level of availability and fault tolerance than existing stand-alone (centralized, i.e., non-replicated) databases, while still ensuring a correct management of data. Traiger et al. [130] suggested the concepts of one-copy equivalence and location, replica, concurrency, and failure transparencies, as desirable features for distributed systems. Bernstein et al. later defined one-copy serializability (1SR) [15] as a correctness criterion. According to it, the interleaved execution of users' transactions must be equivalent to a serial execution of those transactions on a

stand-alone database. 1SR turned immediately to be the accepted correctness criterion for database replication protocols. In order to ensure such guarantees, a conservative approach inherited from stand-alone database systems was followed [14]. Thus, write operations over any data item had to acquire write locks in all its copies prior to updating the local copy. This concurrency control based on distributed locking strongly affected performance, as each write operation must be preceded by a round of messages requesting the corresponding lock in every replica. Moreover, in order to guarantee transaction atomicity, an atomic commit protocol was run for transaction termination. In such protocols, several rounds of messages are required in order to reach a consensus among all participating sites for each transaction commitment, which further penalized performance and scalability.

Several optimizations were added later to database replication systems, trying to provide correct, i.e., 1SR, replication at a reasonable cost. According to the deferred update replication model [119, 130], transactions were processed locally at one server and, at commit time, were forwarded to the rest of system nodes for validation. This optimization saved communication costs as synchronization with other nodes was only done at transaction termination during the atomic commit protocol. Another important optimization consisted in substituting the atomic commit protocol for an atomic (total order) broadcast [2, 123], whose delivery order was then used as a serialization order for achieving 1SR. Wiesmann and Schiper [136] provide a performance comparison of database replication techniques and conclude, among other insightful remarks, that techniques based on total order broadcast significantly outperform traditional database replication protocols like distributed locking, and that this performance difference is larger if the network is slow and subject to contention. This is due to the fact that distributed locking requires many messages, while techniques based on total order broadcast demand less networking resources, typically by restricting interactions to a single broadcast, which makes these techniques very efficient with a slow network.

According to the definition of 1SR [15], replication protocols based on atomic commit and those based on atomic broadcast ensure the same correctness criterion. However, carefully analyzing the behavior of both systems, some differences can be observed. In a traditional stand-alone system, a user could transfer certain amount of money from one account A to another B, commit the transaction, and read the increased balance in B in the following transaction. The same is ensured in replicated systems based on distributed locking and atomic commit, precisely due to these inherited techniques. As both systems behave

identically, the user will then not be able to distinguish between the stand-alone and the replicated one. Moreover, the user will consider this behavior correct, as it perfectly matches their expectations. However, the deferred update propagation and the atomic broadcast of an optimized replicated system may cause that the same operation ends in a worried user when, after committing the bank transfer, they temporarily read the old balance in B, as if the money had disappeared (we show later in the chapter why this may happen). Clearly, the user might consider this behavior incorrect until, after some time, the increased balance can be finally read. This situation would never have occurred in a traditional stand-alone database. Does this mean that the system based on atomic broadcast is not behaving as one copy as it claims to do? Are then one-copy equivalent database replication systems actually behaving as one copy?

In this chapter, we demonstrate that some consistency-related features significantly changed when atomic commit protocols were replaced by termination protocols based on atomic broadcast. These visible differences in consistency, admitted in 1SR, might originate some confusion among users. To show this, we made a historical review through different 1SR systems and chose two representative database replication system models, which are analyzed through a case study to reveal the differences between them. As a solution, and inspired in the memory consistency models used in distributed shared memory (DSM), we propose a new set of correctness criteria for distinguishing among different types of 1SR database replication. We believe that our proposal will remove any ambiguity from current correctness criteria.

Moreover, as we are not only interested in consistency but also in performance, we further distinguish between different server implementations that offer the same consistency as perceived by users but could present important performance differences. This way, establishing correspondences between DSM levels and replica consistency enforced by database replication protocols, we provide a complete formalization of the different levels of replica consistency and our proposed correctness criteria.

The rest of the chapter is structured as follows. Section 3.2 details the assumed system model and provides basic definitions. Section 3.3 states the looseness of the term *one-copy equivalence*, by showing important differences in consistency between two systems providing 1SR. Section 3.4 presents and compares memory consistency models used in the scope of DSM, for later establishing, in

Section 3.5, a complete formalization of our proposed criteria and the correspondence between DSM models and replica consistency levels ensured in database replication systems. Section 3.6 briefly discusses about the consistency provided in current cloud systems. Section 3.7 gives a final discussion.

3.2 System Model and Definitions

The replication systems analyzed in this chapter consider a partially synchronous distributed system composed of database servers, also called sites, nodes or replicas, $R_1, R_2, \ldots, R_n$, and clients or users that access these replicas. Communication between components is based on message passing, as they do not have shared memory. Consequently, they neither have a global clock. Servers can also communicate through atomic broadcast.

Replicas fail independently by crashing (Byzantine failures are not considered). Sites may eventually recover after a crash, i.e., we assume the crash-recovery failure model. Regarding partition failures, we assume the *primary component membership* [28], under which only a majority group is allowed to progress in case of network partition. Such behavior is necessary if we want to provide both consistency and availability, as stated by the CAP theorem [48].

Each server stores a full copy of the database.[1] Queries only need to access one replica, while update transactions are required to modify all replicas. This corresponds to the Read One Write All Available (ROWAA) [15] approach, which was shown [64] to be the best choice for data replication, over quorum-based solutions.[2] Database users are provided with the concept of session, in order to logically group a set of transactions from the same user. Transactions from different users belong to different sessions. However, it can be left to the user the decision of using one or multiple sessions to group their transactions. Transactions of the same client session are submitted sequentially, but may be addressed to different replicas. Each client request is only processed by its delegate, which

[1] Initial systems, such as the one described by Bernstein and Goodman [14], did not require full replication, as they were distributed databases with some degree of replication.

[2] Indeed, the use of atomic broadcast for data replication considerably reduced the interest in quorum-based solutions. During the last decade, database replication systems have scarcely followed approaches other than ROWAA.

later transfers the updates to the remote nodes. In other words, a passive, update-everywhere replication is performed.

Transactions may execute concurrently in a DBMS, raising conflicts. The local concurrency control of the DBMS establishes which executions of concurrent transactions are correct or legal regarding certain isolation level. In this chapter we assume that each local DBMS provides the serializable isolation level [9], which can be achieved, e.g., by the strict two-phase locking (2PL) [15]. Conflicts among transactions executing at remote nodes must also be treated. In optimistic systems, these conflicts are usually resolved during transaction termination with the application of certain validation rules. All the systems considered in this chapter guarantee the correctness criterion of 1SR.

3.3 One-Copy Serializability: a Loose Term

In many cases, implemented systems are quite more restrictive than the correctness models they follow. Situations that would be allowed by the model are rejected in the real system, or they are simply not possible due to implementation issues. This mismatching is usually due to pragmatic considerations, as an exact model reflection would increment the system complexity. Although correctness is never impaired, other aspects such as performance, concurrency or scalability are negatively affected.

This is the case of the distributed concurrency control protocol described by Bernstein and Goodman [14]. It is based on distributed locking and an atomic commit protocol, two-phase commit (2PC), and follows the one-copy serializability criterion. We will refer to this general model as a *2PC-based* system. The 2PC protocol used for transaction termination ensured that all replicas agreed on the set of committed transactions and a serializable history was guaranteed. However, due to distributed locking and atomic commit, replicas did not only commit the same set of transactions but did it in such a way that any new issued transaction was able to see all the changes performed by previous transactions, with independence of the replica where this new transaction was executed. That is, the system nodes were behaving as a traditional one-copy database in that any copy of a written data item was updated before any subsequent access to that copy was allowed: a transaction T always got the most recent vision of the database, as

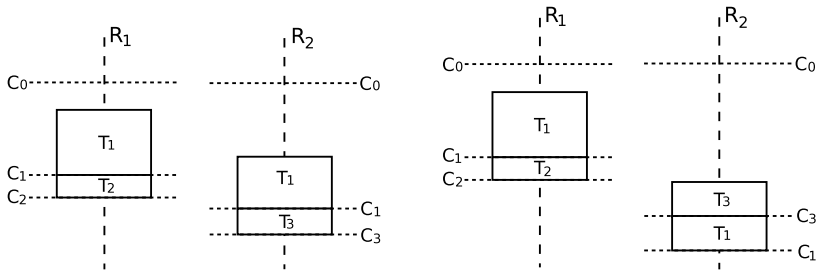


Figure 3.1: Diagrams for Executions 3.1 and 3.2. Two replicas are visualized: R_1 and R_2. Vertical broken lines represent real time at each replica, increasing downwards. Horizontal dotted lines represent commit operations in each replica. A transaction T_0 is assumed to create the initial database state. In 2PC-based systems, T_3 does not start until T_1 finishes. In abcast-based systems, however, transactions may swap. (Note that T_2 is not executed at R_2, and neither does T_3 at R_1, as they are read-only transactions.)

created by the last transaction executed in the system immediately before T. Let us consider a simple database execution to illustrate this feature.

Execution 3.1 (2PC-based systems, Figure 3.1a). A user starts transaction T_1 at replica R_1. The database contains data item x with an initial value x_0. T_1 requests write access to x, which requires that a write lock is acquired at each replica, as stated by the distributed locking concurrency control. After getting all the locks, T_1 modifies the value of x and finishes. During the atomic commit protocol all replicas agree on the commitment, so T_1 can commit, the new value x_1 can be assigned to x, and the write lock on x can be released. After the commitment of T_1, the same user, who wants to be sure about the success of their updates, starts a query, T_2, for reading x. T_2 is also started in replica R_1, where value x_1 is already available. T_2 reads x_1, as expected, and finishes. Suppose that replica R_2 is slower and it did not yet apply the new value to data item x, so it still has value x_0. Now the user starts a second query T_3 for reading x again and, due to some load balancing, T_3 is run at replica R_2. When T_3 tries to read x, a read lock is requested. As the item is not yet updated, the write lock granted to T_1 still holds, and T_3 must wait. Once R_2 applies the update of T_1, the write lock is released and T_3 is able to read value x_1, as expected, and commit. Note that R_2 was not updated when T_3 was launched, but distributed locks prevented T_3 from accessing an outdated value. Both T_2 and T_3 were able to access the value updated by T_1, which is the expected behavior (traditional stand-alone databases

provide this same effect). The equivalent serial order of this execution is T_1, T_2, T_3 or T_1, T_3, T_2.

The observed behavior, i.e., transactions getting the last value of all database items, is not required by the 1SR correctness criterion. Indeed, serializability (and thus 1SR) states that the effect of transactions must be equivalent to that of *any* serial order. No restriction is imposed over this serial order. In particular, no real-time consideration is taken into account, such as respecting the order on which transactions were started by the client. In a traditional stand-alone database providing serializability, however, if a transaction T_2 started *after* the termination of conflicting transaction T_1, the equivalent serial order necessarily respected this precedence (there is only *one* node executing all transactions). Just like traditional stand-alone systems, 2PC-based systems are more restrictive than the correctness model they follow. The stricter observed behavior of stand-alone systems was due to the physical laws of time, while that of 2PC-based systems is a consequence of the techniques used in their implementation. However, these same techniques made those earlier systems slow and barely scalable.

More modern systems [2, 123] proposed the substitution of the atomic commit protocol by a termination protocol based on atomic broadcast that performs the deferred update propagation.[3] In these systems, transactions are locally executed at their delegate replica. When a transaction ends, its readset and writeset are broadcast to the rest of replicas in total order, using the atomic broadcast. After delivery, if no conflict is found in the certification, the transaction commits at every replica. These *abcast-based* systems performed significantly better than previous 2PC-based ones [136]. These improvements were possible thanks to the adoption of the full replication model,[4] where all replicas store a complete copy of the database, instead of the more general model of distribution with partial replication, which forced to use costly distributed transactions. This new approach allowed replication systems to boost performance and scalability while the correctness criterion was claimed to be one-copy serializability as traditionally defined. However, a significant difference with regard to 2PC-based systems was introduced. Let us analyze the new behavior with the same client requests.

Execution 3.2 (Abcast-based systems, Figure 3.1b). The client starts transaction T_1 in replica R_1, where it updates the local copy of data item x, after acquiring the

[3] Deferred update propagation was already used in the distributed certification scheme of Sinha et al. [119] or in the distributed optimistic two-phase locking, O2PL, of Carey and Livny [23].

[4] Partial replication is also possible in abcast-based systems.

local write lock. At transaction termination, the deferred update propagation is done and thus the writeset, i.e., data item x and the updated value x_1, is broadcast to all replicas. A deterministic certification process is run independently at each replica before writeset application. If certification is successful, replica R_1 commits T_1 and the same user immediately starts a query T_2 for reading the updated item. T_2 is directed to replica R_1 and so it reads the updated value and finishes. But again, the user starts a second query T_3 that is addressed to replica R_2 by the load balancer. Similarly to the 2PC-based case, R_2 agrees on committing T_1 but has not yet started to apply the writeset of T_1. The key difference is that now data item x is not locked (assume there are no other transactions running) and T_3 is able to access x without waiting.[5] The accessed version x_0 corresponds to the last *locally* committed transaction that updated that item, and not to the last globally certified transaction in the system. So T_3 gets an unexpected value and the user, who was previously able to read the updated item with T_2, is now not able to see it with the following transaction T_3. Nevertheless, 1SR is guaranteed: the effect of transactions is equivalent to that of the serial order T_3, T_1, T_2. The fact that T_1 precedes T_3 in real time is not considered in 1SR. However, the user reads an outdated value, which clearly is not what they expected.

Now, the stricter behavior of both traditional stand-alone and 2PC-based systems is no longer enforced. In abcast-based systems, 1SR is guaranteed to the letter, but not further. The difference between 2PC-based and abcast-based systems lies in the distinct replica consistency maintained.[6] While in 2PC-based systems distributed locks prevent transactions from accessing outdated values, in abcast-based systems different values for the same data item are accessible at the same time at different replicas. If both system models ensure 1SR, it is because replica consistency was not considered in correctness criteria for database replication. Indeed, the definition of 1SR does not deal with the level of synchronization between replicas, leaving this aspect open to the system designer. The fact of not enforcing any replica consistency level renders the definition loose enough to allow several interpretations. Although this can now lead to ambiguity, that

[5] There are indeed two possibilities for this situation in a locking-based system: either the certification involves getting the necessary locks and R_2 has not yet run the certification for T_1, or the certification is based on history logs and required locks are only set on writeset application. Although the sample execution depicts the second case, both possibilities have the same result: T_3 will read the old value of x and T_1 will finally commit in R_2.

[6] Note that these executions show a unique user performing all accesses. This is only for the sake of a clearer perception of the problem, represented as a user getting different values from an item that they updated in an immediately previous transaction. But the consistency problem is the same if transactions are from different users.

was not the case back when 1SR was defined, as existing techniques allowed only one possibility. However, although different techniques appeared later, no specification regarding replica consistency was added to the definition. This fact is interesting, as a similar concept was already defined and studied in the scope of distributed shared memory, surveyed in multiple papers [1, 29, 91, 124].

3.4 Consistency Models in Distributed Shared Memory

Memory consistency models represent the way on which memories from different sites are synchronized in order to conform a distributed shared memory. The higher the level of consistency, the higher the synchronization and the fewer the divergences on values. According to Mosberger [91], the strictest level of consistency, *atomic consistency* [73] (a.k.a. *linearizability* [56]) considers that operations take effect inside an *operation interval*. Operation intervals are non-overlapping, consecutive time slots. Several operations can be executed in the same slot; read operations take effect at read-begin time while write operations take effect at write-end time. Thus, read operations see the effects of all write operations of the previous slot but not those of the same slot.

Despite the simplicity of the idea and the easiness of design for applications that use atomic memory, this consistency level is often discarded due to its high associated cost [124, 126]. Consequently, a more relaxed model is used in practice as the common level: *sequential consistency* [75]. In a sequentially consistent system, all processors agree on the order of observed effects [91]. According to Lamport [75], the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

The key difference between these two consistency models is the *old-new inversion* between read operations [7]. This issue is precluded in the atomic model but it may arise in the sequential one. It consists in the following: once a process p writes a value v_n onto a variable x whose previous value was v_o, another process r_1 is able to read v_n while later a second reader r_2 reads v_o. Note that this is sequentially consistent since r_2 is still able to read afterwards value v_n, according to the total order associated with sequential consistency. However, such a scenario violates atomic consistency since r_2 reads x once the slot given to value v_n was

already started and all reads in that slot should return value v_n (instead of v_o). This old-new inversion is exactly the situation arisen in previous Execution 3.2.

More relaxed DSM consistency models are the *cache* consistency model [49], the *PRAM* consistency model [81] and the *processor* consistency model [49].

3.5 Formalization of the Correspondence Between DSM Memory Consistency Models and Database Replica Consistency

According to several authors [8, 53], in general distributed executions (i.e., not necessarily related to DBMSs), one-copy equivalence can be guaranteed when all processes see all memory-related events in the same order. This implies that the DSM consistency models able to achieve one-copy equivalence are both the sequential and the atomic ones. Following this trend, Mosberger [91] states that the one-copy serializability concept as defined by Bernstein et al. [15] is equivalent to sequential consistency. On the other hand, Fekete and Ramamritham [45] distinguish between strong consistency (which is explicitly associated with the atomic DSM model and mentioned as a synonym of one-copy equivalence) and weak consistency (where sequential consistency is included). As it can be seen, no agreement on the exact level of replica consistency needed to achieve one-copy equivalence exists nowadays.

Although memory consistency models consider individual operations (write or read a variable in memory), a correspondence to a transactional environment can be established. Indeed, the concept of transaction matches very well the concept of operation interval. As explained by Fekete and Ramamritham [45], a transaction T can be considered as a macro-operation from the DSM point of view, where all read operations can be logically moved to the starting point of T (as read versions correspond to those available when the transaction started),[7] and all write operations logically moved to the termination point of T (when the transaction commits and its updates are visible to other transactions). Thus, we

[7] We can safely assume that a transaction will never read a data item after writing it or that the application is able to cache written versions and serve those read operations without accessing the database.

can consider the interval of time where all transaction operations are done as a *transaction interval*.

The fact that a transaction is a macro-operation also reduces the high costs that led to dismiss atomic consistency in DSM. Although it is true that maintaining strong consistency levels for individual operations is costly, the fact of using transactions that group arbitrarily large sets of operations into one unit of execution allows a reduction in the costs, thus making atomic consistency bearable in a transactional context. With transactions, synchronization between sites must be done only once for the full set of write operations included in a transaction. This writeset is thus regarded as a macro write operation, as opposite to multiple individual write operations.

Although transactions have been widely used in distributed systems, very few works have considered the DSM consistency model resulting from database replication protocols, being that of Fekete and Ramamritham [45] one of such few exceptions. We consider that this is an interesting line of action, as transferring the concept of transaction into traditional memory consistency models makes it possible to provide the strongest level of consistency in database replication systems at a reasonable cost.

3.5.1 Correctness Criteria

Inspired in the idea of a transaction as a macro-operation, and adapting Mosberger's concept of operation interval [91] to its use with transactions, we now present a complete formalization in order to define a new set of correctness criteria and to precisely establish the correspondence between memory consistency models and database replica consistency. Our aim is to refine 1SR correctness criterion by distinguishing among the possible replica consistency levels. In the same manner as old-new inversions allow us to distinguish between the atomic and the sequential memory consistency levels, we will distinguish among our new correctness criteria by means of the presence or absence of similar phenomena in the executions of the system. For this we consider database replication systems that, as stated in Section 3.2, provide full replication and a serializable concurrency control, and guarantee 1SR, as originally defined by Bernstein et al. [15], by following a ROWAA approach.

We first provide the foundations of our reasoning model, based on the concept of intervals. Then we recall the conditions required by 1SR, translated to a system expressed in our model. Finally, we will define additional constraints, also according with the model, that will establish new correctness criteria. Those new criteria divide the original 1SR into disjoint subcriteria.

Definition 3.1. Correct site. A site is correct if it never crashes.

Definition 3.2. Faulty and permanently faulty sites. A site is faulty if it crashes but eventually recovers. A site is permanently faulty if it crashes and never recovers.

Definition 3.3. Database execution. We call database execution to the process of executing some set of transactions in a replicated database. A database execution finishes when all replicas which are not permanently faulty have applied all necessary updates from committed transactions.

Definition 3.4. Replica time slotting. Real time at every replica R_i of a replicated database can be partitioned into consecutive and non-overlapping slots, delimited by the occurrence of transaction commit operations, which are executed at R_i in certain order. The time slot of R_i ended by the commitment of T_j is referred to as $TS_i(T_j)$.

Definition 3.5. Real-time interval. Let R_i be a node of a replicated database, and T_j a transaction committed in R_i. We define the real-time interval of T_j in R_i, $RTI_i(T_j)$ as the interval of real time during which T_j was executed in R_i.

Definition 3.6. (Logical) interval. Let R_i be a node of a replicated database, and T_j a transaction committed in R_i. We define the logical interval of T_j in R_i, or simply the interval of T_j in R_i, $I_i(T_j)$, as the portion of $RTI_i(T_j)$ which overlaps with $TS_i(T_j)$.

Note that aborted transactions do not present intervals at any replica.

Theorem 3.7. *Based on the serializable isolation guaranteed in the nodes, the execution at replica R_i of any committed transaction T_j can be safely considered to have taken place completely inside $I_i(T_j)$.*

Proof. For any transaction T_j committed at replica R_i, $TS_i(T_j)$, $RTI_i(T_j)$ and $I_i(T_j)$ end in the same real-time instant. However, starting instants do not need to coincide. Thus, depending on the database load, the real-time interval $RTI_i(T_j)$

may start at any point before the commit operation of T_j, even out of the bounds of time slot $TS_i(T_j)$. If T_j starts inside the bounds of $TS_i(T_j)$, then $RTI_i(T_j)$ and $I_i(T_j)$ are equal (and $TS_i(T_j)$ contains both of them), and the theorem is trivially proven. If the execution of a transaction T_j starts before the end of the previous time slot, out of the bounds of $TS_i(T_j)$ (in this case, $I_i(T_j)$ and $TS_i(T_j)$ are equal), then concurrency control and the serializable transaction isolation allow us to consider all the operations of T_j as to execute after the immediate previous commitment in the local database. Indeed, all read operations of T_j, if any, are able to access the values written by previous committed transactions, up to the last one; and all write operations of T_j, if any, can be logically moved to the point immediately before the commitment of T_j. If any of these premises does not hold, local concurrency control will abort T_j. Otherwise, if T_j is committed, we can safely consider transaction execution as starting just at the same point that its corresponding time slot $TS_i(T_j)$ at the executing replica R_i, and, thus, inside $I_i(T_j)$. □

Theorem 3.8. *Inside each replica time slot, there is one and only one interval, corresponding to the transaction whose commit ends that slot.*

Proof. Each time slot in replica R_i is delimited by a transaction commitment, i.e., each committed transaction T_j defines a different time slot $TS_i(T_j)$. T_j is executed inside its $I_i(T_j)$, which is contained by definition inside $TS_i(T_j)$, so there is one and only one interval at each time slot. □

Theorem 3.9. *To follow the ROWAA approach, (a) $I_i(T_u)$ exists for each committed update transaction T_u and for each replica R_i; and (b) $I_x(T_r)$ exists for each committed read-only transaction T_r and a single replica R_x.*

Proof. The ROWAA approach requires to apply transaction updates at every node in the system. To apply at R_i the updates of a transaction T_u, T_u must be committed at R_i. This generates an interval $I_i(T_u)$. If all replicas must be updated, there exists a corresponding interval for T_u at each replica. On the other hand, in the ROWAA approach read-only transactions only need to be executed at their delegate node. If the delegate replica of read-only transaction T_r is R_j, there exists $I_j(T_r)$. □

Definition 3.10. Conflicting transactions. Two transactions conflict if they have conflicting operations. Two operations conflict if they belong to different transactions, they access the same data item and at least one of the operations is a write.

Definition 3.11. Independent transactions. Two transactions that do not conflict are called independent transactions. Multiple transactions compose a set of independent transactions if they are independent two by two.

Definition 3.12. Local precedence. We define a total order among all transactions committed in replica R_i, called local precedence of R_i, $<_{l_i}$, in the following way: a transaction T_j precedes transaction T_k in the local order of R_i, $T_j <_{l_i} T_k$, if and only if $I_i(T_j)$ takes place before $I_i(T_k)$.

Definition 3.13. Local history. For each replica R_i, the (already chronologically ordered) sequence of intervals composes the local history of all committed transactions of R_i, H_i.

Theorem 3.14. *The local history at replica R_i is consistent with the local precedence of all the transactions committed in R_i.*

Proof. Let T_j and T_k be two committed transactions at R_i, and let T_j precede T_k in the local order, $T_j <_{l_i} T_k$. Suppose that in the local history, H_i, $I_i(T_k)$ appears before $I_i(T_j)$. This means that the interval of T_k took place before that of T_j, which contradicts the assumed local precedence. □

Corollary 3.15. *Any local history is a serial history, where the local precedence is the serial order.*

Definition 3.16. Complete history. A local history H_i is complete for a database execution of a set $\mathbb{T}$ of transactions if it contains an interval $I_i(T_u)$ for each committed update transaction T_u of $\mathbb{T}$.

With the ROWAA approach, update transactions could commit even if some site is unavailable. When crashed replicas recover, a recovery protocol is run in order to bring those replicas up-to-date, applying the updates of the transactions that committed during the failure. In the absence of permanent failures, even if some transient failures occur, correct replication and recovery protocols should be able to make all local histories complete. However, permanent site failures prevent local histories from being complete.

Definition 3.17. Global history. A global history of a given database execution is a history which contains all intervals from all local histories of that execution, and those intervals appear at the global history respecting their respective local precedence.

Definition 3.18. One-copy history. A centralized, non-replicated database system can be thought of as a replicated system composed of only one replica R_{1C}. As there are no more nodes, both read-only and update transactions present only one interval in the single replica of the system. Thus, intervals do not need to be qualified with the replica identifier, and local histories of R_{1C} are called one-copy histories. As local histories, one-copy histories are serial, where the local precedence is the serial order. Moreover, as the system has a single available node, one-copy histories are trivially complete histories for all database executions that could take place in that system.

As stated by Bernstein et al. [15], a global history H_G from a replicated database is 1SR if it is view equivalent to a serial one-copy history H_{1C}. H_G is view equivalent to H_{1C} if: (a) H_G and H_{1C} have the same reads-from relationships, i.e., T_k reads from T_j in H_G if and only if the same holds in H_{1C}; and (b) for each final write in H_{1C}, there is a final write in H_G for some copy of the same data item. We thus require these conditions to our basic system and now define new constraints in order to subdivide 1SR into more specific correctness criteria.

Independent transactions may execute at any order: as they do not conflict, the results of the transactions and the final state of the database will be the same at any possible execution order. However, when transactions conflict, the order in which they are executed is important, as the results of the transactions and the final state of the database depend on this execution order. One of all possible serial orders of a set of conflicting transactions must be chosen. 1SR admits any of them, but from the point of view of the user, the natural and intuitive order is the order in which transactions have been started in real time.

Definition 3.19. First start. We say that a transaction T_j first-starts when it accesses any item of the replicated database for the first time through any system replica R_i (the delegate replica). This real-time instant corresponds to the start point of $RTI_i(T_j)$.

Real time must be considered in order to capture the point of view of users. However, this does not impose any real-time constraints on the system, nor it requires any notion of global clock among replicas.

Note also that in a locking-based concurrency control all operations over database items must be preceded by the acquisition of the necessary locks. A transaction whose initial operation is held in the database while waiting for the required

locks is not considered as first-started until those locks are granted and the first data item is effectively accessed.

After executing all its operations, if no abortion occurs, a transaction asks for commitment. Each committed transaction T_j must apply its updates at every replica, which can be considered as multiple commit operations for the same transaction. One of the replicas, R_i, will be the first to complete the commit operation and make the updates of T_j visible to local transactions starting at R_i. This instant is when T_j first-commits.

Definition 3.20. First commit. We say that a transaction T_j first-commits when it commits in the replicated database for the first time, through any system replica R_i. This real-time instant corresponds to the end point of $RTI_i(T_j)$.

Definition 3.21. Real-time precedence (RT precedence). We define a partial, transitive order of real-time precedence, $<_{rt}$, in the following way: a transaction T_j precedes transaction T_k in real time, $T_j <_{rt} T_k$, if and only if T_j first-commits before T_k first-starts.

Definition 3.22. Concurrent transactions. Two transactions T_j and T_k are concurrent if both $T_j <_{rt} T_k$ and $T_k <_{rt} T_j$ are false, i.e., there is no real-time precedence between them.

Definition 3.23. Alteration. Let T_j and T_k be two committed transactions at replica R_i. Let T_j precede T_k in real time, $T_j <_{rt} T_k$. We say that an alteration occurs in R_i if T_k precedes T_j in the local order of R_i, $T_k <_{l_i} T_j$, i.e., the local order of R_i is inconsistent with the real-time precedence of the transactions. T_k is called the altered transaction, while T_j is the overtaken transaction.

Definition 3.24. Real-time & conflict precedence (RTC precedence). We define a partial order called RTC precedence, $<_{rtc}$, in the following way: a transaction T_j precedes transaction T_k in RTC order, $T_j <_{rtc} T_k$, if and only if (a) $T_j <_{rt} T_k$, and (b) T_j and T_k are conflicting transactions.

Intuitively, RTC is a *decrement* of the real-time precedence, where independent transactions are not ordered and, thus, transitivity is lost.

Definition 3.25. Inversion. Let T_j and T_k be two conflicting transactions, committed at replica R_i. Let T_j precede T_k in RTC order, $T_j <_{rtc} T_k$. We say that an inversion occurs in R_i if T_k precedes T_j in the local order of R_i, $T_k <_{l_i} T_j$, i.e., the local order of R_i is inconsistent with the RTC precedence of the transactions.

T_k is called the inverted transaction, while T_j is the overtaken transaction. More simply, an inversion is an alteration between two conflicting transactions.

The phenomenon of old-new inversion presented in Section 3.4 corresponds to an inversion of a transaction that overtakes the update transaction from which it was supposed to read a value.

The absence of transitivity in the RTC precedence minimizes the ordering guarantees that replicas must enforce in order to avoid inversions. Let T_a, T_b, and T_c be three transactions such that T_a only writes item x, T_b only reads items x and y, and T_c only writes item y. This way, T_a is conflicting with T_b and T_b is conflicting with T_c, but T_a and T_c are independent transactions. Let R_d be the delegate replica of read-only transaction T_b (R_d is the only node that will execute T_b, following the ROWAA approach). If $T_a <_{rt} T_b <_{rt} T_c$, then $T_a <_{rtc} T_b$ and $T_b <_{rtc} T_c$ both hold and R_d must respect such precedence relationships in order to avoid inversions. However, as T_a and T_c are not related in the RTC precedence, replicas other than R_d may execute T_a and T_c at any order without causing an inversion.

Definition 3.26. α-history. An α-history is a local history which respects the RTC precedence.

Theorem 3.27. *If no inversions, i.e., alterations between conflicting transactions, occur during a database execution, then each local history is an α-history.*

Proof. If no alterations between conflicting transactions occur, then the intervals of conflicting transactions respect the real-time precedence at each node and, therefore, local precedence at each replica is consistent with the real-time precedence of conflicting transactions, i.e., the RTC precedence. As a local history respects the local precedence, each local history respects the RTC precedence and is, therefore, an α-history. □

Definition 3.28. 1ASR correctness criterion. We call 1ASR the correctness criterion that, based on a serializable concurrency control, ensures 1SR and guarantees that each local history of a database execution is an α-history.

The ‘A’ in the name stands for “atomic” as, from the point of view of the users, the results of a database execution are equivalent to those that one could obtain by ensuring that a transaction is committed at every replica before starting the

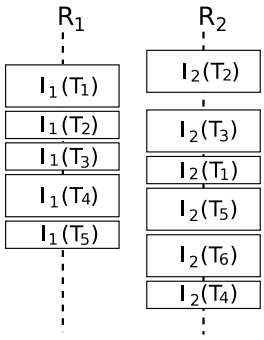


Figure 3.2: Sample execution for a 2PC-based (1ASR) system. Two replicas are visualized: R_1 and R_2. Vertical dotted lines represent real time at each replica, increasing downwards.

next transaction. This way, users perceive an *atomic* nature in the system consistency. [8] Informally, 1ASR restricts the set of all possible equivalent serial orders of a database execution that would comply with 1SR, so that orders violating RTC are not considered. This is the case of the 2PC-based systems described in Section 3.3.

Execution 3.3. Now we illustrate with an example the construction of the global and one-copy histories of a 1ASR database execution, thus providing some guidelines to extract an equivalent serial order for such an execution. Consider a system of two replicas that execute a set of six transactions with the following characteristics: (a) T_1 and T_2 are concurrent and both precede T_3 in real time; (b) T_4 and T_5 are concurrent, both are preceded by T_3 in real time and both precede T_6 in real time; (c) all transactions are update transactions except for T_6, which is read-only; (d) T_3 and T_4 conflict, and T_5 and T_6 conflict, but the rest of transactions are independent. With such a set of transactions, RTC precedence defines only two relationships: $T_3 <_{rtc} T_4$ and $T_5 <_{rtc} T_6$. The database execution depicted in Figure 3.2 corresponds to a 2PC-based system and it is 1ASR, as these precedences are respected at both nodes. Local histories are:

$$H_1 : I_1(T_1), I_1(T_2), I_1(T_3), I_1(T_4), I_1(T_5)$$
$$H_2 : I_2(T_2), I_2(T_3), I_2(T_1), I_2(T_5), I_2(T_6), I_2(T_4)$$

To construct a global history H_G such that it is view equivalent [15] to a one-copy history H_{1C}, we divide transactions in three types and add their intervals to H_G

[8] We will later qualify the correspondence between possible ways of implementing this correctness criterion and the actual level of atomic memory consistency. However note that 1ASR is less strict than the atomic DSM level as it maintains only the property of the absence of inversions.

in three steps. (1) *Pairs of update transactions ordered in RTC precedence*. We scan the partial order defined by RTC precedence. For each pair of ordered transactions, e.g., $T_j <_{rtc} T_k$, if both T_j and T_k are update transactions, their intervals appear at all complete histories and they appear respecting their RTC precedence (incomplete histories also respect RTC precedence but lack some intervals). Iterating over the pairs of this type, for each pair we insert into H_G all intervals (i.e., intervals from all local histories) corresponding to T_j and T_k, in such a way that the local precedence between intervals from the same replica is always respected at H_G (as it is at local histories). In order to do so, we must respect not only the local precedence between the transactions of the current pair, but also between transactions of the current pair and other update transactions already inserted in H_G. These insertions respect RTC precedence, as each local history already respects RTC. (2) *The rest of update transactions*. Once the intervals of all pairs of update transactions ordered in RTC have been appended to H_G, the second step inserts the intervals of the rest of update transactions. The insertion of an interval $I_i(T_j)$ is made respecting the local precedence of R_i between $I_i(T_j)$ and other intervals of R_i already present at H_G. (3) *Queries*. Finally, the intervals of read-only transactions are inserted at H_G, respecting the local precedence of their delegate (e.g., interval $I_i(T_k)$, which was preceded by interval $I_i(T_j)$ in the local sequence of R_i, i.e., $T_j <_{l_i} T_k$, can be added into H_G immediately after the interval $I_i(T_j)$). This completes the construction of the global history, which now contains all intervals from all local histories, respecting their local precedence. Following the provided construction rules for our example, we first add all intervals corresponding to pairs of update transactions ordered in RTC:

$$I_1(T_3), I_2(T_3), I_1(T_4), I_2(T_4)$$

Then we add the intervals of the rest of update transactions, respecting local precedence:

$$I_1(T_1), I_1(T_2), I_2(T_2), I_1(T_3), I_2(T_3), I_2(T_1), I_2(T_5), I_1(T_4), I_2(T_4), I_1(T_5)$$

Finally, there is a single read-only transaction, T_6, which is added respecting the local precedence of delegate node R_2. This finishes the construction of H_G:

$$H_G : I_1(T_1), I_1(T_2), I_2(T_2), I_1(T_3), I_2(T_3), I_2(T_1), I_2(T_5), I_2(T_6), I_1(T_4), I_2(T_4), I_1(T_5)$$

A one-copy history H_{1C} can be constructed from any complete local history H_i, deleting subindexes from the intervals and properly adding the intervals of read-only transactions whose delegate replica is different from R_i. Each one of those intervals can either be ordered in RTC precedence or correspond to an independent transaction. In the first case, the interval (without its subindex) is appended to H_{1C} respecting the local precedence of the delegate but only with regard to conflicting transactions, i.e., respecting the RTC precedence. In the second case

(independent queries), the interval (without its subindex) can be inserted at any place, e.g., at the end. In our example, we take H_1 and add an interval for T_6, whose delegate node is R_2. As T_6 is ordered in RTC precedence, it must be inserted respecting the local precedence of R_2 regarding conflicting transaction T_5:

$$H_{1C} : I(T_1), I(T_2), I(T_3), I(T_4), I(T_5), I(T_6)$$

Both reads-from relationships and the same-final-writes condition of view equivalence are affected by conflicting transactions. H_G is view equivalent to H_{1C} as the only pairs of conflicting transactions are T_3 and T_4, on one hand, and T_5 and T_6, on the other hand, and intervals of these pairs appear at the same order in both histories,[9] maintaining the reads-from relationships (write-read conflicts) and a part of the same-final-writes condition (write-write conflicts), which is completed by the existence in H_G of intervals of all committed update transactions present in H_{1C}. The order followed by H_{1C} is an equivalent serial order (required by 1SR) for the distributed execution and this order respects RTC precedence (required by 1ASR).

Theorem 3.29. *In a 1ASR database execution there is no inversion.*

Proof. Assume an alteration occurs in a 1ASR execution between conflicting transactions T_j and T_k, resulting in an inversion. Let T_j precede T_k in real time, $T_j <_{rt} T_k$. As transactions conflict, the precedence is maintained in RTC order: $T_j <_{rtc} T_k$. The inversion requires that for some replica R_i, T_k precedes T_j in the local precedence of R_i, $T_k <_{l_i} T_j$. Local history H_i respects local precedence and, thus, it violates RTC precedence. As 1ASR executions guarantee that all local histories are α-histories, and α-histories respect RTC precedence, this is a contradiction. □

Corollary 3.30. *From theorems 3.27 and 3.29, and definition 3.28, it is deduced that 1ASR is guaranteed in a database execution if and only if no inversions occur during such an execution.*

Informally, we can say that a database replication system provides the correctness criterion of 1ASR if the user cannot differentiate it from a traditional stand-alone system even when spreading their accesses all over the set of replicas.

[9] In H_G, this order is respected between intervals of the same replica node.

1ASR is suitable for applications that require a strict level of consistency and cannot tolerate any inversion. However, avoiding all inversions may affect performance. We can relax the imposed conditions by only requiring that individual users do not suffer inversions. This way, even if some inversions are allowed, individual users perceive an atomic image, as if the system were 1ASR.

Definition 3.31. Projection of a history for a set of transactions. We define the projection of a history for a set of transactions $\mathbb{T}$ as the result of deleting from the history all intervals corresponding to transactions not belonging to $\mathbb{T}$.

Definition 3.32. Session. Database users are provided with the concept of session, in order to logically group a set of transactions from the same user. Transactions from different users belong to different sessions. However, it can be left to the user the decision of using one or multiple sessions to group their transactions.

Definition 3.33. Session projection. For each session S_i, composed by a set of transactions $\mathbb{T}_{S_i}$, the projection of a history for the set $\mathbb{T}_{S_i}$ is called the session projection of that history for S_i.

Definition 3.34. σ-history for a session. Given a session S_i, a σ-history for S_i is a history whose session projection for S_i respects the RTC precedence.

Definition 3.35. 1SR+ correctness criterion. We call 1SR+ the correctness criterion that, based on a serializable concurrency control, ensures 1SR and guarantees that each local history of a database execution is a σ-history for each existing session.

The extra requirement of 1SR+ (inversion preclusion in sessions) further prevents the occurrence of undesirable conditions over the ensured 1SR foundation: the equivalent serial order must respect RTC precedence over the transactions of the same client session. From the point of view of an individual user grouping all their transactions within the same session, the system cannot be distinguished from a system guaranteeing 1ASR.

Finally, it is obvious that not all applications have the same requirements. Those not sensitive to real-time precedence do not need to care about such an issue.

Definition 3.36. 1SR$'$ correctness criterion. We define 1SR$'$ as the correctness criterion that, based on a serializable concurrency control, ensures 1SR. No guarantees regarding inversions are provided.

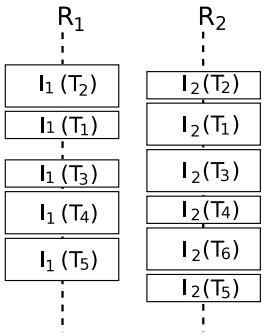


Figure 3.3: Sample execution for an abcast-based (1SR′) system. Two replicas are visualized: R_1 and R_2. Vertical dotted lines represent real time at each replica, increasing downwards.

As no additional constraints are required, this is merely a renaming of 1SR in order to distinguish between the original criterion, which encompasses 1ASR, 1SR+, and 1SR′, and the least restrictive interpretation, which corresponds to 1SR′.

1SR′, as original 1SR, only requires that a given distributed database execution has the same results as a serial execution of the same transactions over a one-copy database. This is the case of the abcast-based systems described in Section 3.3.

Execution 3.4. Figure 3.3 represents the execution in an abcast-based system of the same set of transactions used to show the construction of global and one-copy histories in a 1ASR system (in page 36). In this case, to construct a global history H_G such that it is view equivalent to a one-copy history H_{1C}, we must deal with two types of intervals: (a) intervals of update transactions, and (b) intervals of read-only transactions. The first type of intervals respect the order of their delivery in all replicas, and therefore such an order is also maintained in all local histories. As the order of an abcast is a total order, intervals of update transactions appear following the same sequence at all complete histories (incomplete histories contain only a prefix of that sequence). This ensures the *sequential* nature of the replica consistency perceived by users. Following the order of that sequence, and for each transaction on the sequence and each local history containing an interval for that transaction, such an interval is appended to the global history. Finally, intervals of read-only transactions are inserted into the global history respecting the local precedence of their delegate (e.g., interval $I_i(T_k)$, which was preceded by interval $I_i(T_j)$ in the local order of R_i, i.e., $T_j <_{l_i} T_k$, can be placed into H_G immediately after interval $I_i(T_j)$). This completes the construction of the global history H_G, which now contains all intervals from all local histories,

respecting their local precedence. In our sample database execution, node R_2 presents the inversion of T_6, which overtakes transaction T_5. Local histories are:

$$H_1 : I_1(T_2), I_1(T_1), I_1(T_3), I_1(T_4), I_1(T_5)$$
$$H_2 : I_2(T_2), I_2(T_1), I_2(T_3), I_2(T_4), I_2(T_6), I_2(T_5)$$

To construct the global history, we first append intervals of update transactions and then insert the interval of read-only transaction T_6 respecting the local precedence of node R_2:

$$H_G : I_1(T_2), I_2(T_2), I_1(T_1), I_2(T_1), I_1(T_3), I_2(T_3), I_1(T_4), I_2(T_4), I_2(T_6), I_1(T_5), I_2(T_5)$$

A one-copy history H_{1C} can be constructed from any complete local history H_i, deleting subindexes from the intervals and properly adding the intervals of read-only transactions whose delegate replica is different from R_i. These intervals (without their subindex) are inserted into H_{1C} respecting the local precedence of their delegate, as during the construction of H_G. In our example, we directly use H_2 as it already contains intervals for all committed transactions:

$$H_{1C} : I(T_2), I(T_1), I(T_3), I(T_4), I(T_6), I(T_5)$$

H_G is view equivalent to the serial H_{1C} as the reads-from relationships are the same and the same-final-write condition holds. Indeed, the total order followed to apply writesets at all replicas greatly simplifies the reasoning. If all transactions are update transactions, all replicas would execute all transactions in the same order, resulting in a H_G and a H_{1C} that would also respect that order. Both reads-from relationships and the same-final-writes condition are trivially maintained in that situation. On the other hand, read-only transactions do not affect the same-final-writes condition nor write values that could be read by other transactions, but they read values written by other transactions. As they are included in both H_G and H_{1C} respecting their local precedence,[10] the reads-from relationships are also maintained. Finally, as no guarantees regarding inversions are given, the execution is 1SR′.

We have defined three correctness criteria that subdivide the original 1SR into different possible interpretations. Authors of other works [17, 35, 139], already started to qualify 1SR, although no formalization stated the different approaches, and the underlying reason of the looseness of 1SR was neither clarified. This

[10] Let T_q be a read-only transaction with delegate node R_i. Let T_u be the transaction from which T_q reads in R_i. As replicas run under serializable concurrency control, the updates of T_u are visible only after the commitment of T_u and, thus, T_u will precede T_q in the local order of R_i, i.e., $T_u <_{l_i} T_q$.

way, there are different names in the literature that correspond to the correctness criteria defined here: 1ASR would be equivalent to *strong 1SR*, 1SR+ would match *strong session 1SR*, and 1SR′ would correspond to *(plain) 1SR*.

3.5.2 Synchronization Models

We have expressed 1SR′, 1SR+ and 1ASR correctness criteria with regard to the inversions that are allowed to occur in each of them. As inversions are fully perceived by users, such descriptions of the different levels of consistency match the point of view of users, i.e., they describe the user-centric [129] view of consistency.

Definition 3.37. User-centric view of consistency. The user-centric view of consistency is the perception that users, individually or collectively, have about the consistency of the database replication system in use. This perceived consistency is the user-centric consistency of the system.

Correctness criteria 1ASR, 1SR+ and 1SR′ therefore define three different levels of user-centric consistency: absence of inversions, absence of inversions within sessions, and presence of inversions, respectively.

The user-centric view of consistency depends on the transactions launched by users and is, therefore, intrinsically limited: users view consistency through their transactions. There is a second way of analyzing consistency: from the point of view of the servers, i.e., how consistency is internally provided.

Definition 3.38. Server-centric view of consistency. The server-centric view describes consistency as internally enforced at the replicas of the system. This enforced consistency is the server-centric consistency of the system.

The server-centric view of consistency is a more accurate, more complete description of consistency, as it matches the perception that would have an omniscient observer of the system, not biased by the concrete combination of executing transactions, but with a system-wide knowledge.

The distinction between server-centric and user-centric consistency is very important as strong performance implications are at stake. The stricter the server-centric consistency, the lower the performance. However, a high user-centric consistency may be provided by adding some restrictions to transaction processing

over a system with a relaxed server-centric consistency, i.e., it is possible to relax consistency among servers while still providing users with an image of a high level of consistency, by, e.g., aborting transactions that would reveal to users the occurrence of an inversion. Therefore, distinguishing between user-centric and server-centric consistency allows us to trade consistency for performance while still ensuring a high enough user-centric consistency.

Moreover, there is a tendency in modern applications to use several isolation levels in order to execute different types of transactions. In this case, in a system with a relaxed server-centric consistency, even if it provides users with an atomic –inversions-free– view of consistency through their serializable transactions, other transactions using more relaxed levels of isolation could be able to perceive some inversions.

For these reasons, it is interesting to analyze how consistency is really implemented among servers.

Definition 3.39. Synchronization model for a correctness criterion. In order to enforce a given correctness criterion, servers need to be synchronized. Each possible approach to do so is a synchronization model for such a correctness criterion. Synchronization among servers is performed through different mechanisms: (a) global concurrency control, which establishes when transactions are allowed to proceed, in collaboration with the local concurrency control that ensures certain level of isolation among transactions executed inside the local node; (b) propagation and application of updates, which defines when and how updates are broadcast to other replicas and applied there; and (c) validation rules, which complement the distributed concurrency control by aborting broadcast transactions as needed.

Synchronization models will allow us to distinguish *when* or by means of *which type* of mechanisms the system is able to preclude those inversions it must avoid in order to ensure certain correctness criterion. This distinction is useful in order to estimate the possible differences in performance between the implementations of different synchronization models.

We call *native* those levels of server-centric consistency that can be achieved by using only the mechanisms that control the propagation and application of updates. We consider two native levels of server-centric consistency: natively atomic and natively sequential replica consistency.

Definition 3.40. Natively atomic replica consistency (#A replica consistency). We say that a database replication system providing serializable concurrency control presents the server-centric level of natively atomic replica consistency when, thanks to the mechanisms used for the propagation and application of updates, the occurrence of alterations is absolutely avoided and, thus, no inversions appear. Indeed, with this consistency, no operation is ever able to read the old value of a data item after such an item has been updated by any transaction committed at any replica of the system. More formally, #A replica consistency ensures: (a) for real-time ordered transactions, that no alteration is ever produced (even between independent transactions); and (b) for concurrent transactions, that they present their intervals in the same order at all replicas.

Natively atomic replica consistency is the database version of the atomic DSM memory consistency.

Definition 3.41. 1ASR/#A synchronization model. We define 1ASR/#A as the synchronization model for 1ASR which guarantees natively atomic replica consistency.

As natively atomic replica consistency intrinsically avoids alterations, no additional restriction or mechanism is required for providing 1ASR.

An example of system that follows the 1ASR/#A model would be one in which each transaction is totally ordered before starting execution, and transactions are sequentially executed following that order, in such a way that a transaction must have committed at all available and necessary replicas before the next transaction is started. Such a total order can be determined by a sequencer, by requiring transactions to acquire locks at each replica they must access, or by assigning starting timestamps to transactions, as suggested for the strongly serializable DBMSs [17]. In practice, few systems implement 1ASR/#A.

Theorem 3.42. *1ASR/#A guarantees 1ASR.*

Proof. 1ASR/#A achieves the same serial order at each replica and it moreover ensures that no alteration is ever produced, so it trivially guarantees that no alteration between conflicting transactions is ever produced. □

Definition 3.43. Natively sequential replica consistency (#S replica consistency). A database replication system presents the server-centric level of natively

sequential replica consistency when the mechanisms used for the propagation and application of updates ensure that those updates are committed at each node following the same order.

In natively sequential systems, the sequence of database states (observed or not) is guaranteed to be the same at all replicas. Natively sequential replica consistency is the database version of the sequential DSM memory consistency.

