Sapienza Università di Roma

Dottorato di Ricerca in Ingegneria Informatica

XXVI Ciclo – 2014

# Efficient Protocols for Replicated Transactional Systems

Sebastiano Peluso

Sapienza Università di Roma

Dottorato di Ricerca in Ingegneria Informatica

XXVI Ciclo - 2014

Sebastiano Peluso

# Efficient Protocols for Replicated Transactional Systems

Author's address:
Sebastiano Peluso
Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Sapienza Università di Roma
Via Ariosto 25, I-00185 Roma, Italy
e-mail: `peluso@dis.uniroma1.it`
www: `http://www.dis.uniroma1.it/~peluso`

*To my family...*

# Acknowledgments

The first person who I want to thank is my friend and valuable colleague Diego Didona. This is because I had the incentive to start my doctoral studies thanks to his desire to always do all the best in his work. In addition the hard work that we carried out together for our Master's thesis and the support he gave to me during my first PhD year were essential, and this dissertation would not probably exist without the initial collaboration with him.

On the other hand I think that no appropriate words are sufficient to thank my two advisors prof. Francesco Quaglia and prof. Paolo Romano, but I do my best to thank them anyway. I am so lucky to know Francesco and Paolo and I am really proud of having worked with them. Both luck and pride are the best words to describe what these two extraordinary persons did for me. I talk about luck because it is thanks to their hard work, their full availability, the way they taught me to face problems and the desire of always seeking a solution to any problem, that I successfully achieved the goals of my doctorate. Furthermore, I am proud because I learnt from them something I could have learnt from few people in the world: Francesco and Paolo taught me that honesty and love for the science have to be the only principles regulating the activities in our work. Only thanks to the honesty you can really show your true value, and only thanks to the love for science you can obtain good results. Thank you guys!

About love and honesty, I have to thank those who first taught me how much love and honesty are important in our life. Thank Maria Rosaria (my mother), Antonio (my father), Andrea and Gianluca (my brothers and best friends). The results of this dissertation and all I have in my life are the outcome of what I learn from all of you everyday. I say thanks because every single word in this document has been supported by your sweat and tears, and by your smiles.

In addition, big thanks are for my "big family" too: Maria, Gaetano, Franca, Salvatore, Lina, Stefano, Miryam and Marialavia.

I am also grateful to prof. Pascal Felber and prof. Fernando Pedone for having accepted to serve as external reviewers of this dissertation, and to prof. Leonardo Querzoni for the support during the evolution of this thesis.

I thank my research group in Italy by starting from the leader and scientific father of the group: prof. Bruno Ciciani. Bruno shaped the group as a family, and this was so important in order to have a great environment to work in. Therefore thank my friends and colleagues of the group, i.e., Pierangelo Di Sanzo, Roberto Palmieri, Alessandro Pellegrini, Diego Rughetti and Roberto Vitali, you always made me happy even during the hardest workdays.

Special thanks go to Roberto Palmieri and Marina Sadini for the support they gave to me during the period I wrote this dissertation and at the beginning of my new american adventure.

I would like to thank all people worked with me during the period I spent in Lisbon (Portugal): special thanks go to prof. Luís Rodrigues, and thank my colleagues Nuno Carvalho, Maria Couceiro, Oksana Denysyuk, Diego Didona, João Fernandes, João Leitão, João Paiva, Liliana Rosa and Pedro Ruivo.

Finally I sincerely thank those who had a place in my life and supported and bore my mood in their own way during my doctoral studies. Therefore thank Canzanella family (Mario, Antonietta, Giulio and Silvia) and my friends Margherita D'Errico and Michela D'Errico.

Sebastiano Peluso

# Abstract

Over the last years several relevant technological trends have significantly increased the relative impact that the inter-replica synchronization costs have on the performance of transactional systems. Indeed, the emergence of technologies like Transactional Memory, Solid-State Drives and Cloud computing has exacerbated the ratio between the latencies of replication coordination and transaction processing. The requirements of these environments harshly challenge state of the art techniques for replication of transactional systems, raising the need for rethinking existing approaches to this problem.

This dissertation advances the state of the art on replicated transactional systems by presenting a set of innovative replication protocols designed to achieve high efficiency even in such challenging scenarios.

More in detail, four transactional replication protocols are proposed, which tackle the aforementioned issues from various angles. The first two cope with full replication scenarios, and exploit orthogonal techniques, such as speculation and transaction migration, which allow for amortizing, in different ways, the impact of distributed coordination on system performance. The other two proposals explicitly cope with the issue of scalability, by introducing the first genuine partial replication techniques that support abort-free read-only transactions while ensuring, respectively, One-Copy Serializability and Extended Update Serializability. The core of these protocols is a distributed multi-version concurrency control algorithm, which relies on a novel logical clock synchronization mechanism to track, in a totally decentralized (and consequently scalable) way, both data and causal dependency relations among transactions. The trade-offs arising across the different presented solutions are also discussed and experimentally evaluated by integrating them into state of the art academic and industrial transactional platforms.

Most of the material presented in this dissertation can also be found in the following papers:

1. Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia and Luís Rodrigues
*When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication*
In Proc. of the 32nd IEEE International Conference on Distributed Computing Systems (ICDCS), pages 455–465. Macau, China, June 2012.

2. Sebastiano Peluso, João Fernandes, Paolo Romano, Francesco Quaglia and Luís Rodrigues
*SPECULA: Speculative Replication of Software Transactional Memory*
In Proc. of the 31st IEEE International Symposium on Reliable Distributed Systems (SRDS), pages 91–100. Irvine, California, USA, October 2012.

3. Sebastiano Peluso, Paolo Romano and Francesco Quaglia
*SCORe: a Scalable One-Copy Serializable Partial Replication Protocol*
In Proc. of the ACM/IFIP/USENIX 13th International Conference on Middleware (Middleware), pages 456–475. Montréal, Québec, Canada, December 2012.

4. Danny Hendler, Alex Naiman, Sebastiano Peluso, Paolo Romano, Francesco Quaglia and Adi Suissa
*Exploiting Locality in Lease-Based Replicated Transactional Memory via Task Migration*
In Proc. of the 27th International Symposium on Distributed Computing (DISC), pages 121–133. Jerusalem, Israel, October 2013.

5. Hugo Pimentel, Paolo Romano, Sebastiano Peluso and Pedro Ruivo
*Enhancing locality via caching in the GMU protocol*
In Proc. of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pages 473–482. Chicago, IL, USA, May 2014.

# Contents

# Chapter 1

# Introduction

The explosion of web applications' usage is allowing companies to easily break the wall of national boundary, making their services available to any user in the world. On the one hand, the popularity and the productivity of those services increases significantly due to this ease and wide deployment. On the other hand, the IT systems behind these services have to face the challenge of processing an ever growing volume of requests.

Despite the different types of requests, almost all workloads trigger the execution of procedures for querying (i.e., read interactions) or manipulating (i.e., write interactions) common application state. In this context, a crucial, and long studied, problem is handling concurrent data manipulations efficiently and preserving the consistency of application state, despite multiple simultaneous requests.

Another objective is to keep the service usable, i.e. guaranteeing low user perceived latency, while yielding high service throughput. In addition, since a lot of companies and enterprises base their success on the IT market, or in general, they rely on systems for enlarging their user base, computing systems should meet dependability requirements and ensure the survival of both data and services in case of failures.

Concurrent data accesses are widely managed by means of the transaction abstraction, a well established technique in Database Management Systems (DBMS) that has recently emerged also in the context of parallel programming via the transaction memory paradigm. In this way, applications enclose accesses on shared data, e.g. write and read operations on tables in a database, or simple objects in main memory, within the boundaries of so called transactions. Then the *concurrency control* module is responsible for ensuring that, despite their parallel activation, transactions appear as if they were executed in isolation and atomically (i.e., either all or none of a transaction's operations take effect), thereby allowing only safe and consistent transitions of the application state [12].

On the other hand, data and service replication is a widely adopted technique for dependability, and it is recognized as a practical and effective way for enhancing the availability and fault-tolerance of computer systems. User perceived latency can be reduced by exploiting application locality or by migrating data closer to the requests' source.

Replication applied to transactional systems has been already successfully consolidated in the literature as the reference methodology for building available, fault-tolerant and high performance data management systems. However, current replication protocols for transactional systems do not represent a definitive solution when new requirements arise due to novelties in the architectural trends. The following Section 1.1 discusses (i) the most relevant trends currently driving the process of reorganizing the architecture of transactional systems and (ii) the major shortcoming of state of the art transactional replication protocols in combination with these new trends.

## 1.1 The Need for Rethinking Transactional Replication

During the last years we have seen an evolution of the technological features associated with computing systems and infrastructures, which entailed the need for a reassessment of several solutions specifically tailored for settings and environments that do not match anymore the current ones.

Concerning transactional replication, in fact, there are factors that are concurring to a remarkable reduction of the ratio between effective transactional execution time and replica synchronization time, thus assigning to replication protocols a predominant role in the transactions lifetime. These factors are determined by architectural evolutions that involve the computational resources and the storage components, as well as the system scale, and they have an impact on both the dividend and divisor of the aforementioned quotient. If on one hand transactional executions became more lightweight, because transactions are increasingly used as a means for synchronization in concurrent programming (thus often requiring the accesses to few locations in main memory), and the advent of new storage components made the local execution of even classical transactional profiles faster, on the other hand the replicas synchronization became more expensive due to the trend of increasing the systems scale in order to meet dependability requirements of applications, thus entailing a potential growing number of replicas to be contacted per transaction execution.

Therefore, in this dissertation, three relevant technological changes have been identified, which have posed to the systems designers the need for rethinking transactional replication due to the aforementioned exacerbation of synchronization costs over transactions execution costs. These are (i) the multi-core paradigm that has led to the advent of the Transactional Memory (TM) programming paradigm, (ii) the Solid-State Drive technology that has allowed the implementation of faster storage components and (iii) the scaling of the systems, also enabled by the flexibility of the cloud computing paradigm.

The multi-core paradigm is definitely one of the most disruptive among them. In the last decade, in fact, we have seen the proliferation of multi-processor and multi-core architectures due to physical constraints which place limits on the processor clock: boosting applications performances can no longer be achieved exploiting the increase of processors clock speed and consequently the so called *multi-core revolution* [48] started.

This architectural evolution raised the challenge of how to effectively exploit the computational power of multi-core processors. As a consequence parallel programming, traditionally confined to the niche of high performance computing, stepped into the realms of mainstream application development.

However, parallel programming brings about a number of additional sources of complexity, which can hamper not only applications' performance, but also their development cost, reliability and time to market. Factors that are essential in a market as competitive as the current one.

These considerations have motivated, over the last years, an intense research activity aimed at investigating novel programming paradigms capable of simplifying the development of parallel applications.

One of the most crucial issues to tackle when developing parallel applications is related to how to manage concurrent manipulations to the shared state of the application. The challenge here is to identify mechanisms capable of ensuring adequate consistency levels while being: (i) simple and familiar for the programmers, (ii) highly efficient and scalable, (iii) resilient to failures.

In fact, traditional lock-based solutions have well-known pitfalls: on one hand simplistic *coarse-grained* locking schemes can drastically reduce the parallelism of the applications [2], while on the other hand more sophisticated *fine-grained* locking schemes are complex to design, reason about, verify and debug; in addition, composability [46], an essential principle at the basis of modern software engineering, when using fine-grained locking cannot be easily achieved in a way that avoids deadlocks and dataraces [2].

The *Transactional Memory* programming paradigm, first introduced by Herlihy and Moss [50] and then proposed in its software version, i.e. *Software Transactional Memory* (STM) [32] by Shavit and Touitou [90], is gaining momentum as a promising alternative to locks in concurrent programming. In particular, Transactional Memory middlewares provide programmers with the well-known concept of *transaction*, thus freeing them from the burden of dealing with lower level details underlying synchronization among concurrent operations.

However apart from being a useful concurrent programming abstraction in stand-alone multi-core applications, STMs are very attractive in the context of enterprise systems. In fact, traditional enterprise systems are structured according to a multi-tier architecture in which the logic resides on a replicated middle-tier and relies on back-end relational databases to ensure not only data persistency, but also consistency in presence of concurrent data manipulations. As a result, since centralized DBMSs represent an evident scalability bottleneck in this kind of architectures, emerging alternative system configurations that make use of in-memory caching at the middle-tier are adopted, alleviating the frequency of accesses to the back-end but raising also the issue of how to ensure the consistency of state concurrently observed/manipulated. For this reason STM technologies have in fact been recently integrated in replicated caching frameworks (see, e.g., FénixEDU system [17] and Infinispan [65]) due to the main feature of providing an in-main-memory transactional environment. In addition, this last described feature combined with simple and generic interfaces make the distributed STM flexible enough to be employed as a key building block for data management in large-scale cloud infrastructures.

Spanning on the recent technological innovations, another potential rupture point is related to the evolution of storage components that recently are directly affecting systems performance. In particular we are observing the advent of Solid-State Drive (SSD) technology that is starting to massively replace the well known magnetic-based technology used in the classical Hard Disk Drives (HHD). SSDs do not employ any moving mechanical component, e.g. magnetic platters and actuator arms, thus offering reduced access time/latency and random I/O performance of orders of magnitude better than the one of conventional hard drives.

As a consequence, this evolutionary trend is having a relevant impact for systems, such as DBMSs, that massively generate access requests to the storage components: for these systems the execution costs of transactions would be dramatically reduced, at a speed close to that of accessing data directly in main-memory (resulting in a behavior typical of transactional memories).

Another dominant trend is surely the scaling of the systems in order to meet the services requirements of handling an ever-growing user base. These systems, in fact, may replicate application state even across geographically distributed sites and redirect requests to the closest sites or on the basis of the load distribution in order to try to reduce the user-perceived latency and hence to improve user experience. For instance cloud computing is an example of technology that enables building scalable services through the so called scale-out model by utilizing the elastic pool of computing resources. However as services migrate to these types of large-scale architectures, they force the data management community to face the challenge of designing transactional and storage systems that are capable to scale despite the potentially high communication delays, due to the relevant number of replicas to be synchronized during the updates of the transactional state.

Therefore this evolution entails the deployment of large-scale transactional systems in which severe overheads obstruct the feasibility of adopting classical distributed concurrency control schemes due to the high cost of assuring the coherence among replicated data across all the sites: the replica coordination during the execution of a transaction can be very expensive if compared to the effective execution time of that transaction, and communication costs have a high impact on the transactions' lifetime, thus limiting scalability.

For this reason, to avoid paying the penalty of synchronizing concurrent transactions across all data sites, some distributed data platforms are based on design approaches consisting in the adoption of relaxed data-consistency models, such as eventual consistency [27] and non-serializable isolation levels [11], or restricted transactional semantics, such as single object transactions [58] and static transactions [4]. If on one hand these schemes have been shown to yield significant performance advantages with respect to classic strongly consistent transactional paradigms, on the other hand they add complexity for the programmers, who have to reason on the correctness of complex applications in presence of weak consistency guarantees and/or may need to identify non-trivial work-around solutions to circumvent the limitations of constrained programming paradigms.

Therefore recent architectural evolutions, i.e. multi-core architectures, solid state storage systems and large-scale systems, have generated a significant change in both the structural (e.g. workload types) and non-functional (e.g. performance/response time) characteristics of transactional processing: (i) there are new applications of the classical transactional paradigm that have widened the nature of transactional operations [82, 41], e.g. a means to synchronize computational threads/processes, which become much simpler, e.g. a couple of read and write to a variable in main-memory vs. a complex SQL query evaluation; (ii) the execution times of transactional profiles already adopted in conventional database management systems are noticeably reduced when these systems are deployed on new generation storage components; (iii) the time required for synchronization among the nodes of a distributed transactional system on transactions execution can be noticeably high due to the high number of replicas that need to agree on the transactions' updates.

For this reason the just outlined technological metamorphosis of the computing systems should be seriously considered as a double-sided weapon from the point of view of a system designer: the common benefit in the reduction of the effective execution time associated with a sequence of transactional operations does not necessarily entail an equivalent reduction of the overall transactions' handling time in production distributed environments; on the contrary this process exacerbates the impact of the costs related to existing replication protocols, thus inexorably limiting scalability of complex large-scale transactional systems.

A proof of a real demand to overcome this issue is supported by several studies on the exploitation of a set of techniques adopted to reduce the impact of replicas coordination in transactions execution. On one hand *speculative* techniques are explored to overlap the transactions computation and the replicas synchronization in order to alleviate the replication costs in workload dominated by short-lived transactions [84, 69]. On the other hand, different replication techniques are designed in order to maintain low the replication latency even in large-scale systems via the exploitation of *Genuine Partial Replication* (GPR) [86] as well as the reliance on weak communication primitives, e.g. no total order on messages delivery, and per-node commit grants, i.e. *leases*, in presence of application locality well partitioned among replicas [18].

The proposals of this dissertation leverage on the aforementioned techniques, from *speculation*, *leases management* and *transactions migration* to *GPR*, and advance the literature of replicated transactional systems by providing efficient protocols that, either in full replication or in partial replication scenarios, are able to reduce the ratio between replicas synchronization and transactions execution in common real deployments.

In the following all the innovative contributions proposed in this dissertation are detailed.

## 1.2 Outline of Innovative Contributions

In the light of the aforementioned architectural trends, this dissertation introduces novel replication protocols for distributed transactional systems, which allow for significantly reducing the communication and replication costs incurred for ensuring data consistency.

The proposed solutions constitute a set of distributed concurrency control schemes for transactional processing, i.e. from database systems to transactional memories, and they encompass both *full data replication* and *partial data replication* environments.

In the context of *full replication*, typically adopted in small clusters, two techniques are exploited to enhance the efficiency of the state of the art replication protocols: *speculative execution* and *transactions migration.* With the former one this dissertation proposes a speculative transactional replication protocol that tries to maximize the overlapping of local execution and transaction replication in order to nullify the communication costs in case of advantageous scenarios of low user interaction and high computation demand with the interleaving of both transactional and non-transactional workload. With the latter one this dissertation proposes a lease-based replication protocol in which the transactional processing takes advantage from the application locality to reduce the costs of consensus on transactions outcome among replicas. In particular, in common scenarios of temporal locality, the protocol is able to get close transactional computation and data ownership via fine-grained lease management and transactions migration.

In the context of *partial replication*, typically adopted for large-scale systems as well as large data-sets, two techniques are exploited to minimize the number of contacted replicas per processed transaction: multiversioning and genuine replication protocols (GPR). In particular, the data multi-version model, which optimizes the execution in the common case of read-dominated workloads, is combined with a genuine replication scheme streamlined to reduce the amount of messages exchanged for update transactions. In fact, on one hand, multiversioning makes read-only transactions abort-free while on the other hand, GPR confines the set of nodes that participate to a given transaction to the ones replicating data accessed from that transaction. This dissertation proves the effectiveness of this approach by proposing two transactional replication protocols that explore different trade-offs in the consistency criteria provided for read-only transactions, while guaranteeing fully serializable update transactions. The first proposal in this context, in fact, copes with data freshness by permitting specific “serialization anomalies” to arise for read-only transactions and in case of non-conflicting update transactions. As it will be discussed, these anomalies are expected to be irrelevant for most of the real-life application contexts. The second proposal, instead, departs from weaker consistency criteria, and ensures serializability for both read-only and update transactions, by trading data freshness.

The remainder of this dissertation is organized as follows. Chapter 2 describes the state of the art in the context of transactional systems replication. Chapter 3 details the type of systems targeted by the presented protocols and provides the definitions and all the concepts and terms that will be used throughout the dissertation. Chapters 4 and 5 present and evaluate the solutions targeting full replication, while Chapters 6 and 7 detail the solutions proposed for partial replication. More in detail:

– Chapter 4 presents the SPECULA protocol, which exploits speculative techniques to enhance the efficiency of transactional processing in full replication;

– Chapter 5 presents the Lilac-TM protocol, which reduces the impact of replication by exploiting leases management and transactions migration;

– Chapter 6 presents the GMU protocol, which provides a scalable multi-version solution for genuine partial replication (GPR) by guaranteeing the so called Extended Update Serializability isolation level;

- Chapter 7 explores the additional tradeoffs in the design of scalable multi-version schemes for genuine partial replication (GPR) and presents the SCORe protocol, which, unlike GMU, is able to guarantee One-Copy Serializability by trading data freshness.

Finally Chapter 8 discusses the achieved results and concludes the dissertation.

# Chapter 2

# State of the Art

Software based replication has been widely studied in the literature of distributed systems since it is an effective way to enhance system availability [99]. However replication is far from being a single *ready-to-use* technique especially in the context of databases, or, more in general, transactional systems: since replication is crucial both for performance and fault-tolerance, there is an eternal dispute between consistency and efficiency. On one side, manipulating data in a consistent way is an important requirement to free the application programmers from dealing with errors caused by concurrency and distribution; on the other side, traditional eager replication techniques adopted to achieve that goal have been proved to have serious limitations [99, 38] in terms of performance, e.g. due to overhead, deadlocks, lack of scalability. This is because classical eager replication approaches entailed the synchronous update of every replica in the system on each operation executed by a transaction, according to a deterministic processing scheme, so to ensure that the state trajectories of the replicas remain correctly aligned.

Therefore, supported by the seminal work on the *dangers* of replication [38] or more recent results in which it was informally argued that even in absence of network unpredictability [13, 37] the designers have to choose between low latency and consistency [1], the research focus has also shifted to lazy replication models in which typically only one replica is updated synchronously, i.e. the one in charge of processing the request from the client, while all the remaining ones are lazily updated, i.e. asynchronously, only after the requesting client has been notified about the completion of the requested transaction/operation.

If this last approach certainly offers a greedy way to reduce the replication costs and hence the latency perceived by the end users, it limits the failure resiliency degree. Also, in case of execution of conflicting requests, replicas may reach inconsistent states, and this requires the execution of complex (whether possible) a posteriori reconciliation/compensation logics (a.k.a. eventual consistency) [27]. Other asynchronous replication techniques avoid the need for complex compensation logics, but may not support multi-operation [58] or general-purpose [103] transactions.

However the set of design choices in this context is not binary and it encompasses a spectrum from the most conservative eager approaches to the most optimistic lazy ones. In addition even consistency is not an on/off parameter, and a number of strong consistency semantics exist in literature [3], such as One-Copy Serializability [12], Snapshot Isolation [11, 33], Opacity [40], Update Serializability [3], to name a few. Furthermore, there are also other consistency levels that can be considered reasonable for certain types of applications with the purpose of reducing the user perceived latency, e.g. Causal Consistency or Parallel Snapshot Isolation [62, 63, 91, 5].

For instance as shown by the work in [54], being consistent does not necessarily mean embracing a eager technique. Moreover during the last years a set of techniques have been proposed to reduce potential efficiency problems related to consistent replication [54, 99, 74, 24, 86, 89]. They have shown, in fact, that updating synchronously all replicas does not entail (i) choosing a linear replicas interaction [99, 54] (i.e. one message per operation with communication costs that grow up linearly with the size of transactions), or (ii) adopting deadlock-prone communication primitives (i.e. no assumptions on the order of messages delivery) [99, 74, 24], or (iii) always updating all replicas in the system without creating a selective mapping between a datum and a subset of nodes in the system (e.g. partial replication) [86, 89].

Indeed effective solutions that do not give up consistency have shown that updates can be applied synchronously even with a constant per-transaction interaction among replicas and a total order can be enforced on the delivery of messages so that every replica observes the same sequence of messages delivery and hence distributed deadlocks scenarios are avoided [74].

For this reason, the solutions proposed in this dissertation follow the constant per transaction interaction model, where a constant number of messages is used to synchronize the replicas for a given transaction, unless remote communications are required to execute read operations on data that are not available locally. In addition, only synchronous replication approaches are considered, in order to guarantee that replicas are updated atomically and to avoid the aforementioned drawbacks related to asynchronous replication.

Therefore the remaining part of this Chapter gives an overview of the replication techniques framed precisely in this latter context and the works adopting these techniques, by making a comparison with the design choices made in the solutions proposed by this dissertation. It differentiates between two main replication schemes that are characterized by the roles assumed by the replicas with respect to the possibility of directly executing updates on copies of the transactional state, i.e. *Primary Copy* vs. *Update Everywhere.*

For the latter type of replication it is discriminated whether a transaction is fully executed by all replicas, i.e. *Active Replication*, or by only one, which then broadcasts the transaction's updates to the other replicas, i.e. *Deferred Update Replication*. The overall description encompasses also protocols that exploit speculation to reduce the user-perceived latency and the ones that exploit partial replication to enhance scalability, all with a bias for solutions providing strong levels of consistency and by ruling out lazy replication approaches.

## 2.1 Primary Copy

The Primary Copy replication technique [99], also known as Primary Backup or Passive Replication [98], provides a simple scheme for maintaining up-to-date replicas of the system via an order univocally determined by the so called primary copy. Every datum is associated with a specific site or node that is the primary copy of the datum and that is the only node in charge to process update requests for the datum. Therefore whenever a client issues an update on a datum $d$, the request is first sent to the primary copy of $d$, which processes the update and replicates the outcome to all the other replicas. Even if this approach allows all nodes to process read operations, there is always at most one node that executes updates on a given datum and therefore there is no need to execute an agreement among replicas for determining a common outcome on the execution of two conflicting operations. In fact all the replicas apply the changes on a datum according to the order determined by the primary copy of that datum.

Whenever a request from the client is a transaction, it is sent to the primary node that is in charge of actively processing transactions and of sending the transactions' updates to all the backup nodes. Typically transactions are processed entirely on the primary and, only if they can commit, their updates are sent to the backups.

The communication primitive that this scheme relies on can be a simple FIFO channel [98], so that the order of delivery of messages on the backups follows the order of dispatch on the primary, and the updates can be applied on all nodes in the order in which they are committed on the primary. In case the primary does not adopt any further communication for a committed transaction after having sent the commit of that transaction to all backups, there can be the risk of data loss since the primary sends the reply to the client before it is sure that backups are able to apply the updates. On the contrary, if a second voting phase follows the replication of the updates for a transaction, the overall scheme also guarantees durability. For instance, if at most $f$ nodes can fail in the system, the primary can wait for an acknowledgement from $f + 1$ backups before replying to the client. Alternatively the primary can broadcast the commit of a transaction via a Uniform Reliable Broadcast service (URB) [43], to be sure that whenever a node delivers a commit message then all the other non-faulty nodes must do the same.

Despite the simplicity of the approach, the Primary Backup mechanism has some relevant drawbacks due to the high reconfiguration costs when the primary fails. Further, Primary Backup is not prone to scale since the computation capacity on the primary can become the bottleneck. In fact, to behave correctly even in case the primary fails, the mechanism has to be able to (i) elect a new primary via a leader election facility [36] and (ii) guarantee that updates sent by the new primary will be globally ordered with respect to the updates sent by the old faulty primary, e.g. via a *View Synchronous Broadcast* service [22].

Alternative approaches were proposed to alleviate the scalability issue especially for read-dominated workloads. The work in [85], in fact, proposed a variant of the classical Primary Backup replication in which backups are allowed to process read-only transactions, i.e. transactions that do not issue write operations, in a uncoordinated and independent way with respect to the underlying concurrency control scheme, thanks to the data multi-version support.

The Primary Backup scheme was applied successfully also in the context of in-memory database systems (IMDB) [16] to ensure high availability of an IMDB with a low replication overhead. In this approach, the authors adopted an innovative middleware-level distributed algorithm exploiting assumptions properly valid for IMDB, thus reducing to two communication steps the latency needed to commit update transactions.

Another example of usage of the Primary Backup scheme is found in [96] in which the authors proposed two implementations of that replication scheme tailored for Java objects: one using the Java remote invocation method (RMI) to implement a simple and fast replication scheme for shared objects; the other one, called replica-proxy, that improves the performance of the first one by implementing a more complex mechanism that only relies on Java network packages.

An effective approach to overcome the scalability issues of Primary Backup for all types of workloads could be combining the cheap transaction execution mode locally to the primary with the possibility of all nodes to execute transactions as they were simultaneously the primary node. Therefore, in an ideal scenario in which a client can issue a request to any node because it is sure that the contacted node is the primary copy of all the data to be accessed by its request, would result in a Primary Backup solution in which every node can act as a primary by having the same advantages offered by Primary Backup replication, e.g. no distributed coordination on commit.

Unfortunately the aforementioned scenario can be only ideal because we cannot be sure that a request from a client completely matches the mapping from accessed data to primary copies so that a transaction can always be executed locally at the contacted node and in an uncoordinated fashion. Nevertheless a recent work [18] on replication in the context of software transactional memories [90] proved that the described ideal scenario can be dynamically created by exploiting the application locality and with the usage of the so called asynchronous leases. That work proposed an Asynchronous Lease-based Certification (ALC) scheme in which every node is able to acquire on-demand a set of leases on a set of data $S$ to validate and commit a transaction optimistically executed locally on $S$, so that the node becomes the primary copy of $S$ until another node requires a lease on $S$ or a subset of $S$.

The advantages of this approach are clear in case the applications exhibit temporal locality partitioned among nodes: the probability to commit a transaction without distributed coordination with the other replicas can be high, and in case a transaction can commit locally at a node and that node already has all the necessary leases, the updates can be simply broadcast to all the other nodes via URB.

Clearly, whenever a transaction completes the execution on a node and that node does not own all the necessary leases to commit the transaction, a distributed coordination phase is activated to move the leases from the current owners (if any) to the requesting one, and to ensure that all the nodes agree on the new leases configuration.

This dissertation proposes a replication scheme that uses as base replication idea the same one adopted by ALC, and a novel replication protocol that is able to improve ALC via fine-grained management of leases and transactions migration (Chapter 5).

## 2.2 Update Everywhere

Unlike Primary Copy replication and as its name suggests, in the *Update Everywhere* replication scheme [99] every node of the distributed system is in charge to process operations and therefore there is no distinction between primary and backup nodes. The first proposal of this approach [87], also named *State Machine approach*, defined a general scheme to implement fault-tolerant services by replicating the server nodes and coordinating the client interaction with the server replicas. It formalized a service as a set of *state variables* representing its state and a set of *commands* to perform transitions on its state. The resulting abstract component named *state machine*, constitutes the main unit to be replicated by relying on:

– a communication service used by the nodes to coordinate the replication of the state machine;

– a support for deterministic execution (if needed) of commands on the nodes.

The advantages of this approach are multiple, but the most important is related to fault-tolerance and availability of the services. Since there are no special roles among nodes, the scheme is able to provide full failure masking to the clients because the failure of a node does not directly affect the availability of the service since it does not entail additional costs of reconfiguration in order to elect new special node(s) in charge of executing operations (or in general updates). Furthermore, the possibility to enable all nodes to actively execute commands, makes the State Machine approach prone to scalability, by avoiding the bottleneck of the primary typical of the Primary Copy approach.

The prize to pay for having this flexibility is coordinating the distributed computation among replicas so that even if clients submit conflicting requests, i.e. to execute at least two commands on a same state variable and where at least one of them is an update, all the replicas behave correctly and advance their state deterministically and in the same way.

The two widely established schemes to implement the Update Everywhere approach for transactional processing differ for when replica coordination is carried out along time while processing the transaction. This directly affects whether a transactional execution has to behave deterministically or not, and if all nodes are required to actively process a transaction. In the following the two mechanisms are detailed, namely *Active Replication* and *Deferred Update Replication*.

### 2.2.1 Active Replication

The *Active Replication* scheme [98, 14], besides allowing all nodes to process requests from the clients, forces all nodes to process every update request to ensure replication. In particular the replication of the outcome of a request is achieved by actively replicating the request for processing on all nodes.

The challenge in this scheme is guaranteeing that all nodes execute the same sequence of requests and produce the same state transitions. Therefore, following the general guidelines of the State Machine approach, Active Replication requires both a coordination mechanism among replicas to enforce deterministic delivery guarantees of requests, and a support for deterministic execution of the requests that does not violate the constraints imposed by the coordination layer.

A request is not broadcast from a client to a particular node but it targets nodes as a group, and requests are sent via an Atomic Broadcast (AB) layer [28] to ensure determinism. This means that requests are delivered to all nodes in the same total order and the processing needs to follow that order, so that for the same sequence of input commands replicas produce the same sequence of state changes as result.

The main advantages of this scheme are the ones inherited by the general State Machine approach, namely simplicity (because the scheme does not distinguish special roles for the nodes), scalability improvement if compared to Primary Copy (because all nodes are able to process requests), and complete failure masking.

On the other hand Active Replication has the drawback of the deterministic scheduling of commands on all replicas, which may limit parallelism. Besides scheduling of commands, the commands themselves need to be implemented so as to ensure determinism, i.e. multiple executions of the same command on the same state must produce the same output.

Since determinism is achieved because conflicting requests are finalized in the same order on all nodes, in case semantic information about the commands are known, Active Replication can also be implemented by using weaker coordination services that require nodes' agreement only on the order of commands that are dependent semantically. For instance, allowing two requests that commute to be processed in different order on two nodes, enables the usage of implementations of Generalized Consensus [60] as coordination layer and it can increase parallelism.

For transactional processing, Active Replication is typically adopted under the constant interaction mode [99] according to which a transaction is submitted via AB at its beginning and then processed deterministically on all nodes. The deterministic behavior locally at each node can be achieved either via sequential execution of the totally ordered transactions delivered or, in case of lock-based local concurrency control schemes, via a preventive locks acquisition on the data to be accessed by following the transactions delivery order [55, 92], or any other deterministic locking strategy [68, 69].

Therefore the total execution latency for a transaction is given by the latency to reach an agreement for the ordering of that transaction plus the latency necessary to guarantee the transaction is correctly serialized after all the (conflicting) transactions previously delivered by the AB service, as well as the transaction's execution time.

Since this may result in excessive costs especially when the time required to execute a transaction is negligible with respect to the aforementioned overall overheads required to guarantee its deterministic replication [84], speculative techniques are often employed to reduce the user-perceived latency in Active Replication approaches [55, 56, 64, 68, 69, 70, 77]. Whether requiring an a priori knowledge on the accessed data, e.g. [55], or not, e.g. [69], these protocols rely on an optimistic early knowledge given by the spontaneous network order of messages to overlap transactions execution and coordination. This is achieved by relying on an Optimistic Atomic Broadcast service (OAB) [75] that can early deliver messages as soon as they arrive (typically after a single communication step) by optimistically guessing that the final order will match the arrival order.

These speculative techniques are specifically designed for enhancing scalability and performance of distributed transactional systems, and can be considered orthogonal to the key innovative idea behind one of the proposals of this dissertation, which exploits speculation to prevent threads from blocking till the completion of the replica coordination phase (Chapter 4). Unlike existing solutions, the protocol proposed in this dissertation allows that both transactional and non-transactional code blocks can be (speculatively) executed, and entails a mix of state recoverability techniques at both data-layer and application levels.

The proposals presented in Chapters 6 and 7 of this dissertation are also related to a recent work [92] that combines the determinism of Active Replication with the scalability of partial data replication. The scheme proposed in [92] extends the Active Replication approach by allowing the partitioning of a transaction execution flow on multiple nodes to follow the distributed data mapping.

### 2.2.2 Deferred Update Replication

The *Deferred Update Replication* [74], also known as *Certification-based Replication*, is another *Update Everywhere* replication scheme specifically tailored for transactional processing. As in Active Replication, every node is in charge of processing transactions but, unlike Active Replication, nodes do not need to handle the entire processing of all the requests from the clients. In fact, a transaction can be processed in a non-deterministic way on the first node contacted by a client and, as the technique's name suggests, propagation of the transactions updates to the replicas is deferred until the completion of the execution.

Therefore, as soon as a client requests the execution of a transaction to a given node, the node executes the transaction optimistically by returning to the client the values of the read operations and buffering in a private memory area the outcome of the write operations. The output of a transaction should not be externalized at this stage because its execution advances optimistically without taking care of conflicts with concurrent transactions on other nodes. Then at commit time, namely whenever the transaction requests the commit, the results of the local processing are replicated on all nodes. The replication does not necessarily entail the application of the updates, which can only happen if all the involved nodes determine a successful completion of the transaction.

The decision on whether committing a transaction or not can be made either locally at each node, without any further exchange of messages, or only locally at the transaction's originating node and then propagated to the other replicas. Therefore two types of Deferred Update Replication are distinguished.

The former one, named *non-voting* [99], requires both a deterministic propagation of the updates and a deterministic processing upon the delivery of the updates on a node, so that every node can decide independently of the other nodes and all nodes produce the same outcome, i.e. either commit or abort, for a given transaction. In this case the replication relies on an AB service in order to guarantee a global total order of the committing transactions and it requires that also all the outcomes of the optimistic execution of a transaction are sent via AB [28], i.e. the ones of both read operations (read-set) and write operations (write-set) [74]. As a consequence all nodes process the same sequence of delivered transactions by accessing both the read-set and the write-set and can make the same decisions by relying on a same deterministic function: typically a transaction $T$ can commit if there are no other transactions $T'$ delivered before $T$ and that have updated a datum $d$ after $T$ has read $d$. However, less pessimistic certification functions can be implemented in order to reduce the transactions abort rate [31].

The latter one, named *voting* [99], does not require a deterministic propagation of the updates so that a further voting phase is required among the nodes to determine the outcome of a transaction, i.e. either commit or abort. The replication and the voting phase can be typically implemented by using a 2PC atomic commitment protocol [12], so that the transaction's originating node can decide after having gathered the local decisions from all the other participants. As in the previous *non-voting* scheme the decision has to be made regarding the other concurrent conflicting transactions and needs to determine a final committing order. Despite its simplicity, this scheme is deadlock-prone because without a priori enforcement of a total order on the replication, there can be two concurrent and conflicting transactions that are prepared by the 2PC in two different order on two participants.

Therefore a variant of this approach was proposed [53] that relies on an AB service for replication but, unlike the *non-voting* one, this is only used with the purpose of determining a total order on the commits. Therefore, still the transaction's originating node is the only one that can make the final decision and it has to propagate the decision via a further communication step typically relying on an URB service [43]. The only advantage of this last approach is that the read-sets have not to be sent via AB because only the originating node is in charge to decide, and therefore there can be a gain when compared to the non-voting scheme in workloads dominated by read-intensive update transactions [25], i.e. update transactions that execute a large number of read operations.

Also for the Deferred Update approach, the costs of distributed consensus for determining the outcome of a transaction can be excessive, especially when the time required to execute a transaction locally is negligible with respect to the overall latency for the deterministic replication of its updates [100]. Therefore speculative techniques are employed in order to amortize the latency of the Atomic Broadcast. An example is the protocol in [20] that early validates transactions as soon as they are optimistically delivered and without waiting for the final order established by the OAB, so that the updates of a transaction optimistically validated on a node can be made speculatively visible to other transactions starting on that node. Nevertheless, the execution flow that issued the request of a commit is still required to be stopped till the OAB service determines the definitive delivery order of that commit and the associated transaction is validated according to that order. On the other hand, the speculation-based solution presented in this dissertation overcomes this problem and boosts the optimism by allowing a thread to proceed with the execution after it issued a commit request and without synchronously waiting for the completion of that request (Chapter 4). The protocol, in fact, is able to take a cross-layer approach in which both transactional and non-transactional code blocks can be (speculatively) alternated.

The aforementioned Deferred Update techniques are both effective in case of full replication, namely when a node stores a copy of the whole data-set. On the contrary, in case of partial replication, additional care has to be taken to commit a transaction since a single node has only a partial view of the current transactional state. When considering partial replication schemes, literature proposals can be classified depending on (i) whether they can be considered *genuine*, and on (ii) the specific consistency guarantees they provide. The *genuineness* of a communication protocol [44] is an important requirement that enables scalability because it entails that only nodes that are interested in the content of a message $m$ execute steps of the protocol on $m$. This means that a transactional protocol for partial replication is genuine when a node executes steps for a transaction $T$ (including the exchange of messages) only if that node stores data that are accessed by $T$.

The works in [9, 89] provide non-genuine protocols where the commitment of a transaction requires interactions with all the sites within the replicated system. In particular, to guarantee Snapshot Isolation, entailing that every transaction can safely execute on a consistent snapshot of the transactional state, they requires that all nodes are aware about the execution of all transactions. In [9], in fact, the replication protocol requires that a transaction broadcasts in total order its begin event to all nodes, so that the transaction establishes on every node an upper bound on the commits visible to its read operations; on the other hand, the protocol in [89] adopts a complementary approach by enforcing that the commits of update transactions are communicated to all nodes, so that every node can keep track in the same way the history of changes on the transactional state.

If on one hand both solutions give up genuineness, on the other hand they guarantee another property that is considered as another first class requirement in this dissertation: every transaction is always allowed to observe a consistent transactional state. This allows to drastically simplify the development of concurrent applications because it prevents, for instance, that even transactions that abort can generate, during their execution, unexpected exceptions due to the reading of a non correct state. Moreover read-only transactions (i.e. transactions that do not execute write operations) do not have to undergo additional distributed validation procedures after their execution because they can safely commit.

This last requirement is not ensured by existing protocols for Genuine Partial Replication (GPR) [86]. To guarantee One-Copy Serializability [12], in fact, the protocol in [86] executes a transaction optimistically on the originating node by buffering write operations and by allowing that only the values committed on a node before the first read operation was executed on that node are visible to the transaction. Then at commit time, the transaction (whether it is read-only or not) is validated as in the classical Deferred Update Replication but only on the nodes that store data accessed by the transaction. In particular, a commit message is sent via Genuine Atomic Multicast [44] and only the nodes storing data in the transaction's read-set execute the validation procedure while the other ones storing the data to be updated wait for a set of votes from the validating nodes.

Therefore as in the voting replication type, GPR entails a replication phase that relies on a total order service followed by a voting phase, because even if the validation procedure is deterministic, it is possible that different nodes produce different commit decisions for the same transaction due to the partial view they have on the history of commits.

This technique is improved by the protocol in [88] because it avoids the usage of the Atomic Multicast service by relying on the Atomic Broadcast service within each replication group, i.e. a group of nodes that replicate a certain datum *d*. This has the advantage of a less expensive communication layer without the adoption of any assumption on the intersection of replication groups [44] that is required for the implementation of Genuine Atomic Multicast in asynchronous systems. The price to pay is the implementation of a less permissive validation procedure that has to abort every pair of concurrent and conflicting transactions in order to guarantee One-Copy Serializability. This is because the commit requests of two concurrent and conflicting transactions can be delivered to different nodes in different order if nodes belong to different replication groups. Moreover, this protocol also offers the possibility to avoid the distributed validation of read-only transactions at the additional cost of an asynchronous communication procedure that spreads on all nodes the status of the committed updates [88], hence by impairing the genuineness.

Compared to the aforementioned solutions for partially replicated transactional systems, this dissertation proposes two GPR protocols, i.e. GMU and SCORe (in Chapters 6 and 7 respectively), which avoid to enforce the distributed validation of read-only transactions, since every read operation is always provided with a consistent transactional state.

Other solutions for partial replication that follow the Deferred Update approach and are related to the proposals of this dissertation, can be found in [8, 91]. They both enhance scalability by providing a GPR protocol that guarantees a weaker type of Snapshot Isolation [11], and they spare read-only transactions from expensive distributed validations. Both Jessy and Walter protocols, presented respectively in [8] and [91], improve scalability by allowing two different transactions to observe two non-compatible serialization orders of committed update transactions, by ensuring respectively Non-monotonic Snapshot Isolation and Parallel Snapshot Isolation (PSI). But unlike the GMU protocol presented in this dissertation, these protocols do not restrict the aforementioned anomaly to only aborted or read-only transactions, because they do not guarantee serializability of the committed update transactions. On the contrary GMU guarantees that committed update transactions are fully isolated, and this is an important requirement since those transactions manipulate the state of the system in a non-reversible way.

In addition, Walter serializes transactions at the moment in which transactions start. This design choice can impact the freshness of the data visible by long-running transactions, compared for instance to Jessy and GMU, which attempt to advance the reading snapshot of transactions along their execution. If on one hand this creates similarities with the SCORe protocol in this dissertation, on the other hand SCORe trades-off data freshness in order to achieve a much stronger consistency criterion than PSI, namely Executing One-Copy Serializability [3].

Chapter 3

# Model of the Target Systems and Preliminary Definitions

This Chapter presents the target system models for the proposals of this dissertation. The type of distributed system and communication primitives are defined in Section 3.1, while Section 3.2 specifies how data are maintained on the nodes of the distributed system. In Section 3.3 the definition of transaction and history are provided, and the notion of direct serialization graph on a history is defined. Finally Section 3.4 defines the correctness criteria targeted by the protocols presented in this dissertation.

## 3.1 Distributed Processes and Communication Primitives

A classic distributed system model composed of $\Pi = \{p_1, \ldots, p_n\}$ nodes (also called processes) is considered. Nodes communicate through message passing and do not have access to either a shared memory or a global clock. Messages may experience arbitrarily long (but finite) delays, and no bound on relative site speeds or clock skews is assumed. Further the classic crash-stop failure model is considered: sites may fail by crashing, but do not behave maliciously. A site that never crashes is correct; otherwise it is faulty.

Because of the well-known result of Fischer, Lynch and Paterson (FLP), which implies that in an asynchronous distributed system the problem of consensus cannot be solved in the presence of even a single faulty node [34], the system is supposed to be eventually synchronous, namely there exists a time $t$ after which nodes can communicate with one another in a bounded length of time. In addition, due to the result in [21], at least a majority of nodes are considered correct, unless specified differently.

This is to provide the system with a View Synchronous Group Communication Service (GCS) [22] that integrates two complementary services: *group membership* and *multicast communication.* Informally, the role of the membership service is to provide each participant in a distributed computation with information about which process is active (or reachable) and which one is failed (or unreachable). Such information is called a *view* of the group of participants. It is assumed that the GCS provides a view-synchronous primary-component group membership service [10], which maintains a single agreed view of the group at any given time and provides processes with information on whether they belong to the primary component.

The multicast communication service allows a member to send a message to the group of participants with different reliability and ordering properties. It is assumed the availability of two communication services: Uniform Reliable Broadcast (URB) [43], to guarantee an agreement on the set of messages delivered, and Optimistic Atomic Broadcast (OAB), to guarantee an agreement on the order of messages delivered, like Atomic Broadcast (AB) [75, 56], and with the possibility of exploiting an early delivery before the establishment of the final agreement on the order.

The former broadcast service is defined by the primitives *UR-broadcast*$(m)$ to send a message $m$ and *UR-deliver*$(m)$ to deliver a message $m$, and it guarantees the following properties:

– *Validity.* If a correct process $p$ broadcasts a message $m$, then $p$ eventually delivers $m$.

– *Integrity.* No message is delivered more than once and if a process delivers a message $m$ with sender $p$, then $m$ was previously broadcast by process $p$.

– *Uniform Agreement.* If a message $m$ is delivered by some process then $m$ is eventually delivered by every correct process.

The latter broadcast service is defined by the primitives *AB-broadcast*$(m)$ to send a message $m$ and *AB-deliver*$(m)$ to deliver a message $m$ according to a global final total order. In addition it also provides the primitive *Opt-deliver*$(m)$ to deliver a message $m$ optimistically. Other than guaranteeing the properties of $URB$, $OAB$ also ensures the following:

– *Global Order.* If some process *AB-delivers* some message $m$ before message $m'$, then a process *AB-delivers* $m'$ only after it has *AB-delivered* $m$.

– *Local Agreement.* If a process *Opt-delivers* a message $m$ then it eventually *AB-delivers* $m$.

– *Local Order.* A process first *Opt-delivers* a message $m$ and then it *AB-Delivers* $m$.

It is also supposed that the delivery order of messages exchanged by the $URB$ service or by the $OAB$ service does not violate causality relation among messages. In particular both these communication services guarantee the following property:

– *Causal Order.* If any process delivers a message $m'$, then that process must have before delivered every message $m$ such that $m \rightarrow m'$.

The relation $\rightarrow$ is the happened-before relation among messages [59] and it is defined as follows. For any two messages $m$ and $m'$, $m \rightarrow m'$ iff one of the following conditions is verified:

– Some process broadcast $m'$ after having broadcast $m$.

– Some process delivers $m$ and the it broadcasts $m'$.

– There is a message $m''$ such that $m \rightarrow m''$ and $m'' \rightarrow m'$.

The system also provides the nodes with primitives to send and receive point-to-point messages on reliable channels via the primitive $send(m)$ and $receive(m)$ and such that for any two correct processes $p_i$ and $p_j$, if $p_i$ sends message $m$ to $p_j$ then $p_j$ eventually receives $m$.

## 3.2 Data Model

Each node $p_i$ stores a (either partial or full) copy of data, for which, with no loss of generality, a simple key-value model is assumed. Each data item $d$ is a sequence of versions $ver = \langle val, vid \rangle$, all associated with a key $k$ representing $d$'s identifier, and ordered according to the order of the write operations committed on $d$. The fields $val$ and $vid$ of a version $ver$ are respectively a value of $d$ and a logical identifier associated with the commit of $ver$. How $vid$ is represented depends on the replication protocol being used: in particular it can be a logical scalar timestamp (or scalar clock), i.e. an integer non-negative number, or a logical vector timestamp (or vector clock), i.e. an array of integer non-negative numbers. For the sake of clarity in the presentation, throughout the dissertation it is always supposed that the size of every vector clock is equal to $|\Pi|$, i.e. it is equal to the maximum number of nodes in the system. In practice more efficient techniques can be adopted to represent vector clocks information, e.g. the one proposed for dynamic distributed systems in [61, 97].

Given a sequence of versions associated with $d$ and stored on a node $p_i$, the values of $vid$ have a monotonically decreasing field going from the most recent committed version to the oldest one. More in detail, in case *vid* is a scalar clock, the field is the only integer value of *vid*, while in case *vid* is a vector clock, the field is the integer value at entry $i$ of *vid*, i.e. $vid[i]$.

Throughout the dissertation the binary relation $\leq$ is used to define an order for both scalar values and vector clock values. In case of scalar values the relation is the standard *less-than-or-equal* relation defined for natural numbers. On the contrary, in case of vector clock values the relation has the meaning defined as follows. For each pair of vector clock values $v_1$, $v_2$, the pair $\langle v_1, v_2 \rangle$ is in $\leq$, by also writing $v_1 \leq v_2$, if $\forall i, v_1[i] \leq v_2[i]$. If there exists also an index $j$ such that $v_1[j] < v_2[j]$, where $<$ is the standard *less* relation defined for natural numbers, then $v_1 < v_2$ holds.

For protocols adopting speculative techniques to commit a transaction $T$ there is the necessity to commit at least twice a write operation on $d$ by $T$, e.g. upon a speculative commit and a final commit. Therefore, to support this type of executions, the data model includes an additional sequence of versions having the same characteristics of the previous one but representing the versions written by means of speculative (non-definitive) commits. To differentiate the two sequences, two additional fields are associated with a key $k$: $k.lastFinal$ that identifies the sequence of finally committed versions and that points to the last committed one among them; $k.lastSpec$ that identifies the sequence of speculatively committed versions and that points to the last committed one among them.

Data are subdivided across $m$ partitions, and each partition is replicated across $r$ nodes (in other words, $r$ represents the replication degree for each data item). The set $\Gamma = \{g_1, \ldots, g_j, \ldots, g_m\}$ denotes the set of $m$ groups of nodes, where $g_j$ is the group replicating the $j$-th data partition. Each group is composed of exactly $r$ nodes (to ensure the target replication degree), of which at least a majority is assumed to be correct. In order to maximize flexibility of the data placement strategy, groups are not required to be disjoint (they can have nodes in common), and a node may participate to multiple groups, as long as $\bigcup_{j=1 \ldots m} g_j = \Pi$. In addition $groups(p_i)$ denotes the set of groups which $p_i$ belongs to, and $replicas(S)$ denotes the set of nodes that replicate the data partitions containing all the keys $k \in S$, called also owners of $S$. Note that this model allows for capturing a wide range of data distribution algorithms, such as schemes, currently very popular in NoSQL transactional data stores, which rely on consistent hashing [58, 52] based distribution policies in order to: i) minimize data transfer upon joining/leaving of nodes [26]; ii) ensure the achievement of predetermined replication degrees; iii) avoid distributed lookups to retrieve the identities of the group of nodes storing the replicas of the requested data items.