This native level of server-centric consistency is based in a total order processing of writesets: while read-only transactions are executed and committed in their delegate replica, writesets are broadcast in total order [11] and, if successfully validated, are later applied and committed at all replicas following that order. Intrinsically, if an update transaction T_k starts after the commitment of another update transaction T_j and it is not aborted, its writeset will be placed inside the total order of the delivery *after* the writeset of T_j. In other words, with #S replica consistency, there exists a FIFO total order among the commitment of update transactions. This order subsumes the real-time precedence between update transactions and establishes an arbitrary order for concurrent transactions (not ordered in real-time). Read-only transactions are not required to be broadcast in this server-centric level of consistency.

Theorem 3.44. *In #S replica consistency under serializable isolation, inversions of read-only transactions are allowed, inversions of update transactions are not.*

Proof. There are, a priori, four possible situations of alterations between transactions: (1) an update transaction overtakes another update transaction, (2) an update transaction overtakes a read-only transaction, (3) a read-only transaction overtakes another read-only transaction, and (4) a read-only transaction overtakes an update transaction. Inversions are defined between conflicting transactions and two read-only transactions never present conflicts, so type (3) is not a possible type of inversion. #S guarantees a FIFO total order in the commitment of update transactions. This order intrinsically subsumes RT precedence, so no alteration is possible between update transactions, precluding type (1). Moreover, read-only transactions are only executed and committed at *one* replica, their delegate R_i. By the time a query commits, no transaction started afterwards is able to overtake it, rendering inversions of type (2) impossible. The only case left is (4), where a

[11] Atomic broadcast can be substituted with a FIFO broadcast if only one node sends messages, like in primary copy replication. In any case, the result is a totally ordered delivery of messages.

read-only transaction T_q, with delegate R_i, overtakes a conflicting update transaction T_u. This occurs when transaction T_u first-commits in a node different from R_i before T_q starts in R_i. If no mechanism prevents the execution of T_q from taking place before the updates of T_u are applied at node R_i, then an inversion is produced and the query reads outdated values. #S respects RT precedence for update transactions, but nothing is ensured with regard to read-only transactions, so the inversion of a read-only transaction which overtakes a remote update transaction is possible in #S under serializable isolation. □

Definition 3.45. 1ASR/#S synchronization model. We call 1ASR/#S the synchronization model for 1ASR which guarantees natively sequential replica consistency and enhances it with inversion preclusion.

As natively sequential replica consistency allows alterations of read-only transactions, other mechanisms are required for the avoidance of inversions required in 1ASR. 1ASR/#S systems generally avoid alterations of the real-time precedence only for conflicting transactions by means of their validation rules, thus trying to minimize the performance impact of reduced concurrency. A system that follows the 1ASR/#S synchronization model is BaseCON for strong 1SR [139]. In this system, a total order is established among transactions before their start. Update transactions are executed at every node and committed following the order of their delivery. Read-only transactions are scheduled to replicas that have already committed all previous update transactions from any client. Another possible algorithm is to broadcast in total order each read-only transaction when it is launched by users, start its execution optimistically at the delegate and abort it whenever a conflicting update transaction that precedes it in the total order commits at that node. An aborted read-only transaction is restarted when its delivered message is processed.

Theorem 3.46. *1ASR/#S guarantees 1ASR.*

Proof. 1ASR/#S easily obtains an equivalent serial order based on the delivery order and moreover it precludes inversions, guaranteeing that they are not perceived by the user. □

Apart from the native levels of server-centric replica consistency, other levels must be considered. The periodically atomic and the periodically sequential

replica consistency levels can be achieved with the combination of the mechanisms for global concurrency control and for propagation and application of updates.

The periodically atomic replica consistency maintains the intrinsic preclusion of inversions from the natively atomic level with an increased concurrency. Indeed, precluding alterations for every transaction reduces concurrency and performance and a trade-off immediately appears. Let T_1 and T_2 be two transactions executed in a 1ASR/#A system. If $T_1 <_{rt} T_2$, then the interval of T_1 will not appear after the interval of T_2 at any node. In order to ensure 1ASR/#A, all replicas must wait for T_1 to commit before allowing T_2 to start. When T_1 and T_2 are independent, the database state after committing both transactions will be the same either with a serial order T_1, T_2 or T_2, T_1. The system could execute both transactions concurrently to increase performance without implications on perceived consistency. However, if the local precedence at any replica presents alterations with regard to the real-time precedence, natively atomic replica consistency is lost. In that case, the distributed serializable concurrency control of the system still guarantees that all updates in each database item, which are made by conflicting transactions, will follow the same sequence in every replica (as needed by the cache consistency model [49]). Moreover, most of the traditional distributed database systems maintain a synchronous client-server interaction: the client blocks after sending the request message and resumes its activity once the server sends back the reply. This synchronous interaction allows the program order in each of the client processes to be maintained in their updates (as needed by the PRAM consistency model [81]). As a result, the ensured server-centric consistency when different replicas follow different local orders for non-conflicting transactions drops down to processor consistency [49], i.e., cache+PRAM [3]. This was considered by Traiger et al. [130] as concurrency transparency, a characteristic of their one-copy equivalence definition.

Definition 3.47. Periodically atomic replica consistency (*A replica consistency). A database replication system providing serializable concurrency control presents the server-centric level of periodically atomic replica consistency when the mechanisms used for global concurrency control and for propagation and application of updates ensure that no alteration is produced between conflicting transactions, i.e., that inversions do not occur. This way, in periods when each transaction conflicts with any other transaction, this server-centric consistency assimilates to a natively atomic consistency in that no operation is ever able to read

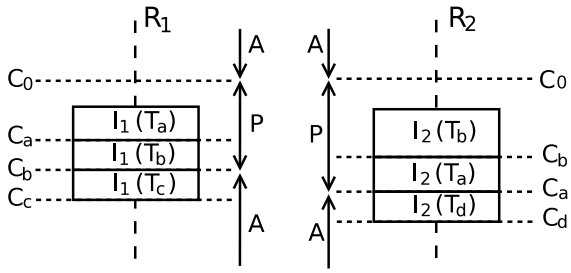


Figure 3.4: A relaxed period (P) in a periodically atomic system

the old value of a data item after such an item has been updated by any transaction committed at any replica of the system. On the other hand, in periods when there exists at least a pair of executing transactions that are independent between them, this replica consistency guarantees only the relaxed level of processor consistency.

The consistency during the relaxed periods of the periodically atomic replica consistency is the database version of the processor DSM memory consistency. In the rest of periods, however, it corresponds with the atomic DSM memory consistency.

Definition 3.48. 1ASR/*A synchronization model. We call 1ASR/*A the synchronization model for 1ASR which guarantees periodically atomic replica consistency.

Algorithms that follow the 1ASR/*A synchronization model are the one described by Bernstein and Goodman [14] and the distributed two-phase locking, wound-wait and basic timestamp ordering algorithms surveyed by Carey and Livny, along with the distributed optimistic two-phase locking algorithm they propose [23]. In these systems, based on two-phase commit, the distributed concurrency control (e.g., to get a write lock at each copy of the data item a transaction wants to update) serializes conflicting transactions but allows independent transactions to run concurrently. In general, we can define 2PC-based systems described in Section 3.3 as 1ASR/*A, with periods of processor consistency when some concurrently executing transactions do not present read-write, write-read nor write-write conflicts. This characteristic alternation between atomic (A) and processor (P) periods is depicted in Figure 3.4. Imagine two non-conflicting

transactions starting at different delegate nodes: T_a writing item x and T_b writing item y. T_a starts at replica R_1 while T_b starts at replica R_2.[12] As transactions do not conflict, write locks are granted and the atomic commit protocol is run concurrently for both transactions. As messages exchanged during 2PC are not ordered, intervals for T_a and T_b may appear at different orders in different replicas. Despite this consistency loss, no observer is able to detect the relaxation within a serializable concurrency control. Suppose an observer (a user) that starts two read-only transactions, T_c and T_d, simultaneously at two replicas that differ in the interval order of T_a and T_b. Both read-only transactions intend to read items x and y. Even if those queries are able to start at their corresponding delegate replica in the precise moment where the first transaction has committed and the second one is still in execution, no inconsistencies could be observed. Suppose that replica R_1 follows local order T_a, T_b and T_c starts between both commit operations. Replica R_2 follows local order T_b, T_a, starting T_d between both commitments. T_c is able to access x but item y is write-locked by T_b. Similarly, T_d can access y but not x. Distributed locks are thus preventing T_c and T_d from perceiving inconsistent (different from each other) database states. Only when both T_a and T_b are committed in both replicas, will T_c and T_d be able to complete their accesses, which will throw the same results in both nodes: the sequence of *observed* database states is the same in all replicas. The consistency-relaxed period spans over the slots containing the intervals of T_a and T_b at each replica. Previous and subsequent periods will present atomic, inversions-free consistency as long as each pair of transactions accesses to, at least, one common item.

Theorem 3.49. *1ASR/*A guarantees 1ASR.*

Proof. 1ASR/*A achieves the same serial order at each replica for all conflicting transactions, and the results of independent concurrent transactions are equivalent to any serialization of those transactions. Moreover, 1ASR/*A ensures that no alteration between conflicting transactions is ever produced, so it guarantees that no inversions are produced. □

This way, 1ASR/*A trades server-centric consistency for performance (in the way of increased concurrency), while still offering a 1ASR image to users.

[12] For the purpose of depicting the alternation of periods, it does not matter if these transactions have a real-time precedence or if they are concurrent. In Figure 3.4, however, T_a and T_b are depicted as concurrent transactions.

The periodically sequential replica consistency maintains the exploitation of the total order delivery of writesets to serialize conflicting update transactions, while increasing concurrency. Indeed, applying all writesets in a sequential manner reduces concurrency. An optimization is possible: to apply in a sequential manner only those writesets that are conflicting, while letting independent transactions to commit at different orders as long as no other running transactions are able to perceive such relaxation of consistency. Similarly to the periodically atomic case, the consistency during the periods when there are pairs of update transactions that are independent to each other is the relaxed level of processor consistency.

Definition 3.50. Periodically sequential replica consistency (*S replica consistency). A database replication system presents the server-centric level of periodically sequential replica consistency when the mechanisms used for global concurrency control and for propagation and application of updates ensure that those updates are committed at each node following the same order as long as they correspond to conflicting transactions. This way, this server-centric consistency assimilates to a natively sequential consistency in that commitments of conflicting update transactions follow the same sequence at every replica. On the other hand, in periods when there is at least one update transaction that is independent to another one, this replica consistency guarantees only the relaxed level of processor consistency, by allowing the concurrent application of independent writesets and letting their commitments to occur at different orders. Global concurrency control mechanisms prevent other running (not committing) transactions from perceiving the loss in consistency caused by such different commit orders.

The consistency during the relaxed periods of the periodically sequential replica consistency is the database version of the processor DSM memory consistency. In the rest of periods, however, it corresponds with the sequential DSM memory consistency.

Definition 3.51. 1ASR/*S synchronization model. We call 1ASR/*S the synchronization model for 1ASR which guarantees periodically sequential replica consistency and enhances it with inversion preclusion.

Periodically sequential replica consistency, similarly to the natively sequential level, allows alterations only of read-only transactions. Indeed, the only difference between both levels involves *independent* transactions, which, by definition,

cannot originate inversions (alterations of *conflicting* transactions). Some mechanisms are needed for the avoidance of inversions required in 1ASR. 1ASR/*S systems generally avoid alterations of the real-time precedence only for conflicting transactions by means of their validation rules, thus trying to minimize the performance impact of reduced concurrency. Moreover, the relaxation in consistency must be concealed from other running transactions by, e.g., getting all necessary locks during validation, before letting independent transactions to freely commit.

Algorithms that follow the 1ASR/*S synchronization model are DBSM-RO-opt and DBSM-RO-cons [96]. Both protocols extend the database state machine replication [101], from which they inherit the optimization over writeset application consisting in allowing independent transactions to concurrently apply their writesets and commit at different orders. This relaxation in consistency is not perceived by other transactions, as writesets get all their write locks as soon as they are certified upon delivery, before they are allowed to proceed. DBSM-RO-opt broadcasts in total order and certifies also read-only transactions, aborting them whenever a potential inversion is detected. In this case, the execution is optimistic but validation rules preclude inversions. DBSM-RO-cons follows a conservative approach, broadcasting in total order read-only transactions when they are launched by users and waiting in the delegate node for all update transactions which appear before in the total order to be committed, before starting the execution of the read-only transaction. In this case, the inversion preclusion is guaranteed by the concurrency control.

Theorem 3.52. *1ASR/*S guarantees 1ASR.*

Proof. 1ASR/*S easily obtains an equivalent serial order for conflicting update transactions based on their delivery order and the results of independent concurrent update transactions are guaranteed to be equivalent to any serialization of those transactions. Moreover, 1ASR/*S precludes inversions, guaranteeing that they are not perceived by the user. □

We have identified four different synchronization models for the 1ASR correctness criterion. These models allow system designers to trade consistency for performance while ensuring that this loss in consistency is not perceived by users, which always obtain a 1ASR image thanks to additional mechanisms that impose certain restrictions over transaction execution. As a brief summary, 1ASR/#A

precludes inversions (in general, any alteration) with the mechanisms used for propagation and application of updates; 1ASR/#S eliminates potential inversions with the validation rules; 1ASR/*A avoids inversions thanks to the global concurrency control; and 1ASR/*S uses validation rules to eliminate potential inversions while ensuring with global concurrency control mechanisms that the periods of relaxed consistency are not perceived. We could say that 1ASR/#A and 1ASR/*A follow a pessimistic approach regarding inversions, while 1ASR/#S and 1ASR/*S use an optimistic approach. This way, both in 1ASR/#A and 1ASR/*A, no operation is able to read the old value of a data item after such an item has been updated by a committed transaction (e.g., all copies were updated on commitment or distributed locks prevent such accesses), while this is indeed possible in 1ASR/#S and 1ASR/*S, where the transaction issuing that operation is ultimately aborted to avoid the appearance of an inversion.

Regarding the 1SR+ correctness criterion, Terry et al. [129] initially proposed *session consistency* as a solution that provides the advantages of precluded inversions while avoiding the costs of atomic consistency. However, the relaxation of their model prevented it from providing one-copy equivalence, as different clients could observe different serial orders for the same set of transactions. Indeed, their guarantees do not attempt to provide atomicity or serializability, as these are considered by the authors as orthogonal issues. More recent papers [35, 36] also propose systems with session semantics, but they build it upon sequential consistency. We follow that trend here.

Definition 3.53. 1SR+/#S synchronization model. We call 1SR+/#S the synchronization model for 1SR+ that guarantees natively sequential replica consistency and enhances it with inversion preclusion on a per-session basis.

The Block and Forward algorithms [35] are examples of systems that follow the 1SR+/#S synchronization model. In both algorithms, a primary site is used to perform all update transactions and lazily send such updates to secondary sites, which apply them sequentially. In the Block algorithm, read-only transactions must wait in their delegate for the required *sequence number*, i.e., for the previous updates of transactions of the same session to be committed in that node. This way, concurrency control is responsible for precluding inversions within a session. In the Forward algorithm, read-only transactions are forwarded to the primary whenever their delegate is not up-to-date, i.e., it has not yet applied all updates from previous transactions of the same session.

Theorem 3.54. *1SR+/#S ensures 1SR+.*

Proof. 1SR+/#S easily obtains an equivalent serial order based on the delivery order and moreover it precludes inversions in sessions, guaranteeing that individual users do not perceive inversions in their sessions. □

Definition 3.55. 1SR+/*S synchronization model. We call 1SR+/*S the synchronization model for 1SR+ that guarantees periodically sequential replica consistency and enhances it with inversion preclusion on a per-session basis.

Taking previous Block and Forward algorithms as a basis, we could design an algorithm for the 1SR+/*S model by making refresh transactions in the secondary sites to acquire locks sequentially but then allowing them to apply updates and to commit in different order. The maintenance of the *sequence number* of the database should also be adapted for this optimization.

Theorem 3.56. *1SR+/*S ensures 1SR+.*

Proof. 1SR+/*S easily obtains an equivalent serial order for conflicting update transactions based on their delivery order and the results of independent concurrent update transactions are guaranteed to be equivalent to any serialization of those transactions. Moreover, 1SR+/*S precludes inversions in sessions, guaranteeing that individual users do not perceive inversions in their sessions. □

Finally, with regard to the 1SR′ correctness criterion, we also take sequential consistency as a basis.

Definition 3.57. 1SR′/#S synchronization model. We call 1SR′/#S the synchronization model for 1SR′ that guarantees natively sequential replica consistency.

The BaseCON for 1SR [139] is an example of system that follows the 1SR′/#S synchronization model. These systems achieve natively sequential consistency by relying in a total order broadcast of update transactions, which establishes the order replicas follow to apply such updates. Indeed, given that many database replication systems apply the set of update transactions in FIFO total order in all replicas, they can be immediately tagged as natively sequential consistent systems that follow 1SR′/#S. In general, we can define abcast-based systems described in Section 3.3 as 1SR′/#S systems, where replica consistency is relaxed and no steps are taken to avoid inversions.

Theorem 3.58. *1SR′/#S ensures 1SR′.*

Proof. 1SR′/#S easily obtains an equivalent serial order based on the delivery order, which is enough for 1SR′. □

Definition 3.59. 1SR′/*S synchronization model. We call 1SR′/*S the synchronization model for 1SR′ that guarantees periodically sequential replica consistency.

Many systems follow the 1SR′/*S synchronization model. For example, DBSM [101] and the SER algorithm [69]. In these systems, non-conflicting writesets are allowed to commit at different orders, while conflicting writesets are enforced to commit in their delivery order. Write locks acquired right after certification prevent other transactions to perceive the relaxation in consistency.

Theorem 3.60. *1SR′/*S ensures 1SR′.*

Proof. 1SR′/*S easily obtains an equivalent serial order for conflicting update transactions based on their delivery order and the results of independent concurrent update transactions are guaranteed to be equivalent to any serialization of those transactions. The achieved serialization guarantees 1SR′. □

3.5.3 Performance Implications

In order to analyze the performance implications that the selection of a synchronization model may have, we define some relations.

Definition 3.61. Relation stricter-than. A correctness criterion CC_1 is stricter than another criterion CC_2 if it prevents more anomalous situations than CC_2 does. This relation is transitive.

Clearly, as 1ASR avoids all inversions, it is stricter than 1SR+, which avoids inversions only in sessions, and than 1SR′, which allows all inversions. Similarly, 1SR+ is stricter than 1SR′.

Definition 3.62. Relation higher-than. A server-centric consistency level CL_1 is higher than another level CL_2 if it corresponds to a higher (according to the hierarchy defined by Mosberger [91]) level of DSM memory consistency. This relation is transitive.

From this definition, we can state that: (a) #A is higher than *A (as natively atomic consistency always provides the atomic level, while periodically atomic consistency alternates between the atomic and the processor levels); (b) #A is higher than #S (as natively atomic consistency provides the atomic level and natively sequential consistency provides the sequential level); (c) #A is higher than *S (as natively atomic consistency provides the atomic level and periodically sequential consistency alternates between the sequential and the processor levels); (d) *A is not comparable with #S (due to the alternation of levels in periodically atomic consistency, as the atomic level is higher than the sequential level, but the sequential level is higher than the processor level); (e) *A is not comparable with *S (although both provide the processor level in their relaxed periods, the alternation with the atomic and the sequential level, respectively, renders these server-centric levels non-comparable, as the atomic level is higher than the sequential level, but the sequential level is higher than the processor level); (f) #S is higher than *S (as natively sequential consistency always provides the sequential level, while periodically sequential consistency alternates between the sequential and the processor levels).

Definition 3.63. Relation harder-than. Let SM_1 and SM_2 be two synchronization models for the same or different correctness criteria. We say that SM_1 is harder than SM_2, $SM_1 >_h SM_2$, if: (a) both models guarantee the same correctness criterion but SM_1 maintains a higher level of server-centric consistency than SM_2 does; or (b) both models maintain the same level of server-centric consistency but SM_1 guarantees a stricter correctness criterion than SM_2 does; or (c) there exists a synchronization model SM_3 such that $SM_1 >_h SM_3$ and $SM_3 >_h SM_2$. This relation is transitive and establishes a partial ordering among synchronization models.

Condition (a) represents the performance penalty of maintaining a tighter synchronization in servers, while condition (b) depicts the cost of the additional mechanisms (or of their application to more situations) that SM_1 requires in order to ensure its more restrictive correctness criterion. Condition (c) establishes the transitivity of the relation.

The fact of using a relaxed server-centric consistency level achieves important performance improvements that increase system scalability. Indeed, performance increases as server-centric consistency relaxes, and the less the needed additional mechanisms, the better. In other words, the harder the synchronization model, the higher the penalty on performance. This relation allows us to estimate beforehand

the differences in performance between two models. The following relationships can be stated:[13]

- 1ASR/#A is harder than 1ASR/*A. They both provide the same correctness criterion and 1ASR/#A guarantees natively atomic consistency, which is higher than the periodically atomic consistency of 1ASR/*A.
- 1ASR/#A is harder than 1ASR/#S. They both provide the same correctness criterion and 1ASR/#A guarantees natively atomic consistency, which is higher than the natively sequential consistency of 1ASR/#S.
- 1ASR/#S is harder than 1ASR/*S. They both provide the same correctness criterion and 1ASR/#S guarantees natively sequential consistency, which is higher than the periodically sequential consistency of 1ASR/*S.
- 1ASR/#S is harder than 1SR+/#S. They both provide natively sequential consistency and the correctness criterion ensured by 1ASR/#S is stricter than that of 1SR+/#S, and thus it requires mechanisms that preclude all inversions, while 1SR+/#S applies such mechanisms only in the context of a single session.
- 1SR+/#S is harder than 1SR+/*S. They both provide the same correctness criterion and 1SR+/#S guarantees natively sequential consistency, which is higher than the periodically sequential consistency of 1SR+/*S.
- 1ASR/*S is harder than 1SR+/*S. They both provide periodically sequential consistency and 1ASR/*S ensures a stricter correctness criterion than 1SR+/*S (1ASR/*S requires mechanisms that preclude all inversions while 1SR+/*S applies such mechanisms only in the context of a single session).
- 1SR+/#S is harder than 1SR′/#S. They both provide natively sequential consistency and 1SR+/#S, to ensure a stricter criterion, requires mechanisms that preclude inversions within sessions, while 1SR′/#S does not.
- 1SR′/#S is harder than 1SR′/*S. They both ensure the same correctness criterion and 1SR′/#S maintains natively sequential consistency, which is higher than the periodically sequential consistency of 1SR′/*S.

[13] As the relation is transitive, we enumerate only direct relationships.

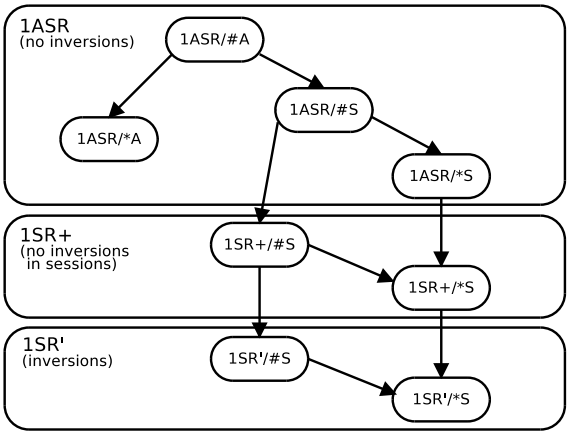


Figure 3.5: Hardness hierarchy in synchronization models for the proposed correctness criteria

- 1SR+/*S is harder than 1SR′/*S. They both provide periodically sequential consistency and 1SR+/*S ensures a correctness criterion that is stricter than the criterion of 1SR′/*S.
- 1ASR/*A is not comparable with 1ASR/#S or 1ASR/*S. They provide the same correctness criteria but their server-centric consistency levels are not comparable.
- 1ASR/*A is not comparable with synchronization models of other correctness criteria, as its server-centric consistency level is not shared with any other model.

Although performance is a property of particular implementations and none of the correctness definitions necessarily leads to poorer performance than others, the harder-than relation allows us to make some estimations and predict that 1ASR/#A systems will likely present the poorest performance, as they ensure the highest server-centric consistency level and guarantee the strictest correctness criterion. For 1ASR systems, those following 1ASR/*S will probably present high performance, while 1ASR/*A systems will depend on their alternation (the more and larger the periods of relaxed consistency, the higher the performance). Some of these estimations are confirmed by comparative studies of performance [35, 96]. Figure 3.5 summarizes the proposed correctness criteria for serializable

database replication, also showing the hardness hierarchy defined by the partial ordering of the harder-than relation among the different synchronization models.

3.6 Consistency in Highly-Scalable Data Systems

In this section, we briefly comment about the consistency provided in current highly scalable systems of cloud environments. The model followed in these systems is substantially different from the system model considered in this chapter: cloud systems generally follow a quorum-based approach, instead of the ROWAA approach; isolation and concurrency control inside each server is usually relaxed; and, moreover, the relational schema is often substituted by a key-value schema, devoid of SQL features.[14]

Almost all cloud systems rely on the partitioning of persistent data [125] among a potentially high number of servers: the whole set of data is partitioned and each partition is stored at a different server. Of course, replication is often used, e.g., storing a given partition in more than one server. This partitioning is made in such a way that users' queries and updates access a minimum number of partitions (ideally, only one), in order to minimize as much as possible the required synchronization among servers (otherwise, the introduced delays would be intolerable for these systems). Other solutions [39] use a mechanism that fragments requests, sends each fragment to the corresponding partition and later unifies all responses in a single reply to the client. In any case, the key to achieve scalability lies in a minimum coordination among servers,[15] and thus consistency is sacrificed for performance. Updates to a data item are lazily propagated and no distributed concurrency control is in use. Thus, server-centric consistency is neither natively nor periodically atomic. During execution, each copy of a data item may present different values and even go through different sequences of values: server-centric consistency is neither natively nor periodically sequential nor even processor. Indeed, the usual consistency level ensured in these systems is the

[14] There are, however, some exceptions to this general tendency. Some systems, e.g., the one described by Kallman et al. [67], maintain the relational model, by moving the DBMS storage to main memory and removing concurrency control and multi-thread management (thus forcing a sequential execution), and even are able to achieve better performance than that of key-value schemas for some applications [100]. Other illustrative examples are Microsoft SQL Azure [22, 87] and Relational Cloud [33].

[15] Nevertheless, coordination is still required in scalable systems for different needs and several services can be used to achieve it, such as Boxwood [84], Chubby [19] or ZooKeeper [66].

eventual consistency [133]: if no more updates are scheduled, all copies of a data item will eventually present the same value, thanks to reconciliation mechanisms that allow diverged copies to agree on a value. Eventual consistency seems the price to pay for an extreme scalability. With such a relaxed consistency, one-copy equivalence is beyond the current possibilities of most cloud systems.

3.7 Discussion and Conclusions

Serializability is the highest level of transaction isolation. Under it, the effect of concurrently executing transactions must be the same as that of *any* serial execution of those transactions. Traditional stand-alone databases providing serializability were actually more restrictive than required by the model: the equivalent serial execution never presented alterations with regard to real-time precedence. Database users became accustomed to this stricter behavior, which ensured that a transaction executed after another is able to see the effects of that previous transaction. This became a fundamental principle of stand-alone systems from the users' point of view, while never required by any correctness criterion.

When databases were distributed and replicated for increased performance, one-copy equivalence was added to serializability to define one-copy serializability (1SR) [15], under which the interleaved execution of transactions must be equivalent to a serial execution of the same set of transactions on a one-copy (non-replicated) database. Early database replication systems based on distributed locking and atomic commit (2PC-based systems), designed to provide 1SR, were again more restrictive than required, due to the employed techniques. Users did not complain: distributed systems behaved as traditional stand-alone ones.

Later, performance-improved systems appeared, where old techniques were substituted with deferred update propagation and atomic broadcast (abcast-based systems). One-copy serializability was guaranteed but nothing enforced the more restrictive behavior to which users were accustomed: a transaction executed after another might not be able to see the effects of that previous transaction, due to what we call an inversion.

The difference between 2PC-based and abcast-based systems is a different consistency level among replicas. This aspect was never considered in database correctness criteria, thus allowing replication systems to freely relax replica consistency, depriving users of the stricter behavior they were familiarized with, while still being as correct as other systems maintaining that behavior. Both system models were equally acceptable under one-copy serializability. This correctness criterion became then a loose term but, up to our knowledge, such a fact has not been stated nor justified yet. However, the consistency level provided by recent systems has been branded as weak, strong, strict, etc., in an attempt to better characterize it with regard to other 1SR systems.

We consider that the ambiguity stemming from the fact of not considering replica consistency in correctness criteria for database replication systems is not trivial, as it may lead to user confusion and unfairness when comparing different replication systems. Our group [95] already raised this topic before. We think that an explicit distinction must be made. Memory consistency models can be borrowed from the distributed shared memory scope in order to clearly state the features of each system. Starting from these models, we propose three different correctness criteria that ensure the original 1SR criterion: 1SR′, 1SR+ and 1ASR. These criteria differ in the inversions that users may perceive and therefore define three levels of user-centric consistency: in 1ASR no inversions are perceived, in 1SR+ no inversions within user sessions are perceived, in 1SR′ inversions are perceived. A complete formalization is also provided. Moreover, we have further distinguished among 1SR database replication systems, according to the exact consistency level actually ensured in the set of replicas (the server-centric consistency). This distinction is important, as the actual DSM consistency model implemented by a replication protocol partially explains its complexity and performance. Indeed, significant performance boosts can be achieved with minor DSM consistency relaxations. Regarding systems that guarantee 1ASR, the highest level of consistency from the point of view of users, while 1ASR/#A systems ensure a strict consistency level in their servers, 1ASR/*A systems offer a relaxed consistency in some periods. Two other groups, 1ASR/#S and 1ASR/*S systems, achieve greater performance by relying on a continuously relaxed server-centric consistency but adding some restrictions to transaction processing and validation. However, it is important to notice that such server-centric consistency differences could be seen by users should their applications not only use a serializable isolation level but also execute part of their transactions under a very relaxed isolation level, an increasingly common practice in modern systems.

We have identified *2PC-based* systems (those using distributed locking and the two-phase commit protocol) as 1ASR/*A systems, where the periodically atomic server-centric consistency jumps from the atomic to the processor level but user-centric consistency is always 1ASR, thanks to the concurrency control mechanisms. On the other hand, general *abcast-based* systems (those locally executing transactions and finally broadcasting them in total order) follow 1SR′/#S (with natively sequential replica consistency) and ensure the correctness criterion of 1SR′, where inversions may arise.

To conclude, there are two answers for the question that started this chapter: "Are one-copy equivalent database replication systems actually behaving as one copy?". According to the theory on isolation levels and consistency models, one-copy equivalent systems do behave as one copy. According to the users' experience, probably accustomed to the behavior of traditional stand-alone databases (more restrictive than the correctness model they enforced), only systems ensuring 1ASR (1ASR/#A, 1ASR/*A, 1ASR/#S and 1ASR/*S) behave as one copy. The transparency required to all distributed systems [127] would not be guaranteed in relaxed, replicated databases providing 1SR+[16] or 1SR′.

[16] Although each individual user of a 1SR+ database is provided with a one-copy image of the system, multiple users (or an individual user working with multiple user sessions) may collectively perceive the relaxation in replica consistency.

Chapter 4

A Characterization Model for Database Replication Systems

In this chapter we present a policy-based characterization model that allows us to decompose database replication algorithms into individual interactions between significant system elements, as well as to define some underlying properties, and to associate each interaction with a specific policy that governs it. With this characterization model, a replication system can be described as a combination of policies. This common framework allows an easy understanding and comparison between protocols.

4.1 Introduction

Since traditional stand-alone database systems started to become distributed and replicated in the mid seventies, many different algorithms for concurrency and replica control have appeared thanks to the contributions of many authors. These proposals came from different communities, each one based on different assumptions and focused on the achievement of different goals. Each new distributed or replicated system defined its own methods, followed its own naming conventions and presented its algorithms in different ways: from descriptions in plain textual form to more or less detailed specifications in its own pseudocode language. In

the midst of this abundance and disparity, it was difficult to find an appropriate solution for a given problem or to compare two apparently similar options to choose the best one for a given scenario.

Some authors from the distributed systems community performed different surveys and classifications [52, 136, 138]. Gray et al. [52] made the first step to study the existing systems for database replication, distinguishing between eager and lazy propagation strategies, and group and master ownership strategies, which combine between them to produce four types of replication systems. Wiesmann et al. [138] proposed a classification based on three parameters, where replication techniques are characterized with regard to their server architecture (either primary backup or update everywhere), their server interaction (either constant or linear) and their transaction termination (either voting or non-voting). Both contributions aimed to classify a broad set of non-related systems, according to some criteria, generally coarse-grained in order to reduce the complexity and the number of equivalence classes. Each one of these criteria thus agglutinated several individual pieces of behavior that were observed together in the studied systems. Another approach was focusing on one set of similar systems, e.g., those mainly based on a given general technique, and characterize and classify them into disjoint subsets, according to other used techniques. This is the case of a work by Wiesmann and Schiper [136], which is focused on replication systems based on total order broadcast –namely, the three most relevant: active, certification-based and weak voting replication– and provides a performance comparison between them and two other widely used techniques –primary copy and lazy replication– that do not rely on group communication.

However, more and more systems appeared and those coarse-grained criteria turned to be insufficient when proposals became hybrid or explored new methodologies not yet categorized. A finer grain is thus necessary to better characterize replication systems and to provide a common ground to compare them all. More than a set of disjoint equivalence classes, what is needed is a common and general framework where different replication systems could be examined and compared. This approach was followed by Bernstein and Goodman in 1981 [13], when they surveyed almost all concurrency control algorithms for distributed databases published until then. In order to do so, they first proposed a framework consisting of a common terminology and a problem decomposition. By unifying concepts and splitting a complex process into several subproblems, a rich characterization and comparison among systems is possible.

Important advances have emerged in the last 30 years since the work of Bernstein and Goodman [13] was published, such as the development of group communication systems with more complex communication primitives, leading to the appearance of new techniques. A new framework for comparing replication techniques was then proposed by Wiesmann et al. [137]. In this framework, five generic phases are identified within a replication protocol: request, server coordination, execution, agreement coordination, and response. According to this, authors then describe different techniques, analyzing how they perform each phase.

Following a similar approach and trying to help researchers and practitioners to make their way through the assorted plethora of database replication systems, this chapter proposes a new characterization model that provides even more detailed descriptions than the framework by Wiesmann et al. [137], by splitting a replication system into a group of policies. This model allows us to describe in detail the nature of the interaction between significative system elements: the underlying local database, the clients and their transactions, and the group of system servers or components. Every time these elements interact, a specific policy regulates the way on which this interaction is performed. Thanks to the fine grain achieved by this model, almost all existing systems can be fully characterized. The resulting detailed descriptions allow an easier comparison between different database replication systems.

The extensive historical survey of Chapter 5, based on the model presented in this chapter, provides not only an empirical proof of the usefulness of our proposal but also a study of the evolution of database replication systems and a reference manual for readers interested in this field, regardless of their background.

The rest of the chapter is structured as follows. Section 4.2 presents the proposed characterization model, and, next, Section 4.3 enumerates the different correctness criteria considered by the surveyed replication systems.

4.2 Interactions, Properties, Strategies and Policies: A Characterization Model

A database replication system can be defined by means of describing the *interactions* among its main components –namely clients, local databases, servers or other system components, and transactions being executed– as well as some basic

behavioral properties. Each one of these interactions may be performed in different ways. Similarly, each property may take different values. All these options are called *strategies*. A replication system must choose, for each interaction and property, one of the available strategies. The selected strategy is the *policy* that such a system follows for such an interaction or behavior. Each system will provide the necessary mechanisms for implementing the selected strategy. The set of policies a system follows can be divided into four policy families, which gather related policies together: the client policy family, the database policy family, the group policy family, and the transaction policy family.

The client policies regulate the interaction between the client and the rest of the database replication system, i.e., the communication from/to the user. The *request* policy specifies which servers in the system must receive the client request, and the *response* policy establishes the number of replies that will arrive to the client with the transaction results.

The interaction between the system and the local underlying database management system is defined by internal properties of the DBMS and regulated by the database policies. These policies determine two aspects: the *isolation* level used whenever the transaction operates in the database, and the level of *replication* of the database, i.e., whether it is fully or partially replicated.

The interaction among the servers or other system components is regulated by the group policies. In order to globally coordinate the execution of transactions, the participation of more than one server is required. Communication is established among the instances of the replication protocol running in different nodes of the system group. These policies control any procedure involving available replicas used to coordinate them with regard to each transaction, i.e., any synchronization (real or logical) between the system nodes required for achieving replica consistency. We distinguish four intervals in the transaction lifetime when different group policies may be applied: *start* (at the start of transaction, before the first access), *life* (during the lifetime of the transaction, e.g., per-operation communications), *end* (at the end of transaction, before the commit operation), and *after* (after termination, i.e., after returning the results to the client).

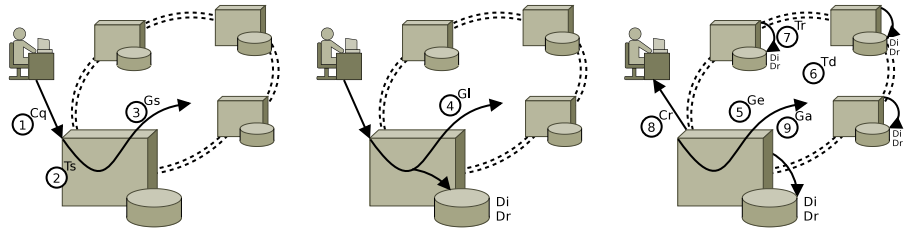
The interaction between the system and the user transactions is regulated by the transaction policies. The *service* policy ensures that the necessary conditions (apart from the obvious resource requirements) hold for the system to serve an incoming request, accepting a transaction for processing. Transaction execution

can be split into two phases: the local phase, in which they are only executed in the delegate server (either interactively or not); and the remote phase, in which their execution spans to the rest of replicas, after some kind of coordination between the system nodes. Right before starting the remote execution, the *decision* policy determines the procedure followed to decide which transactions will commit and which ones must be aborted. If the system decides to commit the transaction, the *remote* policy controls the way the transaction is sent to the database to be applied.

Some *glue* procedures will be usually needed to chain the previous policies in a way that guarantees a correct behavior and the isolation level promised by the replication system.

According to our proposal, the lifetime of a transaction is thus controlled by different policies depending on its current execution step (see Figure 4.1). First, when the client sends its request to the system (identified as interaction 1), the client-request policy determines to which servers this request must be addressed. Once in the appropriate server (or servers), the processing of the transaction is accepted as determined by the transaction-service policy (interaction 2). Once the transaction is accepted for processing, the group-start policy defines if some coordination must be done with the rest of replicas prior to transaction start (interaction 3, e.g., broadcast the start of transaction to the rest of nodes in order to get a global common starting point). After this starting coordination (if it exists), the transaction *enters* the database for the first time, beginning its local execution phase. Database properties affect the transaction execution since this moment on. The database-replication policy will define if there is a copy of the data in the current server or whether the transaction must be distributed among several nodes. At each local DBMS, accesses to the database are controlled by the database-isolation policy. During the local execution phase, a group-life policy may apply, defining a linear coordination among servers (interaction 4). After all operations have been completed, the client asks for the commitment of the transaction.[1] Then, a new communication can be established, following the group-end policy (interaction 5). Prior to transaction termination, a decision process controlled by the transaction-decision policy inspects the transaction and decides if it can be committed or not (interaction 6). If the decision is positive, the transaction enters its remote phase in all nodes. The transaction-remote policy determines when

[1] It is also possible that the client issues an abort operation. In that case, following steps aim to rollback all executed operations instead of committing them.



a. Start of transaction b. Life of transaction c. End of transaction

Figure 4.1: Policies applied during the lifetime of a transaction. Interactions are numbered following the sequence of their first execution.

the transaction can access the local database (interaction 7), where it will be applied according to the database-isolation policy. After transaction completion, the client-response policy regulates the sending of transaction results to the user (interaction 8). Finally, a group-after policy may apply (interaction 9), as in lazy systems.

The sequence of interactions presented above may be adapted to different ways of transaction execution, by selecting the proper strategy to execute each interaction or by denoting that certain interaction will not take place.

Table 4.1 presents the entire classification of the strategies we have found in existing systems for each policy. Identifiers are given to each strategy. This way, we can easily say that a given replication system follows, e.g., the group start policy Gs0, to represent that no communication is established among the group of servers before transaction start. Note that a greater digit in the identifier means a greater effort or a stricter criterion. Now we offer a detailed description for each policy, following the order on which they regulate the transaction lifetime.

Client-request policy Firstly, the client should address its request to the system, directly communicating with one or more elements. Depending on the characteristics of the system, this request may be forwarded to other elements or redirected to a special component by means of another component acting as a proxy (such as a load balancer or scheduler). This policy thus defines the set of servers that finally receive the original user request for processing. When any server is capable of processing a user request (Cq1), it is commonly the client itself who

Table 4.1: Available strategies for each policy

Policy family	Policy	Id	Strategy
Client policies (C)	Request (q)	Cq1	any server
		Cq2	special server
		Cq3	quorum of servers
		Cq4[a]	all servers
	Response (r)	Cr1	one answer
		Cr2	multiple answers
Database policies (D)	Replication (r)	Dr1	partial replication
		Dr2	full replication
	Isolation (i)	Di0	undefined
		Di1	read committed
		Di2	snapshot
		Di3	serializable
		Di4	customized
Group policies (G)	Start (s), life (l), end (e), after (a)	Gx0	no communication
		Gx1[b]	one server [0..1]
		Gx2[b]	several servers [0..n]
		Gx3[b]	all servers
Transaction policies (T)	Service (s)	Ts0[c]	immediate service
		Ts1[c]	deferred service
		Ts2	no local service
	Decision (d)	Td0	no decision
		Td1[d]	one server
		Td2[d]	each server
		Td3[d]	quorum-based
		Td4[d]	agreement-based
	Remote (r)	Tr0	no remote execution
		Tr1[e]	concurrent
		Tr2	non-overlapping

[a] An appended *t* indicates a broadcast in total order.
[b] *n*, no order requirements; *f*, FIFO order; *t*, total order.
For Gs and Gl: *a*, asynchronous; *s*, synchronous.
[c] *n*, no interactivity.
[d] *r*, readset required; *w*, writeset required.
[e] *p*, controlled by the protocol; *d*, controlled by the database.

selects the closest site and sends its request directly. If only one server can process a given user request (Cq2) due, e.g., to some ownership criteria (the primary copy in the system or the server that controls the portion of the data that the user needs to access) or to the decision of a non-trivial scheduler that selects one specific node (not just the least loaded one but one satisfying certain condition), then the user request must be forwarded to this special server. If no kind of request redirection is performed by the system, then the client itself must know how to select the specific node. It is also possible that the user request must arrive to several servers in the system: either a quorum of nodes (Cq3) or the entire group (Cq4). In this last case, some ordering guarantees may be necessary (as in the group policies, as explained below). Thus, a letter 't' appended to the strategy identifier indicates that the multicast must follow total order, which is a usual option for the processing of active transactions. In any case, each system will provide the required mechanisms for implementing this policy.

Transaction-service policy Once the transaction arrives to a node specified by the client-request policy, it enters some sort of queueing system, where it waits for the protocol running in that node to start serving it. This policy reflects the existence of any necessary, non-obvious condition for the node to continue processing incoming requests in general, or this specific request in particular. Waiting for the necessary computational resources (idle threads, available connections to the database, etc.) is considered trivial and included in the default bottom policy (Ts0, immediate service). When the necessary resources are available but any other conditions temporarily prevent the system from processing a transaction, the service is deferred (Ts1). This condition must be locally evaluable, without the participation of other nodes (when a cooperation with the rest of nodes is required to start a transaction, this is reflected in the group-start policy). For example, there may be situations where the node must postpone all incoming requests, e.g., after detecting some inconsistency on the data and until it undergoes reconciliation; or postpone the processing of a query until all pending remote transactions are applied in the node. In a more complex situation, the data could be divided into conflict classes and incoming requests appended to several conflict queues, depending on the data they needed to access. In this scenario, only when the transaction were at the first position in all its queues, it would fulfill the condition to be processed by the system.

These two cases, the immediate and the deferred service, apply to all transactions that have a local execution in their delegate prior to their remote execution

phase: either interactive transactions (where the user sends each transaction operation separately to the database, getting intermediate results as operations are completed), and service requests (calls to stored procedures). In the case of interactivity, it is possible that also intermediate operations of a transaction (not only the first one) are subjected to wait. We extend the concept of deferred service to model also those cases. In order to highlight the non-interactive cases, a letter 'n' will be appended to the strategy identifier.

In distributed (partitioned) and partially replicated databases, the transaction-service policy controls the creation of local subtransactions in each of the participating nodes or cohorts.

Finally, there are also situations where a transaction is not *locally* –individually– processed by any node, but rather has an active execution in all sites at the same time (generally, this precludes interactivity and is mostly used for service requests). We therefore consider that an active transaction has only *remote* phase because, since its starting point, its execution spans all available servers, i.e., there is no previous phase where it is locally executed by one delegate. However, this could be also considered the other way around: as the remote execution of transactions is usually based on the application of logs or writesets (previously created by a delegate node which carried out all the transaction operations), we could say that active transactions are *locally* executed by all nodes and thus have no remote phase. However, we select the first approach –only remote phase– and consider that, in these cases, a policy of no local execution (Ts2) applies. Note that the digit of this last identifier is greater than the previous ones, as an active processing of transactions, where all nodes must perform all operations, is generally more costly than having a local execution phase and a later writeset propagation.

The two last details, i.e., subtransactions of distributed transactions and active execution, are denoted in Figure 4.2. In this diagram, which provides a visual representation of the applied policies during transaction lifetime, the horizontal line is time and it increases rightward.

Group-start policy Before starting a transaction, some coordination among nodes may be necessary. This communication will commonly include some global identifier for the transaction, in order to establish a synchronization point

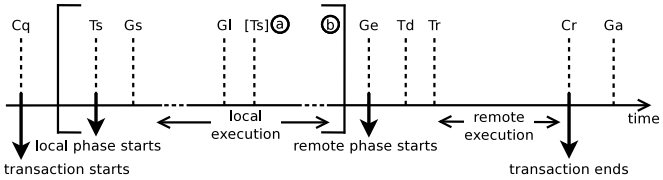


Figure 4.2: Interactions defining the local and remote portions of a transaction. Non-distributed transactions do not initiate subtransactions in multiple nodes (a). Active transactions do not present local phase (b).

before any operation is executed by the transaction. Client-request and group-start policies may seem identical but they present a crucial difference: servers that receive the client request, as expressed in the client-request policy, generally process that request in the same way, i.e. they usually have all the same role regarding to that transaction; while servers contacted at transaction start, as defined in the group-start policy, generally play a different role from that of the first set.

To perform this first coordination, the communication among nodes may or may not need to be synchronous, halting or not the processing of the transaction until some condition holds (e.g., the message is delivered or all replicas reply to the sender).[2] Network communication involves some cost, particularly when some safety or ordering guarantees are required. Thus, an asynchronous communication allows the overlap of the communication cost with the transaction processing, while synchronous communication does not. To distinguish between both situations, an 'a' appended to the policy identifier will denote asynchronous communication, while a 's' will mean the need for synchrony.

All policies of the group family share a common set of strategies, which define the number of servers or other system components the local node must contact with. In the trivial case, no coordination is done (Gx0, where 'x' is 's' for group-start strategies, 'l' for group-life ones, 'e' for group-end options, and 'a' for group-after strategies), e.g., in active replication, the synchronization point is at the beginning of the transaction, so no further synchronization is needed at the end. Non-trivial strategies require the participation of at most other server (Gx1), of a subset of the entire group (Gx2), or the participation of all system nodes (Gx3). In order to implement group coordination, different communication primitives are used. A simple option is the use of node-to-node messages, e.g., some gossip,

[2]Note that although such wait is only necessary in the sender node, it is also part of the required initial coordination, and thus, of the group-start policy.

flooding or cascade mechanisms can be implemented this way. This option was commonly used in initial systems, when GCSs were not yet used for database distribution or replication. More complex primitives include reliable multicasts and broadcasts, with or without order requirements. Multicasts to a quorum are used in quorum-based systems. Reliable broadcast is enough when only one node acts as a sender or when other mechanisms already provide all the order requirements of the system. FIFO or total order[3] broadcasts may be necessary. When a specific ordering guarantee must be provided, an 'f', for FIFO, or a 't', for total, is appended to the strategy identifier. An 'n' will denote that no ordering guarantees are needed.

Database-replication policy Apart from purely distributed systems with no replication at all, not considered in this thesis, replicated databases may enforce replication at different degrees. When each node stores a complete copy of the database, the system features full replication (Dr2). Otherwise, a partial replication is maintained (Dr1).

Database-isolation policy Whenever a transaction is executing in a node (local interactive phase, local non-interactive execution, remote execution and writeset application, and final commit phase), a certain isolation level is enforced in the local database: read committed (Di1), snapshot isolation (Di2), serializable (Di3) or a customized level (Di4), achieved out of the DBMS by directly controlling the locks or making any other management. An additional undefined strategy (Di0) represents that the isolation level was not specified in the system description. Upon the isolation provided in the local database, the replication system is able to enforce certain *global* isolation level for all transactions running in the system.

Group-life policy During the lifetime of a transaction, while it submits operations to the database, some coordination among nodes may be required. When such a coordination exists, it is usually done before or after each single operation (e.g., each SQL statement), sending information about it for, e.g., acquiring locks in remote replicas. Similarly to the group-start case, the execution flow of the transaction may or may not be suspended until this coordination is completed. An appended 'a' ('s') will denote asynchronous (synchronous) communication.

[3] Virtually all GCSs include FIFO guarantees in their total order primitives, so in practice it is assumed that abcast messages respect FIFO ordering.

Group-end policy When the transaction finishes submitting operations, and upon request of commitment, a global coordination is usually needed. During this communication, transaction information, like the readset, writeset and updated values, is commonly spread among system nodes. Sometimes, several rounds are required for appropriate coordination. This is the case of the two-phase or the three-phase commit protocols.

Transaction-decision policy After the transaction has completed its operations and the group-end coordination has been made, a validation process may be run to decide the final outcome of a transaction, i.e., its abortion or commitment, depending on certain conditions. This policy determines which server or servers, if any, are responsible for taking this decision, i.e., for running the decision process. When the rest of the policies is enough –and especially for read-only transactions, which are usually immediately committed in relaxed correctness criteria–, no decision process is executed (Td0). Otherwise, the process may be executed by only one server (Td1), commonly the delegate, which later sends the decision to the rest of nodes; or be performed by each server (Td2) in a symmetric, independent and deterministic way (certification). The process can involve also the collaboration of multiple nodes. This is the case of decisions based on a consensus among a quorum of nodes (Td3), where each server of the quorum informs whether it agrees or not to commit the current transaction; and decisions based on an agreement among all the (available participating) sites (Td4). This latter policy is used when performing a two- or a three-phase commit, where each server says if it agrees to commit the current transaction.

The decision about the final outcome of transactions is usually based on conflicts although it can be also based on some other information (e.g., temporal criteria). When based on conflict checking, an 'r' (respectively, a 'w') after the policy identifier will indicate the use of readsets (writesets) during the decision process. It is important to note the possible different uses of readsets and writesets, which affect performance at different degrees. Thus, decision may be based on conflicts but delegated to the local DBMS (low cost); or it may be necessary to collect those sets and inspect them at middleware level (medium cost); or even to forward them to other servers (high cost).

A final consideration must be done about the decision process. Although normally it is run upon writeset delivery in order to decide the outcome of such a writeset based on the conflicts with previously delivered transactions, in some

systems it is the delivered writeset which, during the decision process run upon its delivery, may cause the abortion of other (local) transactions that, although already broadcast, will be delivered afterwards. Thus, a decision process is run but not for deciding the outcome of the current writeset, but that of future writesets, in a sort of *early* decision.

Transaction-remote policy After completing the local execution phase and getting a positive decision, transactions start their remote execution. For this, transaction information must be somehow provided to every system node. As this is usually done through a GCS, we refer to this information as a delivered transaction, a delivered writeset or, simply, a writeset. At a given node N_i, when the delivered transaction is local, i.e., N_i is its delegate node, only the commit operation is pending. Otherwise, the writeset is remote and its updates must be applied in the local database of N_i prior to final commitment.[4] The access to the local database is controlled by the transaction-remote policy. Two main strategies are considered: either multiple transactions are sent concurrently to the database (Tr1), thus improving performance, or they are sequentially sent, one at a time, following a non-overlapping policy (Tr2) where each delivered transaction must wait for the completion of the previous one. In the first case, conflicting operations must be controlled in order to maintain replica consistency. This control may be performed by the protocol (e.g., the protocol checks for conflicts between writesets before sending multiple, non-conflicting transactions to the database) or by the concurrency control of the database management system (e.g., transactions set write locks in an appropriate sequence before accessing the database). A letter after the identifier specifies if the control is made by the protocol ('p') or by the database ('d').

A third strategy represents the cases where no remote execution is performed (Tr0), namely for read-only transactions that are executed only in their delegate server.

Another aspect that should be mentioned here is the need to abort local transactions, in their local execution phase, holding locks or otherwise preventing remote transactions from being applied in the local database. Such a process is required for protocol liveness but details about its implementation are rarely given in publications. This *clearing* process differs from the early decision commented above

[4] In the case of partially replicated databases, only the updates corresponding to items stored in the node must be applied when processing the delivered writeset.

in that the clearing process aborts transactions that were completely local to the running node (i.e., other nodes had no knowledge of its existence), while an early decision may abort transactions that, although local to the running node, were already broadcast to other nodes. Therefore, in an early negative decision, it will be necessary to broadcast this outcome to remote nodes.

Client-response policy After transaction completion, the client must receive the results of its request. Either one or multiple replies can be sent to the client. Commonly, only the delegate server (or some special node or component in the system) replies, so the client receives only one answer (Cr1). In other cases, multiple replies arrive to the client (Cr2). This distinction is important as, in the latter case, the client has to perform some kind of procedure to select the final answer (the first received, a combination of multiple replies, the most voted, etc.).

Group-after policy After sending the response to the client, once the transaction has committed in one or several nodes, a last coordination may be needed, e.g., for updating remote nodes in lazy systems.

4.3 Correctness Criteria for Replicated Databases

Correctness in replicated databases comprises two characteristics: (a) the isolation level, responsible for the isolation among all concurrent transactions being executed in the system; and (b) the replica consistency, or the degree of admissible divergence among the states of all replicas [95]. The first characteristic is provided by means of a local DBMS in each server and by using certain validation rules at replication protocol level. The second aspect is enforced by the replication protocol and involves synchronization among replicas, which can be made easy by means of a group communication tool. Based on the concepts and conclusions of Chapter 3, we consider the correctness criteria of Table 4.2 for one-copy equivalent systems. The user-centric consistency (i.e., the replica consistency as perceived by users, Definition 3.37, as opposed to the server-centric consistency, Definition 3.38) level where inversions may arise is considered the standard level, while precluding inversions requires a higher effort. For criteria based on isolation levels other than serializability, we choose similar names to those of Chapter 3: e.g., in the case of snapshot isolation, 1ASI corresponds to

systems that preclude inversions, 1SI+ executions ensure the absence of inversions within sessions, and 1SI allows the appearance of inversions. Note that in the context of snapshot isolation, an inversion may occur if a transaction (either a query or an update transaction) is provided with a snapshot which does not correspond to the latest available snapshot in the system, as created by the last committed transaction. If not precluded, either conservatively or optimistically (similarly to the # and * synchronization models of Chapter 3), inversions may appear and a committed transaction T may have read an old value of a data item that was updated by a transaction that committed before the start of T.

Table 4.2: Correctness criteria for one-copy equivalent replicated databases

Criterion	Isolation	Consistency	Short description
1ASR	serializable	no inversions	The effects of transactions are equivalent to a serial execution in only one node. At each single moment, the committed information in every server is exactly the same from the point of view of clients: a user can execute a transaction in one node and change immediately to another server where they will see the updates made by their previous transaction. From the point of view of the servers, 1ASR may be achieved with #A, *A, #S or *S replica consistency levels, as explained in Chapter 3. Strong serializability [17, 35, 139] is another name used in the literature to refer to this correctness criterion.
1SR+	serializable	no inversions on sessions	Replica consistency is more relaxed than in the previous case, but inversions are precluded within client sessions, so a user with a single session perceives an atomic, inversions-free view of the database. Strong session serializability [35, 139] is also used in the literature to refer to this criterion.
1SR′	serializable	inversions	The effects of transactions are equivalent to those of a serial execution in one node. But at a given moment, effects of some transactions may be pending to commit in a server and, thus, a user moving between servers may get inconsistent results.

Continued on next page

Table4.2 – continued from previous page

Criterion	Isolation	Consistency	Short description
1ASI	snapshot	no inversions	Transactions are isolated following the snapshot level. Each transaction T gets the latest snapshot of the entire system (conservative approach) or, at least, the latest snapshot as created by previous transactions that updated data items that T reads (this allows an optimistic approach were transactions are restarted if a conflict is detected). This level is also named conventional snapshot isolation (CSI) [40] or strong SI [36].
1SI+	snapshot	no inversions on sessions	Transactions are isolated under the snapshot level. The snapshot provided to a transaction T corresponds to the latest snapshot created by transactions on the same client session (conservative approach) or, at least, by transactions on the same client session that write data items that T reads (which allows an optimistic approach). This level is also named strong session SI [36].
1SI	snapshot	inversions	Transactions are isolated following the snapshot level. But the snapshot provided to a transaction may be arbitrarily old, due to some transactions pending to commit in its delegate node (i.e., inversions occur). This level is also named generalized snapshot isolation (GSI) [40]. Usually, transactions get the latest snapshot of their delegate server, which is also known as prefix-consistent snapshot isolation (PCSI) [40].
1ARC	read committed	no inversions	The database replication system behaves as only one copy providing read committed isolation. Transactions starting at different nodes at the same time see the same database state.
1RC	read committed	inversions	A read committed isolation level is guaranteed. Inversions may arise.
1CS	cursor stability	inversions	Cursor-stability is enforced in the nodes. This isolation level reduces the abortion of read operations by using short read locks. Inversions may arise.

4.4 Conclusions

In this chapter we have presented a characterization model that provides a common framework to describe and compare different database replication systems. This model is the result of a careful analysis of different community proposals, made since the beginning of this research field. During this study, we identified the relevant steps that are common to all replication protocols, and the different approaches that protocols follow in such steps. A policy was associated with each step, and the different approaches or options were called strategies. Policies were later grouped into families, according to the relation among the interactions they regulated. With this model, we can detail the strategy that each protocol follows for each of its main steps.

This model is used in Chapter 5 in order to characterize more than 50 replication systems, in an extensive survey that reviews the chronological evolution of this research field.

Chapter 5

A Comprehensive Survey of Database Replication Systems

In this chapter we present the research evolution and survey the state of the art of database replication techniques. Our analysis is based on the policy-based characterization model presented in Chapter 4. Over 50 different systems are fully characterized following this model.

5.1 Introduction

As motivated in Chapter 4, a chronological survey of database replication systems based on a common description framework is highly valuable in order to compare and understand different proposals. The survey contained in this chapter describes each proposal by detailing, for each policy, the followed strategy, as well as the enforced correctness criterion. This survey will allow beginners to obtain a global and precise idea of the state of the art of the database replication research field and will also provide a historical vision of the evolution of these systems, allowing us to detect which strategies are the most used and which combinations guarantee each correctness criterion. Moreover, this study makes it easier to identify combinations of strategies that are seldom used but might make sense if new goals are set for replication protocols, such as the support of either more relaxed or stricter consistency models, or the increase of the system

scalability. Furthermore, this survey may enable us to identify which advances at which fields (in database management systems, in group communication systems, in isolation levels and correctness criteria specifications, in replication protocols, etc.) allowed the appearance of each described proposal, as well as to foresee which other advances could have a relevant effect on this evolution.

The rest of the chapter is structured as follows. Section 5.2 presents the comprehensive and chronological survey of database replication systems. Section 5.3 comments about the scope of the characterization model, as observed during the preparation of this survey. Finally, Section 5.4 discusses about the insight the survey offers.

5.2 Replication Systems as Combination of Strategies: A Survey

Any replication system can be defined as a particular combination of strategies, i.e., a set of specific policies. Obviously, not all combinations will create correct or useful replication systems. Some systems proposed in the literature of database replication are chronologically listed in Table 5.1, detailing the followed strategies, which are identified as shown in Table 4.1. The correctness criterion is also specified (see Table 4.2). For simplicity, when detailing communication processes involving several rounds, only the most demanding is showed in the table (and thus, e.g., a total order requirement signaled in the strategy may be only needed in one of the rounds). In Table 5.1, when a system (row) follows different strategies for a specific interaction (column) for different types of transactions, these strategies appear at different lines within that *cell* of the table (column-row). Those types of transactions denote usually the difference between read-only and update transactions, or among the several correctness criteria supported by a given system.[1] Whenever multiple lines are present in a row, columns with only one value mean that such a strategy is shared by all transaction types.

[1] When possible, the strategies at the same line represent the same type of transaction, showing, e.g., the policies for read-only transactions in the first line and those for updates in the second one. However, for more complex cases (e.g., systems distinguishing not only between queries and updates but also among different correctness criteria), Table 5.1 still depicts all followed strategies but such *one-line-one-type* clarification is not made. Please refer to the textual description and the visual representation of Figure 5.1 of those complex systems for a detailed distinction among their transaction types.

A visual representation for each surveyed system is provided in Figure 5.1, also in chronological order. Each *radius* of a graph corresponds to a policy, labeled with its initial letters. Concentric *circles* (hendecagons, to be accurate), mark the scale from 0 (the most inner circle), to 4 (the most external one). The digit associated with the strategy followed by the depicted system for each interaction of each different transaction type is then represented in that scale. Whenever several options are possible for a specific policy (e.g., when users can choose between read committed or snapshot isolation), the least costly option is the one that is represented in the graph (read committed in such a case), thus showing the minimum requirements of the system. Those points are finally connected by lines in order to create a figure for each transaction type of the system.[2] Remember also that the more demanding the strategy, the greater the digit of its identifier (e.g., a group-start policy Gs0 denotes the absence of communication at transaction start, while Gs3 requires a synchronization with all servers). This way, the *bigger* the resulting figure, the more costly the execution of that transaction type. These representations allow us to visually compare different systems as well as to get an idea of their cost. For example, regarding communication costs, all policies involving communication (Cq, Gs, Gl, Ge and Ga) are grouped together in the eastern/northeastern zone of the graph (from 12 until 4:30 in a clock). A figure widening out in that zone depicts a system which relies on communication and thus its performance will depend on the GCS and the network. On the other hand, regarding database requirements, the radius of Di (database-isolation policy) allows a quick comparison between the strictness in the local isolation level required for the correct functioning of different systems.

Next we offer thorough descriptions for all the protocols and systems surveyed. Letters in brackets reference superindexes in the corresponding row of Table 5.1.

Alsberg and Day [4] proposed a protocol following the single primary, multiple backup model, where backups are linearly ordered. The client can address its request to any replica in the system, which will then forward it to the primary [*a*]. The proposed system aims at offering a resilient sharing of distributed resources but it is not specially tailored to any service. In particular, it is not tailored for database replication. Thus, some database-related points are not detailed in the paper, such as the database-isolation [*b*] or the transaction-remote policy [*e*] (for which a conservative, non-overlapping option is assumed in our survey). Authors

[2] Those types are labeled 'q' for queries, and 'u' for update transactions. When different correctness criteria are provided, a distinction is made in parentheses.

Table 5.1: Database replication systems expressed as combinations of strategies

	Client request	**Transaction** service	**Group** start	**Database** replication	**Database** isolation	**Group** life	**Group** end	**Transaction** decision	**Transaction** remote	**Client** response	**Group** after	**Correctness criterion**
Alsberg-Day [4]	Cq2[a]	Ts0	Gs0	Dr2	Di0[b]	Gl0	Ge1-n[c]	Td0[d]	Tr2[e]	Cr1[f]	Ga1-n[g]	1A-[h]
2PL & 2PC [50]	Cq1	Ts0	Gs0	Dr1	Di3[a]	Gl1-n-s[b] Gl2-n-s[c]	Ge2-n[d]	Td4-rw[e]	Tr0 Tr1-d[f]	Cr1	Ga0	1ASR[g]
BTO & 2PC [12]	Cq1	Ts0	Gs0	Dr1	Di4[a]	Gl1-n-s Gl2-n-s	Ge2-n	Td4-rw	Tr0 Tr1-d	Cr1	Ga0	1ASR
Bernstein- Goodman [14]	Cq1	Ts0	Gs0	Dr1	Di3	Gl1-n-s[a] Gl2-n-s[b]	Ge2-n	Td4-rw	Tr0 Tr1-d	Cr1	Ga0	1ASR
OPT & 2PC [119]	Cq1	Ts0	Gs0	Dr1	Di4[a]	Gl1-n-s	Ge2-n[b]	Td4-rw[c]	Tr0 Tr1-d[d]	Cr1	Ga0	1ASR
O2PL & 2PC [23]	Cq1	Ts0	Gs0	Dr1	Di3	Gl1-n-s	Ge2-n[a]	Td4-rw	Tr0 Tr1-d	Cr1	Ga0	1ASR
Bcast all [2]	Cq1	Ts0	Gs0	Dr2	Di3	Gl3-t-s	Ge3-t	Td0	Tr1-d[a]	Cr1	Ga0	1ASR
Bcast writes [2]	Cq1	Ts0	Gs0	Dr2	Di3	Gl0[a] Gl3-t-s	Ge3-t	Td1-rw	Tr0[b] Tr1-d	Cr1	Ga0	1ASR

Continued on next page

Table5.1 – continued from previous page

	Client request	**Transaction** service	**Group** start	**Database** replication	**Database** isolation	**Group** life	**Group** end	**Transaction** decision	**Transaction** remote	**Client** response	**Group** after	**Correctness criterion**
Delayed bcast wrts [2]	Cq1	Ts0	Gs0	Dr2	Di3	Gl0	Ge3-t	Td1-rw	Tr0 Tr1-d	Cr1	Ga0	1ASR
Single bcast txns [2]	Cq1	Ts0	Gs0	Dr2	Di3	Gl0	Ge0 Ge3-t	Td1-rw[a] Td2-rw	Tr0 Tr1-d	Cr1	Ga0	1SR′
Lazy Txn Reordering [104]	Cq1	Ts0	Gs0	Dr2	Di2	Gl0	Ge3-t	Td2-rw	Tr2	Cr1	Ga0	1SR′
OTP-99 [70]	Cq1[a] Cq4-t[b]	Ts0-n[c] Ts2[d]	Gs0	Dr2	Di0[e]	Gl0	Ge0	Td0	Tr0[f] Tr1-p[g]	Cr1	Ga0	1SR′
Fast Refresh Df-Im [97]	Cq1[a] Cq2[b]	Ts0	Gs0	Dr1	Di3[c]	Gl0	Ge0	Td0[d]	Tr0[e] Tr1-d[f]	Cr1	Ga0[g] Ga2-f[h]	1SR′
Fast Refresh Im-Im [97]	Cq1 Cq2	Ts0	Gs0	Dr1	Di3	Gl0[i] Gl2-f-a[j]	Ge0[k] Ge2-f[l]	Td0	Tr0 Tr1-d	Cr1	Ga0	1SR′
DBSM [101]	Cq1	Ts0	Gs0	Dr2	Di3	Gl0	Ge0[a] Ge3-t[b]	Td0[c] Td2-rw[d]	Tr0 Tr1-d[e]	Cr1	Ga0	1SR′
SER [69]	Cq1	Ts0	Gs0	Dr2	Di4[a]	Gl0	Ge0[b] Ge3-t[c]	Td0[d] Td1-rw[e]	Tr0 Tr1-d[f]	Cr1	Ga0	1SR′

Continued on next page

Table5.1 – continued from previous page

	Client request	**Transaction** service	**Group** start	**Database** replication	**Database** isolation	**Group** life	**Group** end	**Transaction** decision	**Transaction** remote	**Client** response	**Group** after	**Correctness criterion**
CS [69]	Cq1	Ts0	Gs0	Dr2	Di4[a]	Gl0	Ge0 Ge3-t	Td0 Td1-rw	Tr0 Tr1-d	Cr1	Ga0	1CS
SI [69]	Cq1	Ts0	Gs0	Dr2	Di2	Gl0	Ge0[a] Ge3-t[b]	Td0[c] Td2-w[d]	Tr0 Tr1-d[e]	Cr1	Ga0	1SI
Hybrid [69]	Cq1	Ts0	Gs0	Dr2	Di2[a] Di4[b]	Gl0	Ge0[c] Ge3-t[d]	Td0[e] Td1-rw[f]	Tr0 Tr1-d[g]	Cr1	Ga0	1SR′
NODO [98]	Cq1[a] Cq4-t[b]	Ts1[c]	Gs0	Dr2	Di2	Gl0	Ge0[d] Ge3-n[e]	Td0[f]	Tr0[g] Tr1-p[h]	Cr1	Ga0	1SR′
REORDERING [98]	Cq1 Cq4-t	Ts1	Gs0	Dr2	Di2	Gl0	Ge0 Ge3-f[i]	Td0	Tr0 Tr1-p	Cr1	Ga0	1SR′
Pronto [102]	Cq2[a]	Ts0	Gs0	Dr2	Di3	Gl0	Ge3-t[b]	Td2[c]	Tr2	Cr2[d]	Ga0	1ASR
DBSM-RAC [121]	Cq2[a]	Ts0	Gs0	Dr1	Di3	Gl0	Ge0[b] Ge3-t[c]	Td0[d] Td4-rw[e]	Tr0[f] Tr1-d[g]	Cr1	Ga0	1SR′
Epidemic restricted [58]	Cq1	Ts0	Gs0	Dr1	Di3	Gl0	Ge0[a] Ge2-n[b]	Td0[c] Td2-rw[d]	Tr0[e] Tr2[f]	Cr1	Ga0	1SR′

Continued on next page

Table5.1 – continued from previous page

	Client request	**Transaction** service	**Group** start	**Database** replication	**Database** isolation	**Group** life	**Group** end	**Transaction** decision	**Transaction** remote	**Client** response	**Group** after	**Correctness criterion**
Epidemic unrestrict. [58]	Cq1	Ts0	Gs0	Dr1	Di3	Gl2-n-s[g]	Ge0 Ge2-n	Td0 Td2-rw	Tr0 Tr2	Cr1	Ga0	1SR′
OTP [71]	Cq4-t[a]	Ts2[b]	Gs0	Dr2	Di0[c]	Gl0	Ge0	Td0	Tr1-p[d]	Cr1	Ga0	1ASR
OTP-Q [71]	Cq1[e] Cq4-t	Ts1[f] Ts2	Gs0	Dr2	Di0	Gl0	Ge0	Td0	Tr0[g] Tr1-p	Cr1	Ga0	1SR′
OTP-DQ [71]	Cq1 Cq4-t	Ts1 Ts2	Gs0	Dr2	Di0	Gl0	Ge0	Td1-rw[h] Td0	Tr0 Tr1-p	Cr1	Ga0	1SR′
OTP-SQ [71]	Cq1 Cq4-t	Ts0[i] Ts2	Gs0	Dr2	Di0	Gl0	Ge0	Td0	Tr0 Tr1-p	Cr1	Ga0	1SR′
RJDBC [44]	Cq1[a]	Ts0	Gs0	Dr2	Di0[b]	Gl3-t-s[c]	Ge3-t[d]	Td0[e]	Tr2[f]	Cr1	Ga0	1-[g]
RSI-PC [107]	Cq1 Cq2[a]	Ts1[b] Ts0[c]	Gs0	Dr2	Di2[d] Di1[e]	Gl0	Ge0	Td0[f]	Tr0 Tr2[g]	Cr1[h]	Ga0[i] Ga2-f[j]	1ASI 1SI 1ARC 1RC
SRCA [79]	Cq1[a]	Ts0	Gs0	Dr2	Di2	Gl0	Ge0[b] Ge3-t[c]	Td0[d] Td1-w[e]	Tr0 Tr2	Cr1	Ga0	1SI

Continued on next page

Table5.1 – continued from previous page

	Client request	**Transaction** service	**Group** start	**Database** replication	**Database** isolation	**Group** life	**Group** end	**Transaction** decision	**Transaction** remote	**Client** response	**Group** after	**Correctness criterion**
SRCA-Rep [79]	Cq1	Ts1[a]	Gs0	Dr2	Di2	Gl0	Ge0[b] Ge3-t[c]	Td0[d] Td2-w[e]	Tr0 Tr1-p	Cr1	Ga0	1SI
DBSM* [141]	Cq1[a] Cq2[b]	Ts0	Gs0	Dr2	Di3	Gl0	Ge0 Ge3-t	Td0 Td2-w	Tr0 Tr1-d	Cr1	Ga0	1SR′
PCSI Distr. Cert. [40]	Cq1	Ts0	Gs0	Dr2	Di2	Gl0	Ge0[a] Ge3-t[b]	Td0[c] Td2-w[d]	Tr0 Tr2	Cr1	Ga0	1SI
Tashkent-MW [41]	Cq1	Ts0	Gs0	Dr2	Di2	Gl0	Ge0[a] Ge1-n[b]	Td0[c] Td1-w[d]	Tr0[e] Tr2[f]	Cr1	Ga0[g]	1SI
Tashkent-API [41]	Cq1	Ts0	Gs0	Dr2	Di2	Gl0	Ge0 Ge1-n	Td0 Td1-w	Tr0 Tr1-d[h]	Cr1	Ga0	1SI
DBSM-RO-opt [96]	Cq1	Ts0	Gs0	Dr2	Di3	Gl0	Ge3-t[a]	Td1-rw Td2-rw[b]	Tr0 Tr1-d	Cr1	Ga0	1ASR[c]
DBSM-RO-cons [96]	Cq1	Ts0	Gs3-t-s[a] Gs0[b]	Dr2	Di3	Gl0	Ge0[c] Ge3-t[d]	Td0[e] Td2-rw[f]	Tr0 Tr1-d	Cr1	Ga0	1ASR
Alg-Weak-SI [36]	Cq1 Cq2[a]	Ts0	Gs0	Dr2	Di2	Gl0	Ge0[b]	Td0[c]	Tr0 Tr1-p[d]	Cr1	Ga0[e] Ga3-f[f]	1SI[g]

Continued on next page

Table5.1 – continued from previous page

	Client request	**Transaction** service	**Group** start	**Database** replication	**Database** isolation	**Group** life	**Group** end	**Transaction** decision	**Transaction** remote	**Client** response	**Group** after	**Correctness criterion**
Alg-Str.-SI / Alg-Str.Ses.-SI [36]	Cq1 Cq2	Ts1[a] Ts0[b]	Gs0	Dr2	Di2	Gl0	Ge0	Td0	Tr0 Tr1-p	Cr1	Ga0 Ga3-f	1ASI 1SI+[c]
One-at-a-time / Many-at-a-time [116]	Cq2[a]	Ts0	Gs0	Dr1	Di3[b]	Gl0	Ge0[c] Ge3-t[d]	Td0[e] Td3-rw[f]	Tr0[g] Tr2[h]	Cr1	Ga0	1SR′
k-bound GSI [6]	Cq1	Ts0	Gs3-t-a[a]	Dr2	Di2	Gl0	Ge0[b] Ge3-t[c]	Td0[d] Td2-w[e] Td1-rw[f]	Tr0 Tr2	Cr1	Ga0	1SR′ 1ASI 1SI
Tashkent+ [42]	Cq2[a]	Ts0	Gs0	Dr2[b]	Di2	Gl0	Ge0 Ge1-n	Td0 Td1-w	Tr0 Tr2	Cr1	Ga0	1SI[c]
Mid-Rep [65]	Cq1	Ts0[a] Ts1[b]	Gs0 Gs3-t-s[c]	Dr2	Di2	Gl0	Ge0 Ge3-t	Td0 Td1-rw	Tr0 Tr2[d]	Cr1	Ga0	1SR′ 1ASI 1SI
SIRC [114]	Cq1	Ts0	Gs0	Dr2	Di1 Di2[a]	Gl0	Ge0[b] Ge3-t[c]	Td0[d] Td2-w[e]	Tr0 Tr2	Cr1	Ga0	1SI 1RC
Serrano et al. [118]	Cq2[a]	Ts1[b]	Gs0	Dr1	Di2	Gl1-n-s[c]	Ge2-n[d] Ge3-t[e]	Td0[f] Td2-w[g]	Tr0[h] Tr2[i]	Cr1	Ga0	1SI

Continued on next page

Table5.1 – continued from previous page

	Client request	**Transaction** service	**Group** start	**Database** replication	**Database** isolation	**Group** life	**Group** end	**Transaction** decision	**Transaction** remote	**Client** response	**Group** after	**Correctness criterion**
MPF/MCF [140]	Cq1[a] Cq2[b]	Ts0	Gs0	Dr2	Di3	Gl0	Ge0 Ge3-t	Td0 Td2-rw	Tr0 Tr1-d	Cr1	Ga0	1SR′
WCRQ [110]	Cq1	Ts0	Gs0	Dr2	Di4[a]	Gl0	Ge2-n[b] Ge3-t[c]	Td3-rw[d]	Tr0 Tr2[e]	Cr1[f]	Ga0	1ASR
AKARA [32]	Cq1	Ts0-n Ts2[a]	Gs3-t-s[b]	Dr2	Di2	Gl0	Ge3-n[c] Ge0[d]	Td0	Tr2	Cr1	Ga0	1SI
BaseCON for 1SR [139]	Cq1[a] Cq4-t[b]	Ts0[c] Ts2[d]	Gs0	Dr2	Di3[e]	Gl0	Ge0	Td0[f]	Tr0 Tr1-p[g]	Cr1[h]	Ga0	1SR′
BaseCON for SC [139]	Cq2[a] Cq4-t[b]	Ts0[c] Ts2[d]	Gs0	Dr2	Di3[e]	Gl0	Ge0	Td0[f]	Tr0 Tr1-p[g]	Cr1[h]	Ga0	1SR+
BaseCON for strong 1SR [139]	Cq4-t[a]	Ts0 Ts2	Gs0	Dr2	Di3	Gl0	Ge0	Td0	Tr0 Tr1-p	Cr1	Ga0	1ASR
*g*B-SIRC [115]	Cq1	Ts0	Gs3-t-a[a]	Dr2	Di1 Di2[b]	Gl0	Ge0[c] Ge3-t[d]	Td0[e] Td2-w[f] Td1-rw[g]	Tr0 Tr2	Cr1	Ga0	1ASI 1SI 1RC

focus on requests that change the state of the replicas, which can be regarded as update transactions. Processing is slightly different depending on whether the client directly addresses to the primary or not. In any case, a two-host resiliency is always ensured. The first backup receives the updates at the end of the transaction [*c*], before a unique reply is sent to the client [*f*]. After this, the rest of backups are updated in a cascade mode [*g*]: whenever a backup commits its new state, it forwards the updates to the following backup. As all transactions are executed in the primary (the rest of nodes only act as backups), the concurrency control of the primary server is enough and no decision process is necessary [*d*]. The resulting correctness criterion includes one-copy equivalence and the guarantee of no inversions (from the user point of view, there is only one centralized server that runs all requests). Server-centric consistency is natively sequential (#S). The exact correctness criterion directly depends on the local isolation of the primary replica [*h*]. If, for example, the isolation in the primary node is serializable, the correctness criterion will be 1ASR.

2PL & 2PC (Distributed Two-Phase Locking with Two-Phase Commit) was originally described by Gray [50]. 2PL is an algorithm for distributed concurrency control, intended for distributed databases where some degree of replication is also probable. This involves the use of distributed transactions. The underlying database provides a serializable level of isolation, with long read and write locks [*a*]. Once in the appropriate cohort [*b*], read operations set a read lock on the local copy of the item and read the data, but updates must be approved at all replicas before the transaction proceeds [*c*]. Thus, write locks are required on all copies in a pessimistic way. All locks are held until the transaction has committed or aborted. Deadlocks can appear and are solved by aborting the youngest transaction. A snoop process periodically requests waits-for information from all sites, detecting and resolving deadlocks. This process rotates among the servers in a round-robin fashion. A two-phase commit [*d*] is executed for each transaction that requests commitment. The initiating server (coordinator) sends a *prepare* message to all nodes. Each server then replies with a positive or negative message. If all messages are positive, then the coordinator sends a commit message. Every server confirms with another message the success of the commitment. If any of the replies in the first phase is negative, the coordinator sends an abort message and all servers report back to the coordinator once the abortion is complete. Decision is, thus, agreement-based [*e*]: aborts are possible due to deadlocks, node or disk failures, problems in the log, etc., so the first phase of 2PC achieves an

agreement among the servers about the decision for the current transaction. In nodes holding copies of replicated items, subtransactions are initiated as in cohorts, but only for update transactions. Locks ensure that conflicting operations are not concurrently made [*f*]. The correctness criterion is 1ASR [*g*], based in a periodically atomic (*A) server-centric consistency.

A similar protocol is Wound-Wait (WW), which was proposed by Rosenkrantz et al. [111]. The only difference with regard to distributed 2PL is the way in which deadlocks are handled. In WW, deadlocks are prevented by the use of timestamps. When a transaction requests a lock which is held by a younger transaction (with a more recent initial start-up time), the youngest transaction is immediately aborted unless it is already in the second phase of its 2PC.

BTO & 2PC (Basic Timestamp Ordering with Two-Phase Locking) was originally proposed by Bernstein and Goodman [12]. BTO is identical to distributed 2PL except for the fact that local isolation is based on start-up timestamps [*a*]. Each data item has a read timestamp corresponding to the most recent reader, and a write timestamp corresponding to the most recent writer. When a transaction requests a read operation, it is permitted if the timestamp of the requester exceeds the write timestamp of the object. A write request is permitted if the timestamp of the requester exceeds the read timestamp of the item. In this case, if the write timestamp of the item is greater than the timestamp of the requester, the update is simply ignored. Write locks must be granted in all remote copies before proceeding with update operations, which are kept in private workspaces until commit time so that other writers are not blocked. On the other hand, approved read operations must wait until the precedent writes are applied in order during the commit operation of previous transactions. The used mechanisms enforce a periodically atomic server-centric consistency (*A) guaranteeing 1ASR.

Bernstein and Goodman later proposed a concurrency control algorithm [14] for achieving 1ASR in replicated distributed databases. This algorithm, which also employs 2PL and 2PC (and so the strategies of both algorithms match), is specially enforced to tolerate failures and recover nodes. Each data item x has one or more copies (x_a, x_b, ...). Each copy is stored at a site. Site failures are clean: when a site fails, it simply stops running (Byzantine failures are not considered);

when it recovers, it knows that it failed and starts a recovery phase. Other sites in the system can detect when a site is down. Neither network partitions nor network failures are considered.

Each data item x has an associated directory $d(x)$, which can be replicated. Each copy of a directory contains two kinds of information: a list of the available copies of x, and a list of the available copies of $d(x)$. Special status transactions change the contents of the directories to reflect site failures. User transactions perform read and write operations over data items. Both types of transactions require an available copy of $d(x)$ for each access over x. Access to data items and directories are both protected by locking. Read and write locks over data items conflict in the usual way. New locks are created for accessing directories: din-locks, in-locks, ex-locks and user-locks. The three first are all conflicting among them. The last one, user-lock, is set by user transactions and it conflicts only with in-locks, being compatible with the rest of directory locks. Two-phase locking (2PL) is used for concurrency control.

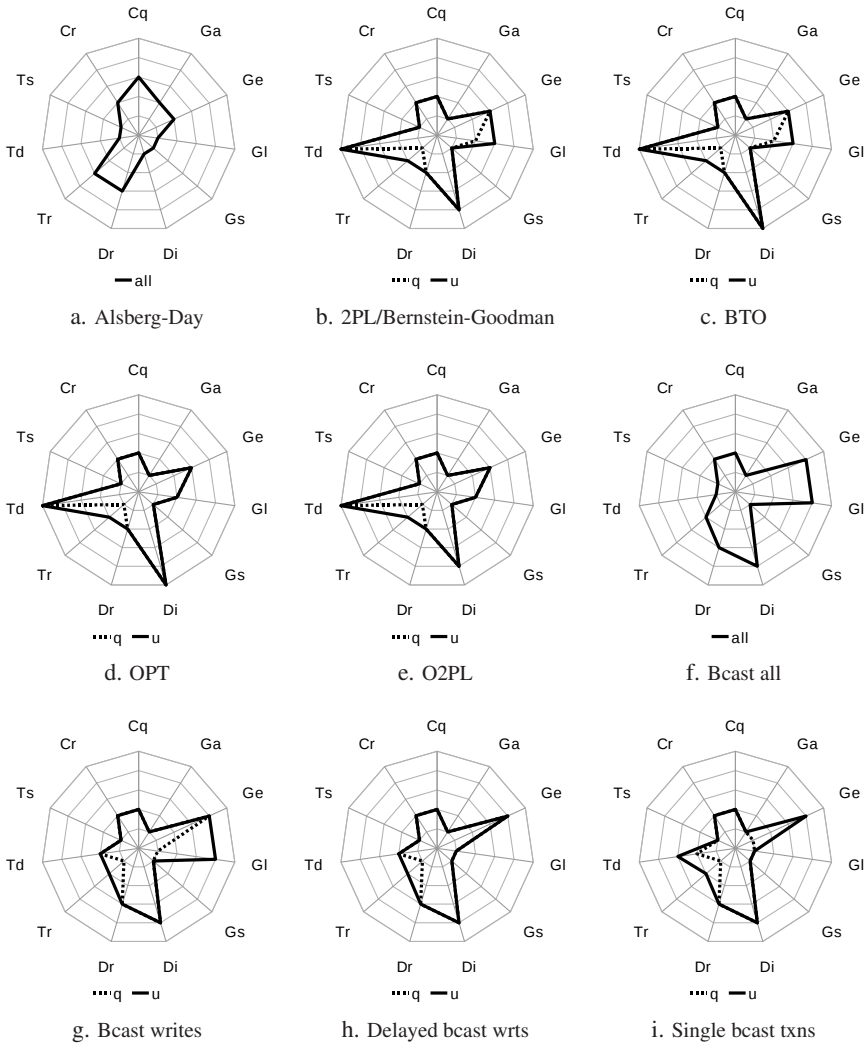
There are three types of status transactions: directory-include, include and exclude. A directory-include transaction, *DIRECTORY-INCLUDE*(d_t), makes directory copy d_t available. It initializes d_t to the "current value" of $d(x)$ and adds d_t to the directory list of every available copy of $d(x)$ (din-locks are required in the original copy of the directory, in the new one and in all the updated copies). An include transaction, *INCLUDE*(x_a), makes data item copy x_a available. It first initializes x_a to the current value of x and then it adds x_a to the data-item list of every available copy of $d(x)$ (in-locks are requested in the local available copy and in all the updated copies of $d(x)$; also, a read-lock is set on the original data item copy of x and a write-lock protects the access to the new copy x_a). Finally, an exclude transaction, *EXCLUDE*(x_a), makes data item copy x_a unavailable (ex-locks are required in the original and in all the updated copies of $d(x)$). When executing any of these status transactions, if it is detected that some directory copy d_u has become unavailable, the transaction also removes d_u from the directory list of every available copy of $d(x)$. The distributed database system invokes exclude transactions when a site fails, and include and directory-include transactions when a site recovers. There is no directory-exclude transaction; d_t becomes unavailable the instant its site fails.

User transactions access data items in read and write operations. When a read operation is requested, the corresponding directory is accessed to find an available copy of x to read, which is then protected with a read-lock [a]. For a write

operation, a user-lock is set on d_t, the local available copy of $d(x)$ (step 1). For each available data-item copy x_a, a write-lock is set [b] (step 2). Finally, all still available copies which were locked in the previous step are written (step 3). As sites may fail at any point in time, exclude transactions can be applied concurrently with user transactions. Copies that could be locked in step 2 but become unavailable before step 3 are ignored. Finally, when the user transaction reaches its locked point (when it owns all of its locks), the following procedure is executed: (1) for each read item x_a, if x_a is not in the data-items list of local directory copy d_t, or if $EXCLUDE(x_a)$ has an ex-lock on d_t, then the transaction is aborted. (2) In parallel, all user-locks and read-locks are released, and the step 3 of the write procedure is finished and all write-locks are also released.

A two-phase commit (2PC) procedure is used to commit transactions. The first phase of 2PC can run before the transaction reaches its locked point. However, phase 2 must wait until the end of the step 1 of the locked-point procedure. Phase 2 of 2PC and step 2 of the locked-point procedure can use the same messages. Due to the use of 2PL and 2PC, the server-centric consistency is periodically atomic (*A) and 1ASR is guaranteed.

OPT & 2PC (Distributed Certification with Two-Phase Commit) corresponds to the first of the two distributed concurrency control protocols proposed by Sinha et al. [119]. As in BTO, all data items have a read and a write timestamp corresponding to the most recent reader and writer, respectively [a]. However, in OPT, transactions are allowed to proceed freely, storing any updates in private workspaces. For each read operation, the transaction must remember the write timestamp of the read item. Before starting the two-phase commit [b], a unique timestamp is assigned to the transaction. A certification is then performed for each transaction in each cohort. If there is some replication, remote updaters (which store copies of the written objects) receive the writeset in the *prepare* message of 2PC and take also part in the certification process. A read request is certified if the version that was read is still the current version and no write with a newer timestamp has been already certified. A write request is certified if no later reads have been locally certified or committed. The term *later* refers to the timestamp assigned at the start of the 2PC. A transaction is certified globally if local certification succeeds for all its cohorts and all its remote updaters [c]. This certification process is run inside the local DBMS, which allows a concurrent execution of writesets in remote updaters while ensuring a right concurrency control [d]. The optimism of this algorithm, which lets read operations proceed



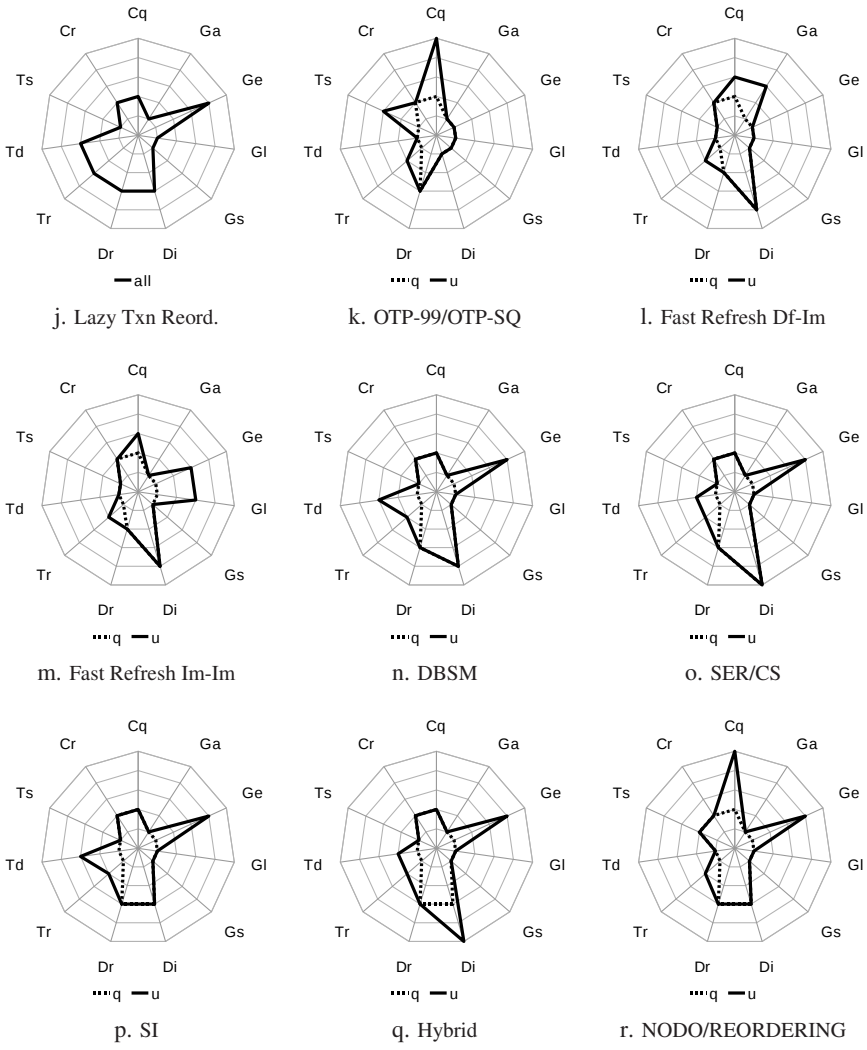
a. Alsberg-Day b. 2PL/Bernstein-Goodman c. BTO

d. OPT e. O2PL f. Bcast all

g. Bcast writes h. Delayed bcast wrts i. Single bcast txns

Figure 5.1: Visual representations of the surveyed systems

without getting any locks, allows the appearance of potential inversions during the execution of transactions. These potential inversions are later detected and aborted during certification, thus providing 1ASR with a periodically sequential (*S) server-centric consistency based on the total order of the timestamps assigned to the transactions.



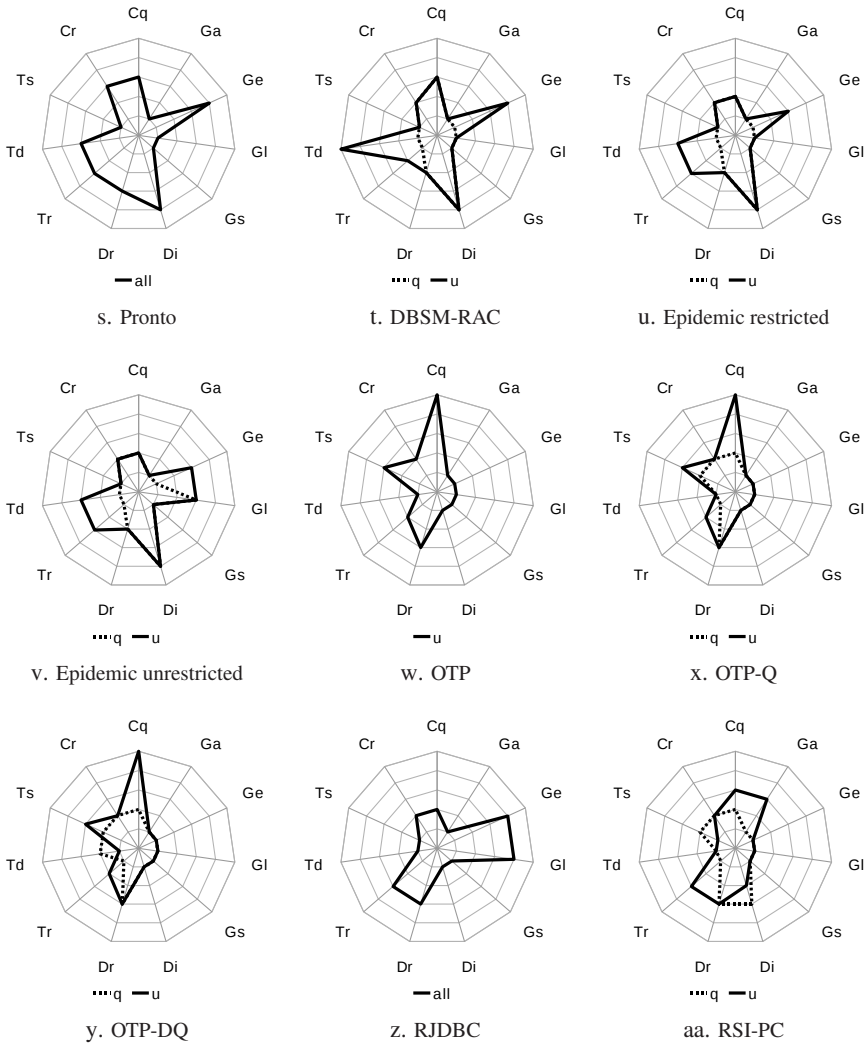
j. Lazy Txn Reord. k. OTP-99/OTP-SQ l. Fast Refresh Df-Im

m. Fast Refresh Im-Im n. DBSM o. SER/CS

p. SI q. Hybrid r. NODO/REORDERING

Figure 5.1: Visual representations of the surveyed systems (cont.)

O2PL & 2PC (Distributed Optimistic Two-Phase Locking with Two-Phase Commit) was proposed by Carey and Livny [23] as an optimistic version of distributed 2PL. Both algorithms are identical in the absence of replication. However, O2PL handles replicated data as OPT does: when a cohort updates a replicated data item, a write lock is requested on the local copy of the item, but the request of



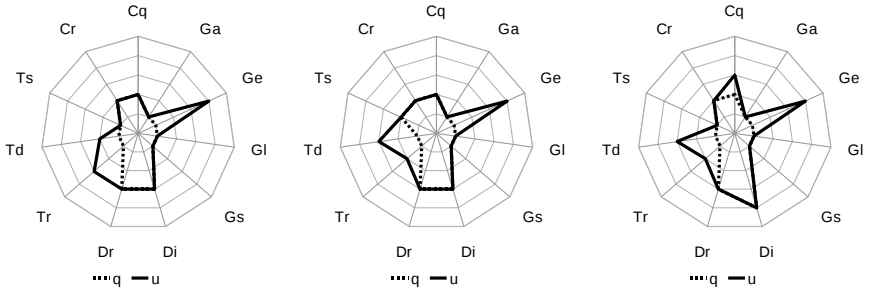
s. Pronto t. DBSM-RAC u. Epidemic restricted

v. Epidemic unrestricted w. OTP x. OTP-Q

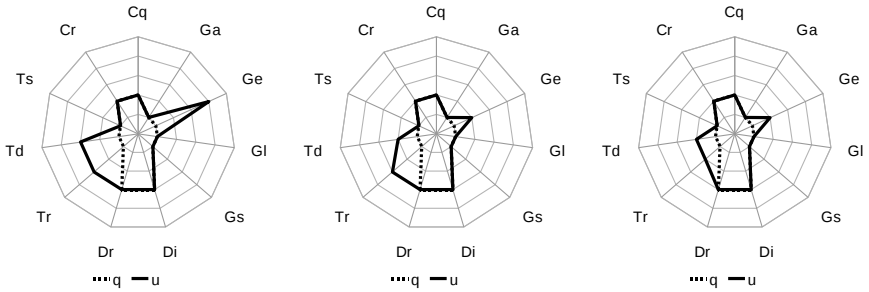
y. OTP-DQ z. RJDBC aa. RSI-PC

Figure 5.1: Visual representations of the surveyed systems (cont.)