In this dissertation two data replication modes are considered: *partial replication* and *full replication.* In the former the replication degree $r$ is less than the number of nodes $|\Pi|$ and therefore a node $p_i \in \Pi$ does not maintain a full copy of the entire transactional data set. This means that as soon as a transaction in execution on a node $p_i$ has to access the value of a datum $d$ that is not maintained by $p_i$, it has to issue a remote operation in order to retrieve the datum from one of the current owners of $d$. On the contrary, in the latter mode, the replication degree $r$ is always equal to the number of nodes $|\Pi|$, so that every node maintains a full copy of the whole transactional data set. Therefore, unlike partial replication, the full one has the advantage of always allowing read operations on transactional data without entailing remote communications.

## 3.3 Transaction Model

Transactions are modeled as a set of *begin*, *read*, *write*, *commit* and *abort* operations on transactional data, and they define a total order in which these operations are executed; therefore a transaction is sequential by nature and no multiple operations of a same transaction can be executed simultaneously (i.e. concurrently). Transactions that do not execute any write operation are called read-only transactions, otherwise they are called update (or equivalently write) transactions.

Since the data model entails one or more versions for the same datum $d$, every operation of a transaction is mapped to a version in the transactional system. In general, a *read* operation can be mapped to versions produced by uncommitted or even aborted transactions, as well as committed ones, but for the majority of concurrency control schemes presented in the dissertation, a read operation can only return a committed value. The only exception is for the speculative concurrency control scheme presented in Chapter 4, in which a transaction makes visible its modifications at a point in time that follows its execution but precedes the completion of its commit.

The $k$-th *write* operation of transaction $T_i$ on a datum $x$, with $k = 1, 2, \ldots$, is denoted with $w_i(x_{i.k})$, meaning that $T_i$ has created the new version $x_{i.k}$ of $x$. Since any concurrency control scheme presented in this dissertation relies on a data model where $T_i$ creates (i.e. makes available for the other transactions) only the version $x_{i.n}$ produced by its last write operation on $x$, the notation $w_i(x_i)$ is always adopted to denote a write operation of transaction $T_i$ on $x$ where:

$$w_i(x_i) \equiv w_i(x_{i.n}), n = max\{k : \exists x_{i.k}\}$$

.

As a consequence, a *read* operation of transaction $T_j$ on $x$ is denoted with $r_j(x_i)$, meaning that $T_j$ has read version $x_i$ created by $T_i$.

The last operation of a transaction $T_i$ is either a *commit* operation, denoted as $c_i$ to indicate that $T_i$ is completed successfully, or an *abort* operation, denoted as $a_i$ otherwise, and there is at most one commit or abort per transaction. In addition the first operation of a transaction is the *begin*, denoted as $b_i$ to indicate that $T_i$ starts its execution at that point in time.

On every node in the system transactions can be executed concurrently and the execution flow of a transaction $T_i$ is identified by a thread $th$. The solutions proposed in this dissertation do not allow for the execution of so called nested transactions [67, 29, 94], and therefore if the begin $b_j$ of a transaction $T_j$ on a thread $th$ follows an operation $o_i$ of a transaction $T_i$, then $b_j$ follows either $c_i$ or $a_i$. On the contrary threads can interleave the execution of transactions and non-transactional code.

### 3.3.1 History and Direct Serialization Graph

A history is the formalization of an execution of transactions on a transactional system. In particular a history $\mathcal{H}$ over a set of transactions $\mathcal{T}$ is constituted by the following two parts:

- a partial order of the events $E$ that reflect the operations of transactions in $\mathcal{T}$;
- a version order $\ll$ that is the total order defined by the creation of the versions for each datum $d$.

Each event in $\mathcal{H}$ corresponds to the execution of an operation of a transaction in $\mathcal{T}$, e.g. $w_i(x_i)$, $r_i(x_j)$, $c_i$, $b_i$, $a_i$ for $T_i, T_j \in \mathcal{T}$, and for simplicity in the exposition an operation $w_i(x_i)$ (respectively $r_i(x_j)$, $c_i$, $b_i$, $a_i$) is treated as an event in $\mathcal{H}$.

The partial order $E$ is such that:

- it preserves the order of the operations within a transaction;
- for any datum $x$ and a pair of different transactions $T_i$, $T_j$, the event $r_j(x_i)$ is always preceded by the event $w_i(x_i)$ (but this does not mean that $w_i(x_i)$ is the last write operation on $x$ before the execution of $r_j(x_i)$);
- for any datum $x$ and transaction $T_j$, if the event $r_j(x_i)$ is preceded by the event $w_j(x_{j.k})$ and no other write event on $x$ is in between $r_j$ and $w_j$ then $x_i = x_{j.k}$.

This means that $E$ cannot reverse the order of operations within a transaction, it must ensure that a read operation returns a version that was previously created and transactions are forced to observe the versions created by their own write operations.

The version order $\ll$ defines the order of versions committed on an object $x$ such that $x_i \ll x_j$ iff version $x_i$ has been committed before version $x_j$.

As defined in [3], the direct serialization graph on a history is considered to define the consistency criteria targeted by the protocols presented in the dissertation as well as their correctness proofs.

A direct serialization graph $DSG(\mathcal{H})$ on a history $\mathcal{H}$ is a graph with a vertex $V_{T_i}$ for each transaction $T_i$ in $\mathcal{H}$ and an edge $V_{T_i} \rightarrow V_{T_j}$ for each pair of conflicting transactions $T_i$, $T_j$ in $\mathcal{H}$. In particular two transactions are conflicting if they both access a common datum and at least one of those accesses is a write operation. The $DSG(\mathcal{H})$ contains three types of edges depending on the three types of conflicts, also called dependencies, that two transactions $T_i$, $T_j$ can have in $\mathcal{H}$:

– $T_j$ *directly read-depends on* $T_i$. $DSG(\mathcal{H})$ contains the read-dependency edge $V_{T_i} \xrightarrow{wr} V_{T_j}$ because there exists a datum $x$ such that both $w_i(x_i)$ and $r_j(x_i)$ are in $\mathcal{H}$.

– $T_j$ *directly write-depends on* $T_i$. $DSG(\mathcal{H})$ contains the write-dependency edge $V_{T_i} \xrightarrow{ww} V_{T_j}$ because there exists a datum $x$ such that both $w_i(x_i)$ and $w_j(x_j)$ are in $\mathcal{H}$, and $x_i$ precedes $x_j$ according to the order defined by $\ll$ in $\mathcal{H}$.

– $T_j$ *directly anti-depends on* $T_i$. $DSG(\mathcal{H})$ contains the anti-dependency edge $V_{T_i} \overset{rw}{\dashrightarrow} V_{T_j}$ because there exists a datum $x$ such that both $r_i(x_k)$ and $w_j(x_j)$ are in $\mathcal{H}$, and $x_k$ precedes $x_j$ according to the order defined by $\ll$ in $\mathcal{H}$.

Unlike the definition provided in [3], $DSG(\mathcal{H})$ is not restricted to only committed transactions in $\mathcal{H}$, but it also considers aborted and executing ones. On the contrary, to specify that the direct serialization graph has vertexes for only committed transactions in $\mathcal{H}$, the notation $DSG(\mathcal{H}^c)$ is used.

## 3.4 Consistency Model

This Section reports the definitions of the correctness guarantees that are targeted by the proposals presented in this dissertation. This is because, having as main objective the efficiency of the replication protocols, some of the proposed concurrency control schemes seek a tradeoff between consistency and efficiency by exploring intermediate transactions isolation levels. Furthermore, even though some schemes explicitly target the well known isolation level of Serializability, they rely on different design choices concerning relevant related consistency criterion, such as preservation of real-time order [71, 40] or of specific consistency guarantees for aborted transactions [40].

The description is arranged from weaker consistency levels to stronger ones, and every definition follows the definitions of the Adya's thesis in [3]; this holds also for the definition of Opacity that has been reformulated in this dissertation in terms of properties on $DSG(\mathcal{H})$ graph.

### 3.4.1 Extended Update Serializability

Extended Update Serializability, also referred to as EUS in this dissertation, was originally defined in its weaker form, i.e. Update Serializability, by the work in [45], in terms of view serializability. Then it was later re-formulated by Adya [3] with the isolation level (E)PL-3U in terms of properties on the direct serialization graph and with the possibility to be extended to also executing transactions.

EUS is a flexible isolation level because it demands different guarantees for update and read-only transactions. In particular, while the committed update transactions are forced to appear as executed serially to always perform correct transitions of the transactional state, read-only transactions, or even executing and aborted ones, are demanded to observe a consistent state generated by one of the possible serialization orders of committed update transactions.

In particular, following the definition in [3], a history $\mathcal{H}$ guarantees EUS if $\mathcal{H}$ proscribes the anomalies *G1a*, *G1b*, *G1c* and *Extended G-update*, which are defined as follows:

- *G1a.* $\mathcal{H}$ contains the operations $w_i(x_i)$, $r_j(x_i)$ and $a_i$. This means that transactions $T_j$ has read a version written by an aborted transaction $T_i$.
- *G1b.* $\mathcal{H}$ contains the operations $w_i(x_{i.k})$, $r_j(x_{i.k})$, and $w_i(x_{i.k})$ is not the last write $w_i(x_i)$ of $T_i$ on $x$. This means that transaction $T_j$ has read an intermediate non-committed value of $x$.
- *G1c.* The $DSG(\mathcal{H}^c)$ graph built on the history $\mathcal{H}^c$ derived from $\mathcal{H}$ by removing aborted and executing (i.e., ongoing) transactions contains an oriented cycle of all dependency edges.
- *Extended G-update.* The $DSG(\mathcal{H}^{upc}_{T_i})$ graph built on the committed write transactions in $\mathcal{H}$ plus transaction $T_i$ in $\mathcal{H}$ contains an oriented cycle with one or more anti-dependency edges.

Informally speaking, proscribing *G1a* and *G1b* means that every transaction can only observe a value that was committed at some point in time. In addition proscribing *G*1*c* and *Extended G-update* means that for any transactions $T_i$, $T_j$ there is an *unidirectional flow of information* from $T_i$ and $T_j$, and if $T_i$ depends on $T_j$ it does not miss the effects of $T_j$ and of all committed update transactions that $T_j$ depends or anti-depends on, according to the *no-update-conflict-misses* property [3].

The graph considered in the *Extended G-update* anomaly only includes committed write transactions and at most one additional transaction $T_i$ belonging to one among the following categories: aborted, executing or read-only transactions. Therefore, EUS does not exclude that the *DSG* built on all the executed (or even only committed) transactions contains an oriented cycle with one or more anti-dependency edges. As a consequence, EUS allows scenarios such that for a pair of read-only (or also executing and aborted) transactions $T_1$ and $T_2$, there are two committed write transactions $T_3$ and $T_4$ such that: (i) $T_1$ depends on $T_3$ but misses the effects of $T_4$, and (ii) $T_2$ depends on $T_4$ but misses the effects of $T_3$. Notice that can happen only if $T_3$ and $T_4$ are not conflicting because EUS proscribes the anomaly *G1c*.

Therefore, roughly speaking, under EUS:

– committed write transactions appear as executed sequentially;

– every transaction always observes a consistent state;

– two read-only or executing/aborted transactions may observe two states produced by two non-compatible sequences of committed transactions, which differ in the serialization order of non-dependent transactions.

As a consequence, the only discrepancies in the serialization orders observable by read-only, executing and aborted transactions are imputable to the reordering of update transactions that neither conflict (directly or transitively) on data, nor are causally dependent. In other words, the only discrepancies perceivable by end-users are associated with the ordering of logically independent concurrent events, which has typically no impact on the correctness of a wide range of real-world applications [3].

### 3.4.2 Serializability

Serializability isolation level is the target correctness criterion for classical transactional systems, e.g. database managements systems, and it guarantees that committed transactions appear as executed serially, i.e. sequentially without overlapping in time. Formally, reporting the definition provided in [3] in terms of conflict-equivalence, a history $\mathcal{H}$ guarantees Serializability if $\mathcal{H}$ proscribes the anomalies *G1a*, *G1b*, *G1c* and *G2*, the latter defined as follows:

– *G2.* The $DSG(\mathcal{H}^c)$ graph built on the history $\mathcal{H}^c$ derived from $\mathcal{H}$ by removing aborted and executing (i.e. ongoing) transactions contains an oriented cycle having one or more anti-dependency edges.

Therefore, a history $\mathcal{H}$ is serializable if the direct serialization graph built on the committed transactions in $\mathcal{H}$ does not contain any oriented cycle, and supposing that transactions always observe committed values.

In a replicated system, usually the designers are interested in a form of Serializability called One-Copy Serializability [12], also 1CS, such that despite the replication of data and the distribution of transactions executions, all the committed transactions appear as executed serially and on a single copy of the transactional system. Since in this dissertation all the proposed concurrency control schemes allow transactions $T$ to only externalize (to make visible) the last write operation executed on a datum $x$ on all the replicas of $x$ and whenever $T$ successfully commits, then the same conditions used for Serializability are adopted to check that a history $\mathcal{H}$ satisfies One-Copy Serializability.

Furthermore, a proposal in this dissertation (Chapter 7) targets 1CS (or equivalently Serializability) extended to also aborted and executing transactions. In particular, a history $\mathcal{H}$ guarantees Executing 1CS, also E1CS, if $\mathcal{H}$ proscribes the anomalies *G1a*, *G1b*, *G1c* and *G2* where the direct serialization graph considered in the definition of the anomalies is $DSG(\mathcal{H})$. Therefore E1CS demands $DSG(\mathcal{H})$ having all transactions in $\mathcal{H}$ to not contain any oriented cycle.

This is a stronger form of 1CS (or equivalently Serializability) because, like EUS, all transactions are forced to observe a consistent transactional state and, unlike EUS, for any pair of transactions $T_i$ and $T_j$ (even executing or aborted ones), the states observed by $T_i$ and $T_j$ are generated by two prefixes of the same sequential history equivalent to $\mathcal{H}$.

Allowing all transactions to observe consistent states is an important property for avoiding that applications executing in non-sandboxed environments, e.g. STMs [2], may behave erroneously upon observing non-serializable histories.

### 3.4.3 Opacity

The Opacity consistency criterion [40] is the reference correctness level for in-memory transactional systems or STMs [90]. The definition of this criterion is reported because the concurrency control schemes presented in this dissertation are targeted also for in-memory transactional systems and for some of them the reference correctness level locally at each node is precisely Opacity (Chapter 5).

Unlike the original definition provided in [40], in this dissertation Opacity has been reformulated in terms of properties on the direct serialization graph. In particular a history $\mathcal{H}$ guarantees Opacity if (i) $\mathcal{H}$ proscribes the anomalies *G1a*, *G1b*, *G1c* and *G2* where the direct serialization graph considered in the definition of the anomalies is $DSG(\mathcal{H})$, and (ii) for any pair of transactions $T_i$ and $T_j$ in $\mathcal{H}$ if $T_i \prec_{\mathcal{H}} T_j$ then $DSG(\mathcal{H})$ does not contain an oriented path from $T_j$ to $T_i$. The $\prec_{\mathcal{H}}$ relation is the happened-before relation [59] or the real-time order between two transactions in an history $\mathcal{H}$, and $T_i \prec_{\mathcal{H}} T_j$ holds if the commit operation $c_i$ of $T_i$ precedes the begin operation $b_j$ of $T_j$ in $\mathcal{H}$.

Therefore Opacity requires that (i) all operations of every committed transaction appears as executed at some single indivisible point between the begin and the commit of the transaction, (ii) operations performed by aborted transactions are not visible, and (iii) every transaction always observes a consistent transactional state.

# Chapter 4

# Exploiting Speculation to Overlap Computation and Distributed Coordination in Fully Replicated Systems

This Chapter aims at addressing the issue of reducing the impact of the replication protocol on the overall transaction execution time for fully replicated systems via SPECULA [76], namely a replication protocol that exploits speculative techniques in order to achieve complete overlapping between replicas synchronization and transaction processing activities. In SPECULA each transaction is executed on a single node, thus avoiding any form of synchronization till it enters its commit phase. Unlike conventional transactional replication protocols, in SPECULA the commit phase is executed in a non-blocking fashion: rather than waiting till the completion of the replica-wide synchronization phase, the results (i.e. the write-set) generated by a transaction that successfully passes a local validation phase are speculatively committed in a local multi-version data container, making them immediately visible to future transactions generated on the same node (either by the same or by a different thread).

To further motivate the potential benefits of a protocol like SPECULA, the impact of state of the art transactional replication mechanisms is showed when they are adopted in today's transactional systems, where the cost of computation can be considered negligible if compared to the cost of data replication. In particular a Distributed Transactional Memory platform (DTM) is evaluated, by using a standard benchmark for Software TM (STM) systems, namely STMBench7 [41], which was deployed on a cluster of up to 8 nodes, each one equipped with 2 quad-core Xeon processors at 2.13GHz and 8GB

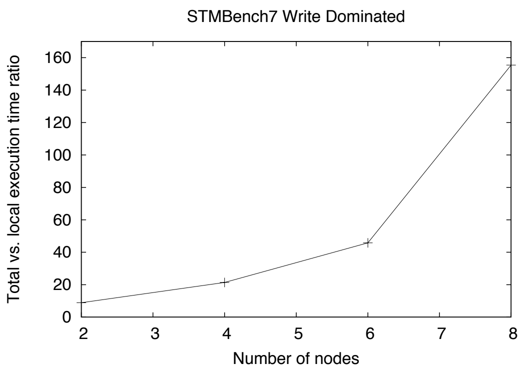


Figure 4.1: Total vs. local execution time ratio in a replicated STM

of RAM and interconnected via a Gigabit Ethernet. The replication scheme adopted is the classical deferred update/certification-based scheme [74] optimized for DTMs [24], where the outcome of a transaction is replicated on all nodes in the cluster via an Atomic Broadcast service and after an optimistic local execution. The plot in Figure 4.1 shows that, even in complex benchmarks comprising long-running transactions, like STMBench7, the duration of the transaction commit phase (which is dominated by the latency of the distributed replica synchronization scheme) is normally 10x-150x longer than the local transaction execution time.

By removing the replica synchronization phase from the critical path of transactions' execution, SPECULA allows threads to pipeline the computation of sequences of transactional and/or non-transactional code fragments, which are processed speculatively. If the outcome of the AB-based transaction certification scheme is positive, namely in absence of conflicts with concurrent remote transactions, SPECULA attains complete overlapping between processing and communication, with obvious benefits in terms of performance. In presence of misspeculations, SPECULA detects data and flow dependencies, squashing any affected speculative execution in a completely transparent fashion. In order to support the rolling-back of multiple speculatively committed transactions, and to reverse the effects of speculatively executed non-transactional code, SPECULA relies on three main mechanisms: i) *continuations* [57], i.e. an abstraction that allows to save/restore threads' stack and point in computation; ii) a speculative multi-versioned STM (called *SVSTM*) that avoids blocking on read/write operations and guarantees One-Copy Serializability (1CS) [12] for committed transactions, as well as serializability of the snapshots observed by all the transactions (even those that are eventually aborted to preserve 1CS); iii) undo-logging techniques to restore the state of the non-transactional heap.

A fully fledged Java-based prototype of SPECULA has been implemented, and, in order to achieve full transparency for applications, this relied on automatic, class-loading-time code instrumentation to inject code necessary to generate undo-logs for the update of objects residing in the non-transactional heap, and to orchestrate the usage of continuations. In addition, SPECULA's runtime automatically detects non-reversible operations (such as calls to native code), transparently injecting forced synchronization points that block speculation in order to guarantee the correct execution of such operations.

SPECULA has been evaluated by using both micro-benchmarks and standard STM benchmarks. The experimental study shows that SPECULA allows achieving up to one order of magnitude speedups when compared with a baseline non-speculative transactional replication protocol.

The remainder of this Chapter is structured as follows. Section 4.1 describes the consistency criterion guaranteed by SPECULA. The SPECULA protocol is presented in Section 4.2 while Section 4.3 reports the results of the experimental analysis.

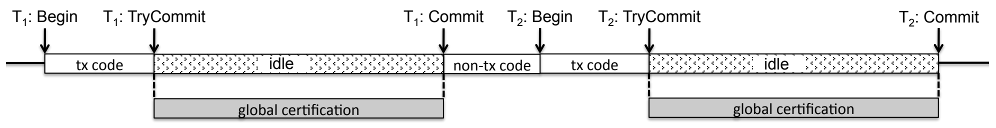
## 4.1 Correctness Criteria

The global consistency criterion for SPECULA is One-Copy Serializability (1CS) which guarantees that the history of the finally committed transactions executed by any node in $\Pi$ is equivalent to a serial history as it was executed on a single node (see Section 3.4.2). At the level of each single (non-replicated) node, SPECULA guarantees a correctness criterion analogous to the Opacity (see Section 3.4.3) criterion (which, having been specified assuming a non-speculative transaction execution model, cannot be straightforwardly applied to SPECULA). Specifically, SPECULA ensures that the snapshot observable by any transaction $T$ is equivalent to that generated by a sequential history of transactions, independently of whether $T$ is eventually aborted or committed. Just like Opacity, this criterion prevents any anomaly that may arise by observing system states that could never be generated in any serializable transaction history, and which could lead, in non-sandboxed environments like STMs, to arbitrary behaviors (such as infinite loops or crashes).

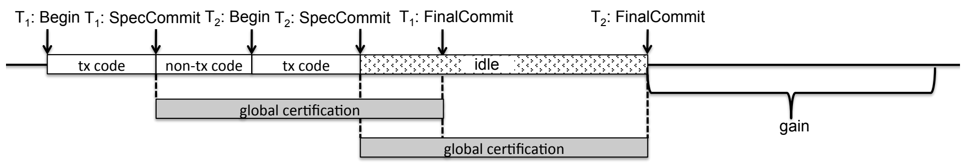
## 4.2 The SPECULA protocol

### 4.2.1 Protocol Overview

Analogously to classical certification-based replication schemes, e.g. [74, 72], in SPECULA transactions are processed locally on their origin nodes without relying on inter-replica synchronization facilities till they enter the commit



(a) Execution example using CERT



(b) Execution example using SPECULA

Figure 4.2: Execution examples

phase. However, unlike conventional schemes, if a transaction invokes commit and passes the local validation stage (that verifies the absence of conflicts with any concurrent transaction committed so far), it is locally committed in a *speculative* manner and the global AB-based transaction certification stage is executed in an asynchronous fashion. This avoids blocking a thread *th* that issued a commit for a transaction $T$ until the outcome of $T$'s global certification is determined. Conversely, $T$'s write-set is speculatively committed in the local STM, making it visible to subsequently starting local transactions, and *th* is allowed to execute immediately any subsequent non-transactional and transactional code block.

This mechanism can lead to the development of two types of causal dependencies with respect to speculatively committed transactions: *speculative flow dependencies* and *speculative data dependencies.* A speculative flow dependency is developed whenever a thread executes code (either of transactional or non-transactional nature) after having speculatively (but not yet finally) committed some transaction. A speculative data dependency arises whenever a read operation of a transaction returns a value created by a speculatively committed transaction. Furthermore, a thread is said to execute *speculatively* if it has developed either a speculative data dependency or a speculative flow dependency. A speculative dependency from a speculatively committed transaction $T$ is removed whenever the final outcome (commit/abort) for $T$ is determined.

In case of an abort event for a speculatively committed transaction, the above dependencies can lead to cascading abort of speculatively executed transactions, and require the restoration of non-transactional state (i.e. heap,

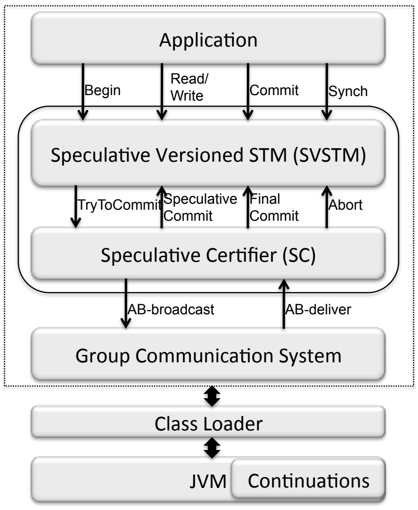


Figure 4.3: Software architecture of a SPECULA instance

stack, and thread control state) updated during the speculative execution of non-transactional code. On the other hand, in case speculatively committed transactions are compatible with the final serialization order established by the AB-based certification scheme, SPECULA achieves an effective overlapping between local processing and distributed certification phases. The diagram in Figure 4.2 illustrates the advantages achievable by SPECULA with respect to a conventional certification-based replication protocol, hereafter named CERT, via an example execution of a sequence of two transactions $T_1$ and $T_2$, interleaved with a non-transactional code block. In this example both $T_1$ and $T_2$ complete their execution without incurring in aborts due to conflicts with other transactions. With both the protocols, $T_1$ is executed locally and, when it enters the commit phase, a *global certification* is started. At this point the two protocols diverge. With CERT, the application level thread is blocked until $T_1$'s global certification finishes. Instead, with SPECULA, $T_1$ is speculatively committed and the application level thread can immediately start processing the subsequent non-transactional code. Then it executes transaction $T_2$. Hence, in this scenario, SPECULA achieves a reduction of the overall execution time equal to the duration of $T_1$'s certification.

### 4.2.2 High Level Software Architecture

The diagram in Figure 4.3 provides a high level overview of the software architecture of a SPECULA replica. The *Application* layer starts a transaction by triggering a *Begin* event on the *Speculative Versioned Software Transactional Memory*, hereafter referred to as SVSTM, which registers the transaction within the SPECULA environment and handles every subsequent read/write operation executed in the context of that transaction.

The SVSTM layer provides isolation guarantees during transaction execution and ensures the atomicity of the state alterations associated with *Abort*, *Speculative Commit* and *Final Commit* events. These properties are guaranteed by a multi-version concurrency control scheme that ensures that read operations performed by a transaction $T$ can be always executed in a non-blocking manner by properly selecting versions belonging to the most recent snapshot that has been committed, possibly in a speculative fashion, before starting $T$. Further, unlike classical multi-version schemes, SVSTM allows a speculatively committed transaction $T$ to externalize its write-set to other locally executing transactions before its final outcome is established. Also, SVSTM atomically removes $T$'s write-set if the certification phase of $T$ determines that it needs to be aborted, in which case cascading abort of transactions having speculative (data and/or flow) dependencies from $T$ is triggered.

Additionally, SPECULA needs to cope with scenarios in which a non-revocable operation (e.g. an I/O operation) is issued by a thread that is executing in speculative mode, i.e. that causally depends on a speculatively committed state. To tackle this case, SVSTM offers a simple programming interface, called *Synch*, that can be used by the applications to force a synchronization point, which blocks speculation on the caller thread until all the transactions speculatively committed so far by that thread are actually committed, or at least one of them is aborted (in which case this speculative execution needs to be squashed).

When the application layer invokes commit (*Commit* event) for a transaction $T$, SVSTM requests $T$'s validation to the *Speculative Certifier* (*SC*), which first attempts to validate $T$ speculatively. If the validation fails, an *Abort* event is triggered. Otherwise, *SC* triggers a *Speculative Commit* event to the upper layer, unblocking the application level thread, and activates a second validation phase aimed at certifying $T$'s consistency on a global basis by relying on the total ordering service provided by the *G*roup Communication System (GCS). If this second certification phase is also successful, a *Final Commit* event is triggered. Otherwise, an *Abort* event is generated.

In order to support the rollback of chains of speculatively committed transactions, possibly interleaved by non-transactional code, SPECULA relies on the *continuation* abstraction, and on the automatic management of non-transactional code via undo-logging techniques. A continuation reifies the control state of a thread, and allows resuming the execution at the point in which the continuation was created. To ensure total transparency for the applications, SPECULA relies on a customized class loader that instruments the application code (at the bytecode level) in order to automatically capture continuations out of the transaction boundaries and transparently create the data structures required for reversing non-transactional executions.

### 4.2.3 Speculative Execution of Transactions

This Section describes in detail the key mechanisms used by SVSTM and SC components to execute and commit a transaction in SPECULA. The pseudocode of the algorithms used by SVSTM and SC is also included, namely the one used to manage the speculative execution and commit of transactions as well as the one used to both speculatively and finally certify transactions.

**Management of Speculative/Final Snapshots**. As in classical multi-versioned STMs, e.g. [15, 82, 83], SVSTM maintains multiple committed versions of data and uses a timestamp-based mechanism to associate each transaction $T$ with a snapshot that is used during $T$'s execution to determine version visibility. Unlike existing multi-version concurrency control algorithms, however, SVSTM allows the addition and removal of speculatively committed versions. These manipulations of the object versions need to take place without endangering the correctness (i.e. serializability) of the snapshots observed by currently executing transactions.

To this end SPECULA follows the data model presented in Section 3.2 and the speculative and final versions maintained for each key $k$ are organized in two separate single-linked lists. As shown in Figure 4.4, a key $k$ is associated with the pointers to the most recent speculatively and finally committed versions of an object, denoted, respectively, as *k.lastSpec* and *k.lastFinal.* A (speculatively/finally) committed version created by a transaction $T$ stores (i) a reference to the previously (speculatively/finally) committed version in the list (if any), (ii) the associated value *val*, (iii) the version identifier *vid* that is represented as a scalar logical timestamp in SPECULA and that identifies the (speculatively/finally) committed snapshot generated by $T$, (iv) a reference to a *Transaction* object identifying $T$.

Since SPECULA has been implemented in a replicated object-based STM for the experimental evaluation, analogously to what other multi-version STMs do, e.g. [15], each key and the associated versions of the data model are mapped in a so called *Transactional Container* (*TC*). Therefore, in the object-based implementation of SPECULA, each transactional object is wrapped in a *TC*, an abstraction that provides two main functionalities: i) allowing cluster-wide object identification, by transparently associating unique IDs to transactional objects (namely the keys of the data model), and ii) maintaining and managing (speculatively/finally) committed versions of each transactional object (by pointing to the versions of speculatively/finally committed versions).

The most recent final committed snapshot is tracked using a single scalar timestamp, called *UpperFinID.* As in conventional (i.e. non-speculative) multi-versioned STMs, whenever a transaction $T$ is final committed, the *UpperFinID* is incremented and the new versions created by $T$ are written back, tagging them with the current value of *UpperFinID*.

The management of speculatively committed snapshots is instead regulated via a *Speculative Window* (*SW*), which is a linked list containing an element for each speculatively committed transaction by the local SPECULA replica. The advancement of speculative commits is regulated by an additional timestamp, called *UpperSpecID*, which identifies the most recent transaction in the speculative window (i.e. the last transaction that has been speculatively committed). Whenever a transaction is speculatively committed, *UpperSpecID* is incremented and, for each data item in the transaction write-set, a new speculative version is added in the associated *TC* and tagged with the new value of *UpperSpecID*. Further, a node is added to the head of $SW$, causing its widening. The speculative window is narrowed whenever a final outcome (commit/abort) is determined for one of the transactions in $SW$, causing the removal of the corresponding node.

As it will be shown, the most recent speculatively committed snapshot by an instance of SVSTM is equivalent to the one obtained by serializing the speculatively committed (and not aborted) transactions contained in the speculative window after the sequence of finally committed transactions up to the one having (finally committed) snapshot id equal to *UpperFinID*.

**Transaction Activation.** When a transaction is activated via the *Begin* operation (Algorithm 1), it is associated with the following data structures: i) the read-set $rs$ and write-set $ws$, maintaining, respectively, the set of items read and written by the transaction; ii) a *state* variable that can assume values in the domain $\{executing, aborted, specCommitted, finCommitted\}$; iii) a scalar timestamp, called *TXUpperFinID*, which determines the most recent final committed snapshot visible by that transaction and which is set, upon starting the transaction, to the current value of *UpperFinID*; iv) a linked list, called $TX\_SW$, which is initialized at transaction's activation time, by creating a copy of the local SVSTM's *SW*; v) a reference to a continuation, called $resumePoint$, which is initialized to `null` and is updated upon the speculative commit of the transaction to capture the thread's execution context at the end of the transaction; vi) an *undoLog* that, as it will be discussed in Section 4.2.4, is used to rollback any update of the non-transactional heap performed by non-transactional code blocks executed after the speculative commit of the transaction.

**Tracking Speculative Flow Dependencies.** In order to track flow dependencies among speculative transactions executed by the same thread, an additional timestamping mechanism is used. Each thread *th* is associated with

**Algorithm 1** SVSTM - Transaction activation

```
1: void Begin(Transaction T)
2:     $T.ws \leftarrow \emptyset$
3:     $T.rs \leftarrow \emptyset$
4:     $T.state \leftarrow executing$
5:     $T.TXUpperFinID \leftarrow UpperFinID$
6:     $T.TX\_SW \leftarrow copy(SW)$
```

a scalar timestamp called *epoch*, which is initialized to zero and incremented upon the *abort* of a transaction speculatively committed by *th*. Further, each replica maintains a table, called *EpochTable* that stores the current epoch of each thread issuing transactions in any of the nodes in $\Pi$. As it will become clearer in the following, this mechanism allows detecting whether an AB-delivered commit request (possibly associated with a remotely originated transaction) should be discarded due to a speculative flow dependency from an aborted speculative transaction.

**Read and Write Operations**. Write operations are managed by simply buffering the new written values in the transaction's write-set (lines 1-4 of Algorithm 2). A read operation issued by a transaction $T$ on a key $k$ is managed as follows. In order to guarantee that transactions observe the values that they previously wrote, it is first necessary to check whether $k$ is already in $T$'s write-set and, in the positive case, the read operation returns that $k$'s value (lines 9-10 of Algorithm 2). Otherwise, it is checked if $k$'s speculative versions' chain contains any version speculatively committed by a transaction present in $T$'s speculative window. If this is true, the most recent of such versions, say $v^*$, is the one that should be observed by $T$, based on the speculative serialization order that was attributed to it upon starting up its processing. However, before returning $v^*$, it is necessary to check whether the transaction that created $v^*$ has been aborted in the meanwhile (recall each version stores a pointer to the creating transaction). In such a case $T$ needs to be aborted as well (lines 11-14 of Algorithm 2).

Finally, if no speculative version of $k$ is visible, the visibility rule of conventional multi-version algorithms is used, namely it is returned the most recent final committed version of $k$ having *vid* less than or equal to *TXUpperFinID* (line 16 of Algorithm 2).

As well as write operations do with the write-set, the read operations buffer the version associated with the returned values in the transaction's read-set, in order to validate the transaction execution at commit time. In addition, both write and read operations always check the status of the transaction and they force an abort in case the transaction was marked as to be aborted. This is because the SC can concurrently abort a transaction $T'$ which $T$ depends on.

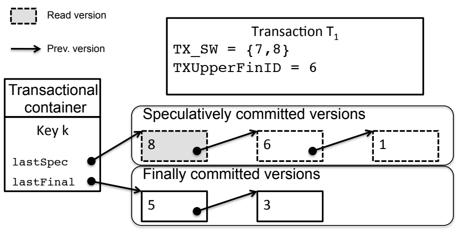
To better clarify the read mechanism, Figure 4.4 shows two example executions in which a read operation is issued by a transaction $T$ having *TXUpperFinID* equal to 6 and having in its $TX\_SW$ two speculatively committed transactions with timestamps 7 and 8. In the example of Figure 4.4(a), $T$ observes the speculative version having *vid* 8, since transaction 8 is in $T$'s $TX\_SW$. In the example of Figure 4.4(b), none of the available speculatively committed versions is observable by $T$, which is forced to read the most recent finally committed version having *vid* equals to 5.

**Algorithm 2** SVSTM - Read and Write operations

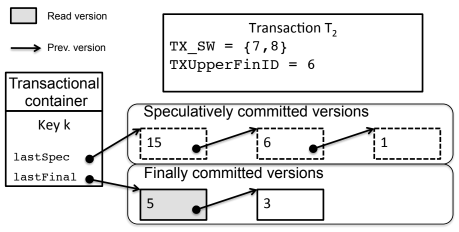
```
void Write(Transaction T, Key k, Value val)
    if $T.state = aborted$ then
        throw $ABORT$;
    $T.ws \leftarrow T.ws \setminus \{\langle k, -\rangle\} \cup \{\langle k, val\rangle\}$

Value Read(Transaction T, Key k)
    if $T.state = aborted$ then
        throw $ABORT$
    if $\exists\langle k, val\rangle \in T.ws$ then
        return $val$
    $Version\ v^* \leftarrow mostRececentInSpecWindow(k.lastSpec, T.TX\_SW)$
    if $v^* \neq null$ then
        if $v^*.commitTx.state = aborted$ then
            throw $ABORT$
    else
        $v^* \leftarrow mostRecentFinal(k.lastFinal, T.TXUpperFinID)$
    $T.rs \leftarrow T.rs \cup \{\langle k, v^*\rangle\}$
    return $v^*.val$
```

**Commit Requests.** The above described visibility rules ensure that every transaction $T$ always observes a snapshot equivalent to the one generated by the sequential history that includes the (globally validated) final committed transactions up to $T$'s *TXUpperFinID* followed by the speculatively committed transactions in $T$'s *TX_SW.* On the other hand, in SPECULA read-only transactions may have to be aborted due to the detection of speculative dependencies.

As the speculative snapshot observed by a transaction is guaranteed to be serializable, SPECULA can spare read-only transactions from the cost of a dedicated (and very expensive) AB-based validation phase. Conversely, SPECULA adopts a lazy validation mechanism that delegates the validation of read-



(a) Read of a speculative version



(b) Read of a final version

Figure 4.4: Versioning in SVSTM and examples of *read* operations

only transactions (that have observed some speculatively committed value) to the first update transaction subsequently executed by the same thread. By this mechanism, whenever a thread $th$ requests the commit for an update transaction $T_{up}$, it does not only validate $T_{up}$ but also any speculative read-only transaction $T_{ro}$ that $th$ executed before $T_{up}$ and after the last update transaction preceding $T_{up}$. This set of read-only transactions is denoted as $RO(T_{up})$.

More in detail, as an update transaction $T_{up}$ requests to commit (lines 1-9 of Algorithm 3), it is first locally validated to check for conflicts with already (both speculatively and finally) committed transactions. This validation simply entails iterating over $T_{up}$'s read-set and checking whether $T_{up}$ would still observe the same versions if its execution were started in that moment (lines 1-6 of Algorithm 4).

If the local validation phase is successful, the commit request of $T_{up}$ is sent via AB, along with the following information: the identifiers of the objects in $T_{up}$'s read-set, its write-set, its speculative window, and the identifiers of the speculative transactions from which there is a read-from dependency (wrt $T_{up}$ and the read-only transactions in $RO(T_{up})$). The latter set of transactions is denoted as $SRF(RO(T_{up}))$. Finally, in order to allow remote nodes to track speculative flow dependencies, the unique identifier of the thread executing $T_{up}$ and its current *epoch* value are also sent via AB.

**Speculative Commit.** The speculative commit of a transaction $T$ by a

**Algorithm 3** SVSTM - Commit request and speculative/final commit (thread *th* of node $p_i$)

```
 1: void Commit(Transaction T)
 2:     if $T.state = aborted$ then
 3:         throw $ABORT$
 4:     if $validateLocal(T)$ then
 5:         if $T.ws \neq \emptyset$ then
 6:             $AB\text{-}broadcast([T, th.epoch, th.id, SRF(RO(T)), RO(T)])$
 7:         $specCommit(T)$
 8:     else
 9:         throw $ABORT$
10:
11: void specCommit(Transaction T)
12:     $UpperSpecID++$
13:     for all $\langle k, val\rangle \in T.ws$ do
14:         $addVersion(k.lastSpec, val, UpperSpecID)$
15:     $widen(SW, T)$
16:     $T.state \leftarrow specCommitted$
17:
18: void finalCommit(Transaction T)
19:     for all $\langle k, val\rangle \in T.ws$ do
20:         $addVersion(k.lastFinal, val, UpperFinID + 1)$
21:     $UpperFinID++$
22:     $put(Spec2FinIDs, T, UpperFinID)$
23:     if $isLocal(T)$ then
24:         $shrink(SW, T)$
25:     $T.state \leftarrow finCommitted$
```

thread *th* implies two main operations. First, a checkpoint of the current thread execution context is generated by creating a continuation and storing it in $T$'s *resumePoint* variable. This continuation will be used in case it is later necessary to rollback the execution of (transactional/non-transactional) code that is processed by *th* in a speculative fashion, following the speculative commit of $T$. If $T$ is an update transaction, it is also necessary to write-back the speculative versions that $T$ updated/created (lines 11-15 of Algorithm 3). This is done by atomically i) increasing *UpperSpecID*, ii) adding the new speculatively committed versions of the keys in $T$'s write-set at the head of the corresponding chains of speculative versions and (iii) adding a new node associated with $T$ to the head of *SW* of the local SVSTM.

**Global Validation and Final Commit.** When an AB message is delivered to node $p_i$, which requests the commit of a transaction $T_{up}$ originated by thread *th* running on node $p_j$, $T_{up}$ is validated to detect whether it is possible to finally commit it. To this end, it is first checked in which epoch $T_{up}$ has been originated. If $T_{up}$'s epoch is less than the current value stored in $EpochTable$ for $th$, it means that $T_{up}$ has a speculative flow dependency from an aborted transaction and can be discarded. If $T_{up}$ is tagged with a greater epoch value than the one currently associated with $th$ at node $p_i$, say $e'$, the corresponding entry in $EpochTable$ is updated (lines 2-9 of Algorithm 5).

Next, it is checked whether the read-only transactions $RO(T_{up})$ and $T_{up}$ have read a valid snapshot. To this end it is checked if the transactions in $SRF(RO(T_{up}))$ have been, in the meanwhile, finally committed at $p_i$. This is achieved by looking up a table, called *Spec2FinIDs*, that maintains a mapping between the identifier of each transaction $T$ that is both speculatively committed and finally committed, and the *UpperFinID* snapshot that was attributed to $T$ when it was finally committed. By the causal ordering property of the AB service, at the time in which $T_{up}$ is *AB-delivered*, it follows that also the update transactions $SRF(RO(T_{up}))$ from which $T_{up}$ (via $RO(T_{up})$) depends indirectly must have already been final delivered. Therefore, if any transaction $T^*$ in $SRF(RO(T_{up}))$ cannot be found in *Spec2FinIDs*, it means that $T^*$ must have been already aborted at $p_i$ (lines 10-15 of Algorithm 5).

Before finally committing a read-only transaction $T^{RO}$ in $RO(T_{up})$, it is necessary to perform read-set validation as if it was an update transaction. In fact, if at this stage transactions in $RO(T_{up})$ do not undergo a validation procedure, there could be executions that violate 1CS in case $T_{up}$ was successfully validated.

To better clarify this issue, an example execution is considered, in which a transaction $T_1$ speculatively commits on node $p_1$ by writing value $val_{x1}$ on object $x$ and transaction $T_2$ speculatively commits on node $p_2$ by writing value $val_{y2}$ on object $y$. Further there is a thread $th$ on $p_2$ that executes a read-only transaction $T^{RO}$ by reading the speculative value $val_{y2}$ committed by $T_2$ and a value $val_{x0}$ of $x$ that precedes the one committed by $T_1$. Even if this could be admissible because $T_1$ and $T_2$ do not conflict and $T^{RO}$ reads from a serializable snapshot in which $T_2$ appears as executed before $T_1$, $T^{RO}$ has to be aborted during the global validation of transaction $T_{up}$ such that $T^{RO} \in RO(T_{up})$. This is because there can exist another read-only transaction $\bar{T}^{RO}$ on $p_1$ that executes two read operations on both $x$ and $y$ and, unlike $T^{RO}$, it observes the speculative commit of $T_1$ and it misses the one of $T_2$. Since both $T_1$ and $T_2$ finally commit, this leads to a non-serializable execution in which $T_1$ appears as committed before $T_2$ according to the observation of transaction $\bar{T}^{RO}$, while $T_2$ appears as committed before $T_1$ according to the observation of $T^{RO}$.

Therefore transactions $T^{RO}$in $RO(T_{up})$ as well as $T_{up}$ are validated by using the *validateGlobal* function (lines 16-20 of Algorithm 5 and lines 8-12 of Algorithm 4), and clearly if any of the transactions in $RO(T_{up})$ is aborted, also $T_{up}$ has to be aborted.

**Algorithm 4** SC - Local and global validation

```
1:  bool validateLocal(Transaction T)
2:      if $T.ws \neq \emptyset$ then
3:          for all $\langle k, v^* \rangle \in T.rs$ do
4:              if $v^*.val \neq k.lastFinal.val \wedge v^*.val \neq k.lastSpec.val$ then
5:                  return $\perp$
6:      return $\top$
7:
8:  bool validateGlobal(Transaction T)
9:      for all $\langle k, v \rangle \in T.rs$ do
10:         if $k.lastFinal.vid > T.TXUpperFinID \wedge$
              $k.lastFinal.commitTx \notin T.TX\_SW$ then
11:             return $\perp$
12:     return $\top$
```

To this end it is first verified whether, for each key $k$ in $T_{up}$'s read-set (respectively $T^{RO}$'s read-set), $k$'s most recent finally committed version, say $k.lastFinal$, has *vid* less than or equal to $T_{up}$'s *TXUpperFinID* (respectively $T^{RO}$'s *TXUpperFinID*). This is certainly a safe read, given that $T_{up}$ (respectively $T^{RO}$) has read a non-speculative version of $k$, which has not been in the meanwhile overwritten by more recent versions, and is hence valid. If this is not the case, it is checked whether the transaction that created $k.lastFinal$, say $T^{\dagger}$, is in the speculative window of $T_{up}$ (respectively $T^{RO}$). If this is false, it means that $T_{up}$ needs to be aborted as it depends on a speculative version that has been either aborted, or overwritten by a more recent (finally committed) version.

If $T_{up}$ is a remote transaction, an additional test is performed before applying its write-set, in order to determine whether $T_{up}$'s commit invalidates any local speculatively committed transaction. To this end it is checked, for each transaction $T'$ in the node's SVSTM whether its read-set intersects with the write-set of $T_{up}$. In this case, $T'$ is aborted, triggering the cascading abort of any other speculative transaction having flow/data dependencies with $T'$.

At this point the node's *UpperFinID* is increased, and the write-set of $T_{up}$ is applied by creating new finally committed versions in the corresponding chains of versions and tagging them with the new *UpperFinID* value as *vid*. Finally, if $T_{up}$ had been previously speculatively committed on node $p_i$ (i.e. $p_i = p_j$), it is removed from the SVSTM's *SW* (lines 17-23 of Algorithm 3).

**Abort.** If a transaction $T$ is aborted while it is still executing, its state is simply marked as *aborted*. The state of a transaction is checked by the SVSTM's layer any time a *Read*/*Write*/*Commit* operation is issued for that transaction, which guarantees that $T$ will be aborted before it can speculatively

**Algorithm 5** SC - Delivery of a commit broadcast message on node $p_i$

```
1: void AB-deliver([T,epochID,thID,SRF(RO(T)),RO(T)])
2:     int epoch ← $get(EpochTable, thID)$
3:     if $epoch \neq null \wedge epochID < epoch$ then
4:         $T.state \leftarrow aborted$
5:         if $isLocal(tx)$ then
6:             $Abort(T)$
7:         return
8:     else
9:         $put(EpochTable, thID, epochID)$
10:    for all $stx \in SRF(RO(T))$ do
11:        if $stx \notin Spec2FinIDs$ then
12:            $T.state \leftarrow aborted$
13:            if $isLocal(T)$ then
14:                $Abort(T)$
15:            return
16:    for all $stx \in RO(tx) \cup \{T\}$ do
17:        if $validateGlobal(stx) = \perp$ then
18:            if $isLocal(T)$ then
19:                $Abort(T)$
20:            return
21:    if $isRemote(T)$ then
22:        for all $stx \in specActiveTx$ do
23:            if $(stx.rs \cap T.ws) \neq \emptyset$ then
24:                $stx.state \leftarrow aborted$
25:                $Abort(stx)$
26:    $finalCommit(T)$
```

commit and externalize its writeset.

Additional care needs to be taken, instead, if the transaction $T$ that is being aborted has already been speculatively committed. In this case, in fact, there may be locally running transactions that include $T$ in their speculative windows, and which may access some of the objects for which $T$ has produced a speculative version while $T$'s abort procedure is being concurrently executed. In order to avoid anomalies, it is also required that the abort of $T$ is performed atomically with the abort of any transaction in $SW$ that has (possibly transitive) speculative dependencies from $T$ (otherwise, concurrently executing transactions may miss the snapshot of $T$ and observe the snapshot created by transactions that depend on $T$). If $th$ is denoted as the thread that executed a transaction $T$ that is being aborted, then the abort procedure recursively aborts all the transactions that were speculatively committed by $th$ after $T$ in reverse chronological order (from the most recent to the the oldest). Next, the abort procedure is called on any other transaction in $SW$ that has read from $T$. Finally, the state of $T$ is set to *aborted*, which allows atomically preventing that future reads can observe any of $T$'s speculatively committed version.

**Garbage Collection.** In order to allow the garbage collection of speculative and final committed versions, whenever a transaction is final committed or aborted, SPECULA evaluates what are the oldest final and speculative snapshots visible by any locally running transaction. These snapshots correspond to the minimum value, across any active transaction $T$, of $T$'s *TXUpperFinID* and, respectively, of the minimum speculative snapshot identifier in $T$'s $TX\_SW$.

### 4.2.4 Speculative Execution of Non-Transactional Code

This Section describes the support of SPECULA for the speculative execution of non-transactional code. Even though the description is strictly related to the specific Java implementation of SPECULA, the presented concepts remain still valid in case other languages are used. In particular, changing the technology only poses the issue of mapping the concepts used here onto different specific solutions.

In case of misspeculation, SPECULA may have to reverse the execution of a sequence of transactional and non-transactional code blocks. As already discussed, continuations are used to allow resuming the execution flow at the end of the most recent valid transaction committed by each thread. In order to rollback the alterations performed on the heap by non-transactional code, SPECULA relies on undo-logs. Undo-logs are built in a totally transparent fashion for the application by intercepting 8 different Java bytecode instructions issuing modifications of fields and arrays. Automatic bytecode instrumentation is achieved by replacing the Java default class-loader with a custom class-loader that performs dynamic bytecode rewriting at the moment in which any Java class is loaded by the JVM.

When any of these bytecode instructions is executed by a thread $th$, the pre-image of the target address is saved in the *undoLog* before being overwritten with the new value. An *undoLog* keeps just one value per memory address, namely the oldest one, and is associated with the last speculatively committed transaction $T$ that was executed by $th$. If the squashing of a speculative execution causes the abort of $th$, the undo-log is used to restore the heap state to the snapshot at the moment in which the continuation that is going to be resumed was captured (that is at the end of transaction $T$).

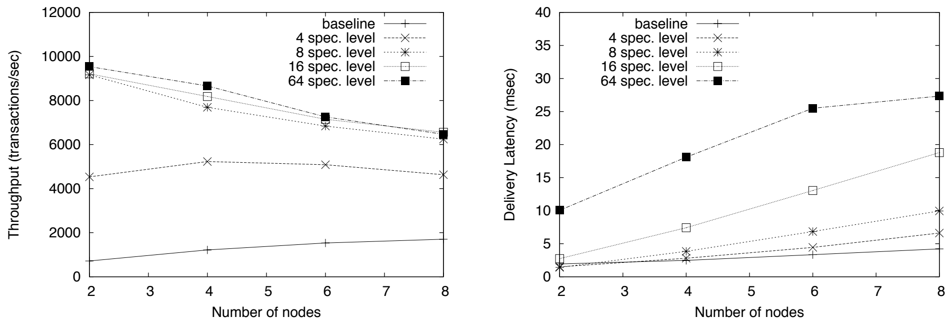
Another issue that is tackled by SPECULA's automatic bytecode instrumentation framework is the transparent addition of forced synchronization blocks preceding any non-revocable operation (e.g. I/O operations). More in detail, SPECULA leverages on the JaSPEx [7] library to detect, at the bytecode level, calls to native code via JNI and other non-reversible operations (e.g. invocations of the java.io.* package), and automatically insert bytecode that forces the thread to wait for the validation of any speculatively committed transaction.