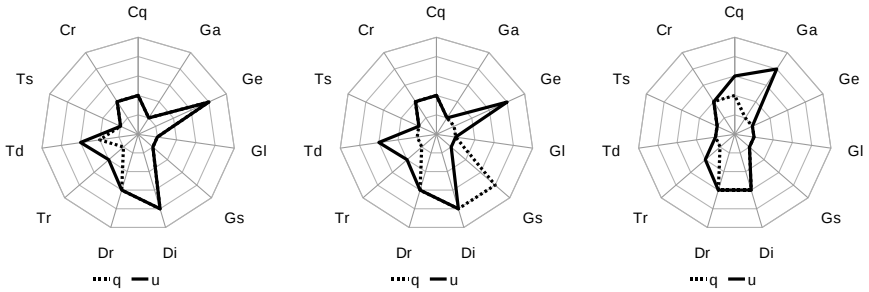
write locks in remote copies is delayed until commit time. During 2PC [*a*], remote nodes must acquire the write locks required by the transaction (this info is in the *prepare* message of the 2PC) before answering to the coordinator. As read operations are required to first get a read lock and certified transactions get all write locks at all replicas before committing, the resulting server-centric consistency is periodically atomic (*A), ensuring the correctness criterion of 1ASR.



ab. SRCA ac. SRCA-Rep ad. DBSM*



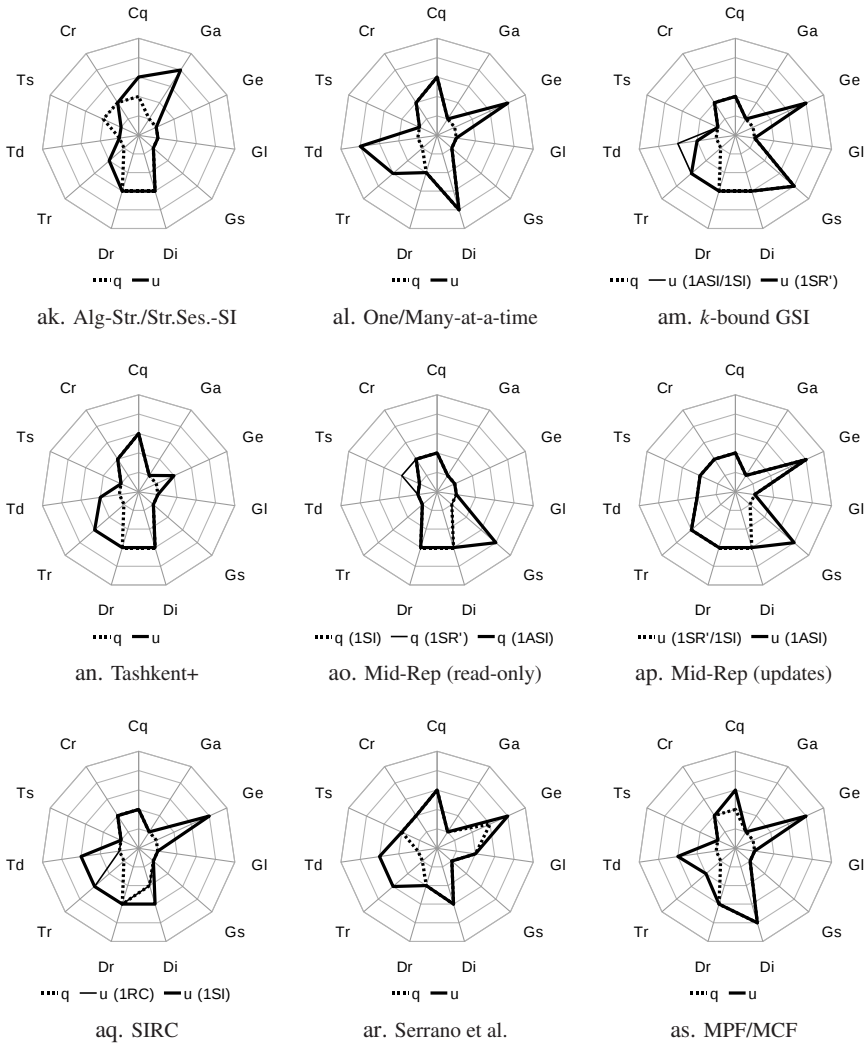
ae. PCSI Distr. Cert. af. Tashkent-MW ag. Tashkent-API



ah. DBSM-RO opt ai. DBSM-RO cons aj. Alg-Weak-SI

Figure 5.1: Visual representations of the surveyed systems (cont.)

Agrawal et al. [2] propose the use of atomic broadcast to simplify the design of replication protocols, thus eliminating the need for acknowledgments, global synchronization or two-phase commitment protocols. Four protocols are proposed.



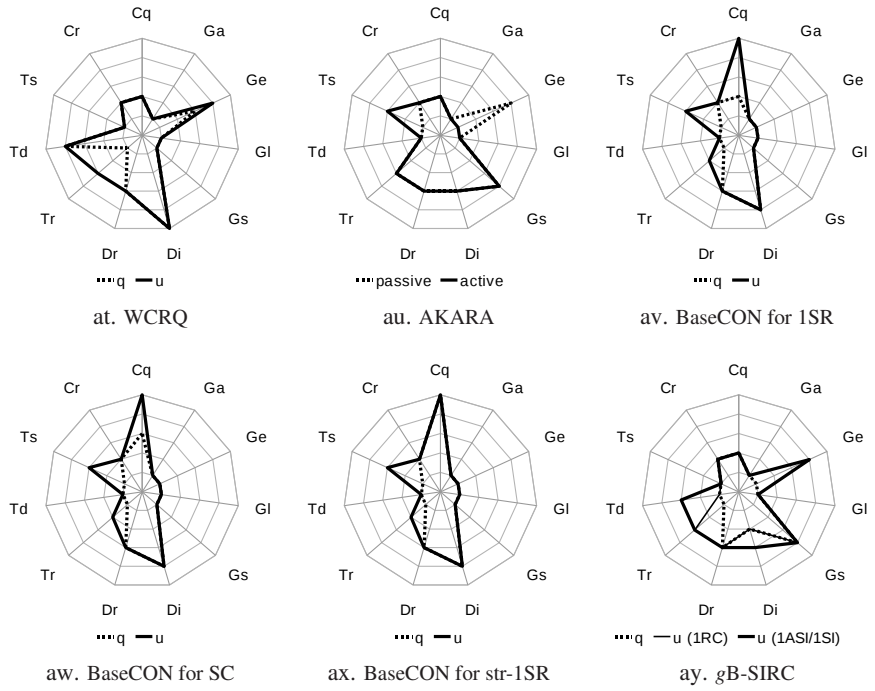
ak. Alg-Str./Str.Ses.-SI al. One/Many-at-a-time am. *k*-bound GSI

an. Tashkent+ ao. Mid-Rep (read-only) ap. Mid-Rep (updates)

aq. SIRC ar. Serrano et al. as. MPF/MCF

Figure 5.1: Visual representations of the surveyed systems (cont.)

Broadcast all is a naive solution that follows the state machine approach [117] broadcasting each operation, read or write, in total order to all replicas and waiting until its delivery to execute it. A final commit operation is also broadcast and applied in the nodes [*a*]. Thus, every site delivers all operations in the same order and a 1ASR correctness criterion is enforced by means of a 1ASR/*S synchronization model, where independent transactions may commit at different orders



at. WCRQ au. AKARA av. BaseCON for 1SR

aw. BaseCON for SC ax. BaseCON for str-1SR ay. *g*B-SIRC

Figure 5.1: Visual representations of the surveyed systems (cont.)

while conflicting operations are ensured to be executed at their delivery order. **Broadcast writes**, optimizes previous algorithm by sparing the broadcast of read operations [*a*]. A delegate node sends a commit or abort message for its transaction T, as it is the only node where the read operations of T were performed. Read-only transactions are committed only in their delegate [*b*], while update transactions are committed at every replica. 1ASR is guaranteed by means of a 1ASR/*S synchronization model. **Delayed broadcast writes**, packs all write operations in a single broadcast at the end of transaction T. When this message is delivered, nodes request write locks and execute writes. When T commits in its delegate node, a commit operation is broadcast to all sites. Assuming that read-only transactions are also broadcast at the end and a validation is performed for them, 1ASR is again guaranteed by means of a 1ASR/*S synchronization model. Finally, **single broadcast transactions**, reduces all communication down to a single broadcast at the end of the transaction, only for update transactions (the decision to commit a read-only transaction is done locally [*a*] and, as a result, inversions may occur). Readset and writeset information is contained in this

message, for each node to be able to independently certify transactions and grant all write locks. The resulting correctness criterion is 1SR′, based on a periodically sequential (*S) consistency level.

These four protocols are referenced to as protocols A1, A2, A3 and A4 and their performance is evaluated by Holliday et al. [57], who claim that the usage of atomic broadcast in database replication protocols simplifies message passing and conflict resolution, thus making replication efficient, even when providing full replication and update-everywhere capability.

The **Lazy Transaction Reordering** protocol was proposed by Pedone et al. [104] as a replication protocol able to reduce the abort rate of existing lazy approaches[3] by reordering transactions during certification when possible. Traditional Kung-Robinson's certification, where a delivered transaction T is aborted if its readset intersects with the union of the writesets of concurrent and previously delivered committed transactions, is thus changed for a new one where serial order does not necessarily have to match up with that of the atomic broadcast used to send the certification message. The reordering protocol tries to find a position in the serial order where T can be inserted without violating serializability. If no position can be found, T is aborted. Natively sequential (#S) server-centric consistency is ensured, but inversions may be increased by the reordering nature of the protocol. As local write operations are tentative and are only confirmed in a non-overlapping manner after certification, a serial execution is achieved despite not using local serializable isolation. As a result, the correctness criterion is 1SR′.

OTP algorithm, Optimistic Transaction Processing, was proposed by Kemme et al. [70] and later refined [71]. In order to distinguish between both versions, we denote the initial one with the year of publication **OTP-99**. In OTP-99, all accesses to the database are assumed to be done through the use of stored procedures. Each stored procedure accesses only one of a set of disjoint conflict

[3] In such a paper [104], an approach is identified as lazy if it locally executes transactions and sends certification information to the rest of the system at commit time, as opposed to eager approaches, which are identified in the paper as those that synchronize each data access by communicating with other nodes during transaction execution. Note that these definitions do not match those of Chapter 2, where eager approaches broadcast the writeset and apply it at all nodes before replying to the client, as opposed to lazy approaches that send and apply writesets at remote nodes after committing in the delegate.

classes into which the database is divided. Each node maintains a queue per conflict class, in order to serialize conflicting transactions at middleware level, and has a mechanism that maintains different versions of the data of each conflict class [*e*]. Read-only transactions can execute at any replica [*a*], using the corresponding snapshot [*c*] and committing locally without further processing [*f*]. To provide consistent snapshots for queries, the different maintained versions of the data are labeled with the index (the position inside the definitive total order) of the transaction that created the version. If T_i was the last processed TO-delivered transaction at the time a query Q starts, then the index for Q is $i.5$ (a *decimal* index). When Q wants to access some data, it is provided with the data corresponding to the maximum version which is lower than the index of the query.

Update transactions are broadcast in total order to all the sites [*b*], where they will be executed in an active way [*d*]. An optimistic atomic broadcast is used, so first transactions are opt-delivered in a tentative order to be later TO-delivered in the definitive total order. When a transaction is opt-delivered, it is inserted in the corresponding queue. All transactions at the heads of the queues can be executed concurrently [*g*], as they do not conflict. When a transaction T is finally TO-delivered, any conflicting transaction T' tentatively ordered before T and not yet TO-delivered must be reordered (as the definitive total order is used as the serialization order for conflicting transactions). If T' already started execution, it must be aborted and later reexecuted. T is then rescheduled before all non-TO-delivered transactions in its corresponding queue. On the other hand, when the definitive order matches the tentative one, T can be committed as soon as it is fully executed. After commitment, T is removed from its queue and the following transaction can be submitted to execution.

Regarding the server-centric consistency, if we consider each conflict class, i.e., each division of the database, as a logically different database, transactions running in any of those divisions are guaranteed a 1SR$'$ correctness criterion, as the snapshots used by queries allow inversions but updates are sequentially applied (#S consistency level).

Pacitti et al. [97] propose **Fast Refresh Deferred-Immediate** and **Fast Refresh Immediate-Immediate**, two refreshment algorithms for a lazy master replication system. In those systems, each data item has a primary copy stored in a master node, and updates to that data item are only allowed in that node [*b*], while read operations are allowed in any replica [*a*] (inversions may arise). A transaction

can commit after updating one copy. The rest of replicas are updated in separate refresh transactions. Partial replication is used in this work, but all transactions are assumed to access only local data (there are no distributed transactions). Secondary copies are stored in other servers. Write operations are propagated to the secondary copies, which are updated in separate refresh transactions. Two types of propagation are considered: deferred (all the update operations of a transaction T are multicast within a single message after the commitment of T) [*h*], and immediate (each write operation is immediately multicast inside an asynchronous message, without waiting for the commitment of T) [*j, l*]. Read operations are not propagated [*e, g, i, k*]. Refresh transactions can be triggered in the secondaries in three different ways: deferred, immediate and wait. The combination of a propagation parameter and a triggering mode defines a specific update propagation strategy. In the deferred-immediate algorithm, a refresh transaction is started as soon as the corresponding message is received by the node; while in the immediate-immediate one, a refresh transaction is started as soon as the first message corresponding to the first operation is received, thus achieving higher replica freshness. Finally, the ordering parameter defines the commit order of refresh transactions. Depending on the system topology, this ordering must be refined in order to maintain replica consistency (secondary copies are updated and no inconsistent state is observable in the meantime). A FIFO reliable multicast with a known upper bound is used. This upper bound, timestamps and drift-bounded clocks in nodes allow the protocol to achieve a total ordering among messages, when necessary, letting refresh transactions to execute concurrently [*f*] but forcing them to wait before their final commit operation, in order to perfectly correspond with the commitment order in the master nodes and thus achieve #S server-centric consistency. A serializable isolation level in local databases [*c*] allows the system to ensure 1SR′. No decision phase is required, as local concurrency control in master nodes is enough [*d*].

DBSM (Database State Machine Replication) [101] applies the state machine model to the termination protocol of a database replication system. Read-only transactions are directly committed when the user requests to, i.e., no communication is established with other nodes and no decision process is performed [*a, c*]. Update transactions atomically broadcast [*b*] their information (readset, writeset, updates) to the rest of nodes, where a certification [*d*] is performed, checking for write-read conflicts. Successfully certified transactions request all their write locks and, once they are granted, the updates are performed [*e*]. To make sure that

each database site will reach the same state, write-conflicting transactions must request their locks (and be applied to the database) in the same order they were delivered. On the other hand, transactions that do not conflict are commutable: they do not need to be applied to the database in the same order they were delivered. Although different sites may follow different sequences of states (depending on such commutable transactions), write locks –held from certification to final commitment–, prevent users from perceiving such inconsistency. This corresponds to the periodically sequential (*S) server-centric replica consistency. On the other hand, as read-only transactions are not certified (they are locally and immediately committed upon request), inversions are possible. The ensured correctness criterion is 1SR′.

During remote writeset application, conflicting local transactions are aborted. Establishing a trade-off between consistency and performance, Correia et al. [30] relax the consistency criteria of the DBSM with Epsilon Serializability. Read-only transactions can define a limit in the inconsistency they import, i.e., the aggregated amount of staleness of read data. Update transactions define a limit in the inconsistency they export to concurrent transactions. This way, during lock acquisition before writeset application, a local mechanism verifies if the inconsistencies introduced by the committing writeset do not force a local query to exceed its limits, or the remote update to exceed its limits. If both limits are not exceeded, the local query can continue and the remote writeset can be applied. Otherwise, the local query is aborted.

SER [69] is a protocol for serializable databases, where long local read locks are requested for read operations, while write operations are delayed until the end of the reading phase [*a*]. When a read-only transaction finishes its operations and requests commitment, it is immediately committed, without any communication with the rest of the system [*b, d*]. On the other hand, update transactions are broadcast in total order [*c*]. When delivered, a validation mechanism is performed [*e*] but not for deciding about this transaction but about local ones not yet delivered. This validation is based on locks. All write locks for the delivered writeset are atomically requested. If there is no lock on the object, the lock is granted. If there is a write lock or all the read locks are from transactions already delivered, then the request is enqueued. If there is a read lock from a transaction T_j not yet delivered, T_j is aborted and the write lock is granted. If T_j was already broadcast, an abort message is sent. After this lock phase, if the delivered transaction was local, a commit message is sent. Thus, a second message, only reliable, is sent

for every broadcast transaction with its final outcome, following a weak voting approach (the commit vote for a transaction T_c is sent after the delivery of the writeset of T_c; the abort vote for a transaction T_a is sent before the delivery of the writeset of T_a). Whenever a write lock is granted, the corresponding operation is performed. The queues of requested locks ensure that conflicting operations are serially performed [*f*]. Updates of non-conflicting transactions can be committed at different orders, but the write locks prevent users from perceiving the lack of sequentiality. Nevertheless, inversions may arise. The exact correctness criterion is 1SR′, based on periodically sequential (*S) server-centric consistency. A PostgreSQL implementation of the SER protocol, Postgres-R, was published and further discussed by Kemme and Alonso [68].

CS [69] is a version of SER for databases where read locks are released after the read operation if the transaction will not update the object later on [*a*]. The algorithm is identical to that of SER, except for the decision phase, where transactions holding short read locks are not aborted when the delivered transaction requests a write lock in the same object. Instead, the delivered transaction waits for the short locks to be released. This way, read operations are less affected by writes, but inversions may increase and the resulting execution may be non-serializable. Regarding server-centric consistency, the update application mechanism, as in SER, allows independent transactions to commit in a different order to that of their delivery, although this is concealed to users by holding write locks from decision time. Therefore, a periodically sequential (*S) replica consistency is used as basis to ensure the correctness criterion of 1CS.

SI [69] is deployed upon a local lock-based database providing snapshot isolation. On request, read-only transactions are immediately committed without any communication with the rest of the system [*a, c*], while update transactions must be atomically broadcast [*b*]. The sequence number of their writesets is used as end-of-transaction (EOT) timestamp. The begin-of-transaction (BOT) timestamp of T_j is set to the highest sequence number EOT_i such that T_i and all transactions with lower sequence numbers have terminated in the replica at the starting time of T_j. Certification [*d*] is made by checking the EOT timestamp of the last transaction currently holding or waiting to acquire a write lock on an object X with the BOT timestamp of the delivered transaction wanting to write X. If both transactions are concurrent ($\text{EOT} > \text{BOT}$), the delivered transaction is aborted. If there

is no write lock on object X, the comparison is made with the EOT of the transaction that wrote its current version. Non-aborted transactions request their write locks in delivery order. As soon as a lock is granted, the corresponding operation is performed [e]. Again, write locks prevent users from perceiving the lack of sequentiality when non-conflicting transactions are applied at different orders in different replicas. This periodically sequential (*S) consistency level gives the 1SI correctness criterion, as inversions are not precluded.

Hybrid [69] is a combination of previous protocols SER and SI. Read-only transactions are executed in snapshot isolation mode [a]: they get a version of the database corresponding to their start time (inversions may arise). Update transactions are executed like in SER [b]. Read-only transactions commit immediately without any communication with the rest of the nodes [c, e], whereas update transactions are broadcast in total order [d]. A validation phase based on locks is performed whenever a transaction is delivered [f]. Only the owner of the transaction is able to decide the final outcome, which is reliably broadcast at the end of the lock phase. As in SER, only local transactions not yet delivered can be aborted. Thus, the decision is not for the delivered transaction, but for transactions that would be subsequent in the total order. Whenever the delivered transaction gets a write lock, the corresponding operation is performed [g], which can produce the reordering of independent transactions. The server-centric consistency is thus periodically sequential (*S), and the correctness criterion is 1SR$'$.

NODO (NOn-Disjoint conflict classes and Optimistic multicast) was proposed by Patiño-Martínez et al. [98] as a middleware-based replication protocol that aims to enhance scalability of existing systems reducing the communication overhead. Data is partitioned into disjoint basic conflict classes, which are then grouped into distinct compound conflict classes. Each compound conflict class has a master or primary site, which allows the protocol to rely on the local concurrency control for deciding the outcome of transactions [f]. A transaction accesses any compound conflict class, which is known in advance. Read-only transactions can be executed in any node [a], as a complete copy of the database is stored at each node. Update transactions, however, are broadcast in total order to all sites [b]. An optimistic delivery allows the overlap of the time needed to determine the total order with the time needed to execute the transaction. For concurrency

purposes, each site has a queue associated to each basic conflict class. When a transaction T is optimistically delivered (opt-delivered), all sites queue it in all the basic conflict classes it accesses. At the master site of T, whenever T is the first transaction in any of its queues, the corresponding operation is executed [*c*]. When T is delivered in total order (TO-delivered), if the tentative order was correct, T can commit as soon as it finishes execution. Then, its writeset is reliably broadcast in a commit message to all sites [*e*], where updates are applied after T is TO-delivered and as soon as it reaches the head of each corresponding queue [*h*]. When all updates are applied, T commits. Committed transactions are removed from the queues. If messages get out of order, any conflicting transaction T' opt-delivered before T and not yet TO-delivered is incorrectly ordered before T in all the queues they have in common. T must be reordered before the transactions that are opt-delivered but not yet TO-delivered. If T' already started its execution, it must be aborted at the master site. Read-only transactions are queued at their delegate node after transactions that have been TO-delivered and before transactions that have not yet been TO-delivered [*c*]. Once a read-only transaction executes, it is locally committed with no further communication [*d, g*]. A performance evaluation of an implementation of this protocol was conducted by Jiménez-Peris et al. [63].

A drawback of NODO is that a mismatch between the tentative and the definitive orders may lead to an abortion. Taking advantage of the master copy nature of this protocol, a new version was also proposed in that paper [98]: the **REORDERING** algorithm, where a local site can unilaterally decide to change the serialization order of two local transactions following the tentative order instead of the definitive one in order to avoid such aborts. Remote nodes must be informed about the new execution order (this information is added to the commit message). Restrictions apply, as reordering is only possible if the conflict class of the reordered transaction, the first one in the local tentative order, is a subset of the conflict class of the so-called serializer transaction, the one that comes first in the definitive total order. The commit message of REORDERING contains the identifier of the serializer transaction and follows a FIFO [*i*] order (several transactions can be reordered with respect to the same serializer transaction, and their commit order in all sites must be the same). When a transaction T_i is TO-delivered at its master site, any non-TO-delivered local transaction T_j whose conflict class is a subset of that of T_i is now committable (it will be committed when it finishes execution, as if it were TO-delivered in the NODO algorithm). Local non-TO-delivered conflicting transactions that cannot be reordered and have

started execution must be aborted. At remote sites, reordered transactions are only committed when its serializer transaction is TO-delivered at that site.

Both NODO and REORDERING allow inversions and ensure, with a periodically sequential (*S) server-centric consistency, the correctness criterion of 1SR′.

Pronto [102] follows the primary-backups approach, so transactions must be addressed to the primary [*a*]. Clients do not need to know which node is the primary at any moment, as the first part of their algorithm is devoted to find the current primary by consecutively asking all the replicas. After transaction execution in the primary, Pronto sends to the backups the ordered sequence of all its SQL sentences [*b*]. This allows heterogeneity in the underlying DBMS as long as they follow the same SQL interface. Possible non determinism is said to be solved by introducing ordering information that allows the backups to make the same non-deterministic choices as the primary. As all replicas completely execute each transaction, Pronto assimilates to an active approach. But unlike active replication, backups process transactions after the primary, allowing the primary to make non-deterministic choices and export them to the backups. The certification process [*c*] does not consider the conflicts between transactions. Instead, a simple integer comparison is performed to check if the transaction was executed in the same epoch where it is trying to commit. A change in the epoch, which results in another server being the primary, occurs when any backup suspects the primary to have failed and broadcasts (also in the total order used for broadcasting transactions) a new epoch message. As these suspicions may be false, the primary may be still running and so it aborts all transactions in execution upon the delivery of the new epoch message. Moreover, due to the time it takes for the message to be delivered, it is possible that multiple primaries process transactions at the same time. To prevent possible inconsistencies, delivered transactions are committed in backups only if they were executed in the current epoch (by the current primary). After termination, all replicas (primary and backups) send the transaction results to the client [*d*]. Server-centric consistency is natively sequential, but inversions are precluded by serving all transactions in the primary, thus following a 1ASR/#S synchronization model that achieves the correctness criterion of 1ASR.

DBSM-RAC, Database State Machine with Resilient Atomic Commit and Fast Atomic Broadcast, was proposed by Sousa et al. [121] as an adaptation of DBSM

for partial replication. In partial replication, nodes maintain only the transaction information that refers to data items replicated in that node. Due to this, certification is no longer ensured to reach the same decision at all nodes. Instead, a non-blocking (to tolerate failures) atomic commit protocol must be run, in order to reach a consensus on transaction termination [*e*]. But atomic commit protocols can abort transactions as soon as a participant is suspected to have failed. This goes against the motivation for replication, as the more replicas an item has, the higher the probability of a suspicion, and the lower the probability of a transaction accessing that item to be finally committed. Resilient atomic commit solves this problem by allowing participants to commit a transaction even if some of the other servers are suspected to have failed, for which it requires a failure detector oracle. The second abstraction presented is Fast Atomic Broadcast, a total order broadcast which can tentatively deliver (FST-deliver) multiple times a message before deciding on the final total order (FNL-deliver). This optimistic behavior allows the overlap of the time needed to decide the total order with the time needed to run the resilient atomic commit, thus overcoming the penalty of the latter. A transaction T must start in a node that replicates all the items accessed by T [*a*]. Read-only transactions are locally committed [*b, d, f*]. Update transactions spread their information using the fast atomic broadcast [*c*]. As soon as T is first FST-delivered, all participating sites (those replicating any item accessed by T) certify T and send the certification result as their vote for the resilient atomic commit protocol. When T is FNL-delivered, if the tentative order was correct, the result of the resilient atomic commit is used to decide the final outcome of T. If T can commit, its write locks are requested and its operations executed as soon as they are granted [*g*]. Whenever the orders mismatch, the certification and resilient atomic commit started for T are discarded and the process is repeated for the final order. As in DBSM, write locks prevent users from perceiving the lack of sequentiality caused by independent transactions committing in different orders in different nodes. The server-centric consistency is periodically sequential (*S) and, as inversions are not precluded, the correctness criterion is 1SR′.

Holliday et al. [58] propose a pair of partial database replication protocols supporting multi-operation transactional semantics and aimed to environments where servers are connected by an unreliable network, subject to congestion and dynamic topology changes, where messages can arrive in any order, take an unbounded amount of time to arrive, or be completely lost (however, messages

will not arrive corrupted). Each site maintains an event log of transaction operations, where the potential causality among events is preserved by vector clocks. Records of this log are exchanged with the rest of servers in an epidemic way with periodic point-to-point messages. This exchange ensures that eventually all sites incorporate all the operations of the system. A node N_i also maintains a table $\mathscr{T}_i$ that contains the most recent knowledge of N_i of the vector clocks at all sites. This time-table, also included in the epidemic messages, ensures the time-table property: if $\mathscr{T}_i[k, j] = v$ then N_i knows that N_k has received the records of all events at N_j up to time v (which is the value of the local clock of N_j).

Transactions are executed locally. In the restricted access approach, **Epidemic restricted**, a transaction T can access only those data items that are permanently stored in the delegate node of the transaction. When T finishes, if it is read-only it is immediately committed without further processing [*a, c, e*]. Otherwise, its readset, writeset (with the updated values) and timestamp are stored in a pre-commit record in the delegate node to be epidemically spread [b]. The timestamp used is the i^{th} row of the time-table of N_i, $\mathscr{T}_i[i, *]$, with the i^{th} component incremented by one (the clock value at each node is incremented every time a new record is inserted into the log). This timestamp allows the protocol to determine concurrency between transactions in order to certify them. When N_i knows, by the clock information from epidemic messages, that this record has reached all sites, N_i must have received any concurrent transactions initiated in other nodes and thus has all the required information to certify T [d]. As there is no order guarantee, when a conflict is found between two concurrent transactions, both transactions must be aborted. Not aborted transactions are applied and committed at each node [f].

In the remote access approach, **Epidemic unrestricted**, remote objects can be read and written by maintaining a local temporary database in memory. When a local transaction wants to read a remote data item, the temporary database and pre-commit records from other sites are inspected trying to get a valid version of the item. If no valid version is available, a request record is added to the event log and epidemically transmitted [g]. A site replicating that item would be able to turn that record into a response one, storing it at its log and transmitting it later. Write operations of remote data items do not require the current value of the item.

Both the restricted and the unrestricted versions of this algorithm allow inversions of read-only transactions. The updates to the local database are applied following

the causal order of the log, thus ensuring periodically sequential (*S) server-centric consistency. As a result, 1SR′ is guaranteed.

OTP was proposed by Kemme et al. [71] (along with OTP-Q, OTP-DQ and OTP-SQ) to achieve high performance by overlapping communication and transaction processing in database replication systems providing full replication and one-copy serializability. OTP is a more refined version of OTP-99 [70], where transactions were restricted to access only one conflict class. OTP only considers update transactions, issued by clients that invoke stored procedures. Whenever a client sends a request to a node, this node forwards it to all sites in an atomic broadcast with optimistic delivery [*a*]. This primitive allows the overlap of the time needed to determine the total order with the processing of the message. To this end, a message is optimistically delivered (opt-delivered) in an initial tentative order. When the order is agreed, the message is delivered in total order (TO-delivered). Tentative and total orders may differ. The processing of transactions is then done in an active way: all sites execute all operations, i.e., there is no delegate node [*b*]. When the request is opt-delivered, all required locks are requested in an atomic step. This consists in queueing a read or write lock entry in the queue corresponding to the accessed data item. These queues are maintained by the protocol, so concurrency control is done at middleware level [*c*], deferring the execution of operations until the corresponding lock is granted [*d*]. This way, transactions are executed optimistically, but the commit operation is not performed until the total order is decided. If a transaction T is already executed when it is TO-delivered, or is already TO-delivered when it finishes execution, this means that the tentative order was correct. T commits and releases all its locks. On the other hand, if a transaction T is TO-delivered before it finishes execution, all its lock entries are inspected. Any transaction T' not yet TO-delivered with a conflicting granted lock is aborted: all its operations are undone, all its locks are released and its execution will be restarted later, as the tentative order was not correct and T must be executed before. Finally, all the locks of T are scheduled before the first lock corresponding to a transaction not yet TO-delivered. As independent transactions may commit at different orders, server-centric consistency is periodically sequential. Moreover, as OTP does not consider read-only transactions, inversions are trivially avoided. 1ASR is guaranteed by a 1ASR/*S synchronization model.

OTP-Q, **OTP-DQ** and **OTP-SQ** complement OTP with the management of read-only transactions. In all these protocols requests must be declared in advance as

queries or update transactions. Queries are only locally executed with no communication overhead [*e, g*]. A basic approach is taken in OTP-Q, a query Q is treated as if it were an update transaction being TO-delivered: any transaction not yet TO-delivered with conflicting granted locks is aborted and the locks of Q are inserted before the first lock entry corresponding to a not yet TO-delivered transaction. Operations are deferred until the corresponding lock is granted [*f*].

Although simple, OTP-Q requires that queries know in advance all the data items they want to access, which might not be feasible due to their usual ad hoc character. Moreover, queries may access many items and execute for a long time. Locking all data at the beginning will thus lead to considerable delay for update transactions. In order to overcome these disadvantages, authors propose OTP-DQ, which treats queries dynamically, allowing queries to request their locks whenever they want to access a new item. To avoid violations of the one-copy serializability, data items are labeled with version numbers corresponding to the position inside the total order of the last transaction that updated them. Each update transaction is also identified with such a version number. Queries maintain two timestamps corresponding to the version numbers of a pair of transactions between which the query can be safely serialized. Each time an update transaction requests a lock on an item read by a query, or whenever the query reads an item, timestamps are adjusted in order to ensure that the query does not reverse the serial order established by the total order. In case that it is detected that the order has been reversed, the query is aborted [*h*].

Both OTP-Q and OTP-DQ place read-only transactions properly inside the serial order but, as server-centric consistency is periodically sequential (*S) and queries are not enforced to respect real-time precedence (their processing is local and the validation rules merely aim for serializability), the correctness criterion is 1SR′.

Finally, OTP-SQ uses multiversioning for providing each read-only transaction with appropriate versions of all data items it wants to access, i.e., with a snapshot. This way, queries do not acquire locks, do not interfere with updates and can be started immediately [*i*]. The server-centric consistency and the correctness criterion are the same of OTP-Q and OTP-DQ.

RJDBC [44] is a simple and easy to install middleware that requires no modification in the client applications nor in the database internals. A client request arrives to a system node [*a*], which, for each operation of the transaction, and

depending on the underlying database concurrency control in use [*b*], decides to broadcast the operation in total order to all replicas [*c*] or not (e.g., read operations in a multi-version concurrency control providing snapshot isolation are not required to be broadcast). If not broadcast, the operation is executed locally. Otherwise, it is sequentially executed upon delivery (the same applies for the final commit operation [*d, f*]). As all nodes execute all significant operations in the same order, no decision phase is necessary [*e*]. Server-centric replica consistency is ensured to be natively sequential (#S). The guaranteed correctness criterion depends on the underlying concurrency control and on the decision to broadcast operations [*g*]. If serializability is used for local isolation but read operations are not broadcast, then 1SR′ is provided. On the other hand, broadcasting also read operations allows the system to achieve 1ASR. Similarly, if snapshot isolation is provided, then 1SI can be achieved without broadcasting read operations, while 1ASI requires such a broadcast.

RSI-PC (Replicated Snapshot Isolation with Primary Copy) was proposed by Plattner and Alonso [107] as a scheduling algorithm for their middleware-based replication platform, Ganymed, where there is a master node and *n* slave nodes. RSI-PC takes advantage of the non-blocking nature of read operations in snapshot isolation (read operations are never blocked by write operations nor cause write operations to wait for readers) by treating read-only and update transactions in different ways, thus providing scalability without reducing consistency. All client requests are addressed to the scheduler, which forwards update transactions to the master node and performs load balancing with read-only transactions among the slaves [*a*]. Updates are started in the master without any delay [*c*] and handled under snapshot (actually, under the serializable mode of the underlying Oracle or PostgreSQL databases, which is a variant of snapshot isolation where conflict detection is performed progressively by using row write locks, ensuring that the transaction sees the same snapshot during its whole lifetime) or read committed isolation [*e*]. No decision phase is necessary [*f*], as the local concurrency control of the master replica is enough. After an update transaction commits in the master, its writeset is sent to the scheduler, which has, for every slave, a FIFO update queue and a thread that applies the contents of that queue to its assigned replica [*j*]. Although this constitutes a lazy behavior (update propagation is done out of the transaction boundaries), this algorithm is equivalent to an eager service as strong consistency can be always guaranteed.

Read-only transactions are processed in the slaves using snapshot isolation [*d*], thus no conflicts appear between writeset application and query processing in the slaves, as readers are never blocked by writers in snapshot isolation. However, to ensure strong consistency for read-only transactions, they are delayed until all pending writesets are applied in the selected slave [*b*], thus providing read-only transactions with the latest global database snapshot. For read-only transactions that cannot tolerate any delay there are two choices: to be executed in the master replica (thus reducing the available capacity for updates), or to specify a staleness threshold. No group communication is established by read-only transactions [*i*].

As transaction-remote and client-response strategies are not detailed in the paper, we assume the most plausible choice [*g, h*]. While server-centric consistency is thus natively sequential (#S), the ensured correctness criterion depends on the isolation mode of update transactions and the staleness toleration of read-only transactions: if queries do not tolerate staleness, they are provided with atomic, inversions-free consistency.

SRCA [79] is a centralized protocol, where all transaction operations must be addressed to the centralized middleware, which redirects the operations to any replica [*a*]. Read-only transactions are locally committed without any global communication [*b, d*]. For update transactions, the group end coordination [*c*] is made *after* the decision [*e*] is taken by the centralized middleware. The sequential application of writesets ensures a natively sequential (#S) server-centric consistency, which combined with snapshot isolation at database level and no mechanisms for inversion preclusion[4] results in the correctness criterion of 1SI.

SRCA-Rep was proposed by Lin et al. [79] as a middleware protocol that guarantees one-copy sequential snapshot isolation in replicated databases. Each replica in the system is locally managed by a DBMS providing snapshot isolation. The database is fully replicated, so transactions can be executed in a delegate replica until the commit operation is requested. Then, read-only transactions are locally committed without any communication [*b, d*], whereas writesets from update transactions are broadcast to the rest of replicas in uniform total order [*c*]. Each

[4]Remember that, in snapshot isolation, inversions are conservatively precluded if the snapshot provided to transactions always corresponds to the latest available snapshot in the entire system. Optimistically, transactions may get an older snapshot but be restarted (getting a new snapshot) when the inversion is detected.

replica performs a certification [*e*] for each writeset, following the delivery order. Successfully certified writesets are then enqueued in the *tocommit* queue, to be later applied and committed in the local copy of the database, and in the *ws* list, which contains all the transactions applied in the system.

To reduce the overhead of the certification, it is performed in two steps. Each successfully certified transaction receives a monotonically increasing identifier called *tid*. When a transaction T requests commitment in its delegate node R_d, a local validation is performed: its writeset is compared against those of the transactions in the *tocommit* queue of R_d. If any conflict is found (non-empty intersection of writesets), T is aborted. Otherwise, the *tid* of the last certified transaction in R_d is set as the *cert* value of T. When T is delivered at remote replica R_r, its writeset is compared against those of the *ws* list whose *tid* is greater than the *cert* of T. Any conflict leads to the abortion of T. Otherwise, T receives its *tid* and is enqueued in both the *tocommit* queue and the *ws* list of each of the replicas.

To improve performance, a concurrent writeset application is allowed. When some conditions are satisfied,[5] several non-conflicting transactions from the *tocommit* queue are sent to the database to be applied and committed. This can alter the commit order, causing *holes* and breaking the sequentiality. Thus, new local transactions must be prevented from starting [*a*] as long as there are holes in the commit order. The server-centric consistency level is thus periodically sequential (*S) and, as inversions are not precluded, the correctness criterion is 1SI.

DBSM* was proposed by Zuikevičiūtė and Pedone [141] as a readsets-free version of DBSM. Local isolation is still managed with 2PL, but the certification test enforces the first-committer-wins rule of snapshot isolation. In order to maintain the original 1SR′, a conflict materialization technique is used. The database is logically divided into disjoint sets, and each one is assigned to a different node, which is responsible for processing update transactions that access that set [*b*]. Read-only transactions, on the other hand, are scheduled independently of data items accessed [*a*]. An additional control table containing one dummy row for each logical set allows the materialization of write-read conflicts, in order to be detected in the certification. This way, a transaction that reads data from a remote

[5]The conditions that must hold to send a writeset T to the database of replica R are: (a) no conflicting writeset is ordered before T in the *tocommit* queue; and (b) either T is local or there are no local transactions waiting to start in R or T does not start a new hole.

logical set is incremented with an update to the corresponding row in the control table. As in DBSM, server-centric consistency is periodically sequential (*S) and, as inversions are not precluded, the resulting correctness criterion is 1SR′.

PCSI Distributed Certification [40] provides prefix consistent snapshot isolation (PCSI), a form of generalized snapshot isolation (GSI), which is equivalent to 1SI. In this distributed certification protocol, read-only transactions directly commit [*c*] without communicating with the rest of nodes [*a*] and update transactions broadcast their writeset in total order [*b*] and are later certified [*d*] and applied at each replica, ensuring a natively sequential (#S) server-centric consistency.

Tashkent-MW and **Tashkent-API** were proposed by Elnikety et al. [41] with the goal of uniting both transaction ordering and durability, whose separation in common database replication systems is claimed by these authors as being a major bottleneck due to the high cost associated to sequential disk writes required to ensure in the database the same commit order decided in the middleware. The replication system proposed is compound of a set of database replicas and a replicated certifier, responsible for validating transactions [*d*] and provide replicas with remote writesets. A snapshot isolated database is used in each replica, where read-only transactions are locally committed (no validation nor communication needed [*a, c, e*]). When an update transaction T finishes in its delegate, it is sent to the certifier [*b*], which replies with the validation result, the writesets generated in remote replicas and the commit order to be enforced in all nodes. The delegate then applies remote writesets and commits or aborts T, depending on the validation result and respecting the global order imposed by the certifier.

In Tashkent-MW, durability is moved to the middleware and, thus, commit operations are fast in-memory operations, which are done serially to ensure the same global order at each replica. Writesets are also serially sent to the database [*f*], but synchronous writes to disk are disabled. On the contrary, in Tashkent-API, commit ordering is moved to the underlying database management system, which is modified to accept a commit order, so multiple non-conflicting writesets can be sent concurrently to the database [*h*] while ensuring the correct commit order. This way, the database can group the writes to disk for efficient disk IO. In both protocols, the transmission of writesets to remote nodes is completely decoupled from transaction execution, as it is done as part of the reply of the certifier to the requests of other nodes [*g*].

Both in Tashkent-MW and Tashkent-API, the commit order followed by replicas is the same. However, the state of the underlying database replicas is updated by grouping multiple commit operations into one single disk write. As this grouping is not forced to be the same in all replicas, servers will not follow the same exact sequence of states: some of them may omit some intermediate states that were present at other servers.[6] Nevertheless, this does not impair consistency and 1SI (as inversions are not precluded) is guaranteed.

DBSM-RO-opt [96] aims to extend the DBSM replication in order to provide atomic, inversions-free consistency [c] among the nodes. To do so, an optimistic approach is followed: read-only transactions are locally executed in their delegate replica but are also atomically broadcast when the user requests to commit [a], so both read-only and update transactions are checked, looking for write-read conflicts: read-only transactions are inspected only by their delegate replica, while update transactions are certified by each replica in the system [b]. This avoids inversions, while the server-centric consistency is, as in DBSM, periodically sequential (*S). As a result, the correctness criterion is 1ASR, provided by means of a 1ASR/*S synchronization model.

DBSM-RO-cons [96] also aims to extend the DBSM replication in order to provide inversions-free consistency but, in this case, a conservative (pessimistic) approach is followed: read-only transactions are atomically broadcast when they begin [a] and are executed only when all update transactions ordered before are

[6] Imagine a Tashkent system with an initial state, or version, v_0. There are three nodes in the system and each one starts the execution of a local update transaction: R_1 executes T_1, R_2 executes T_2 and R_3 executes T_3. If all the transactions are independent, they will all positively pass their validation at the certifier. Suppose T_1 finishes the first. The certifier responds with the positive decision but it does not have any pending writeset for R_1, so this node commits T_1, reaching (from version v_0) version v_1 (corresponding to the updates of T_1). Now R_2 finishes the execution of T_2 and sends it to the certifier, which responds with the positive decision and with the writeset of T_1. As transactions are independent, R_2 sends them together to the database, which writes their commits in a single disk write, thus moving from version v_0 directly to version v_2 (corresponding to the updates of T_1 and T_2). Finally, R_3 finishes the local execution of T_3 and sends it to the certifier, which responds with the positive decision and with the writesets of T_1 and T_2. R_3 applies the three transactions in a single disk write, thus passing from version v_0 to version v_3 (corresponding to the updates of the three transactions). This way, the three replicas end with the same final state and no other possible transaction has been able to see any inconsistent state, but the sequences of database states differ from replica to replica, thus discarding a (natively or periodically) sequential server-centric consistency, as defined in Chapter 3.

committed in the executing replica. This means that not only the group-start communication is synchronous but the query must also wait for all pending writesets to be applied (this extra waiting time could be considered as a late occurrence of a deferred transaction-service, Ts1). Update transactions do not need to be broadcast at start time [*b*]. When finished in its delegate replica, a read-only transaction does not need any further communication [*c*] nor any certification [*e*], but update transactions must broadcast their information in total order [*d*] and undergo the usual conflict checking process [*f*]. The resulting server-centric consistency and correctness criterion are the same than in DBSM-RO-opt.

Alg-Weak-SI [36], as well as Alg-Strong-Session-SI and Alg-Strong-SI, is used in a system with a primary replica and several secondary nodes, where clients send transactions to any replica. Read-only transactions can be executed in the secondaries (without any further communication with the rest of nodes, [*e*]), but update transactions are forwarded to the primary [*a*]. This protocol follows a lazy propagation of updates, so no communication is established during the lifetime of transactions [*b*]. Instead, local concurrency control in the primary replica is the only responsible for deciding the outcome of update transactions [*c*], whose start, updates and final operation (commit or abort) are registered in a log which is later used to lazily propagate [*f*] these operations in order (a FIFO order is required, which provides a total order broadcast when there is only one sender) to the secondary replicas. The sending process inspects each log entry: a start operation is immediately propagated; update operations are inserted in the update list of the transaction they belong to; a commit entry for transaction T causes the broadcast of this operation along with the update list of T; an abort entry of transaction T is also propagated, discarding in this case the corresponding update list. In the secondaries, delivered messages are buffered in the *update* queue and processed in order. When the start message of T_i is processed –after waiting for the *pending* queue to be completely empty–, a refresh transaction T_i' is started. When the commit message of T_i (with the updates associated) is processed, a new thread is created to apply the updates of T_i using transaction T_i', and the commit operation of T_i is appended to the pending queue. This allows the protocol to concurrently apply writesets while ensuring the same commit order of transactions [*d*]. The server-centric consistency is natively sequential (#S) and, as read-only transactions are executed in secondary replicas without inversion preclusion, inversions may occur (queries may get an old snapshot). Thus, the correctness criterion is 1SI [*g*].

Alg-Strong-SI and **Alg-Strong-Session-SI** [36] guarantee natively sequential (#S) server-centric consistency and ensure strong snapshot isolation (1ASI) and strong session snapshot isolation (1SI+), respectively [*c*]. While 1ASI avoids all inversions, 1SI+ prevents inversions within the same user session. In order to provide 1SI+, a version number is assigned to each session, corresponding to the version installed by the last update transaction in that session. When a read-only transaction of the same session wants to start, all writesets with version numbers inferior to the session version number must be applied in the secondary replica prior to the start of the read-only transaction [*a*]. If, instead of having one session per client, there is a single session for the system, then 1ASI is provided. Update transactions can start immediately as they are all executed in the same primary replica [*b*]. Apart from this, these protocols are identical to Alg-Weak-SI.

One-at-a-time and **Many-at-a-time** [116] are two termination protocols that extend DBSM to provide a quasi-genuine partial replication, where a node permanently stores not more than the transaction identifier for those transactions that do not access any item replicated in that node. To avoid consequent unnecessary abortions, a non-trivial validation is performed, based on quorums. A transaction T can only be executed on a site that replicates all items accessed by T [*a*]. Read and write operations are executed locally according to the strict two-phase locking rule [*b*]. When a read-only transaction requests commitment, it is locally committed [*c, e, g*]. In the case of an update transaction, the transaction (identifier, delegate site, readset, writeset with updates, and the logical timestamp of the transaction submission) is broadcast [*d*] in a *weak ordering reliable broadcast*, an optimistic primitive that takes advantage of network hardware characteristics to deliver messages in total order with high probability. A consensus procedure is used to decide the total order of delivered transactions. The non-trivial validation consists in a voting phase where each site sends the result of its validation test to the rest of nodes. Each site can then safely decide the outcome of a transaction T when it has received votes from a voting quorum of T [*f*], i.e., a set of sites such that for each data item read by T, there is at least one site that replicates this item. Instead of first using consensus to determine the next transaction T and then executing the voting phase for T, a different approach is taken, overlapping both processes. In the one-at-a-time algorithm, each site votes for its next undecided transaction T and proposes it for consensus. By the time the consensus decides for transaction T, luckily every site will already have received the votes for T. If

consensus decides a transaction different from that voted by a site, a vote message is sent for the decided transaction. When a transaction is successfully validated, it is applied in the site [h] and the global version counter used to timestamp transactions is increased. This algorithm validates one transaction at a time, which can be a bottleneck if many transactions are submitted. The many-at-a-time algorithm, which does not rely on spontaneous total order, tries to solve this by proposing sequences of transactions and changing the validation test accordingly. As in original DBSM, periodically sequential (*S) server-centric consistency is ensured but inversions are not precluded, so the correctness criterion is 1SR$'$.

***k*-bound GSI** [6] is able to bound the degree of snapshot outdatedness from a relaxed GSI (1SI) to a strong SI (1ASI), while optimistically executing transactions; and it also provides a serializable level for those transactions requiring higher isolation (1SR$'$). As local DBMSs are only required to provide snapshot isolation, serializable transactions are parsed in order to transform SELECT statements into SELECT FOR UPDATE ones. This simplifies the detection of write-read conflicts, which are then governed by the first-committer-wins rule.

As server-centric consistency is natively sequential (#S), two snapshots taken at the same *real* time in different replicas may be different, as only states at the same *logical* time are guaranteed to be consistent. To allow an optimistic execution, before the first operation of each transaction T, an asynchronous $T.ID$ message is broadcast in total order [a], so that the logical starting time of T can be established. Then, the optimistic execution of T overlaps with the time required to complete such initial communication. Moreover, T specifies a value k as the maximum *distance* between the snapshot it took (corresponding to the real time of its start operation) and the snapshot created by the last transaction that committed in any system node before T started (corresponding to the logical time of the start operation of T). This distance is measured as the number of colliding writesets that are applied in the delegate node of T from the real starting time of T until its logical starting time (the delivery of $T.ID$). A colliding writeset is a writeset that has a non-empty intersection with the intended readset of T, which has to be declared in advance. When the number of colliding writesets is greater than k, T is aborted and will be restarted when $T.ID$ is processed. Thus, with $k = 0$, the transaction is executed under 1ASI; with $k > 0$, the achieved correctness criterion is 1SI (authors, to highlight the possibility of defining different

staleness levels, refer to the different *bound values* for the GSI criterion, as opposed to the standard GSI, which occurs with an infinite value of k). Overloading the meaning of k, a value of -1 indicates that T requires serializability.

When a read-only transaction finishes its operations, it is locally committed (after receiving its own *T.ID* message and as long as it has not been aborted in the meantime, for transactions with $0 \leq k < \infty$) without any communication with the rest of nodes [*b, d*], whereas the writeset of an update transaction is broadcast in total order to the rest of replicas [*c*]. A certification process [*e*] is performed in every replica for each delivered writeset. In order to avoid sending readsets, the decision for serializable transactions is taken in their delegate node [*f*] and then broadcast to the rest of nodes.

Tashkent+ [42] was proposed as an evolution of Tashkent-MW where a memory-aware load balancing is performed in order to further minimize the disk IO requirements. Changes to the previous system include the addition of a scheduler, with different scheduling algorithms, and an optimization, called update filtering, for reducing the update propagation load. Transaction types are predefined and the scheduler is able to estimate the amount of memory, called working set, that each type will need. With this information, authors use a bin packing heuristic to group transaction types so that their combined working sets fit together into the available memory, thus avoiding memory contention and subsequent disk IO. Servers are assigned to transaction groups and this allocation can be dynamic for changing workloads. Different proposed scheduling algorithms differ in the way the working sets are estimated. This way, each transaction is dispatched to a server assigned to its transaction group [*a*].

Update filtering consists in identifying unused tables in a replica (those not accessed by the transaction group to which the replica is assigned) and filtering out the updates to those tables, thus reducing the overhead of update propagation. This optimization is only possible under stable workloads, i.e. when the assignment of replicas to transactions groups is permanent. This way, Tashkent+ is essentially a fully replicated design but may, under some conditions, present the advantages of partial replication [*b*].

Apart from the changes explained above, the rest of the system works as in Tashkent-MW. Authors claim that the correctness criterion is 1SI+ (inversions precluded within sessions), as a given connection can execute only one specific

transaction type and will be, thus, always assigned to the same group of replicas. But no details are provided about how all replicas in the same group are atomically updated or how they provide transactions with updated snapshots. Thus, the correctness criterion is here considered to be 1SI [*c*].