### 4.2.5 Correctness Arguments

This Section provides some informal arguments on the correctness of SPECULA with respect to the criteria mentioned in Section 4.1. A formal proof is not reported since it is easy to show in the following how SPECULA guarantees 1CS on the set of finally committed transactions and that every read operation always observes a serializable execution.

The assurance of 1CS stems directly from the fact that every final committed transaction, is validated deterministically and in the same final order by all the replicas in the system. Specifically, if a transaction $T_i$ has developed any speculative dependency from a speculatively committed transaction $T_j$, SPECULA validates $T_i$ only after having finally committed, or aborted, $T_j$. In the latter case, $T_i$ will be aborted either when $T_j$ is delivered, or during the validation of $T_i$ that takes place when the commit request of $T_i$ is *AB-delivered*. Now assume that all the transactions $T_j$ from which $T_i$ has developed a speculative dependency have been committed at the time in which $T_i$ is validated. In this case, the validation of $T_i$ detects whether $T_i$ would observe the same snapshot as if it was executed via a non-speculative multi-version scheme starting on a committed snapshot that includes all the transactions committed before $T_i$ according to the AB-based serialization order.

On the other hand the following description analyzes how SPECULA ensures serializability of the snapshots observed by transactions (including those that are eventually aborted) throughout their execution. When a transaction $T$ is activated at a node $p_i$, SPECULA speculatively serializes $T$ after the totally ordered history of (update) transactions, say $\mathcal{H}$, composed by i) the prefix of finally committed transactions, followed by ii) the sequence of speculatively committed transactions ordered according to $p_i$'s $SW$. The latter ones have been locally validated before being speculatively committed. This guarantees serializability of $\mathcal{H}$. During the execution of $T$, new transactions may commit, either speculatively or finally. In both cases, however, the new versions created by these transactions cannot be observed by $T$ due to the visibility conditions regulating the execution of read operations. Also, the abort of any of the transactions in $T$'s $TX\_SW$, say $T^*$, does not compromise serializability of the snapshot observable by $T$. In fact, the versions speculatively committed by $T^*$ are only removed by the underlying SVSTM after $T$'s execution is completed (see the "Garbage Collection" paragraph in Section 4.2.3).



(a) Throughput. (b) Delivery Latency.

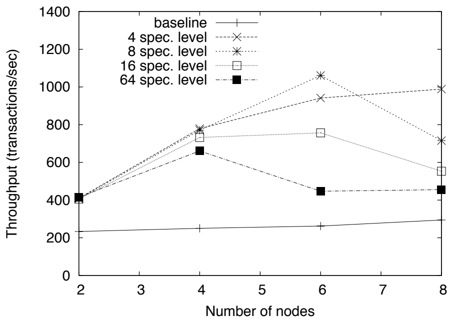
Figure 4.5: Bank benchmark.

## 4.3 Experimental Evaluation

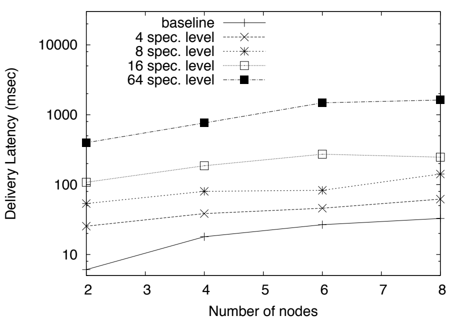
In order to assess the performance gains achievable by SPECULA, a non-speculative multi-versioned replicated STM has been used as baseline, which, analogously to SPECULA, relies on an AB-based distributed certification phase taking place whenever an update transaction reaches the commit stage [74, 72]. The local STM exploited for the baseline is JVSTM [15], a state of the art multi-versioned STM that never blocks or aborts read-only transactions. The AB layer relies on the LCR algorithm [42], a recent ring-based AB algorithm that is known for its robust performance especially at high throughput. The results reported in this section were obtained using the same experimental platform employed to produce the plot reported in Figure 4.1.

In addition to the mechanisms described in Section 4.2, the SPECULA prototype introduced a simple throttling mechanism that limits the maximum number of speculatively committed transactions pending at any node, i.e. the speculation level. By treating the speculation level as an independent parameter, the impact of SPECULA has been assessed on various performance indicators, in particular the latency of the AB layer and the transaction abort rate, as well as the throughput of the system.

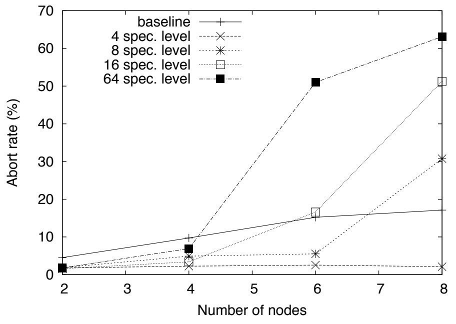
The first considered workload is a synthetic benchmark, called Bank, that was selected to assess the maximum gains achievable by SPECULA. Bank exclusively entails update transactions that simulate the transfers among bank deposits, and was configured not to generate any conflict. These are clearly ideal conditions for SPECULA, and the plots in Figure 4.5 confirm the achievement of striking performance gains, with speedups varying in the range 4x (for 8 nodes) to 13x (for 2 nodes). Despite the simplicity of this scenario, these data allow to draw some interesting conclusions. The plot in Figure 4.5(a)



(a) Throughput.

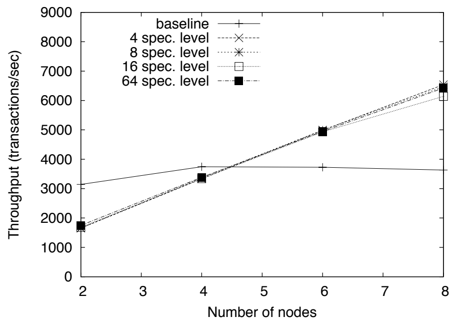


(b) Delivery Latency.

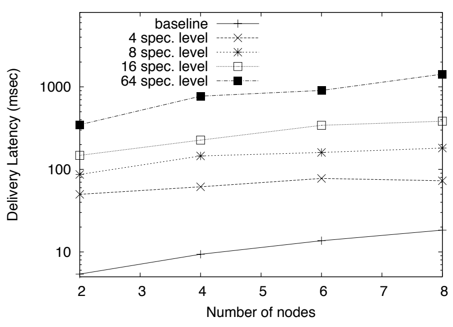


(c) Abort Rate.

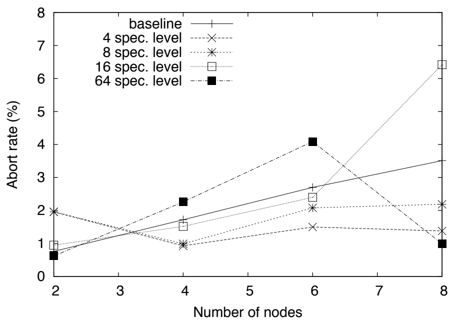
Figure 4.6: STMBench7 write dominated.



(a) Throughput.



(b) Delivery Latency.



(c) Abort Rate.

Figure 4.7: STMBench7 read dominated.

shows that increasing the speculation level beyond 8 does not provide any additional speedup for SPECULA. This is explainable by observing in the plot in Figure 4.5(b) that the AB-delivery latency grows very rapidly for speculation levels above 8, highlighting that SPECULA has already hit the maximum throughput of the underlying AB implementation.

The second considered workload is the write-dominated configuration of STMBench7 [41]. STMBench7 is a complex benchmark that features a number of operations with different levels of complexity over an object-graph with millions of objects. In the write-dominated configuration, this benchmark generates around 50% of update transactions and yields, with the baseline replication protocol, a moderate contention level causing an abort rate ranging from 5% (for 2 nodes) to around 15% (for 8 nodes). Also in this scenario (see Figure 4.6) SPECULA achieves remarkable speedups, up to a 4.4x factor in the configuration with 6 nodes and speculation level equal to 8. Interestingly, unlike in the previously analyzed scenario, with STMBench7, an excessively high setting for the speculation level can actually hinder SPECULA's performance. This is justifiable observing the trends of the abort rate and AB-delivery latency in the two plots of Figure 4.6(b) and 4.6(c). For speculation levels lower than 8, the AB-delivery (in log scale) remains comparable with that of the baseline, and the abort rate is even lower when using SPECULA. This reduction of the transaction abort probability is imputable to the fact that the burst of speculatively committed transactions by each replica has a high chance of being final delivered without the interleaving of messages associated with remote transactions, at least at low/moderate levels of load for the AB layer. As the speculation level increases, on the other hand, the GCS load accordingly increases, and the chances that the burst of transactions speculatively committed by a node are final delivered without interleaving commit requests for remote transactions decrease drastically. The result is a rapid growth of the abort rate, especially as the number of nodes, and hence the concurrency in the system, increases.

Finally, Figure 4.7 analyzes the read-dominated workload of STMBench7, a setting in which this benchmark generates 94% of read-only transactions. This is clearly a least favorable scenario for SPECULA since, on average, 94% of the incoming transactions can be executed very efficiently by the baseline algorithm, i.e. locally and without the risk of incurring in aborts or blocks. Also, the speculative transaction processing mechanism at the core of SPECULA is triggered only for 6% of the transactions. Nevertheless, even in this unfavorable scenario, SPECULA achieves comparable, or even higher throughput in almost all the evaluated scales of the platform (Figure 4.7(a)). The only exception is the case of 2 nodes, where the baseline can benefit from a significantly lower AB-delivery latency (Figure 4.7(b)). The abort rates for this scenario remain below 7% both for the baseline and for SPECULA in all the considered settings (Figure 4.7(c)).

Analogously to the Bank benchmark's scenario, also in this case increasing the speculation level beyond 4 does not pay off in SPECULA. In fact, as two update transactions are on average interleaved by a relatively high number of read-only transactions (that do not incur in any replica synchronization phase in the baseline) the gains achievable in SPECULA by overlapping the phases of processing and replica coordination (for update transactions) are quite reduced.

# Chapter 5

# Reducing Full Replication Costs by Leveraging Transactions Migration

This Chapter advances the state of the art by presenting an innovative, fully decentralized, LocalIty-aware LeAse-based repliCated TM (LILAC-TM) protocol [47], whose aim is to maximize the system throughput via a distributed self-optimizing lease circulation scheme, based on the idea of dynamically determining whether to migrate transactions to the nodes that own the leases required for their validation, or to demand the acquisition of these leases by the transaction originating node.

LILAC-TM builds on and extends the Asynchronous Lease-based Certification (ALC) protocol [18] (which has been already overviewed in Section 2.1). As in conventional transaction certification-based protocols [74, 72, 81], in ALC a transaction is executed exclusively on its originating node, which allows for higher scalability compared to classic active replication schemes [55]. Also, in ALC the replicas run a consensus protocol (embodied by an Atomic Broadcast) only in order to establish *leases* on portions of the data set, rather than to determine the serialization order of every update transaction (as done in certification-based protocols). Leases grant nodes temporary privileges on the management of a subset of the replicated data-set, with the advantage that an update transaction can be locally validated if, at commit time, the execution site owns the leases on the data-objects accessed by the transaction. Further, the node's ownership of a lease on a data set shelters the local transactions from aborts due to conflicts with remote transactions, which are the most expensive conflicts to deal with, as they are detected only at commit time and after an expensive distributed coordination phase.

Therefore the ALC scheme is able to exploit data locality to reduce the impact of data replication because transactions that already own all the necessary leases on the originating nodes can be replicated via a more cheaper reliable broadcast service, i.e. URB, and they do not undergo a more expensive consensus protocol among the replicas.

Nevertheless, if on one hand ALC is proved to provide significant gains if compared to classical certification-based schemes in the particular cases in which the workload is almost completely partitioned among replicas, it can suffer from the same replication costs of certification-based protocols in scenarios that do not reveal considerable gaps among nodes in the frequency of accesses on a given data-subset. In those scenarios, in fact, ALC is not able to maintain low the leases circulation frequency, and the scheme may collapse to a certification-based approach in which an instance of consensus is triggered for any executed transaction.

Lilac-TM's flexibility in deciding whether to migrate code or data not only allows it to take advantage of the data locality present in many application workloads, but also to further enhance it by turning a node $N$ that frequently accesses a set of data items $D$ into an attractor for transactions that access subsets of $D$ (and that could be committed by $N$ avoiding any lease circulation). This allows Lilac-TM to provide two key benefits: (i) limiting the frequency of lease circulation, and (ii) enhancing the contention management efficiency. In fact, with Lilac-TM, conflicting concurrent transactions have a significantly higher probability of being executed on the same node, which prevents them from incurring the high costs of distributed conflicts. The contributions of this Chapter are the following:

1. Lilac-TM is presented, namely a locality-aware lease-based replication protocol for fully replicated transactional systems that optimizes lease circulation by dynamically deciding whether to circulate transactional data or transactional code. A fully-fledged prototype of Lilac-TM is also presented, which implements a fully replicated Software Transactional Memory (STM) based on Java technology.

2. ALC [18] generates *a single* lease request for the entire transaction dataset. This limits the exploitation of data-locality, since another local transaction may reuse the lease only if its data set is a subset of another lease owned by the node. To allow efficient exploitation of data locality by Lilac-TM, a new version of ALC that supports fine-grained leases (FGL-ALC) is presented, which, instead of acquiring one lease for the entire data set, acquires a set of leases, one per each item of the data set.

3. A comprehensive comparative performance analysis is conducted, by evaluating the performance gains obtained by the new locality-aware in-

frastructure and algorithms developed in comparison with ALC. The experimental results establish that replacing ALC by FGL-ALC yields significant performance boost for workloads possessing data locality. When Lilac-TM is used on top of the new lease-management infrastructure, performance gains are increased significantly, providing up to 3.2 times the throughput of the baseline implementation.

The remainder of this Chapter is structured as follows. Section 5.1 gives an overview of the ALC protocol. The Lilac-TM protocol is presented in Section 5.2, while Section 5.3 provides arguments on the correctness of Lilac-TM. Section 5.4 reports the results of the experimental analysis.

## 5.1 Overview of ALC

The original ALC protocol (fully described in [18]) works as follows. A transaction is executed based on local data, avoiding any inter-replica synchronization until it enters its commit phase. At this stage, ALC acquires a lease for the transaction's accessed data items, before proceeding to validate the transaction. Leases are associated with data items indirectly, namely through conflict classes. This allows to flexibly control the granularity of the leases abstraction, trading off accuracy (i.e., avoidance of aliasing problems) for efficiency (amount of information exchanged among nodes and maintained inmemory) [6]. In case a transaction $T$ is found to have accessed stale data, $T$ is re-executed without releasing the lease. This ensures that, during $T$'s re-execution, no other replica can update any of the data items accessed during $T$'s first execution, which guarantees the absence of remote conflicts on the subsequent re-execution of $T$, provided that the same set of conflict classes accessed during $T$'s first execution is accessed again.

To establish lease ownership and disseminate updates of committed transactions, ALC employs the OAB communication service for delivering lease request messages, and the URB service for delivering data items of committed transactions and for lease-release messages.

The ownership of a lease ensures that no other replica will be allowed to validate any conflicting transaction, making it unnecessary to enforce distributed agreement on the global serialization order of transactions. ALC takes advantage of this by limiting the use of atomic broadcast exclusively for establishing the lease ownership. Subsequently, as long as the lease is owned by the replica, transactions can be locally validated and their updates can be disseminated using URB, which can be implemented in a much more efficient manner than OAB.

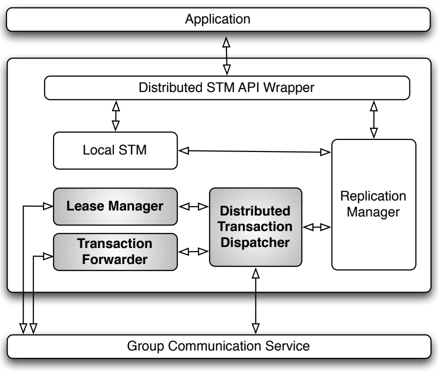

Application
Distributed STM API Wrapper
Local STM
Replication Manager
Lease Manager
Transaction Forwarder
Distributed Transaction Dispatcher
Group Communication Service

Figure 5.1: Middleware architecture of a Lilac-TM replica.

## 5.2 Lilac-TM

Figure 5.1 provides an overview of the software architecture of each replica of Lilac-TM, highlighting in gray the modules that were either re-designed or that were not originally present in ALC.

The top layer is a wrapper that intercepts application level calls for transaction demarcation without interfering with application accesses (read/write) to the transactional data items, which are managed directly by the underlying local STM layer. This approach allows transparent extension of the classic STM programming model to a distributed setting.

The prototype of Lilac-TM has been built by extending the ALC implementation shipped in the GenRSTM framework [19]. GenRSTM has been designed to support, in a modular fashion, a range of heterogeneous algorithms across the various layers of the software stack of a replicated STM platform. Lilac-TM inherits this flexibility from GenRSTM. The object-based implementation of TL2 [30] provided in GenRSTM is also used as the local STM layer. Since the TL2 algorithm is not multi-version based, transactions are always allowed to observe at most the last committed version of a key in the previously defined data model. As in SPECULA, a key is represented by a *Transactional Container* (*TC*) that wraps an object by pointing to the its last committed version.

The Replication Manager (RM) is the component in charge of interfacing the local STM layer with its replicas deployed on other system nodes. The RM is responsible of coordinating the commit phase of both remote and local transactions by: (i) intercepting commit-request events generated by local transactions and triggering a distributed coordination phase aimed at determining transactions' global serialization order and detecting the presence of conflicts with concurrently executing remote transactions; and (ii) validating remote transactions and, upon successful validation, committing them by atomically applying their write-sets in the local STM.

At the bottom layer there is a GCS (e.g. Appia [66]), which provides the view synchronous membership, OAB and URB services.

The role of the Lease Manager (LM) is to ensure that no two replicas simultaneously disseminate updates for conflicting transactions. To this end, the LM exposes an interface consisting of two methods, *GetLease()* and *FinishedXact()*, which are used by the RM to acquire/release leases on a set of data items. This component was originally introduced in ALC and has been re-designed in this work to support *fine-grained leases*. As explained in more detail in Section 5.2.1, fine-grained leases facilitate the exploitation of locality and consequently reduce lease circulation.

The Transaction Forwarder (TF) is responsible for managing the forwarding of a transaction to a different node in the system. The transaction forwarding mechanism represents an alternative mechanism to the lease-based certification scheme introduced in ALC. Essentially, both transaction forwarding and lease-based replication strive to achieve the same goal: minimizing the execution rate of expensive Atomic Broadcast-based consensus protocols to determine the outcome of commit requests. ALC's lease mechanism pursues this objective by allowing a node that owns sufficient leases to validate transactions and disseminate their write-sets without executing consensus protocols. Still, acquiring a lease remains an expensive operation, as it requires the execution of a consensus protocol.

The transaction forwarding scheme introduced in this work aims at reducing the frequency of lease requests triggered in the system, by migrating the execution of transactions to remote nodes that may process them more efficiently. This is the case, for instance, if some node $p_i$ owns the set of leases required to certify and commit a transaction $T$ originated on some remote node $p_j$. In this scenario, in fact, $p_i$ could validate $T$ locally, and simply disseminate its write-set in case of success. Transaction migration may be beneficial also in subtler scenarios in which, even though no node already owns the leases required to certify a transaction $T$, if $T$'s originating node were to issue a lease request for $T$, it would revoke leases that are being utilized with high frequency by some other node, say $p_k$. In this case, it is preferable to forward the transaction to $p_k$ and have $p_k$ acquire the lease on behalf of $T$, as this would reduce the frequency of lease circulation and increase throughput in the long term.

The decision of whether to migrate a transaction's execution to another node or to issue a lease request and process it locally is far from being a trivial one. The transaction scheduling policy should take load balancing considerations into account and ensure that the transaction migration logic avoids to excessively overload any subset of nodes in the system. In Lilac-TM, the logic for determining how to manage the commit phase of transactions is encapsulated by the Distributed Transaction Dispatching (DTD) module. This dissertation proposes two decision policies based on an efficiently solvable formulation in terms of an Integer Linear Programming optimization problem.

In the following, the key contributions of Lilac-TM are described, i.e. the fine-grained lease management scheme, the TF and the DTD.

### 5.2.1 Fine-Grained Leases

In ALC, a transaction requires a *single lease*, associated with its data-set in its entirety. A transaction $T$, attempting to commit on a node, may reuse a lease owned by the node only if $T$'s data-set is a subset of the lease's items set. Thus, each transaction is tightly coupled with a single lease ownership record. This approach has two disadvantages: i) upon the delivery of a lease request by a remote node that requires even a single data item from a lease owned by the local node, the lease must be released, causing subsequent transactions accessing other items in that lease to issue new lease requests; ii) if a transaction's data-set is a subset of a union of leases owned by the local replica but is not a subset of any of them, a new lease request must be issued. This forces the creation of new lease requests, causing extensive use of *AB-broadcast*.

To exploit data-locality, a new lease manager module has been introduced, which decouples lease requests from the requesting transaction's data-set. Rather than having a transaction to acquire a single lease encompassing its entire data-set, each transaction acquires a set of fine-grained *Lease Ownership Records* (LORs), one per accessed conflict class.

The new design only affected the original ALC's *Lease Manager* (LM) while the ALC's *Replication Manager* (RM) was not changed. In fact the latter still interfaces with the LM via the *GetLease()* and *FinishedXact()* methods for acquiring and releasing leases, respectively.

As in ALC, Lilac-TM maintains the indirection level between leases and data items through conflict classes. This allows flexible control of the leases abstraction granularity. The mapping between a data item and a conflict class is abstracted away through the *getConflictClasses()* primitive, taking a set of data items as input parameter and returning a set of conflict classes.

Each replica maintains one main data structure for managing the establishment/release of leases: *CQ* (Conflict-Queues), an array of FIFO queues, one per conflict class. The CQ keeps track of conflict relations among lease requests of different replicas. Each queue contains LORs, each storing the following data: (i) **proc**: the address of the requesting replica; (ii) **cc**: the conflict class this LOR is associated with; (iii) **activeXacts**: a counter keeping track of the number of active local transactions associated with this LOR, initialized to 1 when the LOR is created; and (iv) **blocked** : a flag indicating whether new local transactions can be associated with this LOR - this flag is initialized to false when the LOR is created (in the *createLorsForConflictClasses* primitive), and set to true as soon as a remote lease request is received.

**Algorithm 6** Lease Manager - Requesting leases (node $p_i$)

```
1:  $FIFOQueue\langle LOR\rangle$
2:  $CQ[NumConflictClasses] \leftarrow \{\perp, \dots, \perp\}$
3:
4:  $Set\langle LOR\rangle$ $GetLease(Set\ DataSet)$
5:      $ConflictClass[]\ CC \leftarrow getConflictClasses(DataSet)$
6:      if $\exists(Set\langle LOR\rangle)S \subseteq CQ$ s.t. $\forall cc \in CC(\exists lor \in S : (lor.cc = cc \wedge lor.proc = p_i \wedge \neg lor.blocked))$ then
7:          for all $lor \in S$ do
8:              $lor.activeXacts++$
9:      else
10:         $Set\langle LOR\rangle\ S \leftarrow createLorsForConflictClasses(CC)$
11:         $LeaseRequest\ req \leftarrow LeaseRequest(p_i, S)$
12:         $AB\text{-}broadcast([\mathsf{LeaseRequest}, req])$
13:     wait until $isEnabled(S)$
14:     return $S$
15:
16: upon $Opt\text{-}deliver([\mathsf{LeaseRequest}, req])$ from $p_k$ do
17:     $freeLocalLeases(req.cc)$
18:
19: upon $AB\text{-}deliver([\mathsf{LeaseRequest}, req])$ from $p_k$ do
20:     $Set\langle LOR\rangle\ S \leftarrow createLorsForConflictClasses(req.cc)$
21:     for all $lor \in S$ do
22:         $enqueue(CQ[lor.cc], lor)$
23:
24: $bool\ isEnabled(Set\langle LOR\rangle\ S)$
25:     return $\forall lor \in S, isFirst(CQ[lor.cc], lor)$
```

Algorithms 6 and 7 present the pseudo-code of LILAC-TM's LM. The method *GetLease()* is invoked by the RM once a transaction reaches its commit phase. The LM then attempts to acquire leases for all items in the committing transaction's data-set. It first determines, using the *getConflictClasses()* method, the set of conflict classes associated with the transaction's data-set (line 5 of Algorithm 6). It then checks (lines 6 of Algorithm 6) whether CQ contains a set $S$ of LORs such that i) the LORs were issued by $p_i$, and ii) additional transactions of $p_i$ may still be associated with these LORs (this is the case for each LOR owned by the current node that is not blocked). If the conditions of line 6 are satisfied, the current transaction can be associated

**Algorithm 7** Lease Manager - Freeing leases (node $p_i$)

```
1: $void\ FinishedXact(Set\langle LOR\rangle\ S)$
2:     $Set\langle LOR\rangle\ lorsToFree$
3:     for all $lor \in S$ do
4:         $lor.activeXacts--$
5:         if $lor.blocked \wedge lor.activeXacts = 0$ then
6:             $lorsToFree \leftarrow lorsToFree \cup lor$
7:     if $lorsToFree \neq \emptyset$ then
8:         $UR\text{-}broadcast([\mathsf{LeaseFreed}, lorsToFree])$
9:
10: upon $UR\text{-}deliver([\mathsf{LeaseFreed}, Set\langle LOR\rangle])$ from $p_k$ do
11:     $Set\langle LOR\rangle\ S \leftarrow createLorsForConflictClasses(req.cc)$
12:     for all $lor \in S$ do
13:         $dequeue(CQ[lor.cc], lor)$
14:
15: $void\ freeLocalLeases(ConflictClass[]\ CC)$
16:     $Set\langle LOR\rangle\ lorsToFree$
17:     for all $cc \in CC$ do
18:         if $\exists lor \in CQ[cc]$ s.t. $lor.proc = p_i$ then
19:             $lor.blocked \leftarrow \top$
20:             if $isFirst(CQ[lor.cc], lor) \wedge lor.activeXacts = 0$ then
21:                 $lorsToFree \leftarrow lorsToFree \cup lor$
22:     if $lorsToFree \neq \emptyset$ then
23:         $UR\text{-}broadcast([\mathsf{LeaseFreed}, lorsToFree])$
```

with all LORs in $S$ (lines 7-8 of Algorithm 6). Otherwise, a new lease request, containing the set of LORs, is created and is disseminated using $OAB$ (lines 9-12 of Algorithm 6). In either case, $p_i$ waits (see line 13) until $S$ is enabled, that is, until all the LORs in $S$ reach the front of their corresponding FIFO queues (see the *isEnabled*() method). Finally, the method returns $S$ and the RM may proceed validating the transaction.

When a transaction terminates, the RM invokes the *FinishedXact()* method (line 1 of Algorithm 7). This method receives a set of LORs and decrements the number of active transactions within each record (lines 3-4 of Algorithm 7). Every blocked LOR that is not used by local transactions is then released by sending a single message via the *UR-broadcast* primitive (lines 5-8 of Algorithm 7).

Upon an *Opt-deliver* event of a remote lease request *req*, $p_i$ invokes the *freeLocalLeases()* method, which blocks all LORs owned by $p_i$ that are part of *req* by setting their *blocked* field (line 19 of Algorithm 7). Then, all LORs that are blocked and are no longer in use by local transactions are released by sending a single *UR-broadcast* message (lines 20-23 of Algorithm 7). Other LORs required by *req* that have local transactions associated with them (if any) will be freed when the local transactions terminate. Blocking LORs is required to ensure the fairness of the lease circulation scheme. In order to prevent a remote process $p_j$ from waiting indefinitely for process $p_i$ to relinquish a lease, $p_i$ is prevented from associating new transactions with existing LORs as soon as a conflicting lease request from $p_j$ is *Opt-delivered* at $p_i$.

Upon an *AB-deliver* of a lease request *req* (line 19 of Algorithm 6), $p_i$ creates the corresponding set of LORs, and enqueues these records in their conflict class queues. The logic associated with a *UR-deliver* event (line 10 of Algorithm 7) removes each LOR specified in the message from its conflict class queue.

### 5.2.2 Transaction Forwarder

The TF is the module in charge of managing the process of migrating transactions between nodes. If at commit time the set $S$ of conflict classes accessed by a transaction $T$ is not already owned by its origin node, say $p_i$, the DTD may decide to avoid requesting leases for $T$, and forward its execution to a different node $p_j$. In this case node $p_j$ becomes responsible for finalizing the commit phase of the transaction. This includes establishing leases on $S$ on behalf of transaction $T$, which can be obtained avoiding any distributed coordination, in case $p_j$ already owns all the leases required by $T$. Else, if some of the leases requested by $T$ are not owned by $p_j$, this node has to issue lease requests on behalf of $T$ via the OAB service.

Furthermore the TF can use a remote validation optimization and let $p_j$

perform $T$'s final validation upon arrival (without re-executing $T$) in order to detect whether $T$ has conflicts with concurrently committed transactions. In order to use this remote validation optimization, the TF module must be augmented with a TM-specific validation procedure and append the appropriate meta-data to forwarding messages. TM-specific adaptation and overhead can be avoided by simply always re-executing the forwarded transaction once it is migrated to $p_j$.

To prove that the remote validation optimization (without re-execution) cannot be transparent to the local concurrency scheme and it needs TM-specific adaptation procedure for migrated transactions, in the following an example of execution is considered, which points out the issue of validating a transaction $T$ on a node different from the originating one in Lilac-TM. In particular the following three transactions are considered:

$$T_1 : w_1(x)$$

$$T_2 : r_2(z), w_2(y)$$

$$T_3 : r_3(x), r_3(y), w_3(z)$$

where $T_1$ and $T_2$ are two transactions already validated after a lease acquisition on two different nodes and are going to be committed via URB service. Note that this is admissible because $T_1$ and $T_2$ are not conflicting and therefore they can be associated to non-overlapping conflict classes.

Due to the lack of total order in the URB service $T_1$ is committed on a node $p_i$ before the commit of $T_2$ on $p_i$, while there exists another node $p_j$ where the commit order of $T_1$ and $T_2$ is right the opposite, i.e. $T_2$ is committed before $T_1$ on $p_j$. In addition transaction $T_3$ begins on $p_i$ right after the commit of $T_1$ and it is optimistically executed according to the local TM concurrency control scheme, i.e. TL2, till its commit phase and before $T_2$ commits on $p_i$. Next the TF on $p_i$ migrates $T_3$ on $p_j$. Assume that $T_3$ is delivered on $p_j$ after the commit of $T_2$ but before the commit of $T_1$ on $p_j$. Therefore the executed history $\mathcal{H}$ right after the migration of $T_3$ is the following:

$$\mathcal{H} = w_1(x_1), c_1, r_3(x_1), r_3(y_0), r_2(z_0), w_2(y_2), c_2$$

where $y_0$ and $z_0$ are the initial values of respectively $y$ and $z$, and $x_1$ is the value of $x$ written by $T_1$.

Assume that $T_1$ and $T_2$ are the only transactions committed, and therefore the following advancement of the global clock $gcl$ of the local TMs can be supposed: $gcl$ on $p_i$ is advanced to 1 by the commit of $T_1$ on $p_i$ while $gcl$ on $p_j$ is advanced to 1 by the commit of $T_2$ on $p_j$. This means that transaction $T_3$ started on the snapshot 1 on $p_i$, which in not equivalent to the snapshot 1 on $p_j$. In this case, the local TM on $p_j$ would not be able to distinguish that difference without additional information.

At this point, if LILAC-TM relied on the classical validation implemented by the local TM on $p_j$ (or other timestamp-based TMs usually adopted in DTM [24]), it could commit transaction $T_3$ because it can consider the snapshot read by $T_3$ not older than the current committed snapshot on $p_j$. Clearly this would lead to an incorrect, i.e. non serializable, history because it is not possible to choose a unique serialization point for both $T_2$ and $T_3$: (i) $T_3$ would appear as executed before $T_2$ since it misses the $T_2$'s write on $y$ and (ii) $T_2$ would appear as executed before $T_3$ since it misses the $T_3$'s write on $z$. The result would be the following history $\mathcal{H}'$:

$$\mathcal{H}' = w_1(x_1), c_1, r_3(x_1), w_3(y_0), r_2(z_0), w_2(y_0), c_2, w_3(z_3), c_3$$

The solution adopted in LILAC-TM is associating with every conflict class $C$ on every node $p_i$ a commit counter $cclock_{(C,i)}$ that is incremented whenever a commit of a transaction writes a new value of a datum $x$ belonging to $C$. In addition, upon a read operation on $x$ of a transaction $T$ in execution on node $p_i$, the transaction inserts in its read-set the tuple $\langle x, cclock_{(C,i)}\rangle$.

Therefore the validation procedure after the migration of $T$ on another node $p_j$, checks whether for any tuple $\langle x, cclock_{(C,i)}\rangle$ in $T$'s read-set, the value $cclock_{(C,i)}$ is not strictly less than the value $cclock_{(C,j)}$.

This mechanism is enough to mimic the validation procedure of classical certification-based replication schemes because for any pair of nodes $p_i$, $p_j$, the write commits on a given conflict class $C$ are observed in the same order by both $p_i$ and $p_j$, i.e. LILAC-TM guarantees total order among committed transactions that write on the same conflict class C.

The last property is easily provable by observing communication services in LILAC-TM guarantee *causal order* and therefore:

– For any pair of transactions $T_i$ and $T_j$ committed by the same node $p_i$, if $T_i$ and $T_j$ both write on a common conflict class $C$ then the commits of $T_i$ and $T_j$ are delivered to all nodes in the same order and by following the order of the broadcast of their commit messages.

– For any pair of transactions $T_i$ and $T_j$ committed by different nodes, i.e. respectively $p_i$ and $p_j$, if $T_i$ and $T_j$ both write on a common conflict class $C$ then the commits of $T_i$ and $T_j$ are delivered to all nodes in the same order and the order depends on the order of acquisition of the lease $L$ on $C$ (which is determined by the AB delivery order). If $L$ is acquired by $p_i$ first (respectively $p_j$ first), then it will be released to $p_j$ (respectively $p_i$) after the delivery of the commit message for $T_i$ on $p_i$ (respectively $T_j$ on $p_j$), and therefore the sending of the commit message for $T_j$ from $p_j$ (respectively for $T_i$ from $p_i$) will follow the delivery of $L$ on $p_j$ (respectively $p_i$).

In case of successful validation, $T$ can be simply committed, as in ALC, by disseminating a Commit message via the *UR-broadcast*. Additionally this has the effect of unblocking the thread that requested the commit of $T$ on node $p_i$. On the other hand, if $T$ fails its final validation, it is re-executed on node $p_j$ until it can be successfully committed, or it fails for a pre-determined number of attempts. In this latter case, the origin node $p_i$ is notified of the abort of $T$, and the user application is notified via an explicit exception type. Note that, in order to commit the transaction associated with the re-execution of $T$, which is denoted as $T'$, $p_j$ must own the set of conflict classes accessed by $T'$. This may not be necessarily true, as $T'$ and $T$ may access different sets of conflict classes. In this case, Lilac-TM prevents a transaction from being forwarded an arbitrary number of times, by forcing $p_j$ to issue a lease request and acquire ownership of the leases requested by $T'$.

It must be noted that, in order to support the transaction forwarding process, the programming model exposed by Lilac-TM has to undergo some minor adaptations compared, e.g., with the one typically provided by non-replicated TM systems. Specifically, Lilac-TM requires that the transactional code is replicated and encapsulated by an interface that allows to seamlessly re-execute transactions originating at different nodes. To this end, the transactional logic is wrapped in an object whose attributes encode its input parameters, and which exposes methods (i) supporting its correct serialization and de-serialization, and (ii) allowing to trigger the execution of the transactional logic, possibly on a remote node (similarly to RMI [101]). In order to maximize the generality and flexibility of the programming model, the method that supports the execution of a transaction is allowed to return a (typed) result. In case of re-execution on node $p_j$ of a transaction $T$ forwarded by node $p_i$, the transaction's result is piggybacked on the commit message. This allows to inform the application thread that originated the execution of $T$ on $p_i$ about the result generated by $T$ on $p_j$ (which may be different from the one originally produced by $T$ on $p_i$).

### 5.2.3 Distributed Transaction Dispatching

The DTD module allows encapsulating arbitrary policies to determine whether to process the commit of a transaction locally, by issuing lease requests if required, or to migrate its execution to a remote node. In the following this problem is defined as the *transaction migration problem.* It can be formulated as an Integer Linear Programming (ILP) problem that takes as input the set of leases required by the transaction, the *CPU* utilization of the nodes in the system, and a cost function $C(i, S)$, which returns the cost of acquiring the leases for the transaction on the set $S$ if the node $p_i$ is selected for managing the commit phase of the transaction.

The ILP is defined as follows:

$$\mathbf{min} \sum_{i \in \Pi} N_i \cdot C(i, S) \tag{5.1}$$

subject to:

$$\sum_{i \in \Pi} N_i = 1 \tag{5.2}$$

$$CPU_i \cdot N_i < maxCPU \tag{5.3}$$

The above problem formulation aims at determining an assignment of the binary vector $N$ (whose entries are all equal to 0 except for one, whose index specifies the selected node) minimizing a generic cost function $C(i, S)$ that expresses the cost for node $p_i$ to be selected for managing the commit phase of a transaction accessing the conflict classes in the set $S$. The optimization problem specifies two constraints. Constraint 5.2 expresses the fact that a transaction can be certified by exactly a single node in $\Pi$. Constraint 5.3 is used to avoid load imbalance between nodes. It states that a node $p_i$ should be considered eligible for re-scheduling only if its CPU utilization ($CPU_i$) is below a maximum threshold ($maxCPU$).

Two different policies have been derived for satisfying the above ILP formulation, which are designed to minimize the *long-term* and the *short-term* impact of the decision on how to handle a transaction. First the cost function $LC(i, S)$ has been defined, which models the *long-term cost* of selecting node $p_i$ as the node that will execute the transaction as the sum of the frequency of accesses to the conflict classes in $S$ by every other node $p_j \neq p_i \in \Pi$:

$$LC(i, S) = \sum_{x \in S} \sum_{p_j \in \Pi \wedge j \neq i} \mathcal{F}(j, x)$$

where $\mathcal{F}(j, x)$ is defined as the per time-unit number of transactions originated on node $p_j$ that have object $x$ in their dataset.

In order to derive the *short-term policy*, the function $SC(i, S)$ has been defined, which expresses the immediate costs induced at the GCS level by different choices of where to execute a transaction:

$$SC(i, S) = \begin{cases} c_{URB} & \text{if } i = O \wedge \forall x \in S : \mathcal{L}(i, x) = 1 \\ c_{AB} + 2c_{URB} & \text{if } i = O \wedge \exists x \in S : \mathcal{L}(i, x) = 0 \\ c_{p2p} + c_{AB} + 2c_{URB} & \text{if } i \neq O \wedge \exists x \in S : \mathcal{L}(i, x) = 0 \\ c_{p2p} + c_{URB} & \text{if } i \neq O \wedge \forall x \in S : \mathcal{L}(i, x) = 1 \end{cases}$$

where $p_O$ is the node that originated the transaction, and $c_{URB}$, $c_{AB}$ and $c_{p2p}$ are the costs of performing a URB, an AB, and a point-to-point communication, respectively. The above equations express the cost of the following scenarios (from top to bottom): i) the originating node already owns all the leases required by it; ii) the originating node does not own all the necessary leases and issues a lease request; iii) the originating node forwards the transaction to a node that does not own all the necessary leases; iv) the transaction is forwarded to a node that owns the leases for all required conflict classes. The DTD can be configured to use the long-term or the short-term policy simply by setting the generic cost function $C(i, S)$ in 5.1 to, respectively, $LC(i, S)$ or $SC(i, S)$.

It is easily seen that the ILP can be solved in O($|\Pi|$) time regardless of whether the long-term or the short-term policy is used. The statistics required for the computation of the long-term policy are computed by gathering the access frequencies of nodes to conflict classes. This information is piggybacked on the messages exchanged to commit transactions/request leases. A similar mechanism is used for exchanging information on the CPU utilization of each node. For the short term policy, the costs of the P2P, URB and OAB protocols are quantified in terms of their communication-steps latencies (which are equal to 1, 2, and 3, respectively).

## 5.3 Correctness Arguments

The formal proof about the correctness guarantees of Lilac-TM is not provided because the one provided by the ALC is trivially inherited. In particular the optimistic execution of a transaction on each node is regulated by a local TM (TL2 in the Lilac-TM specific implementation), which guarantees each transaction (even the one that does not commit) always observes a consistent snapshot of the memory, namely a snapshot obtained by the commit of a serializable history of write transactions (as required by Opacity as well). Therefore read-only transactions that do not abort during the execution phase can safely commit and appear as atomically executed at their beginning.

Furthermore the commits of every pair of committed write transactions that conflict on at least one datum $x$ (i.e. both transactions access at least a common datum $x$ and at least one of them executes a write operation on $x$) are serialized (even if committed by different nodes) because Lilac-TM forces a serialization via lease acquisition on $x$. In addition, if the validation mechanism is taken into account, namely upon a lease acquisition a write transaction $T$ undergoes the classical validation procedure to check that no other concurrent transaction (the one that committed after $T$ began) has committed by writing on a datum in $T's$ read-set [74], then the history of write transactions committed by Lilac-TM is 1CS.

Clearly it is not possible to claim either that the whole history of committed transactions (including both read-only and update transactions) is 1CS or that the history of the executed transactions is opaque since, as in ALC, Lilac-TM does not enforce a total order among the commit events of non-conflicting transactions. This can leads to scenarios in which two different transactions $T_i$ and $T_j$ in execution respectively on nodes $p_i$ and $p_j$ can observe the commits of another pair of write transactions, e.g. $T_k$ and $T_h$, in different order.

Nevertheless this scenario can happen only if $T_i$ and $T_j$ are either read-only transactions or update transactions that are eventually aborted. In fact, Lilac-TM, ensures the global serializability for the history restricted to committed update transactions, by certifying them according to the order determined by the lease acquisition scheme.

Technically Lilac-TM guarantees EUS because:

– every history restricted to committed write transactions is serializable;

– every transaction always observes a history of serializable committed write transactions in its past.

## 5.4 Experimental Evaluation

In this Section, the performance of Lilac-TM is compared with that of the baseline ALC protocol. Performance is evaluated using two benchmarks: a variant of the *Bank* benchmark [49] and the *TPC-C* benchmark [93]. The following algorithms are compared: ALC (using the implementation evaluated in [19]), FGL (ALC using the fine-grained leases mechanism), MG-ALC (ALC extended with the transaction migration mechanism), and two variants of Lilac-TM (transaction migration on top of ALC using fine-grained leases), using the short-term (Lilac-TM-ST) and the long-term (Lilac-TM-LT) policies, respectively. The source code of ALC, Lilac-TM and the benchmarks used in this study are publicly available [1].

All benchmarks were executed running 2 threads per node, and using a cluster of 4 replicas, each comprising an Intel Xeon E5506 CPU at 2.13 GHz and 32 GB of RAM, running Linux and interconnected via a private Gigabit Ethernet.

**Bank.** The *Bank* benchmark [20, 49] is a well-known transactional benchmark that emulates a bank system comprising a number of accounts.

This benchmark has been extended with various types of read-write and read-only transactions, for generating more realistic transactional workloads. A *read-write transaction* performs transfers between randomly selected pairs of accounts. A *read-only transaction* reads the balance of a set of randomly-selected client accounts. Workloads consist of 50% read-write transactions and 50% read-only transactions of varying lengths.

[1]`http://aristos.gsd.inesc-id.pt`

Data locality has been also introduced in the benchmark as follows. Accounts are split into *partitions* such that each partition is logically associated with a distinct replica and partitions are evenly distributed between replicas. A transaction originated on replica $r$ accesses accounts of a single (randomly selected) partition associated with $r$ with probability $P$, and accounts from another (randomly selected) remote (associated with another replica) partition with probability $1-P$. Larger values of $P$ generate workloads characterized by higher data-locality and smaller inter-replica contention. Hence, the optimal migration policy is to forward a transaction $T$ to the replica with which the partition accessed by $T$ is associated. Therefore a third variant of Lilac-TM (called Lilac-TM-OPT) using this optimal policy has been implemented and evaluated.[2]

Figure 5.2(a) shows the throughput (committed transactions per second) of the algorithms that have been evaluated on workloads generated by the bank application with $P$ varying between 0% to 100%.

Comparing ALC and FGL, Figure 5.2(a) shows that, while ALC's throughput remains almost constant for all locality levels, FGL's performance dramatically increases when locality rises above 80%. This is explained by Figure 5.2(b), that shows the *Lease Reuse Rate*, defined as the ratio between the number of read-write transactions which are piggy-backed on existing leases and the total number of read-write transactions.[3] A higher lease reuse rate results in fewer lease requests, which reduces in turn the communication overhead and the latency caused by waiting for leases. FGL's lease reuse rate approaches one for high locality levels, which enables FGL and FGL-based migration policies to achieve up to 3.2 times higher throughput as compared with ALC and MG-ALC.

When locality is lower than 80%, the FGL approach yields throughput that is comparable to ALC. Under highly-contended low-locality workloads, FGL's throughput is even approximately 10%-20% lower than that of ALC. This is because these workloads produce a growing demand for leases from all nodes. FGL releases the leases in fine-grained chunks, which results in a higher load on *URB*-communication as compared with ALC.

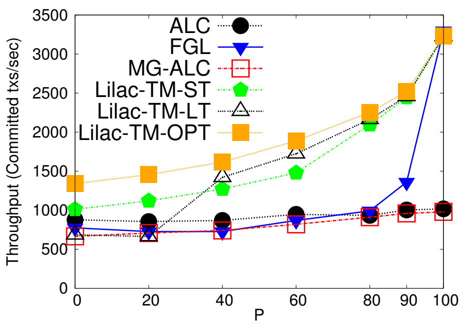
The adverse impact of low-locality workloads on transaction migration policies, however, is much lower. Migrating transactions to replicas where leases might already be present (or will benefit from acquiring it), increases the lease reuse rate, which increases throughput in turn. Indeed, as shown by Figure 5.2(a), Lilac-TM achieves speed-up of between 40%-100% even for low-locality workloads (0%-60%) in comparison with ALC. For high-locality workloads, both FGL and Lilac-TM converge to similar performance, out-

[2]MG-ALC implementation also uses this optimal migration policy.

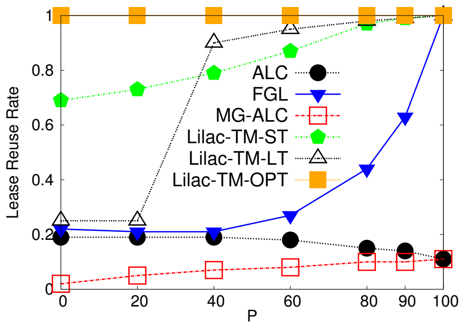
[3]Read-only transactions never request leases.

performing ALC by a factor of 3.2.

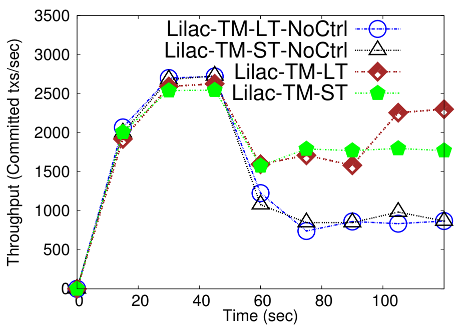
Comparing the performance of ALC and MG-ALC shows that using transaction migration on top of ALC does not improve the lease reuse rate as compared with ALC. This is because migration only helps when used on top of the fine-grained leases mechanism. The slightly lower throughput of MG-ALC vs ALC is due to the overhead of the TF mechanism.



(a) Throughput.



(b) Lease reuse rate.



(c) Overload.

Figure 5.2: Bank Benchmark.

Next the ability of Lilac-TM to cope with load imbalance has been evaluated. To this end, the benchmark is set to access with 20% probability a single partition, $p$, from all the nodes, except for the single node, say $n$, associated with $p$, which accesses only $p$. In these settings, with all the considered policies, $n$ tends to attract all the transactions that access $p$. At second 40 of the test, node $n$ has been overloaded by injecting external, CPU-intensive jobs. The plots in Figure 5.2(c) compare the throughput achieved by Lilac-TM with and without the mechanism for overload control (implementing Inequality (3)), and with both the long-term and the short-term policies. The data highlights the effectiveness of the proposed overload control mechanism, which significantly increases system throughput. In fact, the schemes that exploit statistics on CPU utilization (Lilac-TM-ST and Lilac-TM-LT) react in a timely manner to the overload of $n$ by avoiding further migrating their transactions towards it, and consequently achieve a throughput that is about twice that of uninformed policies (Lilac-TM-ST-NoCtrl and Lilac-TM-LT-NoCtrl).

**TPC-C.** The TPC-C benchmark has been also ported and used to evaluate Lilac-TM. The TPC-C benchmark is representative of OLTP workloads and is useful to assess the benefits of Lilac-TM even in the context of complex workloads that simulate real world applications. It includes a wider variety of transactions that simulate a whole-sale supplying *items* from a set of *warehouses* to *customers* within sales *districts*. Two of the five transactional profiles offered by TPC-C have been ported, namely the *Payment* and the *New Order* transactional profiles, which exhibit high conflict rate scenarios and long running transactional workloads, respectively. For this benchmark transactions have been injected to the system by emulating a load balancer operating according to a geographically-based policy that forwards requests on the basis of the requests' geographic origin: in particular requests sent from a certain geographic region are dispatched to the node that is responsible for the warehouses associated with the users of that region. To generate more realistic scenarios it has been also assumed that the load balancer can do mistakes by imposing that with probability 0.2 a request sent from a certain region is issued by users associated with warehouses that do not belong to that region.

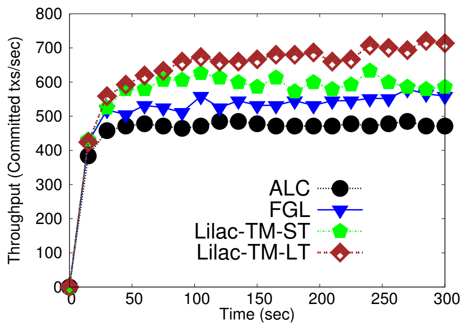


Figure 5.3: TPC-C.

Figure 5.3 presents the throughput obtained by running a workload with 95% Payment transactions and 5% New Order transactions; moreover in this case the throughput is shown varying over time in order to better assess the convergence of the reschedule policies. Even in this complex scenario FGL performs better than ALC due to better exploitation of the application, and a higher leases reuse rate. In addition, using the migration mechanism, driven by either the short term (ST) or the long term (LT) policy, over FGL, it achieves speedups of between 1.2 and 1.5 when compared to ALC. However, unlike the Bank Benchmark, in this case the ST policy achieves only minor gains compared to the LT policy, due to TPC-C's transactional profiles that generate more complex access patterns. In fact, even when the data-set is partitioned by identifying each partition as a *warehouse* and all the objects associated with that *warehouse*, TPC-C's transactions may access more than one partition. This reduces the probability that the ST policy can actually trigger a reschedule for a transaction on a node that already owns all the leases necessary to validate/commit that transaction. On the other hand the LT policy can exploit application locality thus noticeably reducing lease requests circulation, i.e. the number of lease requests issued per second.