Mid-Rep is a pessimistic weak voting protocol proposed by Juárez et al. [65] that provides three different correctness criteria on top of a DBMS supporting SI: 1SR′, 1ASI and 1SI. Transactions define the criterion they require. For 1SR′ transactions, all SELECT statements are turned into SELECT FOR UPDATE ones. For 1ASI transactions, a start message is sent in total order [*c*] and the transaction must wait to its delivery to proceed. When a read-only transaction finishes its operations, it is immediately committed at its delegate replica and no further processing is required. On the other hand, update transactions broadcast their writeset in total order to all available replicas, which will apply them sequentially [*d*] and terminate (commit or abort) each transaction according to the voting message sent by the master site (the delegate) of the transaction, achieving a natively sequential (#S) server-centric consistency. During writeset application, no other potentially conflicting local operation is allowed to start: all write operations and also read operations performed by 1SR′ transactions are thus disabled [*b*] (read operations from 1ASI or 1SI transactions are not deferred [*a*]).

SIRC [114] concurrently supports snapshot and read committed isolation, as long as both levels are provided by the local DBMS [*a*]. Read-only transactions are locally committed without any communication with the rest of replicas [*b, d*], while update transactions are broadcast in total order [*c*]. For SI transactions, a certification based on write-write conflicts is performed [*e*]. RC transactions do not need any decision phase [*d*]. Writeset application following the delivery order guarantees natively sequential (#S) server-centric consistency.

Serrano et al. [118] propose a replication protocol aimed to increase scalability of traditional solutions, commonly based on full replication and on a 1SR′ correctness criterion. These two characteristics are claimed to introduce an important overhead and to limit concurrency. Consequently, their proposal is to use partial replication and a more relaxed correctness criterion, 1SI, working

with snapshot-isolated underlying databases and a natively sequential (#S) server-centric consistency where inversions are not precluded. The client connects to a site [a] that at least stores the data accessed in the first operation of the transaction T. This node acts as the coordinator, assigning a starting timestamp to T and redirecting operations [c] to other nodes when necessary. In those other nodes, T must use the same snapshot, the one corresponding to its starting timestamp. To this end, each node starts dummy transactions each time that a transaction commits. When the redirected operation is the first operation of T in the forwarded node, the corresponding dummy transaction is associated to T (later operations will use the same transaction). Prior to execute each redirected operation, the changes previously produced by T are applied at the forwarded site (naturally, only those affecting data stored at that node) [b] (remember that the transaction-service policy applies at each participating node in the case of distributed transactions). After execution, the forwarded site propagates the result of the operation and all the new changes to the coordinator, which applies these changes before executing the next operation. When the client requests commitment of a read-only transaction, the coordinator multicasts a commit message to all participating sites [d] (there is no need for validation [f] nor execution in remote nodes [h]). In the case of an update transaction, the coordinator broadcasts its writeset in total order [e]. All sites perform then a certification [g]. If certification succeeds, all nodes apply the writeset (those nodes that have already performed some operations of T apply only the missing updates) in a non-overlapping way [i] and T can commit.

Zuikevičiūtė and Pedone [140] proposed a scheduling algorithm for the DBSM replication protocol. Aborts can be reduced if conflicting transactions are executed in the same node, thus letting the local concurrency control appropriately serialize them. On the other hand, parallelism improves performance, reducing response times. Considering this trade-off, a hybrid load balancing technique is proposed, which allows database administrators to give more or less significance to minimizing conflicts or maximizing parallelism. Maximizing Parallelism First, **MPF**, prioritizes parallelism and so it initially assigns transactions to nodes trying to keep the load even. If more than one option exists, then it tries to minimize conflicts. Minimizing Conflicts First, **MCF**, avoids assigning conflicting transactions to different nodes. If there are no conflicts, it tries to balance the load among the replicas. A compromise between the two opposite schemes can be achieved by a factor *f*. This way, update transactions are analyzed and a specific replica is chosen to be the delegate [b]. On the other hand, both techniques assign read-only

transactions to the least loaded replica [*a*]. Apart from this novel scheduling, the followed strategies are the same as in DBSM and, thus, the server-centric consistency and the correctness criterion are also the same.

WCRQ [110] is a bridge between consensus-based and quorum-based replication. Underlying databases provide serializability, using long read locks for reading operations and deferring write operations until the end of the reading phase [*a*]. When an update transaction T finishes in its delegate replica, a uniform total order broadcast [*c*] is sent to the rest of replicas with the transaction writeset. When it is delivered, each replica tries to get write locks for each item in the writeset of T. If there was one or more read locks on an object, every transaction holding them which is not yet serialized is aborted (by sending an abort message in uniform total order if it was already broadcast), and the write lock is granted to T. If there was a write lock in the object, or if some read locks are from transactions serialized before T, T waits until those locks are released. When a replica gets all the locks, it sends a point-to-point acknowledgment message to the delegate. When the delegate gets all the write locks and receives acknowledgment from a write-quorum of replicas, it sends a commit message in a uniform reliable broadcast. When this message is delivered, every replica commits the transaction. As these messages are not ordered, independent transactions may commit at different orders in different nodes. When a transaction commits, all other transactions waiting to get write locks in the updated objects are aborted (their delegate sends an abort message in a uniform reliable broadcast). When a read-only transaction finishes in its delegate replica, a message with the readset is sent to a read-quorum of replicas [*b*]. When this message is delivered, replicas try to get read locks for the items on the readset. When the locks are acquired, if the version is the same as the one read in the delegate, the replica sends back a positive acknowledgment message. Otherwise, a negative acknowledgment message is sent. In any case, read locks are released as soon as this validation is done. When the delegate receives positive acknowledgments from a read-quorum of replicas, it commits the transaction. Otherwise, if any negative acknowledgment is received, the transaction is aborted. For both read-only and update transactions, a quorum of replicas is required to get locks on the items and check that the current versions are equal to the accessed versions in the delegate [*d*]. As transaction-remote and client-response strategies are not detailed in the paper, we assume the most plausible choice [*e, f*]. The periodically sequential (*S) server-centric consistency is used in a 1ASR/*S synchronization model that

avoids inversions by ensuring that queries do not read old values. As a result, the 1ASR correctness criterion is guaranteed.

AKARA [32] allows transactions to be executed either in an active or in a passive manner (in both cases, interactivity is precluded). Upon transaction submission, the type (either active or passive) and the conflict classes of the transaction are computed, and an initial total order broadcast is sent with transaction information. After delivery, and once the transaction is the first in the processing queue, the transaction is started [*b*] (this introduces an additional wait to the synchronous message transmission, for both active and passive transactions). For passive transactions, a local execution phase is performed and afterwards the writeset is reliably sent to the rest of replicas [*c*] to be applied following the total order established by the broadcast sent at transaction start. For active transactions, no local phase exists [*a*]: transactions are initiated, executed and committed in all nodes at the same logical time (that of their slot inside the total order). No extra communication is needed for these transactions [*d*]. The server-centric consistency is periodically sequential (*S) and isolation corresponds to the snapshot level. As no mechanisms avoid inversions, the ensured correctness criteria is 1SI.

Zuikevičiūtė and Pedone [139] characterized different correctness criteria for replicated databases and presented three variants of BaseCON, one for each of the discussed correctness criteria. With **BaseCON for 1SR** transactions are serialized but the causal order may not be preserved (this corresponds to our correctness criterion of 1SR′). In **BaseCON for SC** (session consistency), transactions are serialized and the real-time order of those belonging to the same user session is also preserved and, thus, clients can always read their own previous updates (this corresponds to our correctness criterion of 1SR+). These two variants are identical except for the way the scheduler selects the executing replica for read-only transactions [*a*]: in BaseCON for 1SR, all replicas are considered and the transaction is forwarded to the least loaded one; in BaseCON for SC, the scheduler considers only those replicas where previous update transactions of the same session have been already applied. Once in the executing replica, read-only transactions start as soon as they are received [*c*] and commit locally. On the other hand, update transactions are broadcast in total order to every replica in the

system [*b*] and executed in active manner, so no local phase exists [*d*]. Strict two-phase locking is used to achieve serializability [*e*]. No decision phase is required [*f*] as all transactions can commit, but update transactions must wait [*g*] for all previously delivered conflicting update transactions to commit in this replica before starting. The commit order of all update transactions is required to be the same as their delivery order, thus guaranteeing a natively sequential (#S) server-centric consistency. Transaction results are sent from each executing replica to the scheduler, which sends to the client only the first of the replies [*h*].

The third version of the protocol by Zuikevičiūtė and Pedone [139] is **BaseCON for strong 1SR**, which always preserves the real-time (or causal) order of transactions in their serialization. To this end, some changes are applied to previous systems: read-only transactions are directed to the scheduler but also broadcast in total order to all replicas, like update transactions [*a*].[7] The scheduler then determines, for the read-only transaction, the set of replicas where preceding update transactions of any client have already been committed. From this set, the scheduler selects the least loaded server, where the query immediately starts its optimistic execution. When this transaction is delivered in the chosen replica by the total order broadcast, a test is performed to check if the scheduler has changed since this transaction was scheduled. In this case, the transaction is aborted and restarted. This check allows the system to tolerate failures and cannot be considered as a decision, as the transaction always commits. Server-centrically, replica consistency is natively sequential (#S). A 1ASR/#S synchronization model precludes inversions by scheduling read-only transactions to updated replicas, thus achieving 1ASR correctness criterion.

***g*B-SIRC** [115] is deployed upon a database offering both read committed and snapshot isolation levels [*b*]. This protocol provides several correctness criteria: one based on the read committed isolation (1RC) and another based on snapshot isolation, with a configurable level of staleness, defined by factor g, from 1ASI (or strong SI) with $g = 0$ to 1SI (or standard GSI) with an infinite value of g. Intermediate values of factor g allow transactions to define the exact amount of outdatedness they can tolerate (authors refer to this non-standard criterion as

[7] However, unlike update transactions and despite being the client request addressed to all replicas in the system, only the node chosen by the scheduler will execute the transaction, thus serving the client request.

g-Bound). Similarly to *k*-bound GSI, all SI-based transactions (those providing a value for *g*) broadcast an asynchronous *T.ID* message in total order when they start [*a*], which allows their optimistic execution while establishing a global starting point that would be enforced when transactions abort due to a number of conflicts greater than *g*. Read-only transactions can be locally committed without any global communication [*c, e*]: RC queries commit as soon as they finish their operations, while SI ones must wait to the processing of their *T.ID* message and, if they have not been aborted during the meantime, they can be locally committed. Regarding update transactions, once they finish their operations, their writesets are broadcast in total order [*d*]. For 1RC update transactions, no decision phase is implemented [*e*]. All other update transactions are certified in search for write-write conflicts [*f*]. Whenever a writeset is committed in a replica, local SI-based transactions are validated in search for write-read conflicts with it [*g*], which are tolerated up to *g*. As writesets are applied in a non-overlapping manner, the server-centric consistency level is natively sequential (#S). Inversions are precluded only for 1ASI transactions.

5.3 Scope of the Proposed Model

The policy-based characterization model proposed in Chapter 4 and used for this survey is intended to be general enough to cover all possibilities in replication systems, thus providing a tool able to represent their basic skeleton. The set of strategies followed by a replication system constitutes its operational basis and allows an adequate comparison between systems.

Obviously, many finest-grained details, like optimizations or concurrency control rules, are not covered by this characterization, as intending otherwise would result in an extremely complex model. Thus, there is a trade-off between simplicity and completeness.

Moreover, and despite our efforts, this model is not valid for all replication systems. This is the case of *distributed versioning* [5], a replication protocol tailored to back-end databases of dynamic content web sites, characterized by presenting a low rate of update operations. This protocol aims to achieve scalability while maintaining serializability. The cluster architecture for distributed versioning consists of an application server, a scheduler, a sequencer and a set of database replicas. In order to achieve serializability, a separate version number is

assigned to each table. Each transaction issued by the application server is sent to the scheduler, specifying all the tables that are going to be accessed in the whole transaction and whether these accesses are for reading or for writing. The scheduler forwards this information to the sequencer, which atomically assigns table versions to be accessed by the operations in that transaction. This assignment establishes the serial order to be enforced and allows transactions to concurrently execute operations that do not conflict. All transactions can commit (Td0), those conflicting will follow the serial order dictated by the sequencer. New versions become available when a previous transaction commits or as a result of last-use declarations (an optimization for reducing conflict duration). After the assignment for transaction T is completed, the application server can start to submit the operations of T. The conflict-aware scheduler is able to forward a read operation to the updated least loaded replica. Write operations are broadcast to all replicas and actively executed. This scheduling could be interpreted as a Cq2 for reads and Cq4 for writes. However, in this case it is not the whole transaction which is scheduled but each single operation inside the transaction. It could also be represented as Gl3 and Ge3 for write and commit operations, but the communication initiative is not taken by a server executing the affected transaction. Indeed, no communication is ever established among replicas. Instead, each operation sent from the application server is forwarded by the scheduler to the corresponding replica(s), which execute them independently. Thus, communication is done only between the application server and the scheduler, and between the scheduler and the replicas. Once in a replica, an operation must wait for all its version numbers to be available, which could be represented as Ts1 or Tr1-p (although, again, it is not the transaction start which is deferred but the start of each single operation inside the transaction). Concurrency control is thus made at middleware level (the database isolation level is not detailed in the paper). Once a replica executes the operation, it returns its results to the scheduler. The first reply received by the scheduler is sent back to the application server (Cr1). As read operations must wait also for their version numbers to be available before starting, the correctness criterion is 1ASR. In summary, the main problem for describing distributed versioning with our model is that this system divides transactions into their individual operations and, while the management of each of these operations can be represented with our strategies, the handling of the whole transaction cannot be depicted by our model.

A similar middleware-based system is presented by Cecchet et al. [26] for clustering back-end databases of large web or e-commerce sites. C-JDBC also features

a scheduler that sends update operations to all involved servers while performing load balancing for read operations. But in this case, client requests contact a server and each server contains a scheduler component. C-JDBC supports both full and partial replication, while ensuring atomic, inversions-free consistency between replicas: at any single moment, only one updating operation (write, commit or abort operation) is in progress in the virtual database, and responses are returned to the client only once all servers have processed the request. The correctness criterion will then depend on the isolation level offered by the underlying databases. As for the previous system, the processing of each single operation can be depicted with our model, but that of the whole transaction would constitute a loop of such a representation of single operations.

5.4 Discussion

Chronologically ordered characterizations of Table 5.1 summarize the evolution of database replication systems since their appearance. Earlier systems – distributed databases with some degree of replication– were devoted to provide the highest correctness criterion, 1ASR, using to this end the serializable isolation level in local databases and rigid synchronization mechanisms, inherited from standalone database management systems, such as distributed locking for concurrency control (which involves a linear communication with other servers), or an atomic commit protocol like 2PC (which requires several rounds), in order to reach a consensus among participants about transaction termination and thus ensure consistency. Examples of these earlier systems are 2PL & 2PC [50], BTO & 2PC [12], Bernstein-Goodman [14], OPT & 2PC [119] and O2PL & 2PC [23]. However, these mechanisms restricted concurrency, thus severely reducing performance and scalability.

Research efforts focused then on improving these factors trying to reduce communication and replication overhead with new concurrency control algorithms and more efficient termination management, localizing the execution of operations in delegate or master sites, simplifying termination with the use of group communication systems, using optimistic communication primitives, considering different topologies, relaxing isolation and consistency or introducing partial replication schemas. One example of such relaxed isolation, which is still valid for a wide range of applications, is the snapshot isolation level. An interesting feature of snapshot isolation is that read operations are never blocked by write

operations, nor cause write operations to wait for readers. SI became popular and many database replication systems started to provide this isolation. Some systems exploiting this level and offering correctness criteria based on snapshot isolation are SI [69], RSI-PC [107], SRCA and SRCA-Rep [79], PCSI Distr. Cert. [40], Tashkent-MW and Tashkent-API [41], Alg-Weak-SI, Alg-Str.-SI and Alg-Str.Ses.-SI [36], *k*-bound GSI [6], Tashkent+ [42], Mid-Rep [65], SIRC [114], Serrano et al. [118], AKARA [32] and *g*B-SIRC [115].

Other proposals aimed at adaptability, designing systems able to provide different consistency guarantees that would fit better the requirements of modern applications, which usually include different types of transactions that require different levels of isolation. This led to a new generation of protocols that support different correctness criteria at the same time (such as RSI-PC [107], *k*-bound GSI [6], Mid-Rep [65], SIRC [114] and *g*B-SIRC [115]), which improved performance by executing each transaction at the minimum required level of isolation.

Considering each policy separately, it is clear that in some cases there is a majority strategy with very few exceptions, while in other policies there is no pronounced trend towards any specific strategy. Some choices may have strong implications in consistency or performance, and this may make systems favor ones against others. Let us analyze each policy in detail.

The most used client-request (Cq) strategy is Cq1: any server can process a request. This policy allows an easy management of requests and load balancing, although it requires a correct global concurrency control in order to avoid inconsistencies. On the other hand, systems that use primaries or master sites may rely on the local concurrency control of such nodes but require client requests to be addressed or forwarded to such servers (Cq2). This is the case of Alsberg-Day [4], Fast Refresh Df-Im and Fast Refresh Im-Im [97], Pronto [102], RSI-PC [107], DBSM* [141], Alg-Weak-SI, Alg-Str.-SI and Alg-Str.Ses.-SI [36], and Tashkent+ [42]. Cq2 is also used by systems that provide partial replication and need to address client requests to a server containing the data required by the operation (DBSM-RAC [121], One-at-a-time and Many-at-a-time [116], Serrano et al. [118]). Other systems also use Cq2 because they require transactions to be addressed to updated replicas, in order to provide stronger consistency, such as BaseCON for SC [139]. Finally, scheduling algorithms MPF and MCF [140] may also select a specific server in order to minimize abortions by executing conflicting transactions at the same node, thus relying on local concurrency control to appropriately serialize transactions.

Forwarding the client request to all system nodes (Cq4) in a total order broadcast is a possible approach for establishing an early synchronization point. Systems such as NODO and REORDERING [98], and the families of OTP [70, 71] and BaseCON [139] implement such a client-request policy. Systems that require total order guarantees for synchronizing transaction execution must wait for such a communication primitive to agree on the delivery order. NODO and REORDERING move their synchronization point to the start of the transaction and use an optimistic delivery which allows the system to overlap the time needed by the GCS for the agreement with the time needed to execute the transaction. By the time the delivery order is decided, the transaction has already progressed with its operations in its delegate server. On the other hand, both OTP and BaseCON families follow a Cq4 policy in order to execute update transactions in an active manner, while queries are executed at only one server.

Surveyed systems mainly follow, in their transaction-service policies (Ts), the strategy of immediate service (Ts0), under which transactions are started as soon as the server has enough free resources. In some cases, it is necessary to block the processing of transactions until some condition holds (Ts1). This is the case of several systems: NODO, REORDERING [98], OTP-Q and OTP-DQ [71], where concurrency control is done at middleware level and thus transactions must wait for the end of previous conflicting operations in order to be started; RSI-PC [107], Alg-Str.-SI and Alg-Str.Ses.-SI [36], where the service of transactions is deferred to guarantee stronger consistency; SRCA-Rep [79], where local transactions must be prevented from perceiving the lack of sequentiality; Mid-Rep [65], where potentially conflicting local operations are disabled during writeset application; and the algorithm by Serrano et al. [118], where cohorts of distributed transactions must apply the updates of previous operations of the transaction (served by other nodes) before executing the requested operation in their local database.

Systems that actively execute transactions (OTP-99 [70], OTP, OTP-Q, OTP-DQ and OTP-SQ [71], AKARA [32], BaseCON for 1SR, BaseCON for SC and BaseCON for strong 1SR [139]) are said to implement the strategy of no local service (Ts2) for those active transactions.

Regarding group-start (Gs) strategies, only few systems require to make a communication at transaction start, thus establishing a global starting point for transactions. That is the case of DBSM-RO-cons [96], which totally orders queries to provide 1ASR; k-bound GSI [6], Mid-Rep [65] and gB-SIRC [115], which guarantee 1ASI by totally ordering transaction starts; and AKARA [32], which

moves the required synchronization point to transaction start and allows active and passive transaction processing.

With regard to the degree of replication (Dr), earlier systems (2PL & 2PC [50], BTO & 2PC [12], Bernstein-Goodman [14], OPT & 2PC [119], O2PL & 2PC [23]) were mostly distributed databases where replication was not widely used (Dr1). After the generalization of full replication (Dr2), only few systems (Fast Refresh Df-Im and Fast Refresh Im-Im [97], DBSM-RAC [121], Epidemic restricted and Epidemic unrestricted [58], One-at-a-time and Many-at-a-time [116], Serrano et al. [118]) feature partial replication (Dr1), mainly to minimize the cost of update propagation and application, although other mechanisms or constraints must be applied for the correct management of transaction execution.

To alleviate their complexity and allow replication protocols to focus on their native purpose of ensuring replica consistency, systems usually delegate local concurrency control to the DBMS with the appropriate isolation level (Di) for which the protocol has been conceived. Depending on the correctness criterion, systems require local DBMSs to provide different isolation levels. Thus, earlier systems and those requiring a high level of isolation (2PL & 2PC [50], Bernstein-Goodman [14], O2PL & 2PC [23], Bcast all, Bcast writes, Delayed bcast writes and Single bcast transactions [2], Fast Refresh Df-Im and Fast Refresh Im-Im [97], DBSM [101], Pronto [102], DBSM-RAC [121], Epidemic restricted and Epidemic unrestricted [58], DBSM* [141], DBSM-RO-opt and DBSM-RO-cons [96], One-at-a-time and Many-at-a-time [116], MPF and MCF [140], BaseCON for 1SR, BaseCON for SC and BaseCON for strong 1SR [139]) rely on the serializable isolation level (Di3) of their underlying databases, which adequately serializes transactions executing at that server. Other systems relax their correctness criteria or are able to increase the locally provided guarantees, and thus also relax the isolation level of their local databases. Snapshot (Di2) isolation (Lazy Txn Reordering [104], SI and Hybrid [69], NODO and REORDERING [98], RSI-PC [107], SRCA and SRCA-Rep [79], PCSI Distr. Cert. [40], Tashkent-MW and Tashkent-API [41], Alg-Weak-SI, Alg-Str.-SI and Alg-Str.Ses.-SI [36], *k*-bound GSI [6], Tashkent+ [42], Mid-Rep [65], SIRC [114], Serrano et al. [118], AKARA [32], *g*B-SIRC [115]) or, more rarely, read committed (Di1) isolation (RSI-PC [107], SIRC [114], *g*B-SIRC [115]) are requested by such systems.

Among those systems not specifying a concrete level of isolation (Di0), two of them (Alsberg-Day [4] and RJDBC [44]) are based on the local concurrency control of their DBMSs and may function with different isolation levels at their local

databases. The rest of the systems with a Di0 strategy perform concurrency control at the protocol layer and therefore they do not require any specific underlying isolation. This is the case of OTP-99 [70], which uses a queue per conflict class and allows transactions to proceed when they are at the head position of their queue. OTP, OTP-Q, OTP-DQ and OTP-SQ protocols [71] follow a similar approach but, in this case, there is a queue per data item and so transactions are not restricted to access only one conflict class.

Finally, there are few systems that require some customization (Di4) of their underlying databases. Thus, BTO & 2PC [12] and OPT & 2PC [119] require the maintenance of read and write timestamps for each data item; and SER, CS, Hybrid [69] and WCRQ [110] delay the acquisition of write locks until the remote phase of transactions.

Regarding group-life (Gl) communications, as linear interaction is costly, only few systems make such synchronization. While most of the systems follow a Gl0 strategy (no communication during local transaction execution), systems such as 2PL & 2PC [50], BTO & 2PC [12], Bernstein-Goodman [14], OPT & 2PC [119], O2PL & 2PC [23], Epidemic unrestricted [58] or the protocol by Serrano et al. [118] are obliged to use linear interaction due to their partial replication and the consequent distributed nature of their transactions, which may potentially require to access data items at other nodes. Bcast all, Bcast writes [2] and RJDBC [44] execute their significant operations in an active mode, sending a message for each of such operations to all servers (Gl3). Finally, Fast Refresh Im-Im [97] uses a Gl2 strategy to immediately propagate updates to secondary copies in order to increase their freshness.

Regarding the group-end (Ge) policy, as most of the systems do not apply read-only transactions at remote nodes, they neither broadcast them to the group upon commit request (Ge0). However, in some cases, read-only transactions require a synchronization point. Two of the surveyed systems are able to identify read-only transactions and manage them differently from update transactions while they still require certain synchronization at group-end for queries. This is the case of WCRQ [110], which sends queries to a read-quorum of replicas (Ge2) in order to provide those queries with strong consistency. The algorithm by Serrano et al. [118] also applies a Ge2 strategy for queries, in order to commit the distributed transaction at all participating sites.

In order to ensure replica consistency, a synchronization is always needed for

update transactions, either at the beginning, at the end or after the execution of the transaction in its delegate node. OTP-99 [70], OTP, OTP-Q, OTP-DQ and OTP-SQ [71], the active processing of AKARA [32], BaseCON for 1SR, BaseCON for SC and BaseCON for strong 1SR [139] make the synchronization point of update transactions at the beginning (either with the client-request or the group-start policies), thus rendering unnecessary to synchronize with a group-end strategy (Ge0). Systems such as Fast Refresh Df-Im [97], RSI-PC [107], Alg-Weak-SI, Alg-Str.-SI and Alg-Str.Ses.-SI [36] choose a lazy synchronization after transaction commitment, and thus they also follow a Ge0 strategy.

Apart from the systems synchronizing update transactions at the beginning or after the commitment in the delegate, the rest of the systems make such synchronization at the end of the transaction in the delegate server, i.e., before the final commit operation, with a non-null group-end strategy. Alsberg-Day [4], which makes an update propagation in cascade mode, and the Tashkent family (Tashkent-MW and Tashkent-API [41], Tashkent+ [42]), which sends the write-set to a central certifier, follow the Ge1 strategy that requires the communication with only one server or component in the system. Among the remaining surveyed systems, some of them, based on partial replication, need to send transaction information only to a subset of system nodes (Ge2). This is the case of 2PL & 2PC [50], BTO & 2PC [12], Bernstein-Goodman [14], OPT & 2PC [119], O2PL & 2PC [23], Fast Refresh Im-Im [97], Epidemic restricted and Epidemic unrestricted [58]. The rest of the systems follow a Ge3 strategy, where the transaction information is broadcast to all nodes of the system.

In order to agree on the outcome of a broadcast transaction,[8] systems run the decision process. Weak voting, where a single node decides (Td1) and later communicates it decision to the rest of servers, as well as certification, where all nodes deterministically reach the same decision (Td2) are the preferred strategies. Only three of the surveyed systems base their decisions on the agreement of a quorum (Td3): One-at-a-time and Many-at-a-time [116], and WCRQ [110]. In One-at-a-time and Many-at-a-time, partial replication is used and nodes store information only about transactions that access items replicated at that node. To perform the decision process, nodes vote and then safely decide the outcome of a transaction T once they have received the votes of a voting quorum of T. In WCRQ [110], a read-quorum decides the outcome of a read-only transaction in order to ensure

[8] Those systems that locally commit queries without any communication with the rest of the nodes usually employ the bottom strategy (Td0) for such read-only transactions: no decision process is run for them.

strong consistency, while write-quorums decide the commitment of update transactions. Finally, there are systems that base their decisions on the agreement of all servers (Td4): 2PL & 2PC [50], BTO & 2PC [12], Bernstein-Goodman [14], OPT & 2PC [119] and O2PL & 2PC [23], which all use the 2PC protocol; and DBSM-RAC [121], which employs a non-blocking atomic commit protocol.

The transaction-remote (Tr) policy defines the way transactions are applied at remote nodes. Most of the systems identify read-only transactions and do not apply them at remote servers (Tr0). The only exceptions are: Alsberg-Day [4], which is not specially tailored for database replication and thus it does not identify queries; Bcast all [2], where all operations are broadcast and executed in all the servers; Lazy Txn Reordering [104], where all transactions are broadcast and possibly reordered to minimize abortions; Pronto [102], which assimilates to an active approach by sending to the backups the SQL sentences instead of the writeset; RJDBC [44], where all significant operations (including the commit operation) are broadcast to all replicas; and AKARA [32], which broadcasts all transactions at their starting point. In all these systems, no different treatment is given to read-only transactions. On the other hand, the rest of the surveyed systems do not execute queries at remote nodes, but only update transactions. In order to increase performance, systems usually apply remote transactions in a concurrent manner (Tr1), by controlling, either at the protocol level or inside the database, that conflicting transactions are applied in the same order at all replicas. However, to avoid the possible increase in complexity of such control, many systems apply writesets in a sequential, non-overlapping manner (Tr2).

With regard to the client-response (Cr) policy, only one of the surveyed systems, Pronto [102], returns multiple responses to the client (Cr2), whereas the rest of the systems always opt for returning a single answer (Cr1). The Cr2 client-response policy may require further processing in the client to select or compute a final result if multiple *different* answers are sent.

Group-after (Ga) policies can seriously affect consistency, in that update propagation outside the scope of transactions may lead to inconsistent states in different replicas. Thus, when using lazy propagation special care must be taken to ensure that consistency is maintained or that some reconciliation mechanisms are able to restore the system to a consistent state. Only few of the surveyed systems follow a non-null group-after strategy: Alsberg-Day [4], Fast Refresh Df-Im [97], RSI-PC [107], Alg-Weak-SI, Alg-Str.-SI and Alg-Str.Ses.-SI [36]. All these systems consider a primary copy configuration, where updates are made at only one

node (the primary copy of the system or the master site of the updated data) and are later lazily propagated to the slaves or secondary nodes. As only one node processes updates, no inconsistencies are introduced.

5.5 Conclusions

This chapter offers a historical survey of database replication systems. While many different strategies have been followed in order to accomplish the required system interactions, some of them seem to have been preferred over others, due to several reasons, from performance issues, to easiness of protocol design and implementation. In particular, snapshot isolation has been broadly used for local concurrency control, and many protocols based the transaction termination on atomic broadcast. On the other hand, it is also observable the necessity to offer several modes of execution for user transactions, both by providing different correctness criteria at the same time [6, 65, 107, 114, 115] and by allowing the coexistence of passive and active processing [32]. In the next chapter, we further explore this with the proposal of a metaprotocol for database replication adaptability.

Chapter 6

MeDRA, a Metaprotocol for Database Replication Adaptability

In this chapter we present MeDRA, a middleware metaprotocol developed for providing high adaptability to database replication systems. It faces the problems of dynamism and heterogeneity and features two levels of adaptability.

6.1 Introduction

Many replication protocols have been designed –being the survey in Chapter 5 a proof– and studied, proving that different protocols provide different features (isolation, consistency guarantees, scalability, etc.) and obtain different performance results depending on the environment characteristics (workloads, network latencies, access patterns, etc.). Protocols performance was compared in studies such as the one by Jiménez-Peris et al. [62], which presented the ROWAA approach as the most suitable for the general case, and the one by Wiesmann and Schiper [136], based on total order broadcast. For a particular scenario, i.e., for a particular combination of system environment and running applications with certain workload, access pattern, and isolation and correctness requirements, a specific protocol can be chosen as the most suitable.

The common approach followed by designers of database replication systems is to analyze *once* the general case of their scenario and accordingly elect a *single* replication protocol that will *permanently* remain in their systems. Such combination of a single analysis and a permanent decision over a unique protocol constitutes an approach that is perfect if the scenario is static and homogeneous.

Unfortunately, systems and applications are dynamic and heterogeneous: environments evolve changing its characteristics, which can drastically decrease the performance of the elected protocol; applications undergo updates which may modify or increase the requirements of the application; and multiple heterogeneous applications or procedures can concurrently access the same database. As a result, the initially chosen protocol of the common approach remains in the system while the environment evolves, degrading its performance or even being incapable of meeting new client requirements. This way, when dealing with dynamic scenarios or when concurrent client applications have different requirements (e.g., they work with different isolation levels), a more flexible and adaptable solution is needed.

It is worth noting that we are not specifically addressing mobile systems when talking about dynamic environments, but any replicated system that experiments changes in its main characteristics (e.g., networks with sustained intervals of high load due to massive client requests at some fixed times –like activities related to signing in and out at start and end of the working day–, or periodical administrative procedures that change the standard access patterns).

A sign that this adaptability is demanded by real applications is represented by Microsoft SQL Server. It supports, since version 2005, two concurrency controls simultaneously: one optimistic, based on *row versioning* [131]; and one pessimistic, based on locks. Thus, it concurrently supports different isolation levels –mainly, snapshot isolation and serializable, respectively. Heterogeneous client applications that could benefit from this kind of system while working with the same data would be, for example, a warehouse management application that requires strong consistency in order to know the exact amount of stored items –valuable or perishable goods–, working together with a small querying application used from a shop to consult about the stock of an item without the need to get the exact number.

However, no academic result had yet raised this support to middleware level for database replication systems by enabling several replication protocols to work

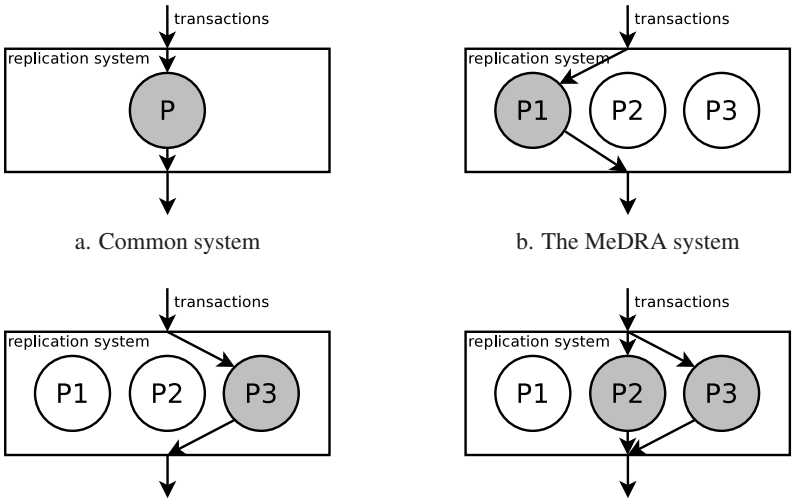
concurrently. To meet this adaptability requirement, we have designed MeDRA, a middleware metaprotocol that supports the concurrency of a set of consistency protocols that follow different replication techniques and may provide different isolation levels. With our metaprotocol, replication protocols can work concurrently with the same data or be sequenced to adapt to dynamic environments. As a result, each concurrent application can take advantage of the protocol that better suits its needs: e.g., an application with long transactions will prefer pessimistic replication to ensure their commitment (as the longer the transaction, the higher the probability of abortion in optimistic techniques due to conflicts with concurrent transactions). Additionally, each protocol can be replaced if the application access pattern is drastically modified or if the overall system performance changes due to specific load variations or network or infrastructure migrations.

Therefore, MeDRA faces the identified problems of dynamism and heterogeneity and features two levels of adaptability. On one hand, forward adaptability allows the system to adapt when dynamic conditions are present in the environment or in the clients themselves. MeDRA allows the exchange of protocols on-the-fly, without halting the processing, in order to always use the protocol that best behaves in the current environment or best fits current requirements. On the other hand, to face heterogeneity and deal with different (even opposite) concurrent requirements, MeDRA provides outward adaptability, which allows the system to run concurrent protocols to meet all client requirements as much as possible.

On the contrary to common database replication systems, where a single protocol manages all transactions all the time, MeDRA hosts multiple replication protocols that can be activated and deactivated in order to provide forward and outward adaptability. Thus, a single or multiple protocols can be activated at the same time, managing transactions as required (see Figure 6.1).

These activations and deactivations can be forced by an administrator following theoretical and empirical studies or they can be automatically triggered according to background performance analysis (response times, abort rates...) executed by a monitoring system. Also it is possible that a given application knows for sure which protocol is more suitable for its needs or that a given transaction access pattern is best served with a certain protocol.

In this chapter we present the design of the metaprotocol, whose overall architecture is depicted in Figure 6.2, and an implementation prototype, used to provide an experimental evaluation. To measure its overhead, we analyze the differences



a. Common system

b. The MeDRA system

c. MeDRA: forward adaptability

d. MeDRA: outward adaptability

Figure 6.1: Activated protocols in database replication systems. While common systems feature a single replication protocol, MeDRA hosts multiple protocols that can be activated and deactivated conveniently. When it is detected that the current protocol is no longer the best, forward adaptability allows the system to deactivate such a protocol and to activate another one. Facing heterogeneity, outward adaptability enables the system to simultaneously activate several protocols.

in performance when comparing the metaprotocol with stand-alone versions of the supported protocols. As we will demonstrate later, our metaprotocol introduces a very low overhead, thus providing a good performance, comparable with that of the stand-alone versions. Another important aspect to measure is the potential performance penalty of every possible combination of protocols when working concurrently. Our tests prove that some combinations have an excellent performance while others clearly degrade as load or system size increases. This will be easily explained considering the behavior of the targeted protocols.

The rest of this chapter is structured as follows. Section 6.2 summarizes the main aspects of the metaprotocol. Section 6.3 presents the experimental results and, finally, Section 6.4 discusses some related work.

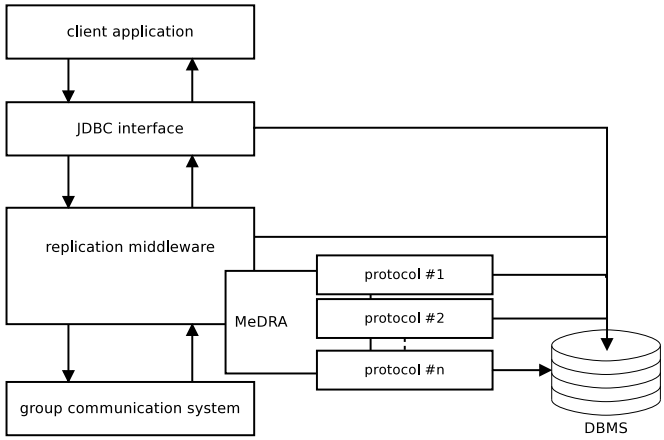


Figure 6.2: Overall architecture of the MeDRA replication system

6.2 Metaprotocol

The metaprotocol function is to support multiple replication protocols concurrently at each replica, properly managing their interaction, i.e., the dependencies between them. It also allows a working protocol to be exchanged when it is detected that another one would fit better (a common situation in dynamic environments). This protocol exchange is seamlessly performed: already started transactions end their execution using the protocol they started with, while new ones use the new protocol. Thus, processing does not need to be halted.

6.2.1 Supported Protocols

When two or more protocols are activated at the same time, they execute in concurrency inside the replica. For this concurrency to be feasible or, at least, practical, some common characteristics are needed. For this reason, the three replication protocols that have been tested with our metaprotocol enforce the same replica consistency level: natively sequential (#S) replica consistency. As seen before in this thesis, this level is very common in database replication systems and is easily achieved upon the guarantees provided by a FIFO total order broadcast used as the communication primitive to share transaction information among nodes. Thus, each targeted protocol is a representative of three protocol families

based on such a kind of broadcast [136]: active, certification-based and weak voting replication. All these families are update-everywhere [138] (to send its request, a client chooses one server, which is known as the delegate server), so they are decentralized replication protocols. On the other hand, each replication protocol may provide different transaction isolation levels, which can be exploited by different client applications.

When different replication techniques work concurrently in the same replicated database, some concurrency issues must be handled with special care in order to ensure their individual correct execution. Thus, first of all it is interesting to understand how the targeted replication protocols work. Moreover, it is important to determine the metadata they need because, when two or more protocols work concurrently, the metaprotocol must generate, for each executed transaction in the system, the metadata needed by each executing protocol. We recall next the description of the targeted protocols.

In *active* replication, the client request is broadcast in total order.[1] Later, all replicas execute and commit the transaction in the order it was delivered. This way, active transactions do not have local phase nor they need any decision process. Due to the sequential, non-overlapping execution of transactions, this replication model does not introduce any additional abortion with regard to a stand-alone system[2] and it can provide any isolation level supported by the local DBMS.

In *certification-based* replication, transactions are first locally executed in their delegate server and, when they request commitment, their writesets and, in some isolation levels, also their readsets, are broadcast in total order to all replicas. After delivery, a deterministic certification process, based on conflicts with concurrent transactions, starts in all replicas to determine if such a transaction can commit or not. The total order established by the broadcast determines the certification result: in case of a pair of conflicting transactions, the newest inside the total order is the one that is aborted. The transactions that commit, do it in the order established by the broadcast, being sent to the database in a non-overlapping way. The transaction information that must be broadcast will depend on the guaranteed isolation level. The writeset is enough for achieving snapshot isolation; for serializability, also the readset must be sent. Some additional metadata is also needed in order to complete the certification; e.g., the transaction start logical timestamp in case of using the snapshot isolation level.

[1] The client usually addresses its request to only one server which immediately broadcasts it.

[2] No system is free of abortions: they may occur due to disk failures, database deadlocks, etc.

In *weak voting* replication, transactions are also locally executed and then their writesets are broadcast in total order. But in this case, upon delivering this message, only the delegate (since readsets are never broadcast in this kind of protocols) is able to validate a transaction: if concurrent conflicting transactions have been committed before (i.e., their writesets have been delivered before), the transaction being analyzed should be aborted. Based on this information, the delegate server does a new broadcast (reliable but without total order) reporting its decision about the outcome of the transaction to all the replicas. It must be noticed that with this technique, broadcasting just the transaction writeset already allows 1SR′, since delegate replicas are able to check for conflicts between the readsets of their own transactions and the writesets of remote transactions.

The transaction metadata needed when all protocols are working is compound by the writeset and a timestamp of the transaction begin. With this information, certification-based and weak voting techniques can provide snapshot isolation. Additionally, if the underlying DBMS features a serializable concurrency control, weak voting techniques provide 1SR′ at no extra cost. On the contrary, certification-based techniques need to collect and broadcast readsets to provide the same correctness criterion. As readset management is costly, certification-based replication is normally used only for snapshot isolation. Thus, we do not consider readsets and work upon an underlying database system that provides snapshot isolation.

In terms of policies, following the model of Chapter 4, the targeted protocols can be described as the combinations of strategies depicted in Table 6.1.

Each of the targeted protocols presents advantages and disadvantages. Active replication is pessimistic and forces all replicas to completely execute every transaction, which increases the system load but ensures that no transaction ever aborts. This is very useful for long transactions that otherwise will have a high probability of being aborted due to conflicts with concurrent transactions. The other two techniques are both optimistic. Weak voting techniques may provide 1SR′ without the need to work with readsets, but they require an extra broadcast that forces non-delegate replicas to wait. Certification-based replication achieves fast certification of transactions, but it is not practical for isolation levels stronger than snapshot, due to readset management. The metaprotocol adaptability allows the system to switch to the most appropriate protocol at any moment to exploit all the advantages, while trying to overcome the disadvantages.

Table 6.1: Policies in MeDRA protocols

	Active	Certification-based	Weak voting
Client request	Cq4-t	Cq1	Cq1
Transaction service	Ts2	Ts0	Ts0
Group start	Gs0	Gs0	Gs0
Database replication	Dr2	Dr2	Dr2
Database isolation	Di2	Di2	Di2
Group life	Gl0	Gl0	Gl0
Group end	Ge0	Ge3-t	Ge3-t
Transaction decision	Td0	Td2-w	Td1-w
Transaction remote	Tr2	Tr2	Tr2
Client response	Cr1	Cr1	Cr1
Group after	Ga0	Ga0	Ga0
Correctness criterion	1SI	1SI	1SI

Once the protocols have been described in behavior and metadata, the interaction between them can be detailed. The meeting point where these replication techniques work concurrently is a pair of shared lists. A transaction is included in both lists when it is delivered by the GCS and, depending on the protocol class, it has not been rejected during the validation phase. Transactions are processed following the list order. Some dependencies may arise due to the behavior of each replication technique.

6.2.2 Metaprotocol Outline

Two shared lists are maintained by the metaprotocol in each replica: the *log* list, with the history of all the system transactions;[3] and the *tocommit* list, with the transactions pending to commit in the underlying database. These lists, initially empty, contain transactions from all the protocols working at the moment. Each transaction has an associated type, which represents its current status and, thus, is modified during the transaction lifetime. Possible types, as summarized in Table 6.2, are the following: (a) *resolved*, a committable (or already committed) transaction with writeset information available; (b) *c-pending*, a transaction with

[3]This list, only needed by the certification-based technique, can be purged as suggested by Wiesmann and Schiper [136].

I. Propagate message M_n related to transaction T_i:
 1. broadcast $M_n\langle T_i\rangle$
II. Upon delivery of M_n related to transaction T_i:
 1. call $T_i.protocol$ to $process(M_n\langle T_i\rangle)$
III. Committing thread:
 1. $T_i \leftarrow head(tocommit)$
 2. if $T_i.committable =$ true then
  a. if $T_i.delegate \neq R_k$ then
   i. call $T_i.protocol$ to $apply(T_i)$
  b. call $T_i.protocol$ to $commit(T_i)$
  c. $L\text{-}TOI \leftarrow T_i.toi$
  d. if $T_i.log\_entry\_type =$ w-pending then
   i. $T_i.log\_entry\_type \leftarrow$ resolved
   ii. $resolve\_w\text{-}dependencies(T_i)$
  e. if $T_i.log\_entry\_type =$ c-pending then
   i. emit vote for T_i
   ii. if $T_i.outcome =$ commit then
    $T_i.log\_entry\_type \leftarrow$ resolved
   iii. else
    delete T_i from log
   iv. $resolve\_c\text{-}dependencies(T_i, T_i.outcome)$
  f. delete T_i from $tocommit$

Figure 6.3: Metaprotocol algorithm at replica R_k

writeset information available but not yet committable (e.g., a weak voting transaction waiting for its voting message); and (c) *w-pending*, a transaction with no writeset information available (i.e., an active transaction not yet committed).

Table 6.2: Log entry types in MeDRA

Type	Description
resolved	a committable (or already committed) transaction with writeset info available
c-pending	a transaction with writeset info available but not yet committable
w-pending	a transaction with no writeset info available

A brief outline of the major steps of the metaprotocol at one replica, R_k, is depicted in Figure 6.3. Roughly speaking, protocols send messages in step I, which are processed in step II depending on the message type (see Table 6.3 for a summary of the different types of messages). The metaprotocol delegates the processing of messages on the activated protocols (Figure 6.4), which have access

a. Active
process($M_n\langle T_i\rangle$): *(of type A)*
a. $T_i.toi \leftarrow$ *N-TOI++*
b. $T_i.log\_entry\_type \leftarrow$ w-pending
c. $T_i.committable \leftarrow$ true
d. append to *log* and *tocommit*

b. Weak voting
process($M_n\langle T_i\rangle$):
1. if M_n contains a writeset then *(type WV-1)*
a. $T_i.toi \leftarrow$ *N-TOI++*
b. $T_i.log\_entry\_type \leftarrow$ c-pending
c. if $T_i.delegate = R_k$ then
i. $T_i.committable \leftarrow$ true
d. else
i. $T_i.committable \leftarrow$ false
e. append to *log* and *tocommit*
2. else *(M_n is a voting message, WV-2)*
a. if $M_n.vote =$ commit then
i. $T_i.committable \leftarrow$ true
ii. $T_i.log\_entry\_type \leftarrow$ resolved
b. if $M_n.vote =$ abort then
i. delete T_i from *log* and *tocommit*
c. *resolve_c-dependencies*($T_i, M_n.vote$)

c. Certification-based
process($M_n\langle T_i\rangle$): *(of type C)*
a. $T_i.toi \leftarrow$ *N-TOI++*
b. *certificate*(T_i)
(check conflicts with concurrent transactions)
c. if *certification* = negative then
(if T_i conflicts with a resolved)
i. if $T_i.delegate = R_k$ then
(transaction is local)
rollback(T_i)
ii. else
discard T_i
d. else
i. if *certification* = positive then
(no conflicts and $\nexists$ w-pending)
$T_i.log\_entry\_type \leftarrow$ resolved
$T_i.committable \leftarrow$ true
ii. if *certification* = pending then
(conflicts with c-pending or $\exists$ w-pending)
$T_i.log\_entry\_type \leftarrow$ c-pending
$T_i.committable \leftarrow$ false
iii. append to *log* and *tocommit*

Figure 6.4: Protocol modules for message processing at replica R_k

to all shared variables. These include the *log* and *tocommit* lists and two integer counters: *L-TOI* and *N-TOI*. Based on their delivery order, transactions are assigned a unique TOI, or total order index. Variable *N-TOI*, initialized to 1, stores the index for the next transaction to be delivered. Variable *L-TOI*, initialized to 0, stores the index of the last committed transaction in the replica. These counters provide logical timestamps for transactions.

Active transactions (Figure 6.4a) do not need a decision phase: all commit in the order established by the broadcast. Thus, when an active message arrives, the transaction is added to both lists and its entry is marked as w-pending and committable. As its writeset will be known only after commitment, it prevents subsequent certifications from being completed, as writesets are needed to determine if transactions present write conflicts. Transaction dependencies will be explained in detail later.

Table 6.3: Message types in MeDRA

Type	Description
A	the whole transaction of an active protocol
C	the writeset of a transaction executed in a certification-based protocol
WV-1	the writeset of a transaction executed in a weak voting protocol
WV-2	the voting message of a weak voting protocol

Messages from the certification-based protocol (Figure 6.4c) contain a writeset that must be certified. This certification is based on two metadata integers representing the transaction start and end, respectively: *bot*, begin of transaction, only needed for certification-based transactions and set before broadcasting to the current value of *L-TOI*,[4] and *toi*, the total order index set at reception. To certify a transaction T_i, it is checked against every transaction T_j of the *log* list such that $T_j.toi > T_i.bot$, i.e., T_i and T_j are concurrent. If the intersection of the writesets of T_i and T_j is non-empty, then a conflict exists. In this case, if T_j is a resolved transaction, then the certification of T_i is negative. With a negative result, transaction T_i is discarded at remote nodes and aborted at its delegate. Otherwise, it is added to both lists. Certification obtains a *pending* result if there are concurrent w-pending transactions or conflicting transactions whose certification/validation phase is incomplete, i.e., c-pending transactions. This creates a dependency between the transaction being certified, T_i, and each of the previously delivered transactions that create the indecision. In this case, T_i is also marked as c-pending and non-committable. Only when the certification result is positive (there are no conflicts with concurrent transactions and there are no w-pending transactions in the *tocommit* list), T_i is marked as resolved and committable.