# Chapter 6

# Changing the Viewpoint: a Scalable Multi-Version Protocol under Genuine Partial Replication

As showed in the previous Chapters, protocols like SPECULA or Lilac-TM are able to noticeably reduce the impact of replication on the transaction processing performance. While the former exploits speculative techniques to tentatively mask the cost of synchronization among nodes, the latter is able to leverage application locality to reduce the probability of remote synchronizations.

Nevertheless, since both the solutions are targeted for a fully replicated system model, i.e. every node stores the whole dataset, their design is inherently not scalable. In fully replicated environments, update transactions need to update synchronously *all* nodes in the system. As the cost of processing update transactions grows (at least) linearly with the number of nodes in the system, full replication schemes are strongly inefficient in large scale systems.

In order to avoid this problem, and enhance scalability, most of today's datastores or transactional systems rely on partial replication schemes, in which each datum is replicated on a (typically small) subset of nodes, i.e. partial replication. Further, these systems often ensure more relaxed consistency models, in order to achieve benefits in terms of execution latency or partition tolerance [27, 58, 4]. This is the approach taken by most of the reference transactional data stores, such as the NoSQL data management platforms (e.g. Cassandra, BigTable, Infinispan) currently playing a core role in cloud based systems.

On the other hand, an orthogonal approach to cope with scalability is represented by the so called Genuine Partial Replication (GPR), e.g. [86]. GPR maximizes scalability by ensuring that, for any transaction $T$, only the sites that replicate the data items read or written by $T$ exchange messages to decide the final outcome (commit/abort) of $T$. Unfortunately, existing GPR solutions introduce considerable overhead, as they require read-only transactions (that are largely predominant in typical applications' workloads [3]) to undergo expensive distributed validation phases.

Therefore, by having in mind as first class requirements both consistency and scalability, this Chapter shifts from the fully replicated system model (targeted by SPECULA and LILAC-TM) by exploiting GPR as the means for coping with (very) large scale systems. In particular it presents GMU [79], namely a Genuine Multiversion Update serializability protocol, which stands as the first GPR proposal guaranteeing that read-only transactions are never aborted or forced to undergo any additional remote validation phase.

The core of GMU is a distributed multiversion concurrency control algorithm, which relies on a novel vector clock [59] based synchronization mechanism to track, in a totally decentralized (and consequently scalable) way, both data and causal dependency relations among transactions.

In terms of formal properties, GMU (as its name suggests) ensures the Extended Update Serializability (EUS) consistency criterion. Therefore it provides guarantees analogous to those offered by classic One-Copy Serializability (1CS) for update transactions, thus ensuring consistent evolution of the system's state. Further it guarantees that all transactions, whether they have to be eventually aborted or not, observe consistent snapshots. This property can be important when applications execute in non-sandboxed environments (such as Transactional Memory systems [2]) and may behave erroneously upon observing non-serializable histories.

The GMU protocol is the first GPR protocol that exploits EUS semantics in order to implement a scalable distributed multiversioning concurrency control scheme that does not introduce any global synchronization point (such as logically centralized clocks [54]), and does not require expensive remote validation phases to commit read-only transactions [86].

In addition, GMU is also able to enrich the EUS semantic, by inheriting an interesting property from Opacity (and hence Strict Serializability) [40, 71]: it enforces a lowerbound on the staleness of the read data. In particular, despite distribution, in a quiescent state a transaction $T$ in GMU is always allowed to see the freshest snapshots produced by transactions that had committed before $T$ started.

GMU has been integrated into Infinispan [65], which is a mainstream open-source transactional in-memory data grid developed by the Red-Hat/JBoss team. Given that the native replication mechanism implemented in Infinispan only supports non-serializable consistency (i.e., Repeatable Read), this data grid represents an ideal baseline to evaluate the additional costs incurred by GMU to ensure EUS consistency. The study has been based on TPC-C [93], an industry standard benchmark for OLTP systems, and YCSB [23], a recent benchmark for distributed key-value stores. The results show that GMU achieves linear scalability up to 40 nodes and provides stronger consistency than Repeatable Read at a negligible cost (less than 10% reduction of the system throughput).

The remainder of this Chapter is structured as follows. The GMU protocol is introduced in Section 6.1, while its formal correctness proof is presented in Section 6.2. Section 6.3 provides a discussion on how GMU is able to reduce the staleness of the data read, while the results of the experimental study are illustrated in Section 6.4.

## 6.1 The GMU protocol

As classical multi-version concurrency control (MVCC) algorithms, GMU stores multiple versions of a same data item, and allows read-only transactions to observe consistent (which for GMU means Update Serializable), but possibly outdated snapshots of the available data. As in typical MVCC implementations (either centralized [12] or fully replicated [54]), in GMU each node arranges the locally stored versions of each data item into a chain tagged with a version identifier *vid* according to the data model described in Section 3.2. In GMU that identifier can be represented either as a scalar clock or a vector clock and the two options condition the way GMU implements read and commit operations. Unlike the protocol presented in [79], in this dissertation a *vid* is represented as a vector clock, resulting in a more intuitive description of the read and commit operations. In fact, for the sake of clarity, in this way there is no need of handling the translation of vector clocks to version identifiers since, as it will be shown later in the description, a transaction relies on vectors clocks both to execute read operations and finalize commit operations[1].

Other than following the total order of commit events on a datum $d$ to represent the sequence of $d$'s versions (as required by the data model described in Section 3.2), GMU also guarantees that the commit events of transactions that update any data item $d$ belonging to a partition $j$ are totally ordered among all replicas that replicate partition $j$ (namely, $g_j$).

[1]Note that this choice barely affects the memory footprint of the metadata required by the protocol, since in both options GMU has to store anyway the vector clock representation of each *vid* in a per-node separate structure. Therefore while in both options each version on every node requires to associate a vector clock identifier, the difference between the two options is that in the former a version also stores a scalar clock and in the latter a version also stores a pointer to the associated vector clock identifier.

More in general, GMU ensures total order among the commit events of update transactions that exhibit (possibly transitive) data dependencies. This is in fact what guarantees that the history restricted to update transactions generated by GMU is 1CS, as demanded by EUS.

However, unlike existing distributed/replicated MVCC protocols [89, 9, 102, 54], GMU does not order transaction commit events by relying on a centralized, or fully replicated, global clock. Conversely, GMU relies on a novel, highly scalable, fully distributed synchronization scheme that exploits vector clocks to achieve the twofold objective of:

1. determining which data item versions have to be returned by read operations issued by transactions;

2. ensuring agreement among the nodes replicating the data items updated by a transaction $T$ on the vector clock to associate with the commit event of $T$ and to be used when locally applying the write-set of $T$.

Before explaining these two key mechanisms of GMU, the main data structures locally maintained at each node $p_i$ are discussed, namely $CommitQueue$, $CLog$ and $LastPrepVC$.

$CommitQueue$ is an ordered queue whose entries are tuples $\langle T, vc, s\rangle$ such that $T$ is a transaction, $vc$ is its current commit vector clock, and $s$ is a value in the domain $\{pending,\ ready\}$. The entries stored in $CommitQueue$ at node $p_i$ are ordered according to the $i$-th entry of their vector clocks, i.e. $vc[i]$, and possible ties are broken using deterministic functions (e.g. hash functions) taking in input the transaction identifier $T$.

The semantic associated with the $s$ field is the following one. If $s$ is equal to *pending*, it means that $T$ is currently successfully prepared to commit on $p_i$ and is waiting for a final commit/abort decision from the transaction coordinator. In the following the $vc$ of a pending transaction is referred as its $prepareVC$. The *ready* value, instead, means that the transaction has already received (a) the commit decision from the transaction coordinator and that (b) it has already been assigned a final vector clock. Such a $vc$ is referred as the $commitVC$. As it will be discussed in the following, a *ready* transaction $T$ will be committed as soon as $T$ becomes the top standing transaction in the $CommitQueue$.

$CLog$ is a simple list that maintains, for each committed transaction, the triple $\langle T, commitVC, updatedKeys\rangle$, such that $T$ is the identifier of a committed transaction, $commitVC$ is the vector clock that $T$ used to commit and $updatedKeys$ is the set of keys locally stored by process $p_i$ and that $T$ has updated by committing. Therefore $commitVC$ is the vector clock pointed by the $vid$ of each new version committed by $T$, i.e. the last version associated to each key in the set $updatedKeys$ upon the commit of $T$.

The elements in the list $CLog$ on a node $p_i$ are totally ordered according to the value $commitVC[i]$ and the order follows the sequence of commits of write transactions that updated a key stored on $p_i$. In addition the vector clock associated to the most recent write transaction committed on $p_i$ is denoted as $CLog.mostRecentVC$, and $CLog$ is initialized as containing only one element $\langle -, commitVC_{init}, - \rangle$, where $CLog.mostRecentVC = commitVC_{init} = [0, \ldots, 0]$ in this case.

$LastPrepVC$, finally, is a vector clock that, as it will be discussed, is used by $p_i$, during the prepare phase of a transaction, to determine its $prepareVC$ on $p_i$. GMU maintains the following invariant on $LastPrepVC$: it is always greater than or equal to the vector clock associated to the last committed transaction in $CLog$, i.e. $\forall j, LastPrepVC[j] \geq CLog.mostRecentVC[j]$.

The following Sections present the details and the pseudocode formalizing the GMU protocol.

### 6.1.1 Transaction execution phase

GMU stores, in the transactional context of each executing transaction $T$, the following information:

1. The transaction vector clock $VC$, namely an array of scalar (integer) logical timestamps, having cardinality equal to the number of nodes in the system, which keeps track of the (data and causal) dependencies developed by the transaction during its execution. When $T$ starts its execution on node $p_i$, $T.VC$ is initialized with the values of $CLog.mostRecentVC$ on $p_i$.

2. The transaction read-set ($rs$ in the pseudocode), which stores the set of identifiers of the keys read by $T$.

3. The transaction write-set ($ws$ in the pseudocode), which stores, as a set of pairs $\langle key, value \rangle$, the identifiers and values of the keys written by the transaction.

4. An array of boolean values, called $hasRead$, which has an entry for each node in the system, and whose $j$-th entry stores the flag $\top$, i.e. true, if $T$ has executed a read operation on a key stored by $p_j$; otherwise the entry stores the flag $\bot$, i.e. false. If $T$ starts on node $p_i$ the $i$-th entry of $T$'s $hasRead$ is initialized to $\top$ in order to indicate that $T$ won't be allowed to read transactions committed on $p_i$ after its beginning.

The pseudocode describing the behavior of a transaction during its execution phase is reported in Algorithm 8 and Algorithm 9.

**Algorithm 8** Write and Read operations (node $p_i$).

```
void Write(Transaction T, Key k, Value val)
    $T.ws \leftarrow T.ws \setminus \{\langle k, - \rangle\} \cup \{\langle k, val \rangle\}$

Value Read(Transaction T, Key k)
    if $\langle k, val \rangle \in T.ws$ then
        return $val$
    if $p_i \in replicas(\{k\})$ then                                    ▷ k is local
        $[val, VC^*, last] \leftarrow doRead(k, T.VC, T.hasRead)$
        $T.hasRead[i] \leftarrow \top$
    else                                                                  ▷ k is remote
        send ReadRequest($[k, T.VC, T.hasRead]$) to all $p_j \in replicas(\{k\})$
        wait receive ReadReturn($[val, VC^*, last]$) from any $p_h \in replicas(\{k\})$
        $T.hasRead[h] \leftarrow \top$
    if $last = \perp \wedge T.ws \neq \emptyset$ then
        throw ABORT
    $T.VC \leftarrow \max(T.VC, VC^*)$
    $T.rs \leftarrow T.rs \cup \{k\}$
    return $val$

upon receive ReadRequest([Key k, VC xactVC, bool[] hasRead]) from $p_j$
    wait until $CLog.mostRecentVC[i] \geq xactVC[i]$
    $[val, VC^*, last] \leftarrow doRead(k, xactVC, hasRead)$
    send ReadReturn($[val, VC^*, last]$) to $p_j$
```

Write operations are simply handled by storing the key and the new value in the transaction write-set (lines 1-2 of Algorithm 8).

If a transaction $T$ issues a read operation on key $k$ at a process $p_i$, it is first checked whether $T$ has already written $k$. In this case, the value stored in $T$'s write-set is returned (lines 5-6 of Algorithm 8). Otherwise, $p_i$ determines whether the key to be read by transaction $T$ is local or not, thus giving rise to the following disjoint execution paths (line 7 of Algorithm 8).

**Local read.** If $p_i$ belongs to $replicas(\{k\})$, the read operation is a local one and the key's value is retrieved from the local data store via the *doRead*() function (lines 7-9 of Algorithm 8). This function (Algorithm 9) iterates over the versions of $k$ and returns the most recent one having a version identifier *vid* that does not exceed an upper bound defined for $T$ on $p_i$. This upper bound delimits the visible snapshot of $T$ on $p_i$ and it is determined in different ways depending on whether it is the first time that $T$ issues a read operation on a key stored by $p_i$.

If this is the case, the most recent local snapshot that is visible by $T$ is determined by iterating over $CLog$ (which stores the totally ordered list of the update transactions that committed at this node) and by discarding all the transactions $\bar{T}$ such that $\bar{T}$'s commit event depends (either directly or transitively) on the set of events associated with the first read operation issued by $T$ on any node (see lines 3-4 of Algorithm 9). Therefore, the remaining transactions $T^*$ are the ones having the commit vector clock $commitVC$ such that $commitVC$ does not exceed the reading transaction's vector clock $T.VC$ on the entries fixed by $T.hasRead$. This is verified if for each entry having value $\top$ in $T.hasRead$, $commitVC$ is not greater than $T.VC$.

Roughly speaking, the first time that $T$ issues a read on a node $p_i$, GMU serializes $T$ after transactions $T^*$, establishing an upper bound $MaxVC$ on the $freshness$ of the snapshots that $T$ can observe during subsequent reads. Specifically, $MaxVC$ is computed by merging via a per-entry maximum operation the $commitVC$ (stored in VisibleSet in the pseudocode) associated to transactions $T^*$ , and it prevents $T$ from observing versions committed, on any node, by transactions that depend/anti-depend (either directly or transitively) on the snapshot identified by $MaxVC$.

$MaxVC$ can therefore be used (line 9 of Algorithm 9) to determine the version of $k$ that is visible by the transaction $T$, namely the most recent version *ver* of $k$ that is committed by one among the transactions $T^*$. This means the selected *ver* has *vid* vector clock that does not exceed $MaxVC$ on the entries fixed by $T.hasRead$.

The behavior in case it is not the first time that $T$ reads on $p_i$ is similar, with the exception that the $MaxVC$ used to determine version visibility is the one already stored in the transaction's $VC$. This is an optimization provided by GMU that avoids to scan the $CLog$ of a generic node $p_j$ on subsequent read operations executed by transaction $T$ on the same (whether local or remote) node $p_j$. Note that if $T$ iterated over the $CLog$ again, it could not produce a $MaxVC$ with the $j$-th entry different from the $j$-th entry of $T.VC$ since $T$ already read on $p_j$. This is because of the rule that GMU uses to select the candidate vector clocks in $CLog$ and the way $T.VC$ is updated upon a read operation, i.e. via the max operator.

Finally, *doRead*() returns the value of the selected version, along with $MaxVC$ and a boolean flag that specifies whether the returned version of $k$ is the most recent currently committed. The opposite case is perfectly acceptable for read-only transactions, which can be serialized in the past as typical of MVCC algorithms. However, that is not for update transactions, and it does doom update transactions to abort (see lines 14-15 of Algorithm 8).

**Remote read.** If $p_i$ does not belong to $replicas(\{k\})$, the read operation is remote, with the meaning that the data to be read must be retrieved from some node $p_j$, with $j \neq i$ (lines 10-13 and 20-23 of Algorithm 8). In this case, a read request is sent to all the nodes replicating the key $k$, and it is waited for the first (namely the fastest) of their replies, e.g. $p_h$. Analogously to the local read case, the *doRead*() function is used by the remote nodes to determine the version of $k$ visible by transaction $T$. However, in this case GMU ensures that, before invoking *doRead*, the remote node $p_j$ has finalized the commit of all the update transactions which $T$ depends on, and that have written keys replicated by $p_j$.

**Algorithm 9** Version visibility logic (node $p_i$).

```
1: [Value,VC,bool] doRead(Key k, VC xactVC, bool[] hasRead)
2:     if $\neg hasRead[i]$ then
3:         $Set\langle VC\rangle\ VisibleSet \leftarrow \{vc : \langle -, vc, -\rangle \in CLog \wedge \forall w\ (hasRead[w] \Rightarrow vc[w] \leq xactVC[w])\}$
4:         $VC\ MaxVC \leftarrow vc : \forall w, vc[w] = \max\{velem[w] : velem \in VisibleSet\}$
5:     else
6:         $MaxVC \leftarrow xactVC$
7:     $Version\ ver \leftarrow k.lastFinal$
8:     $bool\ last \leftarrow \top$
9:     while $\exists w : hasRead[w] = \top \Rightarrow ver.vid[w] > MaxVC[w]$ do
10:        $ver \leftarrow ver.prev$
11:        $last \leftarrow \bot$
12:    return $[ver.val, MaxVC, last]$
```

Independently of whether the read is local or remote, before returning the value of the requested key $k$ to the application, $k$ is added to $T$'s read-set and the process that has provided the read value is flagged as already read by $T$ (lines 9, 13 and 17 of Algorithm 8). Finally, the vector clock of $T$ is updated to reflect the happened before relationship [59] between the commit event of the transaction that wrote the version observed by $T$'s read, and the corresponding read event of $T$ (line 16 of Algorithm 8).

### 6.1.2 Transaction commit phase

As already anticipated, one of the key strength points of GMU is that it allows committing read-only transactions without requiring any kind of local or remote validation phase.

The scheme used to commit *update* transactions in GMU is specified by the pseudocode shown in Algorithms 10 and 11. GMU uses a Two Phase Commit (2PC) protocol, involving *all and only* the set *rep* of nodes that replicate keys read or written by a committing transaction $T$, as well as the node that originated $T$. Exploiting 2PC, GMU can use standard techniques to ensure transaction atomicity and to verify its compatibility with the history of committed (update) transactions. The latter goal is achieved by acquiring read/write locks on all keys read/written by the transaction, at all nodes in which these keys are stored, and then performing a validation of the transaction's read-set (lines 2-4 of Algorithm 11). Note that, to prevent deadlock scenarios, the locks acquisition latency is bound with a timeout, such that when a timer expires the acquisition terminates with a negative outcome.

The key innovative feature of GMU's commit algorithm, however, consists in the scheme employed to establish agreement among the nodes involved in the execution of a transaction $T$, on the commit vector clock to assign to $T$. To this end, GMU blends into the 2PC messaging pattern a distributed consensus scheme that resembles the one used by Skeen's total order multicast algorithm [44].

When node $p_i$ receives a prepare message for transaction $T$ (and after its successful validation), it sends back with the VOTE message the proposal of a new vector clock for $T$. This proposal, i.e. $prepareVC$, is built starting from $LastPrepVC$ to maintain the invariant such that the commit of $T$ on $p_i$ does not precede any other transaction committed on $p_i$. If $T$ is not going to update any key stored by $p_i$, assigning $LastPrepVC$ to $prepareVC$ is enough to maintain the aforementioned invariant (line 9 of Algorithm 11); otherwise, if $p_i$ stores a key that is contained in $T$'s write-set, the $i$-th entry of $LastPrepVC$ is incremented by 1 before performing the assignment (lines 11-12 of Algorithm 11). In addition, in the latter case, the prepare phase is concluded by storing the triple $\langle T, prepareVC, pending\rangle$ in the $p_i$'s $CommitQueue$ (line 13 of Algorithm 11).

The 2PC coordinator gathers all the proposed $prepareVC$, and performs

**Algorithm 10** Commit phase (node $p_i$).

```
bool Commit(Transaction T)
    if $T.ws = \emptyset$ then
        return $\top$
    $VC\ commitVC \leftarrow T.VC$
    $bool\ outcome \leftarrow \top$
    send PREPARE($[T]$) to all $p_j \in replicas(T.rs \cup T.ws) \cup \{p_i\}$
    for all $p_j \in replicas(T.rs \cup T.ws) \cup \{p_i\}$ do
        wait receive VOTE($[T.id, VC_j, res]$) from $p_j$ or timeout
        if $res = \bot \vee$ timeout then
            $outcome \leftarrow \bot$
            break
        else
            $commitVC \leftarrow max(commitVC, VC_j)$
    $int\ xactVN \leftarrow max\{commitVC[w] : p_w \in \Pi\}$
    for all $p_j \in replicas(T.ws)$ do
        $commitVC[j] \leftarrow xactVN$
    send DECIDE($[T, commitVC, outcome]$) to all $p_j \in replicas(T.rs \cup T.ws) \cup \{p_i\}$
    wait until $T.completed = \top$
    return $T.outcome$

bool validate(Set rs, VC xactVC)
    for all $k \in rs$ do
        if $k.lastFinal.vid[i] > xactVC[i]$ then
            return $\bot$
    return $\top$
```

**Algorithm 11** Prepare and Decide messages (node $p_i$).

```
1:  upon receive PREPARE([Transaction T]) from $p_j$
2:      bool $outcome \leftarrow getExclLocksWithTimeout(T.id, T.ws)$
3:      $outcome \leftarrow outcome \wedge getSharedLocksWithTimeout(T.id, T.rs)$
4:      $outcome \leftarrow outcome \wedge validate(T.rs, T.VC)$
5:      if $outcome = \perp$ then
6:          $releaseLocks(T.id, T.ws, T.rs)$
7:          send VOTE($[T.id, T.VC, outcome]$) to $p_j$
8:      else
9:          $VC\ prepareVC \leftarrow LastPrepVC$
10:         if $p_i \in replicas(T.ws)$ then
11:             $LastPrepVC[i]++$
12:             $prepareVC \leftarrow LastPrepVC$
13:             $CommitQueue.put(\langle T, prepareVC, pending\rangle)$
14:         send VOTE($[T.id, prepVC, outcome]$) to $p_j$
15:
16: upon receive DECIDE([Transaction T, VC commitVC, bool outcome]) from $p_j$
17:     if $outcome = \top$ then
18:         $LastPrepVC \leftarrow max(LastPrepVC, commitVC)$
19:         if $p_i \in replicas(T.ws)$ then
20:             $CommitQueue.update(\langle T, commitVC, ready\rangle)$
21:         else
22:             $T.outcome \leftarrow \top$
23:             $T.completed \leftarrow \top$
24:     else
25:         $CommitQueue.remove(T)$
26:         $releaseLocks(T.id, T.ws, T.rs)$
27:         $T.outcome \leftarrow \perp$
28:         $T.completed \leftarrow \top$
29:
30: upon $(\exists\langle T, vc, s\rangle : \langle T, vc, s\rangle = CommitQueue.head \wedge s = ready \wedge (\nexists\langle \bar{T}, \bar{vc}, \bar{s}\rangle \in$
    $CommitQueue : \bar{vc}[i] < vc[i]))$
31:     $\forall\langle k, val\rangle \in T.ws : p_i \in replicas(\{k\})$ do $apply(k, val, vc)$
32:     $CLog.add(\langle T, vc, T.ws\rangle)$
33:     $CommitQueue.remove(T)$
34:     $releaseLocks(T.id, T.ws, T.rs)$
35:     $T.outcome \leftarrow \top$
36:     $T.completed \leftarrow \top$
```

two operations in order to derive the *commitVC* for the transaction (lines 13-16 of Algorithm 10). First, it merges the *prepareVC*'s values with the current *VC*'s values of the transaction using the max operator, which outputs a vector clock having, for each of its entries $j$, the maximum of the $j$-th entry of the vector clocks passed as input. This allows to keep track in the *commitVC* of the causal dependencies developed both by $T$ during its execution as well as by the most recently committed transactions at all the nodes contacted by $T$. Next, the coordinator determines the common value to attribute to the entries of the *commitVC* related to the nodes whose keys have been updated by the transaction (i.e. the nodes $p_j \in replicas(T.ws)$). This is achieved by picking the maximum value among all the entries in the *prepareVC* of all nodes involved in the commit.

At this point the coordinator sends back a decision to all the nodes in *rep*. This triggers the update of the entry associated with transaction $T$ in *CommitQueue*, if any, whose vector clock is replaced with the *commitVC* and whose status is set to *ready* (lines 19-20 of Algorithm 11). In addition, to maintain the invariant such that *LastPrepVC* on $p_i$ is never less than the vector clock of the last committed transaction on $p_i$, *LastPrepVC* is updated accordingly by using the received *commitVC* (line 18 of Algorithm 11). For the sake of clarity, in the pseudocode it has been supposed that the coordinator always sends the decision for $T$ to itself as well, so that it can reply to the client of $T$ as soon as the commit is finalized locally.

In order to finalize the commit of $T$ locally on a node $p_i$, however, it is waited until its entry (if any) has become the first in *CommitQueue* (recall that the *CommitQueue* at process $p_i$ is ordered based on the $i$-th entry of the vector clocks that it contains). This scheme guarantees that all the nodes updated by a transaction $T$ will assign the same *commitVC* to $T$. It also ensures that if a node $p_i$ commits a transaction with a local scalar timestamp (i.e., having as $i$-th entry in its VC the value) equal to $v$, then the local scalar timestamps of the transactions that subsequently commit at $p_i$ will be larger than $v$.

The two aforementioned properties guarantee that all the replicas in $replicas(T.ws)$ commit $T$ by using the same *commitVC* and in the same total order with respect to all the other committed transactions.

Finally, the merging of the causal histories encoded by the transaction's VC and by all the gathered *prepareVC* guarantees that the total order of the commit events is propagated across chains of, possibly transitively dependent transactions. This represents one of key mechanisms leveraged by GMU in order to ensure 1CS of the history of update transactions.

In the case the transaction's coordinator receives at least one negative vote message, i.e. containing *res* equal to $\perp$, it sends to all the participants (including itself) a negative decision, i.e. containing *outcome* equal to $\perp$, which triggers the abort of the commit phase.