When the delivered message contains a writeset from the weak voting protocol (Figure 6.4b), the transaction is added to both lists and marked as c-pending as its outcome is unknown until commit time in the delegate node (which marks the transaction as committable) or until the arrival of the voting message in the rest of nodes (which mark it as non-committable). In the pseudocode outline of Figure 6.3, the validation is based on the local concurrency control: waiting to

[4]The validity of *L-TOI* as logical timestamp for transaction start is ensured by a conflict detection mechanism developed by our group [93].

commit turn and trying to commit the transaction in the delegate. If the commitment succeeds, a positive vote is broadcast (reliably but without total order) to all replicas. Otherwise, a negative vote is sent.

The reception of a voting message for a weak voting transaction changes the transaction status and resolves the dependencies between this transaction and subsequent ones. If the vote is positive, the transaction is marked as resolved and committable. Otherwise, it is deleted from the lists.

The last major step of the algorithm, step III of Figure 6.3, consists in committing the first transaction in the *tocommit* list, provided that it is committable. This is performed sequentially by the metaprotocol, one transaction at a time, following the list order (provided by the total order broadcast). When applying remote writesets, conflicts with local transactions may arise. At this point, our conflict detection mechanism [93] aborts those local transactions allowing the correct remote writeset application. After commitment, active transactions obtain their writeset and mark their status as resolved. Possible dependencies are also resolved. In the case of weak voting replication, conflicts with previously committed transactions force the weak voting transaction to rollback, obtaining an *abort* outcome instead of a *commit* one. If aborted, the transaction is removed from the *log*. Otherwise, it is marked as resolved. In any case, this outcome is used to resolve dependencies and to emit the vote. Finally, the processed transaction is removed from the *tocommit* list.

As seen in the pseudocode outline, the common processing is carried out by the metaprotocol (control of data structures, sending and reception of messages, scheduling of transactions...), which calls the corresponding protocol when protocol-specific steps have to be taken (treatment of messages, commitment of transactions...). On the other hand, communication with client applications remains in the protocols, thus preserving previous client-protocol interfaces. This way, the system presents high modularity and protocols remain very simple and easy to maintain (required adaptations to work within the metaprotocol consist only in simplifications), while they are still able to introduce some optimizations in their specific methods (e.g., a pre-certification process prior to broadcast, which may save useless network communication and subsequent processing).

Please refer to Appendix B for a complete and detailed pseudocode listing of the current prototype implementation. Apart from containing procedures here

omitted for simplicity, it introduces an optimization for the weak voting protocol: a validation phase identical to the certification process, which is run at message delivery in the delegate node and allows an earlier voting in absence of dependencies. This validation requires to also assign a *bot* timestamp to weak voting transactions before broadcasting their writesets. A minor simplification is also included: a transaction T_i is committable depending on the value of T_i.*log_entry_type*, which is there called T_i.*status*. Moreover, the complete pseudocode features a garbage collection mechanism: once a writeset is applied in all replicas, all local conflicting transactions have been aborted, so such a writeset can be removed from the *log* list as it will not be needed any more [136]. Finally, regarding to the protocol exchange possibilities, the complete pseudocode includes the required functionality for another system component (such as a load monitor, an administrator, etc.) to enable and disable protocols in order to best fit the current system requirements.

6.2.3 Dependencies Between Protocols

Concurrency can lead to inefficient systems due to natural differences in the behavior of the protocols. Several dependencies may arise when certifying transactions in certification-based protocols, or when validating weak voting transactions in the delegate node, if such validation is performed in a way similar to the certification.

These dependencies, a natural and inevitable consequence of the concurrency between different replication techniques, imply some delays with regard to the original behavior of the stand-alone protocol. As this may have a notable impact on system performance, it is important to understand these dependencies and to carefully study their implications, as we will do in Section 6.3.

A transaction, in order to be certified (or validated by its delegate, if such validation follows a similar process), must know the writesets of all concurrent and previously delivered transactions that will eventually commit. However, this may not be immediately known, as there is some pending information in the entries of the *log* list (see Table 6.2). First, the writesets of w-pending transactions are unknown until their commit time. Second, the final termination (commit/abort) of c-pending transactions is not yet known (e.g., weak voting transactions waiting for their vote or certification-based transactions waiting for the resolution of any of the previous cases in order to complete their certification phase). This pending

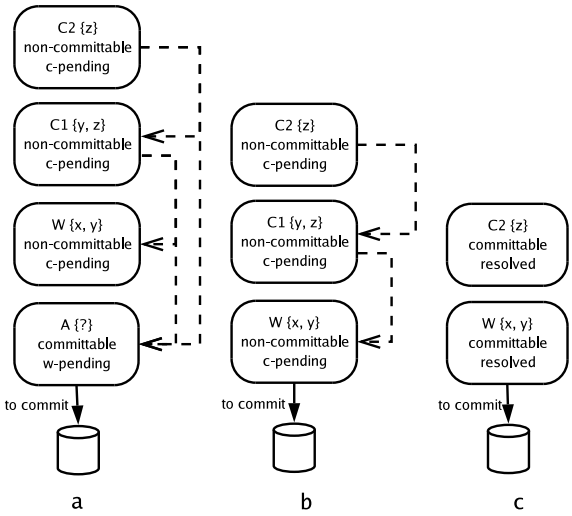


Figure 6.5: Dependencies between transactions

information prevents the certification/validation of a transaction T_i from finishing in two ways: (a) T_i cannot check for conflicts with a w-pending transaction T_w, as the writeset of T_w is not yet available (here we say that T_i has a w-dependency with T_w); and (b) although the writeset of a c-pending transaction T_c is known and thus T_i can check for conflicts, the final outcome for T_c is yet unknown due to a pending vote or another dependency (here we say that T_i has a c-dependency with T_c). Notice that a conflict should cause the abortion of T_i only if the final outcome of conflicting transaction T_c is a commit.

Note also that a transaction T_i may present several dependencies, i.e. depend on several previous transactions, and it will not be considered as resolved until all its dependencies are resolved. At that moment, the dependencies caused by T_i on following transactions will be resolved in cascade.

Let us consider an example situation to review the steps of the metaprotocol. Suppose that all three protocols are running, so transactions from all of them are delivered at replica R_k. Suppose also that, at a given moment, the *tocommit* list is empty. At this moment, an active transaction A is delivered. A is directly appended to the lists, marked as committable and w-pending. The commitment of A begins. Then, a weak voting transaction W, writing objects x and y, is delivered. Suppose R_k is not its delegate replica. Thus, W is added to the lists, marked

as c-pending and non-committable until its vote arrives. A new transaction is delivered: C1, a certification-based transaction that writes objects *y* and *z*. C1 obtains a *pending* result in its certification, as there is a concurrent w-pending transaction (A) and C1 presents write conflicts with W, which is c-pending (thus, C1 has two dependencies). This pending certification forces C1 to be marked as c-pending and, thus, non-committable until both dependencies are resolved. Later, another certification-based transaction C2 is delivered. C2, which writes object *z*, is also marked as c-pending and non-committable because of the conflict with C1 and the existence of A. The current stage corresponds to Figure 6.5a (where dashed arrows represent dependencies). At this moment, A finally ends its commit operation and its writeset is collected: it wrote objects *u* and *v*. Now it is time to resolve the w-dependencies of C1 and C2. As A does not conflict with them, both w-dependencies are just removed. Figure 6.5b represents the current situation. Now, the voting message for W is delivered with a commit vote. So W is now committable and some c-dependencies can be resolved. As W presented conflicts with C1 and W is going to commit, C1 must abort. Due to the termination of C1, more c-dependencies are resolved on cascade, thus removing all the dependencies presented by C2, that becomes committable. This stage is depicted in Figure 6.5c. Any committable transaction at the head position of the *tocommit* list is eventually committed.

6.3 Experimental Results

Naturally, there is a trade-off between the high level of adaptability provided by the metaprotocol and system performance. Differences in protocol behaviors cause dependencies when two or more replication techniques are executed in concurrency. When a transaction requires certain piece of information to proceed but this information is not yet available, a dependency is created and the transaction must wait. These waiting times may increase the completion time of transactions, reducing throughput. In order to assess the efficiency of the system, a metaprotocol prototype was implemented and a suite of tests were conducted to measure two aspects. (a) The overhead introduced by the metaprotocol management. To this end, stand-alone versions of the active, weak voting and certification-based replication protocols were implemented. Later, we run each protocol in both the stand-alone manner and as the only available protocol within the metaprotocol. Differences in performance between each of these two configurations will give a measure of the metaprotocol overhead. (b) The penalty in performance due

to protocol concurrency. All possible combinations of protocols were tested in concurrency within the metaprotocol. Measures were taken separately for each protocol, thus showing, e.g., the variations on completion time for certification-based transactions when executed in a pure certification-based system or when another protocol is also working and some dependencies arise.

System Model We assume a partially synchronous –clocks are not synchronized but the message transmission time is bounded– distributed system where each node holds a replica of the database, i.e., the database is fully replicated. For local transaction management each node has a local DBMS which provides the requested isolation level. On top of it, a database replication middleware system is deployed. This middleware uses a group communication service that provides a total order multicast.

Failures were not considered in the metaprotocol prototype. Our group already worked [37, 106, 112] on recovery protocols for certification-based replication, or, more generally, for a replication technique based on atomic broadcast and guaranteeing sequential consistency. As all the protocols supported by the prototype follow the same approach, any recovery protocol that works for this type of replication, e.g., any of our previously proposed solutions, could be easily adapted to work with the prototype. Indeed, the required state transfers will be similar, and transactions that were running in the crashed node will have aborted and restarted in other nodes. For this reason and for the sake of simplicity, no recovery protocol is described in this thesis.

Protocol implementation Both weak voting implementations (the one used within the metaprotocol and the stand-alone version) include the optimization of starting a validation phase in the delegate node at delivery time, similar to the certification phase of certification-based protocols.

Test Description To accomplish the analysis, we use Spread [122] as GCS and PostgreSQL [108] as underlying DBMS providing snapshot isolation by means of a multiversion concurrency control. Transactions access a database with a single table (the smaller the database, the greater the probability of conflicts and, thus, of dependencies) of 10 000 rows and two columns. The first column is the primary key; the second, an integer field subject to updates made by transactions. A

prior tuning process was performed on PostgreSQL, which showed that a reduced number of 10 connections for client transactions in each replica was the best for our environment. This relates with the work of Milán-Franco et al. [89] which proves that an even more reduced number of database connections is the best option in some environments.

Both the metaprotocol and the stand-alone protocols were tested in our replication middleware MADIS [59] with 2 and 4 nodes. Each node has an AMD Athlon™ 64 Processor at 2.0 GHz with 2 GB of RAM running Linux Fedora Core 5 with PostgreSQL 8.1.4 and Sun Java 1.5.0, and interconnected by a 1 Gbit/s Ethernet. Transactions are initiated at a fixed pace in each replica to obtain system input rates of 20, 40, 60 and 80 TPS (transactions per second). As the system load is the same in the case of 2 and 4 replicas, nodes should be less loaded in the 4-replica system. On the other hand, a greater number of nodes may affect communication. We thus study the influence of the number of replicas in the system.

As the number of database connections is reduced for performance, transactions are started at a certain rate but then they must wait to get a free connection. This waiting time is included in the transaction length. When a transaction obtains a connection, it updates 20 rows (a fixed number of items, as the protocol tasks do not depend on transaction length). These writes may cause conflicts between transactions, which will be detected during protocol validation or certification. Although this workload does not represent any standard benchmark, it was designed for the purpose of stressing the system gradually with higher and higher input rates of write-only transactions causing more and more conflicts and, therefore, dependencies. We discard read operations, which are only locally executed and have no conflicts. As a result, we subject the system to a worst-case environment. This ad-hoc benchmark is, indeed, an unfavorable test for any database replication system. Using a common benchmark would show better performance results, as read-only transactions would constitute an important part of the load.

Concurrency configurations Active (A), certification-based (C) and weak voting (W) techniques were tested in a stand-alone manner and within the metaprotocol. In the latter case, combinations of 1, 2 and 3 protocols were used (a total of 5 concurrency configurations for each technique, e.g., for active replication: Stand-alone A, A, AC, AW and ACW). Each protocol managed a proportional part

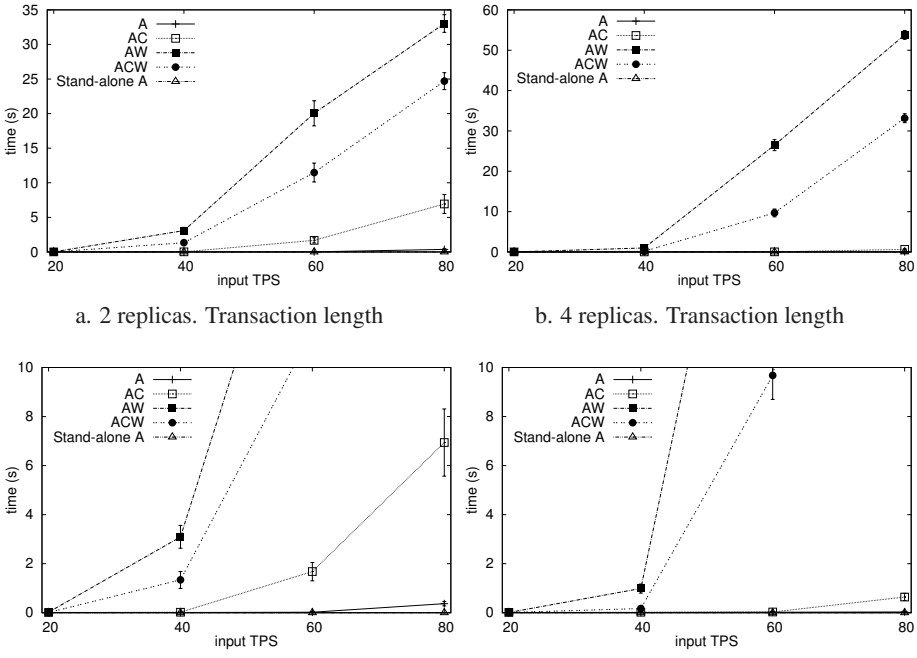
of the issued transactions,[5] which were analyzed separately to compute length and abortion rate.

Results The three aspects considered in our tests (length of committed transactions, length of aborted transactions and abortion rate) were computed for each protocol at each node in each test iteration. Each one of these iterations dismisses initial and final transient phases. A total of 2 000 local transactions were considered at each node (which gives a total of 4 000 values for each time measure in systems of 2 replicas, and a total of 8 000 values when 4 replicas are used). Each plotted result is the mean value obtained after 20 of the previously detailed iterations, and it is shown with its 95% confidence interval. Small confidence intervals prove that the presented results are statistically representative.

Results are presented separately for each protocol family and for each system size (2 or 4 replicas). To allow an easy comparison, we represent in the same graph the evolution of a certain measure in the 5 concurrency configurations previously commented. This way, for example, Figure 6.7b presents mean times for committed certification-based transactions, depending on the concurrency configuration and the input TPS, in a 4-replica system.

Protocols do not show important differences in response time when executed in a stand-alone manner or as the only protocol within the metaprotocol. Indeed, performance is virtually the same at low input rates or when using a system of 4 replicas. Only when 2 replicas must support a high input rate, differences appear. Thus, the main factor to consider is the penalty due to concurrency between different techniques. This concurrency is appropriate when multiple client applications access the same database in different ways. The active technique is pessimistic while the other two are optimistic. This different approach limits the performance: optimistic techniques are forced to wait for the processing of pessimistic transactions, thus reducing the advantages of their optimism. Indeed, as soon as there is one active transaction in the *tocommit* queue, all non-active subsequent transactions in the queue establish a dependency with it. This dependency lasts until the active transaction is committed. Moreover, weak voting replication is handicapped by the second broadcast needed to emit the vote: non-delegate replicas must wait for the delegate to validate the transaction and for the vote to

[5] When two protocols were combined, half of the total issued transactions were managed by each of them. Similarly, when all three protocols were concurrent, a third part of the amount of transactions was managed by each protocol.



a. 2 replicas. Transaction length

b. 4 replicas. Transaction length

c. 2 replicas. Transaction length (zoom)

d. 4 replicas. Transaction length (zoom)

Figure 6.6: Performance of active replication

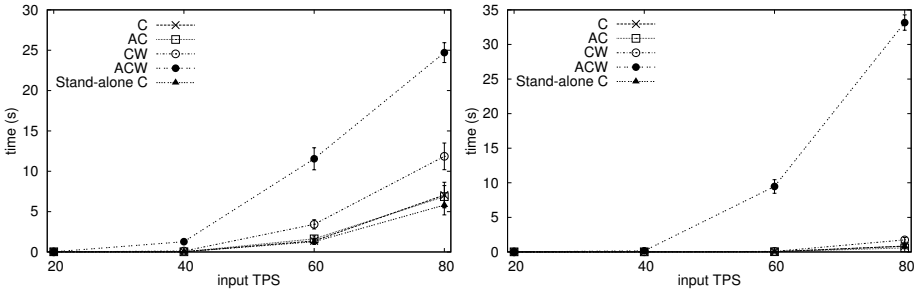
arrive. All these drawbacks join when mixing active and weak voting replication, leading to poorer performance in AW and ACW configurations.

Figure 6.6 corresponds to the active technique (recall that it never aborts transactions). Graphs do not show practical differences between stand-alone and A configurations in either tested system size. When combined with the optimistic certification-based technique (AC), the response time of active transactions increases with the system load in the case of 2 replicas, while it remains near zero when 4 replicas are used, showing a correct scalability for this configuration. On the other hand, configurations which combine active and weak voting replication (AW and ACW) not only do not improve their response time when increasing the number of replicas, but they even degrade their performance, especially with heavy workloads (i.e., with 60 and 80 TPS). This behavioral trend is observed in all presented graphs. The explanation is that active transactions hold dependencies until their commit time, thus preventing many subsequent transactions from

being validated or certified until all previous active transactions have committed. When these subsequent transactions are managed by weak voting replication (AW), this wait may be bearable in delegate nodes, but remember that the rest of nodes must wait until the delivery of the voting message. Let us suppose this time to be a certain amount x of milliseconds. When 2 replicas are used, x milliseconds are wasted per transaction. But when increasing the number of replicas to 4, we are also increasing the number of non-delegate nodes, which then multiplies the wasted time up to 3x for each transaction. In the worst case, all 3 non-delegate replicas may have stopped all commitments waiting for the resolution of the transaction in the head of the *tocommit* list. As the global wasted time is greater, the global completed work is lesser, thus degrading performance. When adding the certification-based technique to the configuration (ACW), resulting times are better, as certification-based transactions are not penalized by voting-waiting times and the dependencies they introduce last less than those of active transactions. Even though, this configuration also degrades when the number of replicas is increased.

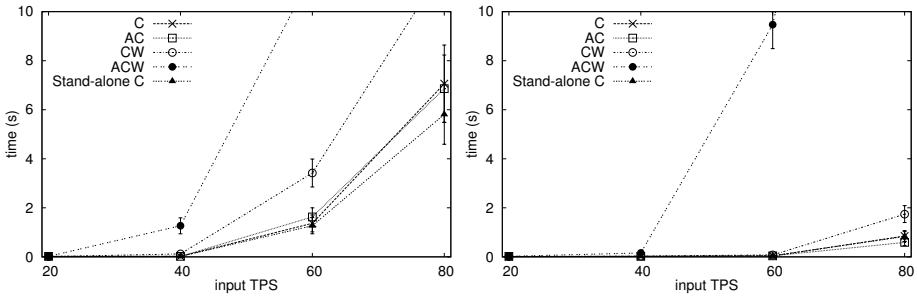
Results from certification-based techniques are presented in Figures 6.7 (committed transactions) and 6.8 (aborted transactions and abortion rate). Focusing on the 2-replica system, times for committed and aborted transactions follow similar curves. Stand-alone and C configurations obtain very similar measures; small differences appear only for a 80 TPS load, when the stand-alone version is better. Times for configuration AC are very close to those observed for active transactions in the same configuration, thus confirming that dependencies caused by active transactions are detrimental only when weak voting transactions are also involved. When combining certification-based and weak voting techniques (CW), dependencies suffered by weak voting transactions increase response times but, as these dependencies do not last as much as those caused by active transactions, the performance of CW is better than that of previous AW. Finally, we obtain the worst response time and abortion rate when mixing all three protocols. Regarding the abortion rate, the most interesting curve is that pertaining to the ACW configuration. As transactions last longer with this combination, more conflicts appear among them. Moreover, as the load grows, a higher level of concurrency is supported by the system, which involves a greater number of abortions. The last section of the curve presents a lesser slope because the system is already almost saturated at 60 TPS and concurrency slightly grows when introducing 80 TPS (see graphs of Figure 6.11).

Some behavioral trends observed in a 2-replica system are maximized in the case



a. 2 replicas. Length of committed transactions

b. 4 replicas. Length of committed transactions



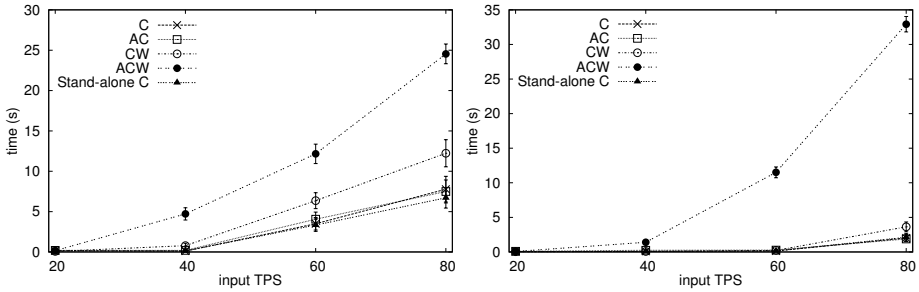
c. 2 replicas. Length of committed transactions (zoom)

d. 4 replicas. Length of committed transactions (zoom)

Figure 6.7: Performance of certification-based replication. Committed transactions

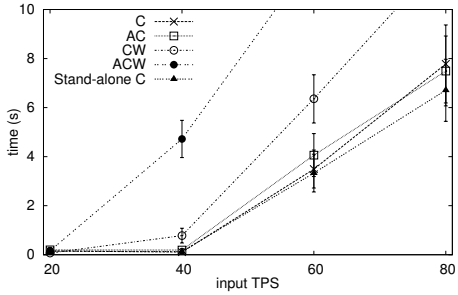
of 4 replicas. Configurations that performed well for 2 replicas have here an even better performance, as each node is less loaded. On the other hand, bad combinations are worsened by the greater number of replicas. Remember that the global time wasted in the weak voting protocol augments with the number of nodes, thus reducing the overall performance.

Graphs from Figures 6.9 and 6.10 show the results of the weak voting technique. In a 2-replica system, committed and aborted transactions follow similar trends, although it is the first time that a configuration using the metaprotocol –the CW combination– is clearly faster than the stand-alone version of the replication protocol. This reinforces the idea of the weak voting technique being penalized by a centralized process –the validation of the delegate– which forces non-delegate

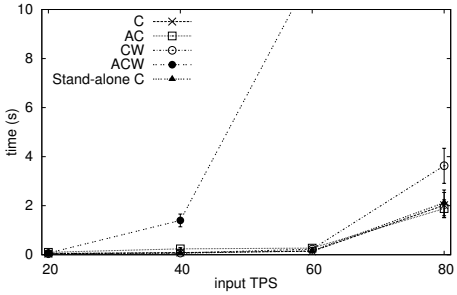


a. 2 replicas. Length of aborted transactions

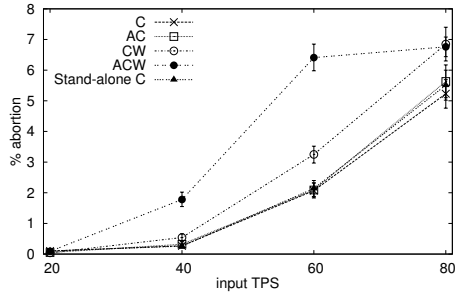
b. 4 replicas. Length of aborted transactions



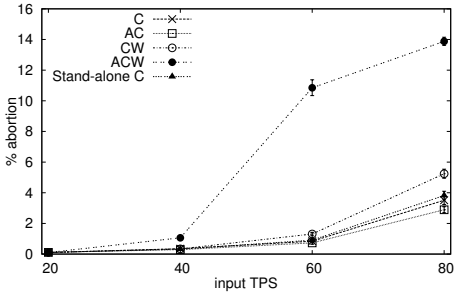
c. 2 replicas. Length of aborted transactions (zoom)



d. 4 replicas. Length of aborted transactions (zoom)



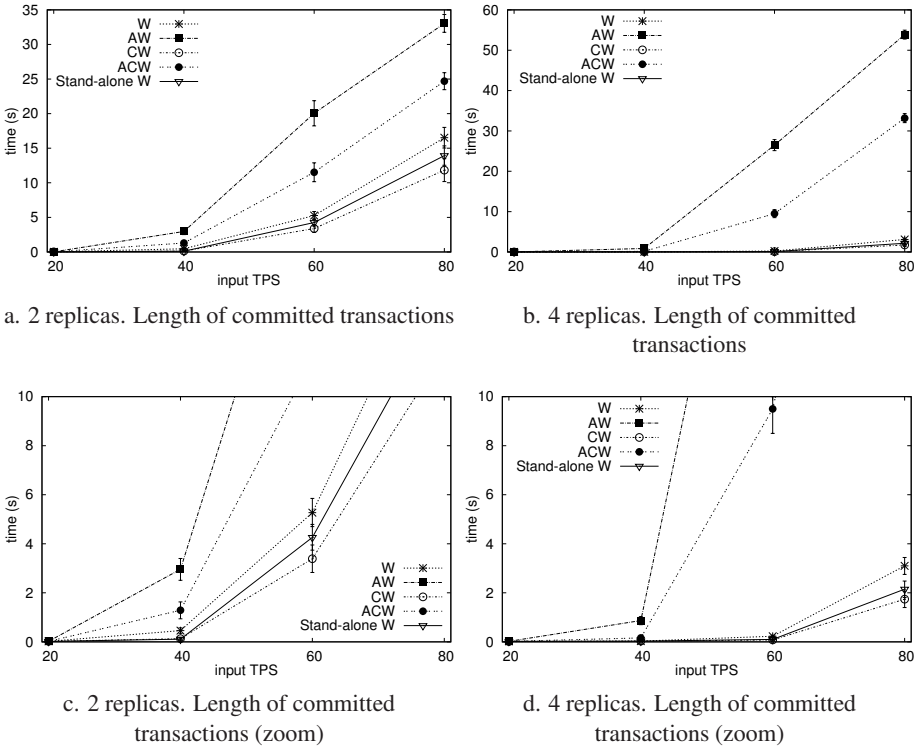
e. 2 replicas. Abortion rate



f. 4 replicas. Abortion rate

Figure 6.8: Performance of certification-based replication. Aborted transactions

nodes to wait for the second broadcast. On the other hand, the weak voting technique is an excellent option when readsets must be also considered for validation, as this technique avoids the collection and transmission of readsets.



a. 2 replicas. Length of committed transactions

b. 4 replicas. Length of committed transactions

c. 2 replicas. Length of committed transactions (zoom)

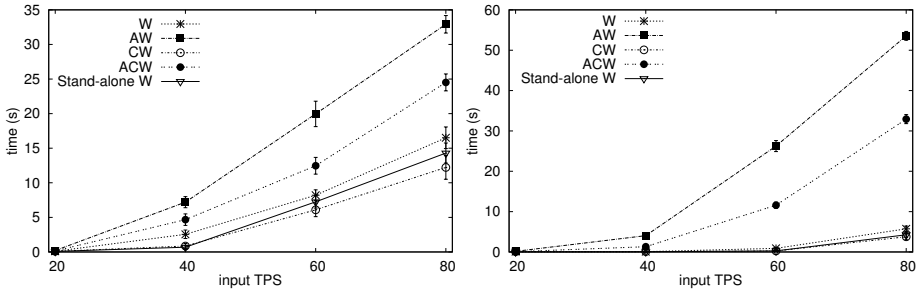
d. 4 replicas. Length of committed transactions (zoom)

Figure 6.9: Performance of weak voting replication. Committed transactions

As seen in the graphs, mixing certification-based transactions with weak voting ones lightens the system, achieving lower response times (certification-based transactions are not penalized by a second broadcast). On the other hand, as previously noticed, mixing active transactions with weak voting ones has undesirable consequences, which can be improved adding certification-based transactions to the combination (the dependencies will be less and, thus, also the system load).

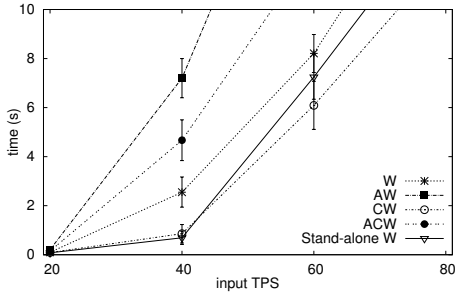
With regard to the abortion rate, curves corresponding to combinations AW and ACW stand out from the rest. The same as before, the longer the transactions, the greater the possibility of conflicts with concurrent transactions.

The behavior observed in the 2-replica system is again maximized in the case of 4 replicas and we clearly see how AW and ACW combinations obtain times and abortion rates dramatically higher than those of other configurations.

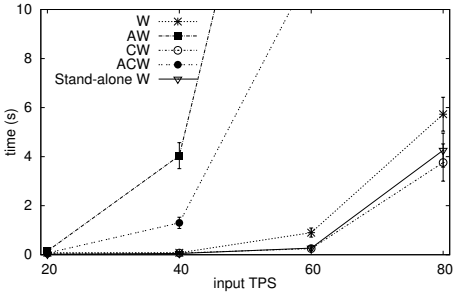


a. 2 replicas. Length of aborted transactions

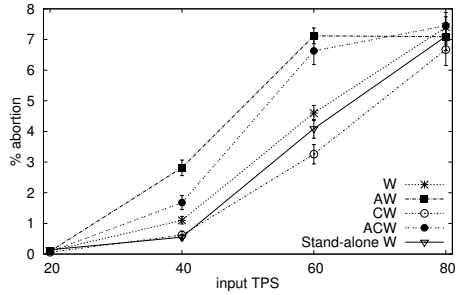
b. 4 replicas. Length of aborted transactions



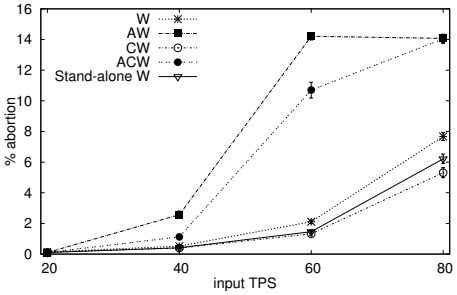
c. 2 replicas. Length of aborted transactions (zoom)



d. 4 replicas. Length of aborted transactions (zoom)



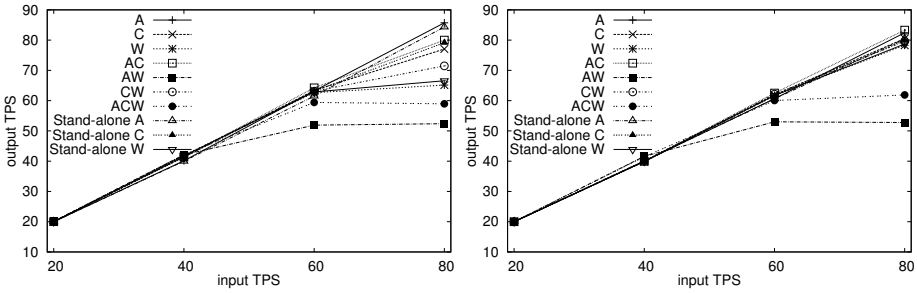
e. 2 replicas. Abortion rate



f. 4 replicas. Abortion rate

Figure 6.10: Performance of weak voting replication. Aborted transactions

Figure 6.11 shows the output TPS, i.e., the amount of committed transactions per second in the whole system. As already seen in previous graphs, all configurations are similar at low loads, but their curves diverge when more and more transactions are initiated per second. Again, configurations AW and ACW show



a. 2 replicas b. 4 replicas

Figure 6.11: Output TPS

the poorest performance, being the gap with the other curves greater when increasing the system size, as already explained. To sum up, it is clearly seen how some combinations have an excellent performance while others degrade as load or system size increases.

Final remarks The trade-off between adaptability and performance must be carefully analyzed in each system. In the system used for our tests, configurations that combine active and weak voting replication should be avoided if high loads need to be supported or large system sizes are used, as such protocol concurrency reduces system performance in these cases. Nevertheless, if performance is not as important as supporting concurrent client applications of different requirements, these combinations may be very useful. On the other hand, the CW combination has shown an excellent performance. This can be combined with the flexibility already offered at DBMS level by Microsoft SQL Server, which concurrently supports snapshot and serializable isolation levels. Thus, when the middleware is deployed on top of such a DBMS, our metaprotocol can provide a straightforward support for both isolation levels in replicated environments. To our knowledge, no other solution offers at a middleware layer such a degree of transparency and functionality.

6.4 Related Work

The metaprotocol presented here is a continuation of the work started by our group with MADIS [59], a platform designed to host a wide range of replication protocols. MADIS was designed to support different kinds of pluggable protocols, whose paradigms range from eager to lazy update propagation, from optimistic to pessimistic concurrency control, etc. Consequently, MADIS was thought to maintain a wide range of metadata in order to cover the most common database replication protocol requirements. Particularly, it was thought to switch from one consistency protocol to another, as needed, without the need to recalculate metadata for the newly plugged-in protocol. The resulting architecture allowed the administrator to change the used protocol, stopping first the old one and starting later the new one, as no concurrent execution was supported.

An exchanging algorithm for database replication protocols was presented by Castro-Company and Muñoz-Escoí [24], who designed a protocol for supporting a closed set of database replication protocols. The metaprotocol presented in this chapter follows this work, providing a seamless and fast protocol exchange and enhancing the modularity of the system.

Apart from the work of our group, no academic result presents a valid solution of a database replication system supporting concurrent replication protocols.

A different approach is to develop a single replication protocol that supports multiple isolation levels [10, 92, 114]. This was one of the aims in the GlobData project and some initial solutions were provided by Muñoz-Escoí et al. [92]. But the isolation levels defined in GlobData were not the ANSI standard ones, since GlobData was a system with an object-oriented interface able to provide an object-oriented replicated database using relational database replicas, and for those systems there were considered another set of behaviors. Bernabé-Gisbert et al. [10] propose a general scheme for designing middleware database replication protocols supporting multiple isolation levels. The authors based it on progressive simplifications of the validation rules used in the strictest isolation level being supported, and on local (to each replica) support for each isolation level in the underlying DBMS. Unfortunately, the resulting protocol is complex and it lacks the modularity and maintainability of our metaprotocol.

The idea of a single replication protocol providing a flexible behavior was also studied by Correia et al. [32], with the AKARA protocol. With it, a transaction

can be executed in an active or a passive manner, thus taking advantage of the best characteristics of each replication protocol. Unfortunately, AKARA needs the database to be partitioned in conflict classes and classifies each transaction regarding the accessed classes. This allows a straightforward conflict detection but forces to know the entire transaction before its execution, thus precluding interactive transactions. Moreover, conflict classes are usually entire tables, which leads to a coarse-grained conflict detection. On the other hand, our metaprotocol (using block detection mechanisms [93] developed in our group) provides a row-level conflict detection, which allows the execution of interactive transactions, as no partition in the database is needed. Finally, although AKARA performs better, it has to be noted that it uses the TPC-C benchmark where a new request is only triggered by the completion of the previous one, and read-only transactions are included in the load. Our ad-hoc benchmark is much more stressing for the system, as it involves only update transactions which execute over one single database table and are initiated at a fixed pace, independently of whether previous requests have been completed or not. This way, conflict probability increases, which also raises dependencies between transactions and, thus, completion times. Despite this, our system is still able to perform in an acceptable way.

Considering general distributed systems, i.e., not necessarily related to database replication, adaptability can also be provided by self-optimization. Taton et al. [128] propose a queue clustering solution for message oriented middlewares. A clustered queue is a set of queues each running on different servers and sharing clients. Its self-optimization fairly distributes client connections among the queues belonging to the clustered queue and dynamically adds and removes queues in the clustered queue depending on the load. As a result, the system uses the adequate number of queues at any time.

Communication protocols are another distributed system research branch that has also studied techniques for dynamically changing protocols. In this case, the goal is to select each time the group communication system which best fits with the network layer and changing load profile of the system. There are different approaches to provide this support. Mocito and Rodrigues [90] make a proposal in order to switch between total order broadcast communication protocols. Their switching system, instead of buffering messages or blocking their propagation, spreads each message using both protocols during the switching phase. This mechanism is less aggressive than others [82, 88] but presents some overload in the network. On the other hand, some more aggressive mechanisms [82, 88] benefit from the simplification of not caring about the existence of transactions.

The general problem of protocol exchange, or dynamic protocol update (abbr., DPU), has been broadly discussed [16, 27, 47, 113]. However, DPU solutions provide adaptability by replacing the working protocol by means of some type of synchronization between nodes, and do not consider protocol concurrency (except, perhaps, for a short transition phase).

Bhargava et al. [16] study the adaptability provided by RAID-V2. RAID-V2 is a distributed database system where three components have built-in adaptability features: the concurrency controller, the replication controller and the atomicity controller. Each of these elements implements several algorithms and offers the mechanism to convert from one algorithm to another. In that system, protocol replacement is based on a fully replicated relation, the control relation, which contains one row for each site and is updated by special control transactions. An update of a row in this table is interpreted by the corresponding server as a dynamic adaptability request, issuing a protocol replacement.

Fritzke et al. [47] also focus on DPU in the context of database replication. They provide adaptability to mobile systems, where disconnected nodes can work with weak consistency levels but switch to stronger levels when connected. This protocol replacement is based on uniform reliable multicast and uniform consensus.

Chen et al. [27] presented a general solution for DPU, where each layer of a distributed system can be adaptive and algorithms are changed in a three-step process: change detection, agreement and adaptive action. The changeover does not halt processing or message exchange but it requires communication with the protocol modules, which forces to extend them. To solve this, Rutti et al. [113] propose a modular, highly flexible architecture where protocol modules are not even aware of replacements. This solution is based on services (specifications of distributed protocols) rather than on protocols (implementations of distributed protocols).

6.5 Conclusions

Adaptability is a desirable feature for all systems, especially for those more sensitive to changes in the environment. Moreover, client applications of database replication systems may demand different requirements that can be better served with different replication techniques.

We propose here a metaprotocol that is the key piece of an adaptable system allowing the dynamical change of the replication protocol used in a replicated database as well as the concurrent execution of several protocols. The idea is to select each time the replication technique(s) that best fits with the changing requirements and dynamic environment characteristics, trying to provide always the best achievable performance. This selection can be made at system level, at application level or even at transaction level (using, for each transaction, the protocol which is the most appropriate for that transaction).

As first approach, the goal of the metaprotocol prototype is to provide support for the concurrent execution of the most relevant database replication protocol families based on total order broadcast: active, certification-based and weak voting replication.

Experimental results demonstrate that our metaprotocol introduces very low overhead when compared with stand-alone versions of the same replication protocols. On the other hand, inherent differences in the protocol behaviors may penalize performance when concurrency is exploited. We show and explain that certain combinations of protocols should be avoided if performance is a major system goal. Other protocol combinations, however, showed excellent performance and good scalability.

Chapter 7

Integrity Awareness in Database Replication at Middleware Level

Integrity constraints are usually forgotten in distributed and replicated databases, while they constitute an important part of the database: the semantic consistency. This chapter analyzes the problems an improper integrity management can originate, proposes simple solutions for abcast-based replication protocols and provides a comparative experimental evaluation.

7.1 Introduction

Data usually needs to be characterized beyond the basic database schema. There is a wide range of semantic constraints that belong to the domain of the application. Those constraints specify the *semantic consistency* of data, which complements the *transaction consistency* (guarantees of atomicity and isolation) and the *replication consistency* (accordance between replicas in their individual copies of common data items). The semantic consistency must be guaranteed; otherwise the data is inconsistent and does not correspond with the reality the application works with. Carefully checking at application level that semantic consistency is always kept would be costly and hard to maintain. Therefore, it is defined at database level by means of integrity constraints whose observance is ensured at data management level, inside the DBMS. The restrictions expressed by these

constraints can vary from a very simple condition affecting one single piece of data (e.g., a car rental company could require the age of drivers to be greater or equal than 21) to a complex relationship involving several tables (e.g., the sum of the salaries of the employees belonging to certain department cannot exceed the amount assigned to human resources in that particular department, and, in turn, the sum of these amounts of all departments cannot be greater than the total amount available in the company), including all the constraints commonly supported by any relational database (domain constraints, uniqueness constraints, not-null constraints, primary and foreign keys). Although the current support in commercial databases is far away from being standard, they will never commit a transaction if it violates any of such integrity constraints. To ensure this, the DBMS checks the constraints either at each update operation (immediate checking) or at the end of the transaction, before the final commitment (deferred checking). In any case, actions of concurrent transactions must be taken into account. If any violation is detected, the transaction responsible for it is aborted to guarantee that integrity is preserved. This works fine in centralized, stand-alone systems but can originate several problems when replicating the database.

In a database replication middleware, the replication protocol is responsible for maintaining the consistency of replicas, deciding which transactions to commit and which to abort. These decisions are based on the accesses made by transactions, so that two concurrent transactions are not allowed to conflictingly access the same object. If a transaction T successfully passes this validation, it is sent to the database to be committed. At the same time, the protocol informs the client about the transaction success, and it regards T as confirmed in order to abort subsequent concurrent transactions that try to conflictingly access the same objects. All this operation would be correct if no error could prevent the actual commitment of the transaction. But, e.g., T might violate any constraint and thus be aborted by the DBMS during deferred checking. Integrity violation is a permanent error, so re-attempts of applying the transaction are in vane. At this point, several undesirable conditions arise. First, the accesses made by the DBMS during integrity checking remain unnoticed by the protocol, which cannot consider them for conflict checking (we call this the problem of *increased accesses*). Second, the client was positively informed about the updates made by T but now it cannot see the state it expects (*premature client notification*). Third, the protocol discarded some transactions due to conflicts with T, which was never actually committed, causing unnecessary abortions (*compromised validation*). Lastly, the protocol is blocked trying to apply the violating transaction while subsequent

transactions accumulate to be applied in turn (*infinite reattempting*). Good news: integrity was preserved. Bad news: everything else is a mess.

Many database replication protocols have been proposed, as already shown in this thesis, mostly oriented towards dependability details; i.e., they are focused on the problems that should be overcome regarding performance, availability, and scalability. Unfortunately, none of these proposals has assessed the support of semantic consistency defined by integrity constraints, when precisely those constraints are one of the important differences that make databases quite specialized with regard to other regular components of a distributed application. This is an interesting matter to consider because, at least in middleware-oriented systems [20, 25, 31, 43, 59, 85, 93, 99], complications may arise if constraints are checked in deferred mode, i.e., at effective commit time, after conflict validation by the replication protocol. Most modern database replication protocols use total order broadcast for propagating sentences or writesets/readsets of transactions to other replicas. Some classes of replication protocols are able to seamlessly deal with the integrity support of the underlying DBMS, but others are not. Following again the classification proposed by Wiesmann and Schiper [136], we analyze here the integrity support that can be provided in various classes of replication protocols, proposing extensions for those that cannot directly manage constraints. Our aim is to ensure that each database replication protocol class provides at middleware level the same integrity support than its underlying DBMS. Finally, we include an experimental study that shows the negative effects of an improper integrity management as well as the performance penalty of an appropriate one.

Section 7.2 describes the architecture and database model. Section 7.3 recapitulates the main database replication classes identified by Wiesmann and Schiper [136]. Section 7.4 describes the problems that arise without an appropriate constraint management in a replicated system. In Section 7.5, we propose solutions for those identified problems. Later, Section 7.6 presents an experimental comparison between a correct and an incorrect integrity constraint management in middleware protocols. Related work is analyzed in Section 7.7.

7.2 System Model

We consider here a partially synchronous distributed database with full replication, where the process that executes the replication protocol belongs to a middleware [11] layer. A solution for DBMS-core replication protocols could be trivially derived from the solutions presented here.

Failures are not specifically considered in this chapter, as integrity maintenance is a problem orthogonal to failures and recovery. A system wanting to implement any of the solutions proposed here could use any available recovery protocol in accordance with the replication protocol and failure model assumed by such a system. However, some possible solutions can have negative effects in case of failures and we signal this later.

The data stored in the database is subject to integrity constraints, which are enforced at data management level, i.e., not at protocol or application levels (which would be costly and hard to maintain as it would require the examination of every update made by concurrent transactions to data items involved in a constraint).

The underlying DBMS directly provides support for integrity maintenance, by raising exceptions or reporting errors in case of constraint violation. Such exceptions and error messages are then managed by the replication protocol. Thus, they do not reach the user-level application, unless the replication protocol decides so. The DBMS also supports the isolation level for which the replication protocol has been conceived. Thus, the replication protocol may focus on its native purpose of ensuring replica consistency, and delegate local concurrency control and integrity maintenance to the DBMS.

7.3 Database Replication Protocols

As already stated before in this thesis, Wiesmann and Schiper [136] present a comparison of the three most relevant database replication techniques based on total order broadcast (active, certification-based and weak voting replication) along with other approaches that do not rely on group communication (primary-copy [18] and lazy replication). All the studied protocols use a *constant interaction* [138] principle: the number of messages used to synchronize the servers for a given transaction is constant and independent of the number of operations

in the transaction. Such a paper concludes that the replication protocol classes with the best performance are those that combine the usage of a single delegate replica per transaction with eager total order update propagation. Lazy replication is included in the study only for comparison purposes (it requires a minimal amount of synchronization but it later needs reconciliation techniques for ensuring consistency, which makes this technique not practical). We recall here the description of active, certification-based and weak voting replication, along with the primary copy technique:

Active replication (AR) The client sends its request to a given replica R_d, which forwards the whole transaction to all replicas by a single total order broadcast. Each replica independently and deterministically processes the transaction. Total order propagation ensures the same scheduling in all replicas, which get the same results for each transaction. Once the transaction is completed, R_d returns its results to the client.

Certification-based replication (CBR) A transaction is first locally executed in a single delegate replica. Once the client application requests its commitment, the transaction readset and writeset are collected and propagated to all replicas using a total order broadcast. Once such a message is delivered, a deterministic certification phase starts in all replicas to determine if such a transaction can commit or not. This certification is based on conflicts with concurrent transactions previously delivered that were already accepted for commitment. This validation stage is symmetrical since all replicas hold the same history log of previously certified readsets and writesets. Once certified, accepted transactions are applied and committed, and their writesets and readsets are temporarily held in order to certify subsequent transactions. Otherwise, i.e., when the transaction has been aborted in the certification stage, its readset and writeset are discarded.

Weak voting replication (WVR) As in the previous class, a transaction is executed in a single delegate replica. When it requests its commitment, its readset and writeset are collected but only its writeset is broadcast in total order to all replicas. In this case, once the writeset is delivered, only the delegate is in the position to validate the transaction against concurrent transactions previously delivered. Note that write-read conflicts are also detectable, since the delegate replica

knows the readset of its local transaction and also the writesets from remote replicas. The result of the validation is propagated using a reliable broadcast [28] to all replicas, which behave accordingly.

Primary copy replication (PCR) All transactions must be executed by the same primary replica which may rely on its local concurrency control mechanisms to decide whether transactions can be committed or not. Once a transaction has been executed in the primary replica, its writeset is propagated to the other replicas (that behave as backups) and applied there. Finally, the transaction is committed in all replicas, and the results are sent to the client.

7.4 Integrity Problems in Replication Protocols

Integrity constraints can be checked either in immediate or in deferred mode. When using immediate checking, constraints are checked each time an update is attempted in a transaction. With deferred checking, those integrity checks are delayed until the transaction requests its commitment. In centralized, stand-alone settings, some integrity constraints have been traditionally managed in deferred mode [21, 72, 83], thus avoiding intermediate checks that could have uselessly required time and resources and, more important, allowing temporary inconsistencies, sometimes unavoidable, that can be solved before transactions end. In any case, not only the effects of the current transaction but also those of concurrent ones must be considered during integrity checking. This leads to the mandatory use of deferred checking when dealing with distributed or replicated databases, where concurrent transactions can be executed in different delegate nodes. In that case an immediate checking is unable to see the effects of all concurrent transactions since they can be in execution in remote nodes and their updates need some time to be propagated and applied. Moreover, the set of concurrent transactions to consider when checking the integrity consistency of a given transaction is unknown until the global commit order is decided. For most replication protocols [136], which rely on total order broadcast, this happens only after the client requests transaction commitment. As a result, deferred checking is the appropriate and single choice.

However, deferred checking gives raise to some problems at current replication protocols. There are some common behaviors that are incorrect when integrity

constraints are involved. A generic replication protocol featuring all of these behaviors would execute transactions in one delegate node and, later, use two steps for transaction management once updates were broadcast and delivered. The first step would be devoted to conflict checking among concurrent transactions, based on a history log containing previously (positively) validated transactions. This validation would decide whether the current transaction could be committed or not, informing the client. The second step would take place only when the transaction were positively validated. In this case, the transaction would be added to the history log and sent to the database to be applied in non-delegate replicas and finally committed. If the commit operation failed, it would be considered a temporary error (e.g., a deadlock) and the operation would be reattempted again and again until its successful completion. These reattempts are necessary to avoid that temporary and local errors prevent a transaction from committing in one node, while it committed in the rest of replicas. In this generic protocol, an integrity violation detected in deferred checking turns some of the previous actions into serious problems:

Premature client notification The protocol admitted the transaction as correct in the conflict-checking step and reported its success to the client application. This early client notification reduces the completion time perceived by applications. However, the client expects the database to be updated accordingly but it will never be.

Increased readset/writeset Integrity management is an internal process of the DBMS and, thus, is highly vendor-dependent. Broadly speaking, read and even write operations could be performed during this checking. As they are within the scope of the transaction, they should be considered as part of its readset or writeset for conflict checking. But validation has already been performed. A deeper study of the integrity management done in commercial databases (PostgreSQL, Oracle, Microsoft SQL Server, Sybase, IBM DB2 and MySQL) concludes that no write operation is allowed during deferred checking, as all cascade actions, when supported, are restricted to immediate mode. Thus, only the readset is increased in such databases. As a result, replication protocols using readsets to validate transactions during conflict checking are missing some read operations.

Compromised validation A transaction successfully validated[1] in the conflict-checking step is appended to the history log of the protocol to be considered when validating subsequent transactions. This is to ensure that two concurrent transactions with any conflict in their accesses are not both committed. When an integrity-violating transaction T is positively validated and added to the history log, it will possibly (depending on conflict rate) cause the rejection of some subsequent transactions. But T will not be actually applied in the database, so those rejections could have been avoided. We call this *abort errors*. These abort errors cause, in turn, that some other transactions are accepted (*commit errors*) when they would not have passed the validation in a system properly handling integrity-related abortions. In short, the history log does not reflect the transactions actually applied in the database, leading to more and more validation errors.

Infinite reattempting The protocol notices the abortion of the transaction, but it assumes it is due to some temporary error and, thus, requests the operation again, maybe after an exponential backoff. But an integrity violation is a permanent error, so those reattempts are useless because the database will always raise the integrity exception. As transactions in abcast-based replication protocols usually follow a global commit order equal to the delivery order of the atomic broadcast, infinite reattempting eventually stops all committing processes. The loss of protocol liveness cannot be avoided even though some optimizations are used. The *concurrent writeset application* technique [79] allows several non-conflicting transactions to be sent to the database at the same time. This way, when indefinitely reattempting an integrity-violating transaction, subsequent non-conflicting transactions can be also sent to the database, allowing the committing process to continue. However, this optimization cannot be applied when the next transaction presents conflicts with the ones already in the database, so the protocol will eventually stop all processing.

These problems will arise in any database schema with integrity constraints, if the replication protocol follows any of the identified incautious behaviors, independently of the load or the conflict rate or the amount or kind of constraints.

[1] Remember that we use validation as the general term to refer to the decision process of a replication protocol. On the other hand, we use the term certification to specifically refer to a validation performed by each system node in an independent, deterministic and symmetrical manner.

7.5 How To Support Constraints

In order to adapt current database replication protocol families to properly manage integrity constraints, we must first analyze which of the previously identified problems present each of the targeted classes.

Active replication (AR) protocols do not perform any kind of validation. As transactions are completely executed in parallel at every replica, local DBMS management is enough. As a result, compromised validation does not appear in this class. Moreover, although transaction readsets will be extended, this does not pose any problem, as readsets are not considered at protocol level.

In these protocols, clients are informed only after transaction completion, which discards any premature client notification problem. In conclusion, only a slight modification will be necessary for an AR protocol in the case it reattempts transaction commitment. In such cases, the protocol must be able to recognize the cause of the abortion, as reported by the DBMS, rejecting transactions that violate integrity.

Primary copy replication (PCR) protocols neither require a validation step since only one replica is processing the transaction. So, in this regard, it is equivalent to a non-replicated architecture. Consequently, extended readset and compromised validation are not raised in this protocol family.

Client notification is delayed in PCR protocols until transaction is applied at the primary replica, preventing any premature client notification. Similarly to AR protocols, the only possible problem would be infinite reattempting.

Therefore, AR and PCR are able to deal with integrity constraints in a seamless way, by relying on the support provided by the underlying DBMS. Unfortunately, neither AR nor PCR protocols (direct translations into the database field of the general active and passive replication models, respectively) provide good performance [136]. Fortunately, there are specialized variations of these protocol classes able to break some of their limitations. For instance, the database can be partitioned and the subsets distributed, according to predefined conflict classes, among multiple primary copies (each one being responsible for each subset) [63]. These adaptations significantly improve the performance of PCR protocols, while the reliance on the integrity support from the underlying DBMS is not impaired.

On the other hand, the two classes that provide best performance and scalability (CBR and WVR) also feature several of the previously identified behaviors that give rise to problems with regard to integrity constraints. Our aim is to carefully study these two last classes in order to find some adaptations of their protocols that ensure a correct integrity support, while keeping their performance as good as possible. Next we recall the original pseudocode of these protocols and propose and explain extended versions that correctly manage integrity constraints.

7.5.1 Weak Voting Replication Protocols

```
1:  Execute T
2:  On T commit request:
3:    TO-bcast(ℝ, ⟨wset(T), R_k⟩)
4:  Upon ⟨wset(T), R_d⟩ reception:
5:    if R_k = R_d then
6:      status_T ← validate(T)
7:      R-bcast(ℝ, status_T)
8:      send(c, status_T)
9:    else DB.apply(wset(T))
10: Upon status_T reception:
11:   if status_T = commit then
12:     DB.commit(T)
13:   else DB.abort(T)
```

a. Original

```
1:  Execute T
2:  On T commit request:
3:    TO-bcast(ℝ, ⟨wset(T), R_k⟩)
4:  Upon ⟨wset(T), R_d⟩ reception:
5:    if R_k = R_d then
6:      status_T ← validate(T)
7:      R-bcast(ℝ, status_T)
8:      // Client notification delayed until line 15
9:    else DB.apply(wset(T))
10: Upon status_T reception:
11:   if status_T = commit then
12:     status_T ← DB.commitIA(T)
13:   else DB.abort(T)
14:   if R_k = R_d then
15:     send(c, status_T)
```

b. Extended

Figure 7.1: Integrity support in weak voting protocols

The pseudocode listing of Figure 7.1a represents the three event-driven blocks of a WVR protocol. Block I (lines 1–3) starts at transaction beginning and locally executes T. It lasts until T is requested to be committed, when its writeset, $wset(T)$, is broadcast in total order to the set $\mathbb{R}$ of alive replicas. The identifier of the local replica, R_k, is included in such a message, as the delegate identifier. The following two blocks describe what is executed in all replicas.

In block II (lines 4–9), action is taken upon writeset delivery. The delegate replica validates the transaction, broadcasting its decision to the rest of nodes and also notifying the client, c. Non-delegate replicas apply the delivered writeset (through the local DBMS interface, *DB*) and wait to the arrival of the voting

message. Note that line 6 is shaded in gray. This means that the validation operation must be executed in mutual exclusion, one transaction at a time. Each sequence of shaded rows in following listings corresponds to a block of protocol steps that are mutually exclusive.

In block III (lines 10–13), action is taken upon arrival of the voting message, resulting in a commitment or abortion request. As the commit operation was assumed to be always successful in the original WVR protocol, any failure is interpreted as temporary and the commitment is retried. For clarity, this reattempting policy is assumed inside the $DB.commit(T)$ operation.

Considering this pseudocode, the integrity problems of WVR are the premature client notification and the infinite reattempting. Problems related to validation (increased readset and compromised validation) do not necessarily appear here. Indeed, instead of using a history log, validation can be based on local concurrency control and follow several approaches. A possibility is to defer update operations until commit time and ask for the corresponding write locks as soon as a writeset is delivered. This solution, only possible for lock-based DBMSs, allows the protocol to obtain an immediate response: whenever a lock cannot be granted, a conflicting transaction was validated before and so the validation is negative. This approach, however, assimilates to one based on a history log and leads to a compromised validation when integrity constraints are involved.

Another possibility for a validation based on local concurrency control was suggested in Chapter 6: the delegate may wait until the end of the commit operation and adopt DBMS response as its vote for that transaction. This approach would provide a correct integrity management but would involve a potential problem. In a system following this approach and with a crash-recovery failure model, if a delegate replica R_d fails, for any positively validated transaction T between its commitment and the broadcast of its voting message, the delegate replica has already committed the transaction while all others are still waiting for its commit/abort message. This violates the uniformity principle [105] required for preserving in modern replication protocols the same functionality than in the traditional 2PC one. Such uniformity is preserved in the original WVR protocol using uniform total order broadcast for writeset propagation, and uniform reliable broadcast in the termination/voting final message. In the system using the commit response as a vote, in case of failure one of the non-delegate replicas must be selected as the new transaction delegate, in order to broadcast that message. This complicates such transaction termination, demanding much more time. Additionally, this also

complicates the recovery protocols since the state maintained in a crashed delegate replica for its transactions is uncertain (it may have applied their updates or not), violating also the assumptions of the *persistent logical synchrony* [94] model that was specifically tailored for replicated database recovery.

A validation based on local concurrency control that does not suffer from the above problems is the one where the delegate notices the abortion of its transactions, caused by the commitment of previously delivered writesets. This approach is similar to the previous one, but it does not commit the transaction before sending its vote. Indeed, when a transaction T *survives* to the commitment of all previously delivered transactions, the delegate sends a positive vote for T. Otherwise, a negative vote is sent as soon as the abortion of T is detected. We assume both the original and the extended version of WVR follow this approach.[2]

The extended protocol of Figure 7.1b, where original line numbers are maintained, includes the adaptations needed for appropriate integrity management. First, the operation responsible for requesting the commitment is now integrity-aware ($DB.commitIA(T)$): if the operation fails due to integrity violations, it will not be reattempted, solving the infinite reattempting problem. Moreover, this operation returns a value with the final outcome of the transaction (either committed or aborted by integrity violation). On the other hand, client notification is delayed until transaction termination (from original line 8 to new lines 14 and 15), informing the application about the actual final termination of the transaction, thus eliminating the premature client notification problem.

It is important to note that despite a positive voting of the delegate replica for a given transaction, this transaction could be later aborted due to a constraint violation. But this abortion will occur in every replica as each local DBMS performs integrity checking. This way, no inconsistency will appear and such mismatch between voting and final outcome does not affect any other replication processes.

Although these extensions demand a minor effort, they do have some impact from the point of view of the client. In the original algorithm, as it is described by Wiesmann and Schiper [136], control was returned to the client before the actual commit was requested to the DBMS, thus reducing the transaction completion time *perceived* by the user application. When integrity constraints are involved, such behavior is not correct. However, *actual* transaction completion

[2] Approaches subject to the compromised validation problem can be solved as in the CBR case.

time remains the same in the extended version, so proper integrity management does not carry performance implications in WVR protocols.

7.5.2 Certification-Based Replication Protocols

CBR protocols are quite similar to WVR ones but, instead of a validation where only the delegate replica decides and broadcasts later its decision, CBR protocols perform a symmetrical evaluation stage: the certification.

To provide a serializable isolation level, readset propagation is needed in CBR protocols, since only the delegate replica maintains such information when transactions request their commitment, but all replicas need it when the certification is executed. However, readset collection and propagation can be costly if row-level granularity is used for managing readsets and writesets.[3] So, in practice, certification-based protocols are rarely used for implementing serializable isolation. Nonetheless, CBR is the preferred protocol class for providing the snapshot isolation (SI) level [9], where readsets do not need to be checked. So we focus on SI-oriented CBR protocols, whose pseudocode is displayed in Figure 7.2a.

```
1:   Execute T
2:   On T commit request:
3:     TO-bcast(ℝ, ⟨wset(T), R_k⟩)
4:   Upon ⟨wset(T), R_d⟩ reception:
5:     status_T ← certify(wset(T), wslist_k)
6:     if status_T = commit then
7:       append(wslist_k, wset(T))
8:       if R_k ≠ R_d then
9:         DB.apply(wset(T))
10:      DB.commit(T)
11:    else DB.abort(T)
12:    if R_k = R_d then
13:      send(c, status_T)
```

a. Original

```
1:   Execute T
2:   On T commit request:
3:     TO-bcast(ℝ, ⟨wset(T), R_k⟩)
4:   Upon ⟨wset(T), R_d⟩ reception:
5:     status_T ← certify(wset(T), wslist_k)
6:     if status_T = commit then
7:       // append delayed until line 10b
8:       if R_k ≠ R_d then
9:         DB.apply(wset(T))
10:      status_T ← DB.commitIA(T)
10a:     if status_T = commit then
10b:       append(wslist_k, wset(T))
11:    else DB.abort(T)
12:    if R_k = R_d then
13:      send(c, status_T)
```

b. Extended

Figure 7.2: Integrity support in SI-oriented certification-based protocols

[3] It is also possible to use table-level granularity, but its coarse grain may increase abortion rate.

Initially, a transaction T is locally executed in R_k. When T requests commitment, its writeset is collected and broadcast in total order to the set $\mathbb{R}$ of alive replicas, along with the identifier of its delegate replica, R_k. Upon the reception of such a message, each replica certifies the incoming writeset against concurrent transactions (logical timestamps are used to determine the set of concurrent transactions), looking for write-write conflicts (line 5). This certification process returns a *commit* result if no conflicts are found, or an *abort* result otherwise. This result is regarded as the status of the transaction, $status_T$. Note that the certification is symmetrical since each replica holds the same history log of previously delivered and successfully certified writesets, $wslist_k$. If no conflict is found, the writeset is included in the history log (which should be pruned to avoid indefinite growth, as suggested by Wiesmann and Schiper [136]) and the local DBMS interface, *DB*, is used to apply the writeset in non-delegate replicas and to commit it in each node (if writeset application is impeded, e.g., by T being involved in a deadlock and aborted by the DBMS, it is reattempted until it succeeds). On the other hand, if a conflict appears during certification, T is aborted in its delegate and discarded in the rest of replicas (for simplicity, the pseudocode represents both actions as the $DB.abort(T)$ operation, although no operation is requested to the DBMS in non-delegate replicas). Finally, the delegate informs the client c with the transaction outcome, represented by $status_T$.

SI-oriented CBR protocols are not affected by the increased readset problem, as readsets are not considered. However, they base certification on a history log that adds transactions as soon as they pass their validation, ignoring the results of their commit requests. This leads to a compromised validation when integrity constraints are involved. In addition, infinite reattempting appears when trying to commit an integrity-violating transaction. As the processing for such a transaction will stop at line 10, the protocol will never reach line 13, which trivially avoids any premature client notification.

The extensions for managing integrity constraints in SI-oriented CBR protocols are displayed in Figure 7.2b. Line 10 is modified to perform an integrity-aware commit request, which solves the infinite reattempting problem by detecting abortions due to integrity violation. The result of this commit attempt is also recorded and, only if it is successful, the writeset of such a transaction is appended to the history log. This is done in lines 10a and 10b, instead of in line 7, in order to avoid the compromised validation problem. Moreover, as the status of the transaction is updated with the result of its commit attempt, the client is appropriately notified.

These extensions, although simple, may have a notable impact on system performance. Typical SI-oriented CBR protocols [40, 41, 79, 93] use some optimizations in order to achieve better performance. A possible optimization consists in minimizing the set of operations to be executed in mutual exclusion in the part of the protocol devoted to managing incoming messages. The related protocol section in Figure 7.2a encompasses only lines 5 to 7. As a result, new certifications can be made once the current writeset is certified. In our extended protocol, no new writeset can be certified until a secure decision on the current one is taken. Indeed, no possible assumption about the current writeset is secure: if we assume that the current writeset T will commit and it later aborts due to integrity constraints, subsequent writesets that were negatively certified due to T were unnecessarily aborted. If, on the other hand, we assume that the current writeset T will abort due to integrity, subsequent writesets that were positively certified due to the absence of T from the history log were incorrectly validated and they should be rejected. In any case, either if we add the writeset to the log and later remove it, or if we wait for its commitment before adding a writeset to the log, a fatal situation may arise if the access to the log is not prevented in the meantime. Indeed, replica divergence may occur due to the different pace at which replicas apply transactions in the database. If the final outcome of a writeset modifies the history log (either removing that transaction from the log or adding that transaction to the log), such a modification may take place at different instants in different replicas. This natural lack of synchrony may cause different outcomes at different nodes in the validation of new delivered transactions, which may in turn lead to replica divergence.

As a result of the problems just mentioned, no new writeset can be certified until the real outcome of the current one is known. That only happens once the transaction termination is completed in the DBMS. This might require quite some time, but must be done one writeset at a time. Thus, the mutual exclusion zone in the extended protocol of Figure 7.2b includes lines 5 to 10b.

Another possible optimization [41] for CBR protocols consists in grouping multiple successfully certified and non-conflicting writesets, applying all of them at once in the underlying DBMS. This reduces the number of DBMS and I/O requests, thus greatly improving the overall system performance. Unfortunately, extensions needed to prevent a compromised validation make impossible the formation of such groups of transactions. And even if it were possible, an integrity violation detected when applying one of such batches should lead to the rejection of the whole batch, as the protocol could not find out which one of the individual

writesets caused the violation. Still worse problems can raise. This is the case of the optimization proposed by Lin et al. [79], where certified and non-conflicting writesets can be concurrently sent to the database in order to parallelize writeset application. As we will see in Section 7.7, this technique may lead the protocol to break the fundamentals of database replication, by allowing different replicas to commit different sets of transactions.

A final consideration must be done. As stated before, our aim in supporting integrity checking in database replication protocols is to guarantee at middleware level the same support present in the DBMS. Although several DBMS products (PostgreSQL, Oracle, Microsoft SQL Server...) support the snapshot isolation level, some of them label it as serializable not enforcing a true serializable level but their own flavor of the SI one. These DBMSs are actually providing an enhanced integrity management in their systems, equivalent to the one provided in a serializable level. The resulting isolation level cannot be tagged as a pure SI one (as it was first defined [9]) but as an extended SI level able to support integrity constraints in a serializable way [80]. As a result of this, *pure SI* certification-based protocols [40, 69, 79, 93] should not be criticized for omitting the extensions shown above.

7.5.3 Compromise Solutions

As we will see in the experimental study of Section 7.6, a proper integrity management may cause performance degradation. From the identified problems, the most serious one is the infinite reattempting issue, as it causes the loss of protocol liveness. The solution to this problem, as showed above, is simple and carries no performance implication, so all systems will surely want to apply it if integrity constraints are in use.

Other problems, however, are not as severe as infinite reattempting and may be tolerated by some systems. Thus, the increased readset problem can be safely ignored by protocols that do not use readsets for validation. In other systems, client notifications might be simply used as an informal notice for the user, being unimportant if they carry the actual outcome of the transaction.

Compromised validation, although it involves unnecessary abortions as we will see later, does not cause more important problems like replica divergence, as long as all nodes arrive to the same decision for each transaction. In WVR,

the decision of the delegate is assumed by remote nodes, so such agreement is guaranteed. In CBR, as long as all replicas certify over the same history log (which is ensured if transactions are processed in their delivery order and no modification –like removing transactions aborted due to integrity– is made to the log outside the corresponding mutual exclusion zone), they will make the same certification errors.[4] As a result, the same set of transactions is committed and aborted in all replicas. If those unnecessary abortions caused by the compromised validation are bearable, it may be worth maintaining such a compromise solution due to the performance penalty of an increased mutual exclusion zone.

An example of a compromise solution would be a WVR system where delegates validate a transaction by asking for write locks as soon as its writeset is delivered. As stated before, this approach, although based on local concurrency control (and not in a history log), is still compromised: as a writeset asks for locks upon delivery, it holds its locks from delivery-time until its termination. This may result on some subsequent transactions being negatively validated based on the assumption that the writeset will be committed. If the writeset is later aborted due to integrity violation, those transactions were unnecessarily aborted.[5] However, as said before, as the rest of replicas assume the decision of the delegate, no divergence occurs.

In summary, each system should assess which problems matter to it, correct such issues, and accept the consequences, if present, of the rest.

7.5.4 Metaprotocol Extensions

In order to correct all the identified integrity-related problems that could arise in our metaprotocol MeDRA, similar extensions as the proposed above need to be applied.

As readsets are not used in the metaprotocol, the increased readset problem can be safely ignored. Moreover, control is not returned to the client until after the

[4] In Section 7.6, we will force a situation where different nodes execute different protocol versions and so the condition of having the same history log will not hold.

[5] Once a writeset aborts due to integrity constraints, it will release its locks and thus the replica will stop sending unnecessary abort votes due to that writeset.

commit operation is invoked at the DBMS (see the complete metaprotocol pseudocode in Appendix B), and therefore the metaprotocol does not suffer from the premature client notification.

In order to avoid infinite reattempting, the metaprotocol needs to identify the abortion cause as reported by the DBMS. This can be easily done by distinguishing among the different error codes and discarding the reattempt whenever an integrity violation occurs.

Finally, both the validation of the weak voting protocol and the certification of the certification-based one are implemented in the metaprotocol by the execution of a procedure that checks for write conflicts with concurrent previously delivered transactions, as stored in the history log (Algorithm 4 of Appendix B). Due to the dependencies that appear among transactions, the validation/certification result is often pending. Our dependency tracking, however, would allow us to easily adapt the metaprotocol in the case of an integrity violation of an active transaction. Indeed, whenever an active transaction T_A exists in the *tocommit* list, all subsequent, non-active transactions establish a w-dependency with it. Should T_A abort due to integrity violation, the metaprotocol could notice it and, instead of marking T_A as resolved and aborting all conflicting w-dependent transactions, the metaprotocol would delete T_A from the lists and simply remove all the w-dependencies it caused. This allows a correct, non-compromised integrity support for active transactions without any additional overhead, as the current dependency tracking serves as an extended mutex.

On the other hand, in order to support integrity violations by weak voting or certification-based transactions, the metaprotocol could add a new log entry type, *i-pending*, that would substitute the current *resolved* type for WVR and CBR transactions positively validated but not yet committed. This new status would mean that the final outcome of the transaction is not yet known, as it depends on the deferred integrity checking. The solutions proposed above extended their mutual exclusion zone. Likewise, with this new status, we extend our dependency tracking in order to support integrity constraints. This way, the status of a CBR or WVR transaction that is free of c- and w-dependencies but it is not yet committed will be i-pending instead of resolved. Remember that whenever a c-pending T_C transaction removes all its c- and w-dependencies, a process is run for resolving the dependencies it caused in cascade. For a correct integrity support, as said before, the status of T_C would become i-pending and existing c-dependencies

on it will be transformed into i-dependencies during this cascade process.[6] The implications of an i-pending transaction for the validation of subsequent transactions would be identical to those of c-pending transactions: whenever a delivered transaction T overlaps with an i-pending transaction T_I, a new i-dependency is created and T is said to i-depend on T_I. It is important to note that, as opposed to a c-pending transaction, an i-pending transaction is committable, i.e., it can be sent to the database to try to commit (another thing is whether it will be able to actually commit or not, depending on integrity constraints). Only after the end of the commit operation of T_I, the i-dependencies it caused will be resolved: if T_I achieved to commit, its status changes to resolved and all its i-dependent transactions are aborted. Otherwise, T_I is removed from the log and such i-dependencies are also removed.

Of course, an easier and less penalized approach would be to accept the consequences of the compromised validation and thus provide a compromise solution that tolerates integrity constraints although makes some unnecessary abortions. In this case, without i-dependencies, transactions aborted due to integrity violation must remain in the log in order to avoid replica divergence.

7.6 Evaluation

The extensions or adaptations proposed in this chapter have performance implications only in the case of CBR protocols, where the greater mutual exclusion zone of the extended version represents a serious disadvantage of the proper integrity management. On the other hand, in WVR protocols, although the client perceives higher response times in the extended than in the original version, the truth is that the premature client notification of the original protocol actually provided clients with an unreal vision of transaction completion times. As long as the approach followed for validation is not compromised, the extensions of the proposed version do not entail performance implications. Otherwise, when validation is compromised and the system requires an absolutely correct integrity support, an extension of the mutual exclusion zone, similar as the one made in CBR, is also required. In such a case, one may expect the performance penalty of the WVR extended version to be similar as in the CBR case, which is analyzed in this section.

[6] The transformation of c-dependencies on i-dependencies can be only logical, as both dependencies could be resolved after commit operation likewise.

In order to study the costs of a correct constraint management in the CBR protocol class, this section shows some experimental results, which confirm the increment of transaction completion times and also reflect the problems that an incorrect integrity management causes in a database replication middleware.

Experimental tests were performed in order to observe in a real system both the negative effects of an incorrect integrity management and the performance degradation due to a proper one. To this end, the original and the extended versions of the SI-oriented CBR protocol family were implemented and tested in our replication middleware system, MADIS [59, 93]. From the original SI-oriented CBR class shown in Figure 7.2a, we derived an integrity-unaware protocol, IntUnaware, which corrects the infinite reattempting problem (it is able to identify those transactions that raise integrity exceptions when tried to be committed and so it does not indefinitely reattempt them) and also notifies the client with the real outcome of the transaction. These two modifications were needed to keep protocol liveness and to be able to compute real abortion times. The only remaining problem in IntUnaware is the compromised validation: it keeps in the history log those transactions that were aborted due to integrity violations. On the other hand, the extended pseudocode shown in Figure 7.2b was implemented in an integrity-aware version, IntAware.

Test Description To accomplish the analysis, we use Spread [122] as our GCS and PostgreSQL [108] as the underlying DBMS. Client transactions access a database with two tables, *tbl1* and *tbl2*, each with 2 500 rows and two columns. The first column of each table is declared as its primary key. The second column of *tbl1* is an integer field that is subject to updates made by transactions. The second column of *tbl2* is a foreign key, set to be evaluated in deferred mode, that references the first column of *tbl1*. The primary and foreign key constraints define the integrity consistency of our database.

Two types of transactions are used: (a) transactions that preserve integrity, called IntS –integrity-safe– transactions; and (b) transactions that violate an integrity constraint. The latter update the foreign key column of *tbl2* with a value that has no referenced value in the primary key column of *tbl1*, and are called IntV –integrity-violating– transactions. In the test runs of our analysis, we varied the proportions of IntS and IntV transactions. More precisely, we analyzed test runs with 0, 3, 6, 9, 12 and 15% of IntV transactions,[7] in order to analyze the evolution

[7] Higher percentages would not be realistic.

of the system as more and more transactions originate integrity violations. Thus, all shown graphs display this percentage in their X axis.

Both protocols were tested using MADIS with 2 replica nodes. Each node (as those used for the evaluation of the metaprotocol in Chapter 6) has an AMD Athlon™ 64 Processor at 2.0 GHz with 2 GB of RAM running Linux Fedora Core 5 with PostgreSQL 8.1.4 and Sun Java 1.5.0. They are interconnected by a 1 Gbit/s Ethernet. In each replica, there are 4 concurrent clients, each of them executing a stream of sequential transactions. Two environments have been studied: a low-load one, with 100 ms of pause between each pair of consecutive transactions from the same client; and a high-load environment, with no pause between transactions. Both IntS and IntV transactions access a fixed number of 20 rows from table *tbl1* for writing. These accesses may cause conflicts, which will be detected during certification and will cause the abortion of some transactions. Besides this, every transaction also updates a row from table *tbl2*: IntS transactions do it preserving integrity, whilst IntV ones always violate the related foreign key constraint.

Evaluating Incorrect Decisions To clearly show the differences in the decisions made by each protocol, one replica node works with the IntAware version and the other one uses the IntUnaware protocol. So, for convenience, we call nodes as the aware and the unaware node, respectively. With this configuration, it is easy to detect the incorrect decisions made by the IntUnaware protocol. Suppose that an IntV transaction T_v is delivered in the system, presenting no conflicts with concurrent transactions, and therefore being positively certified by both nodes. The aware node tries to commit T_v and discards it when the database notifies the integrity violation, preventing T_v from be appended to the history log. In the unaware node, T_v is first inserted in the log and later sent to the database for the commit operation. Although the integrity violation is detected and so T_v is not indefinitely retried (one of the corrections we added to the original protocol, as explained above), the violating transaction remains in the history log.

Problems arise when a subsequent IntS transaction T_s presents write conflicts only with IntV transactions (see Figure 7.3). In the unaware node, some of these IntV transactions –those that were positively certified– are present in the history log and thus are considered during the certification of T_s. As a result, T_s may fail the certification phase and thus be aborted, while it is allowed to commit in the aware node, which does not present any IntV transaction in its history log. Let

Event	History logs
Non-conflicting IntV transaction T_v:	
successfully certified in all nodes	IntAware: $\{\}$
nodes try to commit T_v and violation is detected	IntUnaware: $\{T_v\}$
IntS transaction T_s conflicts only with T_v:	
IntAware successfully certifies and commits T_s	IntAware: $\{T_s\}$
IntUnaware detects conflicts and aborts T_s (abort error)	IntUnaware: $\{T_v\}$
IntS transaction T_{s2} conflicts only with T_s:	
IntAware detects conflicts and aborts T_{s2}	IntAware: $\{T_s\}$
IntUnaware successfully certifies and commits T_{s2} (commit error)	IntUnaware: $\{T_v, T_{s2}\}$

Figure 7.3: Compromised validation in integrity-unaware nodes. An incorrect behavior for IntV transactions leads to incorrect decisions about subsequent IntS transactions.

us call this situation an *abort error*. A transaction T_a incorrectly aborted in the unaware node is committed in the aware one, and it appears in the history log of the aware node but not in the history log of the unaware one. Now suppose that a subsequent IntS transaction T_c is delivered. If T_c presents conflicts only with T_a, then, due to the differences in the history logs, T_c will abort in the aware node but commit in the unaware one. This is a *commit error*.

As a result of the improper management of the unaware node, its history log incorrectly includes some transactions (IntV and erroneously committed IntS transactions) and it also incorrectly misses others (erroneously aborted IntS transactions). This way, the unaware node may certify incoming transactions in a wrong way, i.e., with different certification result that the aware node, or, even, with the same final certification decision but based on conflicts with different transactions. In our tests, we compute the differences in the final outcome of transactions from both nodes. As IntV transactions always end in abortion, certification errors for IntV transactions remain unnoticed in our tests. Thus, Figure 7.4 shows both abort and commit errors over IntS transactions, made by the unaware node as a result of its compromised validation. Values are expressed in absolute numbers over the total of transactions issued (16 000 in each test).

Mainly, detected errors consist in abort errors, i.e., aborting transactions that conflict with others incorrectly included in the history log. Commit errors are less usual as transactions in an unaware node are certified against a greater number of transactions (compared to the aware node), thus being much more likely to

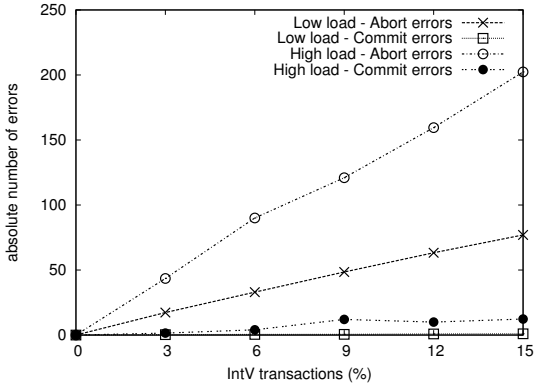


Figure 7.4: Errors of the IntUnaware protocol

get aborted by mistake than to successfully pass certification. Indeed, for an IntS transaction to be erroneously committed, it can present conflicts only with previous IntS transactions erroneously aborted. The graphs show that, as expected, the greater the percentage of IntV transactions, the greater the number of errors made by the unaware node, as opposed to the always correct behavior of the aware node. This trend is maximized when the load increases.

Evaluating the Abortion Rate Figure 7.5 shows the average percentage of local transactions that are aborted in a node due to a failed certification caused by write conflicts with concurrent transactions previously delivered. Abortions due to integrity violations at commit time are not computed here. As seen before, the history logs of both protocol versions differ when transactions involve any integrity violation. This may lead to a different set of transactions to be checked for conflicts against the one being certified. Therefore, the abortion rate differs from aware to unaware nodes: the abort errors of the unaware node result in an abortion rate in this protocol which is higher than in aware nodes.

In a low-load environment, the abortion rate remains constant in the IntUnaware protocol, while it linearly decreases in the IntAware node, where, as the percentage of IntV transactions increases, more and more transactions are not inserted into the history log due to integrity violation, leading to a smaller probability for local IntS transactions to fail certification due to conflicts with the remaining ones. In a high-load environment, this behavioral trend is maintained; i.e., the

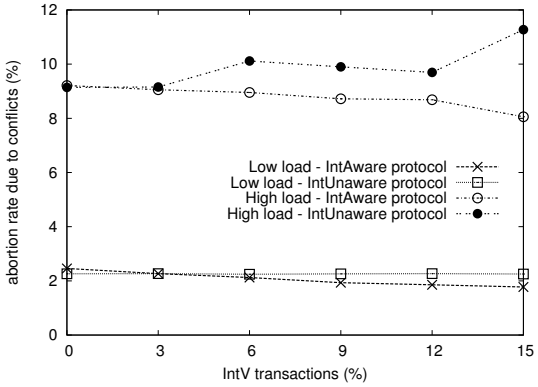


Figure 7.5: Abortion rate due to actual certification conflicts

IntAware protocol aborts less transactions than the IntUnaware one. However, the abortion rate of the unaware node should be constant (as it was in the low-load case) as the conflict rate does not change in our test. The upward trend observed in the graph for the IntUnaware protocol under a high load is surely due to the fact that the environment chosen as high load was really stressing for our middleware, which became saturated in many test iterations whose results we were forced to discard. Remaining results, as they are variable and non-representative, might lead to a biased interpretation of the evolution of the abortion rate in high-load environments.

Evaluating the Length of Transactions Figure 7.6 presents the length of local committed transactions (in ms). No important differences appear between the protocols in the low-load environment, as the arrival rate is low enough and transactions do not have to wait for accessing the mutual exclusion zone. On the other hand, when the load becomes higher, it is clearly seen that the IntAware protocol performs worse than the IntUnaware one. Indeed, completion times increase up to 16.18% in the case of 12% of IntV transactions. As before, it can be observed an upward trend in the high load case which is, again, due to the reduced number of test results caused by system saturation.

Regarding aborted transactions, those that abort due to integrity violation present completion times only slightly higher than those that commit. This is expected, as the processing of those two types of transactions is the same except for the

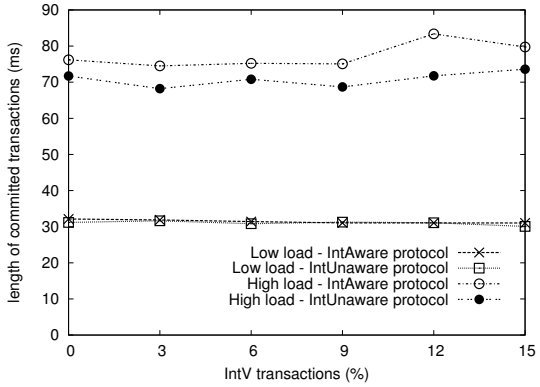


Figure 7.6: Length of committed transactions in presence of integrity constraints

final integrity exception and subsequent rollback suffered by the first type. On the other hand, transactions that suffer some block when locally executing at their delegate, due to conflicts with other concurrent transactions, experiment an increment in their duration which increases the probabilities of being aborted during certification. As a result, transactions aborted due to conflicts present completion times noticeable higher than the rest.

Recall that the proper management of integrity constraints prevents the IntAware protocol from applying any optimization proposed for certification-based replication protocols. Moreover, the aware protocol requires a greater mutual exclusion zone. Obviously, these conditions penalize performance and such a drawback is more noticeable at higher loads. On the other hand, the IntUnaware protocol applies only the optimization consisting in certifying newly delivered writesets concurrently with the application of previous ones. This means that the performance difference observed between the two considered protocols would be bigger when comparing the IntAware algorithm with an optimized version of the SI-oriented CBR protocol type. Likewise, when increasing the number of replicas and clients, the performance gap will also increase. Nevertheless, also recall that the IntUnaware version used for our tests was a correction over the original SI-oriented CBR protocol and thus it was able to identify abortions due to integrity violation, avoiding the problem of infinite reattempting and also notifying the client with the actual outcome. The original SI-oriented CBR protocol would simply have lost liveness in our tests.

7.7 Related Work

The literature on integrity checking in replicated database systems is extremely scant; exceptions are few and peripheral (e.g., Veiga and Ferreira [132] focus on supporting *referential integrity* in the world wide web, i.e., they attempt to avoid broken links). Apart from the study presented here, which continues the work initiated by Muñoz-Escoí et al. [94], and up to our knowledge, only Lin et al. [80] make a contribution in this topic. We next analyze such a contribution.

In that paper, Lin et al. present a formal framework for reasoning about snapshot isolation in a replicated environment. They later derive a new isolation level, denoted SI+IC, which corresponds to the DBMSs that guarantee the maintenance of integrity constraints upon the snapshot isolation level. Authors then extend their dependency graphs in order to be able to identify a replicated history as 1-copy-SI+IC. These contributions are very important, as all the existing literature about integrity management was based on the serializable isolation level. Lin et al. are thus the first to formally reason about integrity in snapshot isolation and they achieve to provide a formal characterization for an isolation level that was already supported in commercial systems but lacked of sound formalism, and to adapt such a formalization for a replicated environment.

In order to show the applicability of their framework to real replication protocols, authors show three sample protocols and describe their integrity support. The first protocol is SRCA [79] (included in the survey of Chapter 5). As originally described by Lin et al. [79], SRCA was a simple replica control algorithm for SI and its pseudocode did not include any special treatment for possible errors detected during the commit operation. Nevertheless, Lin et al. [80] assert that the middleware is able to track the failure of a commit operation accordingly (unfortunately, they do not provide details about what actions are made by this tracking). From their description of the protocol and following our classification of integrity-related problems, we can say that SRCA does not suffer from infinite reattempting nor it does a premature client notification, thanks to such failure tracking. Moreover, as readsets are not considered in snapshot isolation, the increased readset problem can be safely ignored. However, the compromised validation remains as a problem: even if SRCA removed an integrity-violating transaction from its history log (which is not said in the paper, as details are not provided), other transactions could have been incorrectly validated in the meanwhile. Truly, as all nodes apply the same writesets, no divergence would occur

among replicas. However, as showed in our experimental results (Section 7.6), a compromised validation entails a higher abortion rate than a correct integrity management. In short, SRCA is a compromise solution for integrity support and it may make validation errors. As a result, it does not provide the same integrity support than the underlying DBMS, as it may make unnecessary abortions that would not occur in a local DBMS.

The second sample replication protocol presented by Lin et al. [80] is there called SRCA-Ex (which corresponds to the SRCA-Rep protocol included in the survey of Chapter 5). This protocol allows non-conflicting writesets to be concurrently sent to the database in order to parallelize such writeset application. As a result, different replicas may commit transactions in different orders. The loss of sequentiality that follows is concealed from clients by not allowing new local transactions to start whenever a *hole* in the commit order exists. In the absence of integrity constraints, authors prove that SRCA-Ex provides one-copy snapshot isolation. However, as authors highlight, SRCA-Ex allows different replicas to commit different sets of transactions when integrity constraints are involved. Imagine two non-conflicting transactions that violate integrity if they are concurrent and both commit (e.g., the insertion of an employee into an empty department and the deletion of that department). If these transactions concurrently execute at different delegate replicas and their validation success (suppose no other conflicting transactions are executing), then in SRCA-Ex they are both immediately committed at their delegate node. Later, when the second transaction tries to commit, the violation is detected and that second transaction, which is different at each node, aborts. As a result, a different set of transactions is committed by these two replicas: not only integrity constraints are not supported but also the correctness criterion of one-copy snapshot isolation is violated.

The third protocol described by Lin et al. [80] tries to unify the benefits of the concurrent writeset application of the second protocol with the integrity support of the first one. Thus, SRCA-2PC follows the same algorithm as SRCA-Ex but it runs a 2PC protocol for the commitment of each transaction, after its writeset has been applied at all nodes. This allows replicas to concurrently apply writesets while ensuring that they will later agree on the set and the order of committed transactions. Unfortunately, SRCA-2PC requires two consensus per transaction (one for the abcast and one for the 2PC). This may have a cost similar to our proposed solutions in a fast network, but it may seriously penalize performance in networks with low bandwidth or high latency.

In summary, the contributions of Lin et al. [80] regarding integrity support in replicated databases are the definition of the new correctness criterion of 1-copy-SI+IC and the detailed presentation of a formal framework that allow us to prove if a replication protocol guarantees such a criterion. However, as the main objective of the paper is not the same as ours, the paper does not show the integrity-related problems that may appear with an incautious integrity management[8] and it does not detail how replication protocols must be designed in order to provide the new criterion. Indeed, from the two protocols providing 1-copy-SI+IC which are described in the paper, SRCA and SRCA-2PC, the first one is said to directly and seamlessly support integrity although, always based on the given information, it actually constitutes a compromise solution; and the second one, which inherits the compromised validation from SRCA, requires an atomic commit protocol additional to the consensus of the abcast which might severely penalize performance. On the other hand, Lin et al. [80] were mainly focused in the formal characterization of snapshot isolation in replicated databases, and so they make an extensive and pioneering study of it.

7.8 Conclusions

Integrity constraints define what is a consistent database state, by requiring certain conditions to be invariant across updates. Due to the physical distribution of a replicated database, concurrent transactions may execute at different servers without being aware of each other. Thus, to guarantee the ACID properties of transactions [55], integrity, i.e., semantic consistency, has to be checked at deferred mode, when the set of concurrent transactions is finally known. Except for very rare exceptions, replication protocols do not consider semantic consistency at all. This poses a problem when the database schema includes integrity constraints: as protocols sanction transactions as ready to commit if no conflicting accesses to shared data resources are detected, some accepted transactions are later unexpectedly aborted by the DBMS to guarantee integrity.

[8] SRCA-Ex, as a vast majority of replication systems, does not support integrity, but the consequence this entails is the loss of one-copy equivalence when integrity constraints are involved. As a result, it would not make sense to study the possible quality degradation of the protocol in the presence of integrity constraints, as it cannot be used as a correct replication protocol in such an environment.

We have identified four problems related to an improper or null integrity management: premature client notification, increased readset/writeset, compromised validation, and infinite reattempting. Their implications vary from the loss of protocol liveness to a misinformed client. We have also analyzed several well-known classes of abcast-based replication protocols. For some of them, support for integrity checking can be seamlessly integrated. For others, some modifications are needed to make them work well also when integrity is checked by the DBMS at hand. The solution to their problems is not difficult and has been described in this chapter by providing the required extensions for each critical class. Unfortunately, the extensions for CBR and WVR protocols (which have the best performance properties [136]) demand longer critical sections and prevent them from using most of the optimizations that are responsible for their good reputation, thus leading to a performance decrease.

We have presented an experimental study of the negative effects of not correctly managing integrity constraints, along with the performance implications of a correct integrity management. This has been accomplished by comparing the behavior of two protocols. The first one reflected the traditional behavior of protocols which do not care about integrity maintenance, based on the incautious assumption that all transactions are programmed in such a way that they will preserve integrity. On the other hand, the second protocol studied in our analysis properly handles semantic consistency as declared by integrity constraints. We have showed that an improper processing of integrity-violating transactions entails a history log that does not reflect the transactions actually applied in the database, which leads to errors when validating subsequent transactions. This causes not only incorrect abortions but also incorrect commits. Moreover, resulting from those errors, incorrect nodes present higher conflict-related abortion rates than correct nodes. On the other hand, results show that transactions suffer some delay when managed by the extended, integrity-aware protocol version. This delay is due to the greater mutual exclusion zone needed to safely access the history log, which forces transactions to wait before accessing it. This way, as the load increases, also the queue length (and, therefore, the waiting time) becomes larger. This opens a new line of research in the field of database replication, that could lead to efficient integrity-aware protocols, if new protocol optimizations, compatible with a correct integrity constraint management, are designed for overcoming the current limitations, as identified in this thesis.

Chapter 8

Conclusions

This thesis is devoted to the consistency, characterization, adaptability and integrity of database replication systems. Among its results, the following insightful conclusions can be highlighted:

- The correctness criterion of 1SR, widely accepted and used, can be subdivided into three different levels of user-centric consistency, regarding the possibility of inversions between transactions that present a real-time precedence among them. The strictest level avoids all inversions; the intermediate level avoids inversions within user sessions; and the most permissive level allows any inversion. These three levels correspond to new proposed correctness criteria: 1ASR, 1SR+ and 1SR$'$, respectively.
- Such division into different criteria is interesting because it removes all the possible confusion among users faced to one-copy equivalent databases: users now know exactly what to expect from the system, both in terms of transaction isolation and replica consistency.
- Different synchronization models based on different levels of server-centric consistency can be used to guarantee each of the proposed criteria. Designers, providers and administrators can then benefit from this distinction, as the exact level of replica consistency maintained among the set of replicas may have a notable influence upon the performance of the system.

- A fine-grained characterization model based on interactions, policies and strategies can precisely describe a database replication system while being open enough to cover the great majority of the existing systems.[1]

- By means of a metaprotocol, different combinations of policies, i.e., different replication protocols, can be executed concurrently in the same database system while achieving a performance similar[2] to that obtained by the original protocols executed in a stand-alone manner. Such concurrency allows the system to face heterogeneity, providing it with what we have called outward adaptability.

- The same metaprotocol can be used for the on-the-fly exchange of protocols, with a transition phase where protocols are executed in concurrency. This allows the system to adapt to dynamic conditions in the environment, the applications and the users, achieving forward adaptability.

- Not only transaction and replica consistency are important: semantic consistency defines constraints that must be enforced so that data appropriately reflects reality as needed by the semantics of the application. While stand-alone databases commonly support such integrity constraints, distributed databases usually do not consider them, which may lead to several anomalies of different gravity: from unnecessary abortions to replica divergence or loss of protocol liveness.

- Some replication protocols, maybe unintendedly, already avoid most of such integrity-related problems, allowing a seamlessly support for integrity constraints with minimum adaptations. Other protocols, however, require further extensions that penalize performance by preventing some well-known optimizations. Trading complete correctness in integrity support for less penalized performance, compromise solutions can be designed. Such solutions tolerate constraints without increasing costs but must assume some negative consequences such as unnecessary abortions.

The results presented in this thesis meet the established objectives and constitute a contribution to the state of the art of database replication at the time the respective

[1] Upon 60 different analyzed protocols, only 2 presented some difficulties to be described with the model, which represents a percentage over 96.6% of applicability of the model.

[2] From the studied combinations, only those mixing the weak voting and the active techniques suffered a significant performance penalty in our test bed.

works were started. Moreover, such results are also relevant because they open the door to prospective interesting contributions in different lines of work:

- In the light of the survey and based on the policies followed by different replication systems, a handful of criteria could give a non-expert designer some guidance on their election of the best protocol to use in their system, depending on the characteristics of the system, the applications or the workload.
- Starting from the characterization model and the survey, the metaprotocol could be further explored by studying the gains and penalties of the concurrency of different sets of protocols or, individually, of different sets of strategies for each policy.
- Based on the previous prospective results, different metrics could be identified as relevant in order to be measured by an on-line monitoring component able to deduce the likely optimal combination of policies for a given scenario in a dynamic and heterogeneous system that exploits the metaprotocol.
- Analyzing the combinations of strategies used by the systems studied in the survey, not yet explored combinations could be identified in order to evaluate their adequacy if new goals are set for replication protocols.
- The historical study of replicated databases may also help researchers to identify which advances at which fields allowed the appearance of each proposal, as well as to foresee which other advances could lead to new systems.
- New optimizations could allow integrity-aware database replication protocols to overcome the performance penalties identified in this thesis.

All these lines of work could lead to new and significant contributions that would join many other encouraging results of the research community in order to further improve and enrich the state of the art of database replication.

Conclusiones

Esta tesis está dedicada a la coherencia, la caracterización, la adaptabilidad y la integridad de los sistemas de replicación de bases de datos. Entre sus resultados, se pueden destacar las siguientes conclusiones:

- El criterio de corrección de 1SR, ampliamente aceptado y usado, puede subdividirse en tres niveles diferentes de coherencia desde el punto de vista del usuario, según la posibilidad de inversiones entre transacciones que presentan una precedencia de tiempo real entre ellas. El nivel más estricto evita todas las inversiones; el nivel intermedio evita inversiones en el contexto de las sesiones de usuario; y el nivel más permisivo permite cualquier inversión. Estos tres niveles corresponden con los nuevos criterios de corrección propuestos: 1ASR, 1SR+ y 1SR′, respectivamente.

- Esta división en diferentes criterios es interesante porque elimina toda posible confusión entre los usuarios de bases de datos equivalentes a una copia: los usuarios saben ahora exactamente qué esperar del sistema, tanto en términos de aislamiento de transacciones como de coherencia de réplicas.

- Para garantizar cada uno de los criterios propuestos, pueden usarse diferentes modelos de sincronización basados en distintos niveles de coherencia desde el punto de vista de los servidores. Los diseñadores, proveedores y administradores pueden así beneficiarse de esta distinción, ya que el nivel exacto de coherencia de réplicas que se mantiene entre el conjunto de servidores puede tener una influencia notable en el rendimiento del sistema.

- Un modelo de caracterización de grano fino y basado en interacciones, políticas y estrategias puede describir con precisión un sistema de replicación

de bases de datos y ser a la vez lo suficientemente abierto como para cubrir una amplia mayoría de los sistemas existentes.[3]

- A través de un metaprotocolo, se pueden ejecutar concurrentemente en el mismo sistema de bases de datos diferentes combinaciones de políticas, es decir, diferentes protocolos de replicación, consiguiendo un rendimiento similar[4] al conseguido por los protocolos originales ejecutados individualmente. Esta concurrencia permite al sistema afrontar la heterogeneidad, proporcionándole lo que hemos llamado adaptabilidad hacia fuera.

- El mismo metaprotocolo puede usarse para el intercambio en caliente de protocolos, con una fase de transición en la que ambos protocolos se ejecutan concurrentemente. Esto permite al sistema adaptarse a condiciones dinámicas en el entorno, en las aplicaciones y en los usuarios, proporcionándole adaptabilidad hacia delante.

- No sólo la coherencia de transacciones y de réplicas es importante: la coherencia semántica define restricciones que deben respetarse para que los datos reflejen apropiadamente la realidad, tal y como necesita la semántica de la aplicación. Mientras que las bases de datos centralizadas generalmente soportan tales restricciones de integridad, las bases de datos distribuidas suelen no considerarlas, lo que puede dar lugar a varias anomalías de diferente gravedad: desde abortos innecesarios a divergencia de las réplicas o pérdida de la viveza del protocolo.

- Algunos protocolos de replicación, quizá sin pretenderlo, ya evitan la mayoría de los problemas relacionados con la integridad, lo que permite un soporte directo de las restricciones de integridad con mínimos cambios. Otros protocolos, sin embargo, requieren mayores extensiones que penalizan el rendimiento al impedir ciertas optimizaciones bien conocidas. Sacrificando la completa corrección en el soporte de la integridad por un rendimiento menos penalizado, pueden diseñarse soluciones de compromiso. Estas soluciones toleran las restricciones sin incurrir en costes pero deben asumir algunas consecuencias negativas, como los abortos innecesarios.

[3] De los 60 protocolos analizados, sólo 2 presentaron dificultades a la hora de ser descritos con el modelo, lo que representa un porcentaje de aplicabilidad del modelo de más del 96,6 %.

[4] De las combinaciones estudiadas, sólo las que mezclan las técnicas de votación débil y de replicación activa sufrieron una penalización notable en su rendimiento en nuestro banco de pruebas.

Los resultados presentados en esta tesis cumplen los objetivos establecidos y constituyen una contribución al estado del arte de la replicación de bases de datos en el momento en que se iniciaron los trabajos respectivos. Estos resultados son relevantes, además, porque abren la puerta a posibles contribuciones futuras:

- A la vista del estudio del capítulo 5 y basándose en las políticas seguidas por diferentes sistemas de replicación, un pequeño conjunto de criterios podrían orientar a un diseñador no experto en su elección del mejor protocolo para usar en su sistema, dependiendo de las características del mismo, las aplicaciones o la carga de trabajo.
- A partir del modelo de caracterización y el estudio del capítulo 5, puede explorarse el metaprotocolo en mayor profundidad, analizando las ganancias y penalizaciones de la concurrencia de diversos conjuntos de protocolos o, individualmente, de diferentes conjuntos de estrategias para cada política.
- En base a los puntos anteriores, se pueden identificar diferentes métricas como relevantes para su medición por un componente de monitorización en línea capaz de deducir la combinación de políticas probablemente óptima para un escenario dado en un sistema dinámico y heterogéneo que explote el uso del metaprotocolo.
- Analizando las combinaciones de estrategias usadas por los sistemas estudiados en el capítulo 5, pueden identificarse combinaciones aún no exploradas con el fin de evaluar su adecuación en el caso de que se establezcan nuevos objetivos para los protocolos de replicación.
- El estudio histórico de los sistemas de bases de datos replicadas puede también ayudar a los investigadores a identificar qué avances en qué campos permitieron la aparición de cada propuesta, así como a predecir qué otros avances podrían dar lugar a nuevos sistemas.
- Nuevas optimizaciones podrían conseguir protocolos de replicación de bases de datos que sean conscientes de las restricciones de integridad pero que no sufran las penalizaciones de rendimiento identificadas en esta tesis.

Todas estas líneas de trabajo podrían dar lugar a nuevas y significativas contribuciones que se unirían a muchos otros alentadores resultados de la comunidad científica con el fin de mejorar y enriquecer aún más el estado del arte de la replicación de bases de datos.

Appendix A

On the Correctness of MeDRA

We provide here some hints about the correctness of our metaprotocol MeDRA. Although this appendix does not pretend to be a formal and complete correctness demonstration, it includes some guidelines that intuitively reason about the goodness of MeDRA.

A.1 Correctness Arguments for the Supported Protocols

Pedone et al. [105] presented a replication scheme based on atomic broadcast. The correctness proof for that replication protocol, as presented in the technical report version of that work, is based on the Multiversion Graph theorem [15]. The demonstration first proves that every two processes produce the same multiversion serialization graph and, then, it proves that every graph is acyclic, concluding that the proposed replication protocol guarantees one-copy serializability. This reasoning is directly applicable to each of the protocol classes currently supported by the metaprotocol, as they are also based on a total order broadcast, which determines the order on which transactions are validated and finally committed. Therefore, the same reasoning can be used to sketch an intuitive correctness proof for these three protocol families, which follow the same basic scheme as the algorithm by Pedone et al. [105]: (a) transactions are locally executed in one process without interaction with other processes (this step is void in active replication); and (b) when the transaction requests its commitment, its

writeset is propagated to the other processes, so that the transaction can be validated and, if possible, committed (in active replication, this step contains the full processing of the transaction). As each process receives the same input (i.e., transactions) in the same order (that of the atomic broadcast), and each process follows a deterministic algorithm to commit or abort transactions, the resulting history of applied transactions will be the same in all processes. As a result of these steps, each protocol achieves one-copy equivalence with natively sequential replica consistency.

Deployed upon a local DBMS that provides SI, and enforcing in their validation phase the rules for SI, these protocols guarantee, at least, a snapshot isolation level among all the system transactions. Active replication, for its part, increases the isolation level until the serializable one if transactions are executed in a non-overlapping manner.

A.2 Principle of No Interference

We claim that MeDRA does not interfere with the original behavior of the supported protocols when executed either as the only protocol inside the metaprotocol or in concurrence with other protocol families. The sequence of steps followed by the protocols, the rules that govern their decisions, and the behavior of their transactions are the same as in a stand-alone execution.

A.2.1 MeDRA Running Only One Protocol

The metaprotocol does not interfere nor it introduces any modification in the behavior of a supported protocol affecting to its isolation guarantees or its level of replica consistency. The execution of the stand-alone version and the version running inside MeDRA are identical. Indeed, the aspects of a system which contribute to the correctness criterion provided are: (a) the global concurrency control in collaboration with the local concurrency control of the underlying database; (b) the communication guarantees used to propagate the transaction information and the way writesets are applied; and (c) the rules for the validation phase of the protocol, if any. None of these change when using the metaprotocol.

A.2.2 MeDRA Running Multiple Protocols

Transactions from several or all the protocols coexist in the local database and in the queues of the middleware metaprotocol: *tocommit*, with all the delivered transactions pending to be processed in this node, and *log*, which includes the history of all transactions applied in this node. The isolation level provided by the DBMS is the same for all: snapshot isolation. During validation,[1] all concurrent transactions are considered, regardless of their protocol. In order to validate a transaction T_i in a node R_k, its writeset $T_i.wset$ is compared against the writesets of all concurrent transactions T_j that were previously delivered at R_k and that are or will be finally committed. Thus, we need to know which concurrent transactions committed or will commit and which are their writesets. An active transaction is not executed until it reaches the first position in the *tocommit* queue and, thus, its writeset is unknown before its commitment. A weak voting transaction in non-delegate replicas must wait to the arrival of the vote of the delegate to know if it will commit. These conditions create what we call dependencies between the transaction being validated and all previous transactions whose pending information prevents the validation from finishing. These dependencies are stored as part of the transaction information. As the *tocommit* queue is processed and active transactions are committed, and as voting messages arrive from delegate nodes, dependencies are resolved, leading to the abortion of the transaction if a conflict is detected or to its acceptance and later commitment otherwise. Dependencies introduce additional waiting times but liveness is ensured. The turn for an active transaction eventually arrives, as well as the voting message of a weak voting transaction. (In case of failure of a node, not originally considered, all replicas can safely abort all weak voting transactions whose delegate replica was the crashed one, i.e., the failure of a replica can be understood as a negative voting message for all its local transactions.) This way, the behavior of the protocols is not changed, only possibly delayed. The complete history of each replica will guarantee SI. Interleaved in this history, active transactions remain completely serialized between them but with snapshot isolation with regard to the rest of transactions.

[1] See Algorithm 4 of Appendix B.

A.3 Correctness criterion

On the basis of the principle of no interference, which states that the metaprotocol does not pervert the basic scheme used in the original protocols, even in case of concurrency, we can say that the correctness criterion provided by the metaprotocol prototype deployed upon a full replicated database where local DBMSs guarantee snapshot isolation by means of a multiversion concurrency control is 1SI. Indeed, the metaprotocol, based on the snapshot isolation of the underlying database and on the validation rules of the running protocols, ensures an isolation level of SI among all transactions executed in the system (higher isolation would involve the management of readsets). On the other hand, as update propagation and application is based on a total order broadcast and on a non-overlapping processing, server-centric replica consistency is natively sequential (#S). Finally, no mechanisms avoid the appearance of inversions. As a result, the correctness criterion is 1SI.

Appendix B

Pseudocode of MeDRA

Algorithms 1 to 13 reflect the pseudocode of the current MeDRA prototype.

Algorithm 1 Metaprotocol pseudocode (in replica R_k)

```
 1: Initialization:
 2:     for each available protocol P_q do
 3:         P_q.activated ← suitable(P_q)          ▷ true if P_q is currently suitable
 4:     end for
 5:     tocommit ← ∅                               ▷ transactions pending to commit in R_k
 6:     log ← ∅                                    ▷ history log of all system transactions
 7:     L-TOI ← 0   ▷ Total Order Index (TOI) of the last committed transaction
 8:     N-TOI ← 1                                  ▷ TOI for the next transaction to be delivered
 9:     GCcounter ← 0                              ▷ Counter for garbage collection heartbeat
10:
11: I. Protocol P_q has to be deactivated:                          ▷ deactivate a protocol
12:     P_q.activated ← false
13:     if P_q.locals = 0 then
14:         unload(P_q)
15:     end if
16:
17: II. Protocol P_q has to be activated:                           ▷ activate a protocol
18:     P_q.activated ← true
19:     load(P_q)
20:
```

Algorithm 2 Metaprotocol pseudocode (in replica R_k) (cont.)

```
III. New local request for a transaction T_i in protocol P_q:
    if P_q.activated then
        P_q.locals ← P_q.locals + 1
        call P_q to serve(T_i)                          ▷ request is granted
    else
        deny request
    end if

IV. A local P_q transaction terminates:
    P_q.locals ← P_q.locals − 1
    if (P_q.activated = false) ∧ (P_q.locals = 0) then
        unload(P_q)
    end if

V. P_q asks to send message M_n related to transaction T_i:
    if M_n contains a writeset then
                        ▷ from a certification-based or weak voting transaction
        T_i.bot ← L-TOI              ▷ logical timestamp of begin of transaction
    end if
    piggyback garbage collection info (L-TOI)   ▷ R_k committed until L-TOI
    GCcounter ← 0              ▷ reset counter for garbage collection heartbeat
    broadcast(M_n⟨T_i⟩)              ▷ in total order, except for voting messages

VI. Upon delivery of M_n related to transaction T_i:
    call T_i.protocol to process(M_n⟨T_i⟩)
    do garbage collection                                          ▷ step VIII

VII. Garbage collection heartbeat:
    if GCcounter = GCMAX then
          ▷ GCMAX must be set according to node capabilities, workload...
        broadcast garbage collection info (L-TOI)
                          ▷ no guarantees required (or reliable if GC is critical)
        GCcounter ← 0
    end if
```

Algorithm 3 Metaprotocol pseudocode (in replica R_k) (cont.)

```
56: VIII. Garbage collection:
57:     remove from the log until the last writeset applied by all replicas
58:
59: IX. Committing thread:
60:     T_i ← head(tocommit)                          ▷ oldest transaction in tocommit
61:     if T_i.status ≠ c-pending then                                ▷ T_i is committable
62:         if T_i.delegate ≠ R_k then
63:             call T_i.protocol to apply(T_i)       ▷ apply in non-delegate replicas
64:         end if
65:         call T_i.protocol to commit(T_i)                          ▷ commit in all replicas
66:         L-TOI ← T_i.toi                                           ▷ update L-TOI
67:         if T_i.status = w-pending then
68:                                 ▷ a w-pending transaction obtains its writeset now
69:             T_i.status ← resolved                 ▷ T_i is not w-pending any more
70:             resolve_w-dependencies(T_i)                           ▷ resolve dependencies
71:         end if
72:         tocommit ↪ T_i                                            ▷ delete T_i from tocommit
73:         GCcounter ← GCcounter + 1
74:                             ▷ increment counter for garbage collection heartbeat
75:     end if
76:
```

Algorithm 4 Common procedures (in replica R_k): validate

```
1:  procedure VALIDATE(T_i)
2:      ▷ check for write conflicts with concurrent previously delivered writesets
3:      for each T_j ∈ log such as T_j.toi > T_i.bot do          ▷ concurrent with T_i
4:          if (T_j.status = resolved) ∧ (T_j.wset ∩ T_i.wset ≠ ∅) then
5:                                                  ▷ a resolved conflicts with T_i
6:              remove_dependent(T_i)                  ▷ remove references to T_i
7:              return negative                  ▷ T_i did not pass its validation
8:          end if
9:          if (T_j.status = w-pending)∨
10:             ((T_j.status = c-pending) ∧ (T_j.wset ∩ T_i.wset ≠ ∅)) then
11:                 ▷ w-pending and conflicting c-pending cause dependencies
12:             T_j.dependent ↩ T_i                ▷ consider T_i as dependent on T_j
13:             T_i.dependencies ← T_i.dependencies + 1
14:                                                 ▷ increase dependency counter
15:         end if
16:     end for
17:     if T_i.dependencies > 0 then
18:         return pending                  ▷ some dependencies were found
19:     else
20:         return positive                          ▷ T_i successfully validated
21:     end if
22: end procedure
```

Algorithm 5 Common procedures (in replica R_k): remove_dependent

```
1:  procedure REMOVE_DEPENDENT(T_i)
2:                  ▷ T_i must abort, remove references from all dependent lists
3:      for each T_j ∈ log such as T_i ∈ T_j.dependent do
4:          T_j.dependent ↪ T_i                                  ▷ remove T_i
5:      end for
6:  end procedure
```

Algorithm 6 Common procedures (in replica R_k): resolve_w-dependencies

```
1:  procedure RESOLVE_W-DEPENDENCIES(T_j)
2:                ▷ T_j was a w-pending transaction and now its writeset is known
3:      for each T_i ∈ T_j.dependent do                ▷ for each dependent transaction
4:          T_j.dependent ↪ T_i                        ▷ remove T_i from T_j.dependent
5:          if T_i.wset ∩ T_j.wset = ∅ then            ▷ no conflicts between T_i and T_j
6:              T_i.dependencies ← T_i.dependencies − 1
7:                                                     ▷ decrease dependency counter
8:              if T_i.dependencies = 0 then           ▷ all dependencies are resolved
9:                  call T_i.protocol to set_resolved(T_i)        ▷ T_i is resolved
10:                 resolve_c-dependencies(T_i, commit)
11:                                                    ▷ resolve dependencies in cascade
12:             end if
13:         else                                       ▷ conflicts between T_i and T_j
14:             log ↪ T_i; tocommit ↪ T_i              ▷ remove T_i from both lists
15:             remove_dependent(T_i)                  ▷ remove references to T_i
16:             resolve_c-dependencies(T_i, abort)
17:                                                    ▷ resolve dependencies in cascade
18:             if T_i.delegate = R_k then
19:                 call T_i.protocol to rollback(T_i)        ▷ rollback in delegate
20:             end if
21:         end if
22:     end for
23: end procedure
```

Algorithm 7 Common procedures (in replica R_k): resolve_c-dependencies

```
1:  procedure RESOLVE_C-DEPENDENCIES(T_j, termination))
2:                ▷ T_j was a c-pending transaction and now its outcome is known
3:      for each T_i ∈ T_j.dependent do                ▷ for each dependent transaction
4:          T_j.dependent ↪ T_i                        ▷ remove T_i from T_j.dependent
5:          if termination = commit then               ▷ if T_j committed, T_i must abort
6:              log ↪ T_i; tocommit ↪ T_i              ▷ remove T_i from both lists
7:              remove_dependent(T_i)                  ▷ remove references to T_i
8:              resolve_c-dependencies(T_i, abort)
9:                                                     ▷ resolve dependencies in cascade
10:             if T_i.delegate = R_k then
11:                 call T_i.protocol to rollback(T_i)          ▷ rollback in delegate
12:             end if
13:         else                                ▷ T_j aborted, so the dependency is removed
14:             T_i.dependencies ← T_i.dependencies − 1
15:                                                    ▷ decrease dependency counter
16:             if T_i.dependencies = 0 then           ▷ all dependencies are resolved
17:                 call T_i.protocol to set_resolved(T_i)          ▷ T_i is resolved
18:                 resolve_c-dependencies(T_i, commit)
19:                                                    ▷ resolve dependencies in cascade
20:             end if
21:         end if
22:     end for
23: end procedure
```

Algorithm 8 Protocol modules (in replica R_k): active

```
1:  procedure SERVE(T_i)                          ▷ serve an incoming local request
2:      T_i.proxy ← R_k
3:      ask metaprotocol to send M_n⟨T_i⟩
4:          ▷ R_k acts as a proxy to broadcast the request on behalf of the client
5:  end procedure

6:  procedure PROCESS(M_n⟨T_i⟩)                    ▷ process an incoming message
7:      T_i.toi ← N-TOI                           ▷ assign total order index
8:      N-TOI ← N-TOI + 1                         ▷ increase N-TOI for next transaction
9:      T_i.status ← w-pending
10:                 ▷ active transactions are w-pending until commitment
11:     log ↩ T_i; tocommit ↩ T_i                 ▷ append T_i to both lists
12: end procedure

13: procedure APPLY(T_i)                          ▷ apply a transaction in the database
14:                 ▷ in active replication there is no writeset to apply
15: end procedure

16: procedure COMMIT(T_i)                         ▷ commit a transaction in the database
17:     DB.execute(T_i)                           ▷ call DBMS to execute T_i
18:     DB.commit(T_i)                            ▷ call DBMS to commit T_i
19:     if T_i.proxy = R_k then
20:         return to client
21:     end if
22: end procedure
```

Algorithm 9 Protocol modules (in replica R_k): weak voting

```
1:  procedure SERVE(T_i)                          ▷ serve an incoming local request
2:      T_i.delegate ← R_k
3:      while operation ≠ commit do
4:          let T_i execute the operation locally
5:                          ▷ in case of abortion, the service of T_i ends
6:      end while
7:      T_i.wset ← DB.writeset(T_i)               ▷ collect the writeset
8:      ask metaprotocol to send M_n⟨T_i⟩          ▷ broadcast the writeset
9:  end procedure
```

Algorithm 10 Protocol modules (in replica R_k): weak voting (cont.)

```
10: procedure PROCESS(M_n⟨T_i⟩)                          ▷ process an incoming message
11:     if M_n contains a writeset then
12:         T_i.toi ← N-TOI                                          ▷ assign total order index
13:         N-TOI ← N-TOI + 1                  ▷ increase N-TOI for next transaction
14:         if T_i.delegate = R_k then                                        ▷ T_i is local
15:             validation ← validate(T_i)                ▷ perform a validation process
16:             if validation = negative then          ▷ T_i conflicts with a resolved
17:                 rollback(T_i)              ▷ T_i must abort due to negative validation
18:             else
19:                 if validation = positive then  ▷ no conflicts and ∄ w-pending
20:                     set_resolved(T_i)                              ▷ T_i is now resolved
21:                 else                                          ▷ validation is pending
22:                     T_i.status ← c-pending
23:                 end if
24:                 log ↩ T_i; tocommit ↩ T_i                  ▷ append T_i to both lists
25:             end if
26:         else                                                      ▷ T_i is not local
27:             T_i.status ← c-pending
28:               ▷ non-delegate nodes mark it as c-pending until the vote arrives
29:             log ↩ T_i; tocommit ↩ T_i                      ▷ append T_i to both lists
30:         end if
31:     else                                                  ▷ M_n is a voting message
32:         if T_i.delegate ≠ R_k then                          ▷ ignore in the delegate
33:             if M_n.vote = commit then
34:                 T_i.status ← resolved          ▷ if vote is positive, T_i is resolved
35:             else                                                ▷ M_n.vote = abort
36:                 log ↪ T_i; tocommit ↪ T_i                ▷ delete T_i from both lists
37:             end if
38:             resolve_c-dependencies(T_i, M_n.vote)
39:                                   ▷ outcome is known, resolve dependencies
40:         end if
41:     end if
42: end procedure
```

Algorithm 11 Protocol modules (in replica R_k): weak voting (cont.)

```
43: procedure SET_RESOLVED(T_i)                    ▷ T_i is now resolved
44:     T_i.status ← resolved                      ▷ update status
45:     if T_i.delegate = R_k then
46:         vote(T_i, commit)                      ▷ emit a positive vote in the delegate
47:     end if
48: end procedure

49: procedure APPLY(T_i)                           ▷ apply a transaction in the database
50:     DB.apply(T_i.wset)                         ▷ call DBMS to apply the writeset of T_i
51: end procedure

52: procedure COMMIT(T_i)                          ▷ commit a transaction in the database
53:     DB.commit(T_i)                             ▷ call DBMS to commit T_i
54:     if T_i.delegate = R_k then
55:         return to client
56:     end if
57: end procedure

58: procedure ROLLBACK(T_i)                        ▷ rollback a transaction in the delegate
59:     vote(T_i, abort)                           ▷ emit a negative vote
60:     DB.rollback(T_i)                           ▷ call DBMS to rollback T_i
61:     return to client
62: end procedure
```

Algorithm 12 Protocol modules (in replica R_k): certification-based

```
1:  procedure SERVE(T_i)                          ▷ serve an incoming local request
2:      T_i.delegate ← R_k
3:      while operation ≠ commit do
4:          let T_i execute the operation locally
5:                                  ▷ in case of abortion, the service of T_i ends
6:      end while
7:      T_i.wset ← DB.writeset(T_i)                          ▷ collect the writeset
8:      ask metaprotocol to send M_n⟨T_i⟩                    ▷ broadcast the writeset
9:  end procedure

10: procedure PROCESS(M_n⟨T_i⟩)                   ▷ process an incoming message
11:     T_i.toi ← N-TOI                                      ▷ assign total order index
12:     N-TOI ← N-TOI + 1                  ▷ increase N-TOI for next transaction
13:     validation ← validate(T_i)  ▷ check conflicts with concurrent transactions
14:     if validation = negative then                ▷ T_i conflicts with a resolved
15:         if T_i.delegate = R_k then
16:             rollback(T_i)                        ▷ rollback T_i in the delegate
17:         else
18:             discard(T_i)                 ▷ discard T_i in non-delegate replicas
19:         end if
20:     else
21:         if validation = positive then      ▷ no conflicts and ∄ w-pending
22:             T_i.status ← resolved                        ▷ T_i is now resolved
23:         else                                          ▷ validation is pending
24:             T_i.status ← c-pending
25:         end if
26:         log ↩ T_i; tocommit ↩ T_i                    ▷ append T_i to both lists
27:     end if
28: end procedure
```

Algorithm 13 Protocol modules (in replica R_k): certification-based (cont.)

```
29: procedure SET_RESOLVED(T_i)                    ▷ T_i is now resolved
30:     T_i.status ← resolved                      ▷ update status
31: end procedure

32: procedure APPLY(T_i)                           ▷ apply a transaction in the database
33:     DB.apply(T_i.wset)                         ▷ call DBMS to apply the writeset of T_i
34: end procedure

35: procedure COMMIT(T_i)                          ▷ commit a transaction in the database
36:     DB.commit(T_i)                             ▷ call DBMS to commit T_i
37:     if T_i.delegate = R_k then
38:         return to client
39:     end if
40: end procedure

41: procedure ROLLBACK(T_i)                        ▷ rollback a transaction in the delegate
42:     DB.rollback(T_i)                           ▷ call DBMS to rollback T_i
43:     return to client
44: end procedure
```

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[2] Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, and Ioana Stanoi. Exploiting Atomic Broadcast in Replicated Databases (Extended Abstract). In *Proceedings of the 3rd International Euro-Par Conference on Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science (LNCS)*, pages 496–503, Passau (Germany), August 1997. Springer-Verlag.

[3] Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The Power of Processor Consistency. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 251–260, Velen (Germany), June 1993. ACM Press.

[4] Peter Alsberg and J. D. Day. A Principle for Resilient Sharing of Distributed Resources. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, pages 562–570, San Francisco, California (USA), October 1976. IEEE-CS Press.

[5] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Distributed Versioning: Consistent Replication for Scaling Back-End Databases of Dynamic Content Web Sites. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *Lecture Notes in Computer Science (LNCS)*, pages 282–304, Rio de Janeiro (Brazil), June 2003. Springer-Verlag.

[6] José Enrique Armendáriz-Íñigo, José Ramón Juárez-Rodríguez, José Ramón González de Mendívil, Hendrik Decker, and Francisco D. Muñoz-Escoí. *k*-bound GSI: A Flexible Database Replication Protocol. In

Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), pages 556–560, Seoul (Korea), March 2007. ACM Press.

[7] Hagit Attiya. Robust Simulation of Shared Memory - 20 Years After. *Bulletin of the EATCS*, (100):100–114, February 2010.

[8] Hagit Attiya and Jennifer L. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley & Sons, Inc., 2nd edition, 2004. ISBN 0-471-45324-2.

[9] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10, San Jose, California (USA), May 1995. ACM Press.

[10] Josep M. Bernabé-Gisbert, Raúl Salinas-Monteagudo, Luis Irún-Briz, and Francisco D. Muñoz-Escoí. Managing Multiple Isolation Levels in Middleware Database Replication Protocols. In *Proceedings of the 4th International Symposium on Parallel and Distributed Processing and Applications (ISPA)*, volume 4330 of *Lecture Notes in Computer Science (LNCS)*, pages 511–523, Sorrento (Italy), December 2006. Springer-Verlag.

[11] Philip A. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM (CACM)*, 39(2):86–98, February 1996.

[12] Philip A. Bernstein and Nathan Goodman. Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems. In *Proceedings of the 6th International Conference on Very Large Data Bases (VLDB)*, pages 285–300, Montreal, Quebec (Canada), October 1980. IEEE-CS Press.

[13] Philip A. Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[14] Philip A. Bernstein and Nathan Goodman. An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases. *ACM Transactions on Database Systems (TODS)*, 9(4):596–615, 1984.

[15] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN 0-201-10715-5.

[16] Bharat K. Bhargava, Abdelsalam Helal, Karl Friesen, and John Riedl. Adaptability Experiments in the RAID Distributed Database System. In *Proceedings of the 9th Symposium on Reliable Distributed Systems (SRDS)*, pages 76–85, Huntsville, Alabama (USA), October 1990. IEEE-CS Press.

[17] Yuri Breitbart, Hector Garcia-Molina, and Abraham Silberschatz. Overview of Multidatabase Transaction Management. *The VLDB Journal*, 1(2):181–239, October 1992.

[18] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The Primary-Backup Approach. In S. J. Mullender, editor, *Distributed Systems*, chapter 8, pages 199–216. Addison-Wesley, ACM Press, 2nd edition, 1993.

[19] Michael Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–350, Seattle, Washington (USA), November 2006. USENIX Association.

[20] Lásaro J. Camargos, Fernando Pedone, and Marcin Wieloch. Sprint: A Middleware for High-Performance Transaction Processing. In Ferreira et al. [46], pages 385–398.

[21] Stephanie J. 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*, pages 88–97, Portland, Oregon (USA), May 1989. ACM Press.

[22] David G. Campbell, Gopal Kakivaya, and Nigel Ellis. Extreme Scale with Full SQL Language Support in Microsoft SQL Azure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1021–1024, Indianapolis, Indiana (USA), June 2010. ACM Press.

[23] Michael J. Carey and Miron Livny. Conflict Detection Tradeoffs for Replicated Data. *ACM Transactions on Database Systems (TODS)*, 16(4):703–746, 1991.

[24] Francisco Castro-Company and Francisco D. Muñoz-Escoí. An Exchanging Algorithm for Database Replication Protocols. Technical Report ITI-ITE-07/02, Instituto Tecnológico de Informática, Valencia, Spain, 2007.

[25] Emmanuel Cecchet. C-JDBC: a Middleware Framework for Database Clustering. *IEEE Data Engineering Bulletin*, 27(2):19–26, June 2004.

[26] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *Proceedings of the FREENIX Track, USENIX Annual Technical Conference*, pages 9–18, Boston, Massachusetts (USA), June 2004. USENIX Association.

[27] Wen-Ke Chen, Matti A. Hiltunen, and Richard D. Schlichting. Constructing Adaptive Software in Distributed Systems. In Dasgupta and Zhao [34], pages 635–643.

[28] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys (CSUR)*, 33(4):427–469, December 2001.

[29] Vicent Cholvi and Josep Bernabéu. Relationships Between Memory Models. *Information Processing Letters*, 90(2):53–58, April 2004.

[30] Alfrânio Correia Jr., António Luís Sousa, Luís Soares, Francisco Moura, and Rui Carlos Oliveira. Revisiting Epsilon Serializabilty to improve the Database State Machine (Extended Abstract). In *Proceedings of the SRDS Workshop on Dependable Distributed Data Management (WDDDM)*, October 2004.

[31] Alfrânio Correia Jr., José Pereira, Luís Rodrigues, Nuno Carvalho, Ricardo Vilaça, Rui Carlos Oliveira, and Susana Guedes. GORDA: An Open Architecture for Database Replication. In *Proceedings of the 6th IEEE International Symposium on Network Computing and Applications (NCA)*, pages 287–290, Cambridge, Massachusetts (USA), July 2007. IEEE-CS Press.

[32] Alfrânio Correia Jr., José Pereira, and Rui Carlos Oliveira. AKARA: A Flexible Clustering Protocol for Demanding Transactional Workloads. In Meersman and Tari [86], pages 691–708.

[33] Carlo Curino, Evan P. C. Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Sam Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational Cloud: A Database-as-a-Service for the Cloud. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 235–240, Asilomar, California (USA), January 2011.

[34] Partha Dasgupta and Wei Zhao, editors. *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona (USA), April 2001. IEEE-CS Press.

[35] Khuzaima Daudjee and Kenneth Salem. Lazy Database Replication with Ordering Guarantees. In *Proceedings of the 20th International Conference on Data Engineering (ICDE)*, pages 424–435, Boston, Massachusetts (USA), March 2004. IEEE Computer Society.

[36] Khuzaima Daudjee and Kenneth Salem. Lazy Database Replication with Snapshot Isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 715–726, Seoul (Korea), September 2006. ACM Press.

[37] Rubén de Juan-Marín, María Idoia Ruiz-Fuertes, Jerónimo Pla-Civera, Luis H. García-Muñoz, and Francisco D. Muñoz-Escoí. On Optimizing Certification-Based Database Recovery Supporting Amnesia. In *XV Jornadas de Concurrencia y Sistemas Distribuidos (JCSD)*, pages 145–157, Torremolinos, Málaga (Spain), June 2007.

[38] Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. Revisiting the Paxos Algorithm. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG)*, volume 1320 of *Lecture Notes in Computer Science (LNCS)*, pages 111–125, Saarbrücken (Germany), September 1997. Springer-Verlag.

[39] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, San Francisco, California (USA), 2004. USENIX Association.

[40] Sameh Elnikety, Fernando Pedone, and Willy Zwaenepoel. Database Replication Using Generalized Snapshot Isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 73–84, Orlando, Florida (USA), October 2005. IEEE-CS Press.

[41] Sameh Elnikety, Steven G. Dropsho, and Fernando Pedone. Tashkent: Uniting Durability with Transaction Ordering for High-Performance Scalable Database Replication. In *Proceedings of the 2006 EuroSys Conference*, pages 117–130, Leuven (Belgium), April 2006. ACM Press.

[42] Sameh Elnikety, Steven G. Dropsho, and Willy Zwaenepoel. Tashkent+: Memory-Aware Load Balancing and Update Filtering in Replicated Databases. In Ferreira et al. [46], pages 399–412.

[43] Javier Esparza-Peidro, Antonio Calero-Monteagudo, Jordi Bataller, Francisco D. Muñoz-Escoí, Hendrik Decker, and José M. Bernabéu-Aubán. COPLA - a Middleware for Distributed Databases. In *Proceedings of the 3rd Asian Workshop on Programming Languages and Systems (APLAS)*, pages 102–113, Shanghai (China), November 2002.

[44] Javier Esparza-Peidro, Francisco D. Muñoz-Escoí, Luis Irún-Briz, and José M. Bernabéu-Aubán. RJDBC: A Simple Database Replication Engine. In *Proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS)*, pages 587–590, Porto (Portugal), April 2004.

[45] Alan Fekete and Krithi Ramamritham. Consistency Models for Replicated Data. In Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors, *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science (LNCS)*, pages 1–17. Springer-Verlag, 2010.

[46] Paulo Ferreira, Thomas R. Gross, and Luís Veiga, editors. *Proceedings of the 2007 EuroSys Conference*, Lisbon (Portugal), March 2007. ACM Press.

[47] Udo Fritzke Jr., Rodrigo Pereira Valentim, and Luiz Alberto Ferreira Gomes. Adaptive Replication Control Based on Consensus. In *Proceedings of the 2nd Workshop on Dependable Distributed Data Management (SDDDM)*, pages 1–10, Glasgow, Scotland (UK), March 2008. ACM Press.

[48] Seth Gilbert and Nancy A. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT News*, 33(2):51–59, June 2002.

[49] James R. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, SCI Committee, March 1989.

[50] Jim Gray. Notes on Database Operating Systems. In Rudolf Bayer, Robert M. Graham, and Gerhard Seegmüller, editors, *Advanced Course: Operating Systems*, volume 60 of *Lecture notes in Computer Science (LNCS)*, pages 393–481. Springer-Verlag, 1978.

[51] Jim Gray and Leslie Lamport. Consensus on Transaction Commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, March 2006.

[52] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Quebec (Canada), June 1996. ACM Press.

[53] Rachid Guerraoui, Benoît Garbinato, and Karim Mazouni. The GARF Library of DSM Consistency Models. In *Proceedings of the ACM SIGOPS European Workshop*, pages 51–56. ACM Press, 1994.

[54] Vassos Hadzilacos and Sam Toueg. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Technical Report 94-1425, Department of Computer Science, Cornell University, May 1994.

[55] Theo Härder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[56] Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, July 1990.

[57] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. The Performance of Database Replication with Group Multicast. In *Proceedings of the 29th IEEE International Symposium on Fault-Tolerant Computing Systems (FTCS)*, pages 158–165, Madison, Wisconsin (USA), June 1999. IEEE-CS Press.

[58] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. Partial Database Replication using Epidemic Communication. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 485–493, Vienna (Austria), July 2002. IEEE-CS Press.

[59] Luis Irún-Briz, Hendrik Decker, Rubén de Juan-Marín, Francisco Castro-Company, José Enrique Armendáriz-Iñigo, and Francisco D. Muñoz-Escoí. MADIS: A Slim Middleware for Database Replication. In *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science (LNCS)*, pages 349–359, Lisbon (Portugal), August 2005. Springer-Verlag.

[60] Hans-Arno Jacobsen, editor. *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, volume 3231 of *Lecture Notes in Computer Science (LNCS)*, Toronto (Canada), October 2004. Springer-Verlag.

[61] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Gustavo Alonso, and Sergio Arévalo. A Low-Latency Non-blocking Commit Service. In *Proceedings of the 15th International Conference on Distributed Computing (DISC)*, volume 2180 of *Lecture Notes in Computer Science (LNCS)*, pages 93–107, Lisbon (Portugal), October 2001. Springer-Verlag.

[62] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Bettina Kemme, and Gustavo Alonso. How to Select a Replication Protocol According to Scalability, Availability, and Communication Overhead. In *Proceedings of the 20th Symposium on Reliable Distributed Systems (SRDS)*, pages 24–33, New Orleans, Louisiana (USA), October 2001. IEEE-CS Press.

[63] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Bettina Kemme, and Gustavo Alonso. Improving the Scalability of Fault-Tolerant Database Clusters. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 477–484, Vienna (Austria), July 2002. IEEE-CS Press.

[64] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Gustavo Alonso, and Bettina Kemme. Are Quorums an Alternative for Data Replication? *ACM Transactions on Database Systems (TODS)*, 28(3):257–294, September 2003.

[65] José Ramón Juárez-Rodríguez, José Enrique Armendáriz-Íñigo, José Ramón González de Mendívil, Francisco D. Muñoz-Escoí, and José Ramón Garitagoitia. A Weak Voting Database Replication Protocol Providing Different Isolation Levels. In *Proceedings of the 7th International Conference on New Technologies of Distributed Systems (NOTERE)*, pages 261–268, Marrakech (Morocco), June 2007.

[66] Flavio Paiva Junqueira and Benjamin Reed. The life and times of a ZooKeeper. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, page 4, Calgary, Alberta (Canada), August 2009. ACM Press.

[67] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael

Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, August 2008.

[68] Bettina Kemme and Gustavo Alonso. Don't Be Lazy, Be Consistent: Postgres-R, a New Way to Implement Database Replication. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, pages 134–143, Cairo (Egypt), September 2000. Morgan Kaufmann.

[69] Bettina Kemme and Gustavo Alonso. A New Approach to Developing and Implementing Eager Database Replication Protocols. *ACM Transactions on Database Systems (TODS)*, 25(3):333–379, September 2000.

[70] Bettina Kemme, Fernando Pedone, Gustavo Alonso, and André Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS)*, pages 424–431, Austin, Texas (USA), May 1999. IEEE-CS Press.

[71] Bettina Kemme, Fernando Pedone, Gustavo Alonso, André Schiper, and Matthias Wiesmann. Using Optimistic Atomic Broadcast in Transaction Processing Systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(4):1018–1032, August 2003.

[72] Gilles M. E. Lafue. Semantic Integrity Dependencies and Delayed Integrity Checking. In *Proceedings of the 8th International Conference on Very Large Data Bases (VLDB)*, pages 292–299, Mexico City (Mexico), September 1982. Morgan Kaufmann.

[73] Leslie Lamport. On Interprocess Communication. *Distributed Computing*, 1(2):77–101, 1986.

[74] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News* (Distributed Computing Column), 32(4):18–25, December 2001.

[75] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28 (9):690–691, 1979.

[76] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[77] Butler W. Lampson. Atomic transactions. In Butler W. Lampson, M. Paul, and Hans-Jürgen Siegert, editors, *Advanced Course: Distributed Systems*, volume 105 of *Lecture Notes in Computer Science (LNCS)*, pages 246–265. Springer-Verlag, 1981.

[78] Butler W. Lampson. How to Build a Highly Available System Using Consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG)*, volume 1151 of *Lecture Notes in Computer Science (LNCS)*, pages 1–17, Bologna (Italy), October 1996. Springer-Verlag.

[79] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware based Data Replication providing Snapshot Isolation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 419–430, Baltimore, Maryland (USA), June 2005. ACM Press.

[80] Yi Lin, Bettina Kemme, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and José Enrique Armendáriz-Íñigo. Snapshot Isolation and Integrity Constraints in Replicated Databases. *ACM Transactions on Database Systems (TODS)*, 34(2):1–49, June 2009.

[81] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[82] Xiaoming Liu, Robbert van Renesse, Mark Bickford, Christoph Kreitz, and Robert L. Constable. Protocol Switching: Exploiting Meta-Properties. In Dasgupta and Zhao [34], pages 37–42.

[83] François Llirbat, Eric Simon, and Dimitri Tombroff. Using Versions in Update Transactions: Application to Integrity Checking. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, pages 96–105, Athens (Greece), August 1997. Morgan Kaufmann.

[84] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 105–120, San Francisco, California (USA), 2004. USENIX Association.

[85] Miguel Matos, Alfrânio Correia Jr., José Pereira, and Rui Carlos Oliveira. Serpentine: Adaptive Middleware for Complex Heterogeneous Distributed Systems. In Wainwright and Haddad [134], pages 2219–2223.

[86] Robert Meersman and Zahir Tari, editors. *Proceedings (Part I) of the OTM Confederated International Conferences*, volume 5331 of *Lecture Notes in Computer Science (LNCS)*, Monterrey (Mexico), November 2008. Springer-Verlag.

[87] Microsoft Corporation. Windows Azure: Microsoft's Cloud Services Platform. *http://www.microsoft.com/windowsazure*, 2011.

[88] Emili Miedes, María del Carmen Bañuls, and Pablo Galdámez. Group Communication Protocol Replacement for High Availability and Adaptiveness. In *XIII Jornadas de Concurrencia y Sistemas Distribuidos (JCSD)*, pages 271–276. Thomson Paraninfo, 2005.

[89] Jesús M. Milán-Franco, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Bettina Kemme. Adaptive Middleware for Data Replication. In Jacobsen [60], pages 175–194.

[90] José Mocito and Luís Rodrigues. Run-Time Switching Between Total Order Algorithms. In *Proceedings of the 12th International Euro-Par Conference on Parallel Processing*, volume 4128 of *Lecture Notes in Computer Science (LNCS)*, pages 582–591, Dresden (Germany), August 2006. Springer-Verlag.

[91] David Mosberger. Memory Consistency Models. *ACM SIGOPS Operating Systems Review*, 27(1):18–26, 1993.

[92] Francisco D. Muñoz-Escoí, Luis Irún-Briz, Pablo Galdámez, José María Bernabéu-Aubán, Jordi Bataller, and María del Carmen Bañuls. Consistency Protocols in GlobData. In *X Jornadas de Concurrencia y Sistemas Distribuidos (JCSD)*, pages 165–178, Jaca (Spain), June 2002.

[93] Francisco D. Muñoz-Escoí, Jerónimo Pla-Civera, María Idoia Ruiz-Fuertes, Luis Irún-Briz, Hendrik Decker, José Enrique Armendáriz-Íñigo, and José Ramón González de Mendívil. Managing Transaction Conflicts in Middleware-based Database Replication Architectures. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 401–410, Leeds (UK), October 2006. IEEE-CS Press.

[94] Francisco D. Muñoz-Escoí, Rubén de Juan-Marín, José Enrique Armendáriz-Iñigo, and José Ramón González de Mendívil. Persistent Logical Synchrony. In Werner [135], pages 253–258.

[95] Francisco D. Muñoz-Escoí, Josep M. Bernabé-Gisbert, Rubén de Juan-Marín, José Enrique Armendáriz-Iñigo, and José Ramón González de Mendívil. Revising 1-Copy Equivalence in Replicated Databases with Snapshot Isolation. In *Proceedings (Part I) of the OTM Confederated International Conferences*, volume 5870 of *Lecture Notes in Computer Science (LNCS)*, pages 467–483, Vilamoura (Portugal), November 2009. Springer-Verlag.

[96] Rui Carlos Oliveira, José Pereira, Alfrânio Correia Jr., and Edward Archibald. Revisiting 1-Copy Equivalence in Clustered Databases. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC)*, pages 728–732, Dijon (France), April 2006. ACM Press.

[97] Esther Pacitti, Pascale Minet, and Eric Simon. Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, pages 126–137, Edinburgh, Scotland (UK), September 1999. Morgan Kaufmann.

[98] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. Scalable Replication in Database Clusters. In *Proceedings of the 14th International Conference on Distributed Computing (DISC)*, volume 1914 of *Lecture Notes in Computer Science (LNCS)*, pages 315–329, Toledo (Spain), October 2000. Springer-Verlag.

[99] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R: Consistent Database Replication at the Middleware Level. *ACM Transactions on Computer Systems (TOCS)*, 23(4): 375–423, 2005.

[100] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 165–178, Providence, Rhode Island (USA), June 2009. ACM Press.

[101] Fernando Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, 1999.

[102] Fernando Pedone and Svend Frølund. Pronto: A Fast Failover Protocol for Off-the-shelf Commercial Databases. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 176–185, Nurnberg (Germany), October 2000. IEEE-CS Press.

[103] Fernando Pedone and André Schiper. Optimistic Atomic Broadcast. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*, volume 1499 of *Lecture Notes in Computer Science (LNCS)*, pages 318–332, Andros (Greece), September 1998. Springer-Verlag.

[104] Fernando Pedone, Rachid Guerraoui, and André Schiper. Transaction Reordering in Replicated Databases. In *Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 175–182, Durham, North Carolina (USA), October 1997. IEEE-CS Press.

[105] Fernando Pedone, Rachid Guerraoui, and André Schiper. Exploiting Atomic Broadcast in Replicated Databases. In *Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, volume 1470 of *Lecture Notes in Computer Science (LNCS)*, pages 513–520, Southampton (UK), September 1998. Springer-Verlag.

[106] Jerónimo Pla-Civera, María Idoia Ruiz-Fuertes, Luis H. García-Muñoz, and Francisco D. Muñoz-Escoí. Optimizing Certification-Based Database Recovery. In *Proceedings of the 6th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 211–218, Hagenberg (Austria), July 2007. IEEE-CS Press.

[107] Christian Plattner and Gustavo Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In Jacobsen [60], pages 155–174.

[108] PostgreSQL Global Development Group. PostgreSQL Database Management System. *http://www.postgresql.org*, 2011.

[109] Yoav Raz. The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment. In *Proceedings of the 18th*

International Conference on Very Large Data Bases (VLDB), pages 292–312, Vancouver, British Columbia (Canada), August 1992. Morgan Kaufmann.

[110] Luís Rodrigues, Nuno Carvalho, and Emili Miedes. Supporting Linearizable Semantics in Replicated Databases. In Werner [135], pages 263–266.

[111] Daniel J. Rosenkrantz, Richard Edwin Stearns, and Philip M. Lewis II. System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems (TODS)*, 3(2):178–198, 1978.

[112] María Idoia Ruiz-Fuertes, Jerónimo Pla-Civera, José Enrique Armendáriz-Iñigo, José Ramón González de Mendívil, and Francisco D. Muñoz-Escoí. Revisiting Certification-Based Replicated Database Recovery. In *Proceedings (Part I) of the OTM Confederated International Conferences*, volume 4803 of *Lecture Notes in Computer Science (LNCS)*, pages 489–504, Vilamoura (Portugal), November 2007. Springer-Verlag.

[113] Olivier Rütti, Paweł T. Wojciechowski, and André Schiper. Structural and Algorithmic Issues of Dynamic Protocol Update. In *Proceedings of the IEEE 20th International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island (Greece), April 2006. IEEE-CS Press.

[114] Raúl Salinas, Josep M. Bernabé-Gisbert, Francisco D. Muñoz-Escoí, José Enrique Armendáriz-Íñigo, and José Ramón González de Mendívil. SIRC: A Multiple Isolation Level Protocol for Middleware-based Data Replication. In *Proceedings of the 22nd International Symposium on Computer and Information Sciences (ISCIS)*, pages 1–6, Ankara (Turkey), November 2007. IEEE-CS Press.

[115] Raúl Salinas, Francisco D. Muñoz-Escoí, José Enrique Armendáriz-Íñigo, and José Ramón González de Mendívil. A Performance Evaluation of g-Bound with a Consistency Protocol Supporting Multiple Isolation Levels. In *Proceedings of the OTM Confederated International Workshops and Posters*, volume 5333 of *Lecture Notes in Computer Science (LNCS)*, pages 914–923, Monterrey (Mexico), November 2008. Springer-Verlag.

[116] Nicolas Schiper, Rodrigo Schmidt, and Fernando Pedone. Optimistic Algorithms for Partial Database Replication. In *Proceedings of the 10th International Conference on Principles of Distributed Systems (OPODIS)*, volume 4305 of *Lecture Notes in Computer Science (LNCS)*, pages 81–93, Bordeaux (France), December 2006. Springer-Verlag.

[117] Fred B. Schneider. Synchronization in Distributed Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):125–148, April 1982.

[118] Damián Serrano, Marta Patiño-Martínez, Ricardo Jiménez-Peris, and Bettina Kemme. Boosting Database Replication Scalability through Partial Replication and 1-Copy-Snapshot-Isolation. In *Proceedings of the 13th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 290–297, Melbourne, Victoria (Australia), December 2007. IEEE-CS Press.

[119] Mukul K. Sinha, P. D. Nanadikar, and S. L. Mehndiratta. Timestamp Based Certification Schemes for Transactions in Distributed Database Systems. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, pages 402–411, Austin, Texas (USA), May 1985. ACM Press.

[120] Dale Skeen. Nonblocking Commit Protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, pages 133–142, Ann Arbor, Michigan (USA), April 1981. ACM Press.

[121] António Luís Sousa, Rui Carlos Oliveira, Francisco Moura, and Fernando Pedone. Partial Replication in the Database State Machine. In *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA)*, pages 298–309, Cambridge, Massachusetts (USA), October 2001. IEEE-CS Press.

[122] Spread Concepts LLC. The Spread Toolkit. *http://www.spread.org/*, 2009.

[123] Ioana Stanoi, Divyakant Agrawal, and Amr El Abbadi. Using Broadcast Primitives in Replicated Databases (abstract). In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, page 283, Santa Barbara, California (USA), August 1997. ACM Press.

[124] Robert C. Steinke and Gary J. Nutt. A Unified Theory of Shared Memory Consistency. *Journal of the ACM*, 51(5):800–849, September 2004.

[125] Michael Stonebraker. The Case for Shared Nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, March 1986.

[126] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, 1995. ISBN 0131439340.

[127] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems. Principles and Paradigms*. Prentice-Hall, 2002. ISBN 0130888931.

[128] Christophe Taton, Noel De Palma, and Sara Bouchenak. Adaptive Middleware for Message Queuing Systems. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 47–50. Springer-Verlag, 2009.

[129] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 140–149, Austin, Texas (USA), September 1994. IEEE-CS Press.

[130] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, and Bruce G. Lindsay. Transactions and Consistency in Distributed Database Systems. *ACM Transactions on Database Systems (TODS)*, 7(3):323–342, September 1982.

[131] Kimberly L. Tripp and Neal Graves. SQL Server 2005 Row Versioning-Based Transaction Isolation. Technical report, Microsoft, 2006.

[132] Luís Veiga and Paulo Ferreira. RepWeb: Replicated Web with Referential Integrity. In *Proceedings of the 2003 ACM Symposium on Applied Computing (SAC)*, pages 1206–1211, Melbourne, Florida (USA), March 2003. ACM Press.

[133] Werner Vogels. Eventually Consistent. *Communications of the ACM (CACM)*, 52(1):40–44, January 2009.

[134] Roger L. Wainwright and Hisham Haddad, editors. *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC)*, Fortaleza, Ceara (Brazil), March 2008. ACM Press.

[135] Bob Werner, editor. *Proceedings of the 7th IEEE International Symposium on Network Computing and Applications (NCA)*, Cambridge, Massachusetts (USA), July 2008. IEEE-CS Press.

[136] Matthias Wiesmann and André Schiper. Comparison of Database Replication Techniques Based on Total Order Broadcast. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 17(4):551–566, April 2005.

[137] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Understanding Replication in Databases and Distributed Systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS)*, pages 464–474, Taipei (Taiwan), April 2000. IEEE-CS Press.

[138] Matthias Wiesmann, André Schiper, Fernando Pedone, Bettina Kemme, and Gustavo Alonso. Database Replication Techniques: A Three Parameter Classification. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 206–215, Nurnberg (Germany), October 2000. IEEE-CS Press.

[139] Vaidė Zuikevičiūtė and Fernando Pedone. Correctness Criteria for Database Replication: Theoretical and Practical Aspects. In Meersman and Tari [86], pages 639–656.

[140] Vaidė Zuikevičiūtė and Fernando Pedone. Conflict-Aware Load-Balancing Techniques for Database Replication. In Wainwright and Haddad [134], pages 2169–2173.

[141] Vaidė Zuikevičiūtė and Fernando Pedone. Revisiting the Database State Machine Approach. In *Proceedings of the VLDB Workshop on Design, Implementation and Deployment of Database Replication*, Trondheim (Norway), August 2005.