### 6.1.3 Garbage Collection

GMU integrates an efficient distributed garbage collection protocol that relies on background dissemination (either via gossip [95] and/or via piggybacking) of the $lbVC_i$, i.e. the lower bound vector clock computed locally at each node $p_i$. In particular $lbVC_i$ is the result of the minimum operator applied on the set of VC vector clocks associated to the active transactions on node $p_i$.

This lightweight rumor-mongering mechanism allows each node to determine a conservative estimate of the global lower bound vector clock, say $glbVC$, that delimits the set of snapshots that may be still visible by active transactions in the system, and allows identifying the ones that cannot be reached anymore by any read operation. This means that it is possible to garbage collect every version having $vid < glbVC$ without risking to remove versions that may be later requested by some transaction.

Therefore once a node $p_i$ is aware about the current value of $lbVC_j$ of each node $p_j \in \Pi$, it computes $glbVC$, where $glbVC[w]$ is the minimum value among the ones stored in $lbVC_j[w]$. Afterwards it can safely detach and delete from the $CLog$ every node $\langle T, commitVC, updatedKeys\rangle$ such that $commitVC < glbVC$. Then for each key $k$ in $updatedKeys$ it can remove every version $ver$ such that $ver.vid < glbVC$ and $ver$ is not the last committed version of $k$.

### 6.1.4 Failure Handling and Dynamic Process Groups

For the sake of simplicity, GMU is layered on top of a 2PC protocol, which is well known to be blocking upon failure of the coordinator. However, the issue of how to ensure high availability of the transaction coordinator state is well understood, and a range of orthogonal solutions have been proposed in literature to deal with such failure scenarios. One may use, for instance, protocols such as Paxos Commit [39] or other consensus based abstractions [35, 73], to replicate the state of the coordinator of a transaction $T$ across the replicas of any of the data partitions accessed by $T$. Note that, since a majority of nodes is assumed to be correct for each replica group, failures of transactions' participants will not lead to blocking scenarios during the execution of a remote read operation. Failures of transactions' participants can, instead, lead to aborts during the commit phase, as the coordinator unilaterally aborts the transaction if it times out while waiting for some reply during the prepare phase. To ensure the liveness of the commit protocol, GMU relies on an underlying Group Communication System [22] in order to handle the removal of faulty replicas from the system and manage its reconfiguration, which may imply the re-distribution of data across replicas to guarantee a desirable replication degree.

Aiming at ensuring a strong consistency criterion, GMU opts for sacrificing availability (by aborting transactions that span remote nodes) in order to ensure consistency in presence of network partitions. This is not surprising, given the existence of well known results, such as the CAP theorem [13, 37], concerning the impossibility of achieving both availability and consistency in presence of partitions. In particular, GMU can be categorized as a PC/EC protocol in accordance with the PACELC-based classification of replication protocols provided in [1]. GMU, in fact, refuses to give up consistency independently of whether in presence of network partition or not, and it will pay the availability and latency costs to achieve it. Therefore, if there is a partition (P, i.e. Partition) the system chooses consistency (C, i.e. Consistency) instead of availability (A, i.e. Availability). Also, when the system runs normally (E, i.e. Else), it prefers consistency (C) to minimal latency (L, i.e. Latency).

Finally, GMU does not introduce additional issues concerning the management of dynamic process groups with respect to classic 2PC-based transactional replication systems. Conversely, its supports for multiversion simplify significantly the design of state-transfer mechanisms [51] aimed to synchronize the state of newly joining nodes.

### 6.1.5 On the support for read operations

Section 6.1.1 has shown that one of the key role for determining the correct version to be returned during a read operation is played by the commit log $CLog$. In fact the commit log on a node $p_i$ is used by a transaction $T$ to select the set of write transactions whose committed values can be observed by, i.e. are correctly serialized before, $T$. Then after that selection, $T$ can computes the $MaxVC$ vector clock that establishes a upper bound for its read operations on $p_i$.

Even though the visibility rule (as described in Section 6.1.1) ensures that a transaction cannot observe an inconsistent state, it can suffer from high execution costs whenever the number of committed write transactions on $p_i$ starts growing. This is because a naive implementation of line 3 of Algorithm 9 may require a transaction to scan the whole $CLog$ in order to find the set of visible commits, and clearly the temporal cost of this operation has a linear dependence on the size of the $CLog$.

Therefore the implementation of GMU follows a slightly modified rule that is able to scan only a subsequence of the elements in $CLog$ to compute the $MaxVC$ vector clock. This new rule has its foundation on the fact that for a set of elements $S = \{e_1, \dots, e_{j-1}, e_j, e_{j+1}, \dots, e_m\}$, if $\forall h \in [1, j-1]$ and $\forall k \in [j, m]$, $e_h \leq e_k$, then the following equation is verified:

$$max\{e : e \in S\} = max\{e' : e' \in S' = \{e_j, e_{j+1}, \dots, e_m\}\} \tag{6.1}$$

So the new visibility rule uses Equation 6.1 to optimize the computation of

$MaxVC$. In particular, given $CLog$ on $p_i$, the rule scans $CLog$ from the most recent element, in order to find the element $e^* = \langle T^*, commitVC^*, updatedKeys^* \rangle$ such that for all the elements $e' = \langle T', commitVC', updatedKeys' \rangle$ that follow $e^*$ (i.e. newer than $e^*$) in $CLog$ and for all the elements $e'' = \langle T'', commitVC'', updatedKeys'' \rangle$ that precede $e^*$ (i.e. older than $e^*$) in $CLog$, then $commitVC'' \leq commitVC^*$ and $commitVC'' \leq commitVC'$. Therefore, the values taken into account to produce $VisibleSet$ (see Algorithm 9) and hence to compute $MaxVC$ are only the values $commitVC'$ and $commitVC^*$.

Finding the element $e^*$ in GMU requires scanning all and only the elements in $CLog$ from the most recent to $e^*$ itself without looking at all the remaining elements $e''$. This is because for any element $e'$ in $CLog$, GMU stores which was the last element $lbe'$ in $CLog$ at the time $T'$ was prepared on $p_i$. In this case GMU guarantees that $commitVC'$ is greater than or equals to:

- the commit vector clock associated to $lbe'$ and
- all the vector clocks associated to the elements in $CLog$ that precedes $lbe'$.

Therefore, named (i) $S'$ the set of elements $e'$, (ii) $LS'$ the set of elements $lbe'$, (iii) $min(S')$ the oldest committed element in $S'$, and (iv) $min(LS')$ the oldest committed element in $LS'$, then the element $e^*$ is equal to $min(LS')$ iff $min(LS')$ immediately precedes $min(S')$ in $CLog$.

## 6.2 Correctness Proof

This Section proves that GMU accepts only Extended Update Serializable histories by showing that the protocol avoids *G1a* and *G1b* anomalies, plus *G1c* and *Extended G-update* anomalies by taking also executing transactions into account (as described in Chapter 3).

To simplify the explanation and without loss of generality, throughout the following proof, both an executing and an aborted transaction at time $t$ are treated as a read-only transaction constituted by its prefix at time $t$ that contains all its read operations performed until time $t$, except the read operation which has triggered an abort (if any). This is an admissible reduction since write operations are buffered during the execution of a transaction and they are externalized (i.e. the updates are applied) only upon a successfully completed commit phase. Moreover, the read operation that triggers an abort for a transaction $T$ can be safely discarded because it does not return any value to the application layer, and hence it does not generate any dependency in the $DSG(\mathcal{H})$ graph.

Since write operations are externalized only by committed transactions then *G1a* anomaly is trivially avoided by the fact that a transaction $T_j$ cannot read a value written by an aborted transaction $T_i$; on the other hand *G1b* anomaly is avoided because only the final modification produced by a transaction $T$ on a key $k$ is made visible after transaction $T$ commits.

The rest of the proof is organized in two parts: the former proves that the *G1c* anomaly is avoided, namely the *unidirectional flow of information* is guaranteed; the latter proves that the *Extended G-update* anomaly is avoided, namely the *no-update-conflict-misses* property extended to also executing and aborted transactions is guaranteed.

### 6.2.1 Unidirectional flow of information

As already described in Chapter 3, for each history $\mathcal{H}$ containing committed, aborted and executing transactions, and pair of transactions $T_i$, $T_j$ in $\mathcal{H}$, there is an unidirectional flow of information from $T_i$ to $T_j$ if $DSG(\mathcal{H})$ does not contain a directed cycle consisting entirely of dependency edges from $T_i$ to $T_j$.

To prove the last statement it will be shown that the sub-graph $DSG(\mathcal{H}^{upc})$ does not contain any directed cycle consisting entirely of dependency edges, where $\mathcal{H}^{upc}$ is derived from $\mathcal{H}$ by removing all the executing, aborted and read-only transactions in $\mathcal{H}$. In other words $\mathcal{H}^{upc}$ considers only the committed update transactions in $\mathcal{H}$. This simplification is admissible due to the fact that:

1. by excluding anti-dependency edges, read-only transactions are necessarily sink nodes of the $DSG(\mathcal{H})$, i.e. they do not have outgoing edges, as they can only develop incoming read dependency edges in this analysis;

2. each executing or aborted transaction can be treated as a read-only transaction.

Consequently it is proved that for each edge $V_{T_i} \xrightarrow{wr} V_{T_j} \in DSG(\mathcal{H}^{upc})$ and $V_{T_i} \xrightarrow{ww} V_{T_j} \in DSG(\mathcal{H}^{upc})$, then $T_i.commitVC < T_j.commitVC$ holds, where $T_i.commitVC$ (respectively $T_j.commitVC$) is the vector clock used to commit transaction $T_i$ (respectively $T_j$), and hence that $DSG(\mathcal{H}^{upc})$ cannot contain any oriented cycle with (write and read) dependency edges.

Therefore if an edge $V_{T_i} \xrightarrow{E} V_{T_j}$ is in $DSG(\mathcal{H}^{upc})$, two cases are distinguished, depending on whether $T_j$ directly read-depends on $T_i$, i.e. $E = wr$, or $T_j$ directly write-depends on $T_i$, i.e. $E = ww$.

### $T_j$ directly read-depends on $T_i$

In this case $T_j$ reads a value *val* associated to a version *ver* committed by $T_i$ on key $k$ stored on a node $p_n$. Let $T_j.VC$ be the vector clock associated to $T_j$ after $T_j$ has read from $T_i$; then it is proved that both $T_i.commitVC \leq T_j.VC$ and $T_j.VC < T_j.commitVC$ hold.

**Lemma 6.2.1.** *If a transaction $T_j$ directly read-depends on a transaction $T_i$ then $T_i.commitVC \leq T_j.VC$, where $T_j.VC$ is the vector clock associated to $T_j$ right after $T_j$ has read from $T_i$.*

*Proof.* Right after $T_j$ has read from $T_i$, $T_j.VC$ is equal to the vector clock obtained by maximizing each entry of $T_j.VC$ with its counterpart in the $VC^*$ vector clock (see line 16 of Algorithm 8), implying that

$$VC^* \leq T_j.VC \tag{6.2}$$

.

At that point two cases have to be distinguished, depending on whether this is the first read by $T_j$ on node $p_n$ or not. In the first case $VC^*$ is equal to the vector clock $MaxVC$ computed from $p_n$'s $CLog$ and such that

$$VC^* \leftarrow MaxVC \leftarrow vc : \forall w, vc[w] = max\{velem[w] : velem \in VisibleSet\} \tag{6.3}$$

(see line 4 of Algorithm 9) and where

$$VisibleSet \leftarrow \{vc : \langle -, vc, - \rangle \in CLog \land \forall w(hasRead[w] \Rightarrow vc[w] \leq xactVC[w])\} \tag{6.4}$$

(see line 3 of Algorithm 9), where $xactVC$ was the old value of $T_j.VC$ right before the read operation.

In addition, since $T_j$ has read version *ver* of key $k$ from $T_i$, the following condition is verified

$$\forall w : hasRead[w] = \top \Rightarrow T_i.commitVC[w] \leq MaxVC[w] \tag{6.5}$$

by the visibility rule at line 9 of Algorithm 9, and therefore

$$MaxVC \geq T_i.commitVC \tag{6.6}$$

because $T_i.commitVC \in VisibleSet$ by Equations 6.3, 6.4 and 6.5.

Since $T_i.commitVC \leq VC^*$ by Equations 6.3 and 6.6, it follows that $T_i.commitVC \leq T_j.VC$ by Equation 6.2.

Now it is considered the case in which the read by $T_j$ on key $k$ is not the first read on node $p_n$. In this case $T_i.commitVC \leq T_j.VC$ still holds because $T_i.commitVC$ was in the $VisibleSet$ computed by $T_j$ upon the first read on

$p_n$. If this was not the case, it would mean that at the time of that first read there existed an index $h$ such that $T_j.hasRead[h] = \top$ and $T_i.commitVC[h] > T_j.VC$. However, by the visibility rule at line 9 of Algorithm 9, $T_j.VC$ is always updated by means of a max operator and that once $T_j.hasRead[h]$ is set to $\top$ then $T_j.VC[h]$ cannot change anymore. Hence the claim follows.

□

**Lemma 6.2.2.** *For each committed update transaction $T_j$, $T_j.VC_{MAX} < T_j.commitVC$, where $T_j.VC_{MAX}$ is the vector clock associated to $T_j$ before $T_j$ enters the commit phase.*

*Proof.* Let $I$ be the set of the identifiers of the nodes in the system.

$T_j.VC_{MAX} < T_j.commitVC$ because:

- $T_j.VC_{MAX}[h] \leq T_j.commitVC[h]$, $\forall h \in I$, since $T_j.commitVC$ is initialized with the values of $T_j.VC_{MAX}$ (see line 4 of Algorithm 10) and it is later modified by means of a *max* operator (see line 13 of Algorithm 10).
- $\forall s \in I$ such that $T_j$ writes on a key maintained by node $p_s$, then $T_j.VC_{MAX}[s] < T_j.commitVC[s]$ holds, since $T_j.commitVC[s]$ is set to $xactVN$, which is a new scalar version number derived from the increment of $LastPrepVC[s]$ scalar clock on $s$ (see lines 14-16 of Algorithm 10 and line 11 of Algorithm 11) and $T_j.VC_{MAX}[s] \leq LastPrepVC[s]$ before the increment.

□

**Lemma 6.2.3.** *If a transaction $T_j$ directly read-depends on a transaction $T_i$ then $T_i.commitVC < T_j.commitVC$*

*Proof.* Since $T_j.VC \leq T_j.VC_{MAX}$ by definition, $T_i.commitVC \leq T_j.VC$ by Lemma 6.2.1 and $T_j.VC_{MAX} < T_j.commitVC$ by Lemma 6.2.2 then $T_i.commitVC < T_j.commitVC$ .

□

#### $T_j$ directly write-depends on $T_i$

In this case $T_j$ overwrites a key $k$ already written by $T_i$.

**Lemma 6.2.4.** *If a transaction $T_j$ directly write-depends on a transaction $T_i$ then $T_i.commitVC < T_j.commitVC$*

*Proof.* Since (i) a write is actually executed when a transaction commits, (ii) an exclusive lock for each key to be written is acquired during the prepare phase and (iii) all the locks are released at the end of the commit phase, then $T_j$ commits after $T_i$ has already committed and both $T_i$ and $T_j$ commit on at least a common node $p_n$ (the node that stores the key overwritten by $T_j$). In addition, by line 13 of Algorithm 10, it follows that

$$prepareVC_{j,n} \leq T_j.commitVC \tag{6.7}$$

where $prepareVC_{j,n}$ is the prepare vector clock used to prepare $T_j$ on node $p_n$. Moreover, since $T_j$ starts the prepare phase after $T_i$ has already inserted $T_i.commitVC$ in $p_n$'s $CLog$ and $prepareVC_{j,n}$ is greater than any vector clock in that $CLog$ (see line 11 of Algorithm 11), it follows that

$$T_i.commitVC < prepareVC_{j,n} \tag{6.8}$$

Therefore $T_i.commitVC < T_j.commitVC$ by (6.7) and (6.8).

□

**Theorem 6.2.5.** *For each history $\mathcal{H}$ containing committed, aborted and executing transactions, $DSG(\mathcal{H})$ does not contain any oriented cycle with (write and read) dependency edges.*

*Proof.* As already described in the introduction of section 6.2.1 proving that the $DSG(\mathcal{H}^{upc})$ does not contain any directed cycle with dependency edges is a sufficient condition to prove that $DSG(\mathcal{H})$ does not contain such cycles. The history $\mathcal{H}^{upc}$ is obtained by removing all the aborted, executing and read-only transactions from $\mathcal{H}$.

Therefore the $DSG(\mathcal{H}^{upc})$ cannot contain any directed cycle with dependency edges because if, by contradiction, such a cycle $C$ existed, then, by Lemma 6.2.3 and Lemma 6.2.4, there would be the absurd such that for each transaction $T_i$ in $C$, $T_i.commitVC < T_i.commitVC$.

□

### 6.2.2 No-update-conflict-misses

The no-update-conflict-misses property is guaranteed if, for each history $\mathcal{H}$ and aborted/executing/committed transaction $T_i \in \mathcal{H}$, the $DSG(\mathcal{H}^{upc}_{T_i})$ containing all committed update transactions of $\mathcal{H}$ and transaction $T_i$ does not contain any oriented cycle with anti-dependency edges.

Since every aborted/executing transaction can be reduced to a read-only transaction (as described in the introduction of Section 6.2), the following proof considers $T_i$ as a generic (i.e. either read-only or update) committed transaction.

The proof is structured as follows: Lemma 6.2.6 proves that the result of Lemmas 6.2.3 and 6.2.4 can be extended to the case of anti-dependency edges; then Lemma 6.2.7 shows that the $DSG(\mathcal{H}^{upc}_{T_i})$ cannot contain oriented cycles in the case $T_i$ was a read-only transaction and at the end, Theorem 6.2.8 proves that the no-update-conflict-misses is guaranteed by using the results of Lemmas 6.2.3, 6.2.4, 6.2.6 and 6.2.7.

**Lemma 6.2.6.** *If a transaction $T_j$ directly anti-depends on a transaction $T_i$ then $T_i.commitVC < T_j.commitVC$*

*Proof.* In this case $T_i$ reads a version of a key $k$ older than the one created by $T_j$ on $k$ and later it successfully commits. Since the protocol ensures that an update transaction $T$ is not aborted if and only if (i) it successfully acquires locks on all the keys in its write-set and read-set at the beginning of its prepare phase and (ii) it passes the validation phase, namely other transactions have not concurrently altered the versions chain of the keys in $T$'s read-set (see lines 21-25 of Algorithm 10), then $T_i$ commits before $T_j$ has committed and the two sets of nodes involved in their commit operations are not disjoint. This is the same scenario analyzed in Lemma 6.2.4, which considered the case in which $T_j$ directly write-depends on $T_i$. Therefore, using identical arguments, one can easily show that $T_i.commitVC < T_j.commitVC$.

□

**Lemma 6.2.7.** *For each history $\mathcal{H}$ and read-only transaction $T^{RO} \in \mathcal{H}$, the $DSG(\mathcal{H}^{upc}_{T^{RO}})$ containing all committed update transactions of $\mathcal{H}$ and transaction $T^{RO}$ does not contain any oriented cycle involving $T^{RO}$.*

*Proof.* As $T^{RO}$ is a read-only transaction, it follows that any incoming edge from an update transaction $T_i$ to $T^{RO}$ must be a read-dependency edge, namely $V_{T_i} \xrightarrow{wr} V_{T^{RO}}$. Further, the only outgoing edges from $T^{RO}$ to update transactions $T_j$ must be anti-dependency edges, namely $V_{T^{RO}} \overset{rw}{\dashrightarrow} V_{T_j}$.

In the case $DSG(\mathcal{H}^{upc}_{T^{RO}})$ contains an edge $V_{T_i} \xrightarrow{wr} V_{T^{RO}}$, by Lemma 6.2.1 it follows that $T_i.commitVC \leq T^{RO}.VC$, where $T_i.commitVC$ is the $T_i$'s commit vector clock and $T^{RO}.VC$ is the vector clock associated to $T^{RO}$ right after $T^{RO}$ has developed the dependence on $T_i$, i.e. $T^{RO}$ has read a version committed by $T_i$.

On the other hand, if $DSG(\mathcal{H}^{upc}_{T^{RO}})$ contains an edge $V_{T^{RO}} \overset{rw}{\dashrightarrow} V_{T_j}$, by definition of anti-dependency edge it follows that $\exists k \neq j$ and a key $x$ such that $T^{RO}$ has read a version $x_k$ on a node $p_n$, $T_j$ has committed a version $x_j$ on $p_n$ and $x_k \ll x_j$ ($x_k$ is committed before $x_j$).

In this case there always exists an index $h$ such that $T^{RO}.VC_{MAX}[h] < T_j.commitVC[h]$. In particular, if $T_j$ has already committed version $x_j$ at the time $T^{RO}$ has read version $x_k$ then $\exists$ $p_h$ from which $T^{RO}$ has already read such that $T^{RO}.VC[h] < T_j.commitVC[h]$ (following the condition at line 9 of Algorithm 9); since $T^{RO}.VC[h]$ cannot change anymore after $T^{RO}$ has read on $p_h$, then $T^{RO}.VC_{MAX}[h] < T_j.commitVC[h]$ holds.

Otherwise, if $T^{RO}$ misses $T_j$ updates because its read operation on key $x$ is delivered on $p_n$ before $T_j$'s commit, then $T^{RO}.VC[h] < T_j.commitVC[h]$ holds, where $h$ is equal to $n$, because the $p_n$'s commit log $CLog$ is always totally ordered according to the commit vector clocks' values with index $n$. Also in this case $T^{RO}.VC_{MAX}[h] < T_j.commitVC[h]$ holds because $T^{RO}.VC_{MAX}[h]$ cannot change anymore after $T^{RO}$ has read on $p_n$.

Note that no other cases are possible since the reading rule defined at line 21 of Algorithm 8 prevents the transaction $T^{RO}$ from missing the updates of $T_j$ on $p_n$ when $T^{RO}.VC[n] \geq T_j.commitVC[n]$ holds, and therefore $T^{RO}.VC_{MAX}[n] \geq T_j.commitVC[n]$ since $T^{RO}.VC_{MAX} \geq T^{RO}.VC$ by construction.

At this point if $DSG(\mathcal{H}^{upc}_{T^{RO}})$ has an oriented cycle involving $T^{RO}$, without loss of generality such a cycle $C$ can be assumed as follows: $V_{T_1} \rightarrow \ldots \rightarrow V_{T_i} \rightarrow V_{T^{RO}} \rightarrow V_{T_j} \rightarrow \ldots \rightarrow V_{T_1}$. Then by Lemmas 6.2.3, 6.2.4 and 6.2.6 it follows that $T_1.commitVC < T_i.commitVC$ and $T_j.commitVC < T_1.commitVC$. In addition $T_i.commitVC \leq T^{RO}.VC$, after $T^{RO}$ has read a version committed by $T_i$, and there always exists an index $h$ such that $T^{RO}.VC_{MAX}[h] < T_j.commitVC[h]$. Since $T^{RO}.VC \leq T^{RO}.VC_{MAX}$ by construction, it follows the absurd:

1. for each update transaction $T$ in $C$, $T.commitVC[h] < T.commitVC[h]$;
2. $T^{RO}.VC_{MAX}[h] < T^{RO}.VC_{MAX}[h]$.

Therefore $DSG(\mathcal{H}^{upc}_{T^{RO}})$ does not contain any cycle involving $T^{RO}$.

□

The no-update-conflict-misses property is guaranteed by the following Theorem.

**Theorem 6.2.8.** *For each history $\mathcal{H}$ and transaction $T_i \in \mathcal{H}$, the $DSG(\mathcal{H}^{upc}_{T_i})$ containing all committed update transactions of $\mathcal{H}$ and an arbitrary (read or update, committed, executing or aborted) transaction $T_i$ does not contain any oriented cycle with at least an anti-dependency edge.*

*Proof.* In the case $T_i$ is a committed update transaction then if such an oriented cycle $C$ existed, it would follow the absurd according to which $T.commitVC < T.commitVC$ for each transaction $T$ in $C$ by Lemmas 6.2.3, 6.2.4 and 6.2.6.

On the other hand, in the case $T_i$ is a read-only transaction, $DSG(\mathcal{H}^{upc}_{T_i})$ does not contain any cycle by Lemma 6.2.7.

Since no other cases have to be considered because any executing or aborted transaction can be treated as a read-only transaction (see introduction of section 6.2), $DSG(\mathcal{H}^{upc}_{T_i})$ does not contain any oriented cycle and therefore not even a cycle with an anti-dependency edge.

□

## 6.3 On the Data Freshness

In the transactional systems world it has been argued in various works [40, 71] that executing transactions without violating the real-time order among them may represent a necessary condition for the correctness of the applications. Informally, according to Strict Serializability or Opacity, it can be required that for each history $\mathcal{H}$ and for each pair of transactions $T_1$ and $T_2$ in $\mathcal{H}$, $T_2$ must appear as executed after $T_1$, i.e. $T_2$ must be serialized after $T_1$ in the $DSG(\mathcal{H})$, if $T_1 \prec_{\mathcal{H}} T_2$. The $\prec_{\mathcal{H}}$ relation is the happened-before relation [59] between two transactions in a history $\mathcal{H}$. In particular $T_1 \prec_{\mathcal{H}} T_2$ if the commit operation $c_1$ of $T_1$ precedes the begin operation $b_2$ of $T_2$ in $\mathcal{H}$ (see also Section 3.4.3).

This is an important property since, for multi-version concurrency control schemes, it entails to establish a limit on the staleness of data returned by read operations. This means that, even though a transaction $T_2$ is allowed to observe an old version of a datum, it must see at least the values committed by every transaction $T_1$ such that $T_1 \prec_{\mathcal{H}} T_2$.

Unfortunately GMU is not able to guarantee the real-time order as it has been defined above, due to two main reasons that can be explained by the following two executions admitted by GMU which exactly violate real-time order.

In the first execution there are two update transactions $T_1$, $T_2$ such that (i) $T_1 \prec_{\mathcal{H}} T_2$; (ii) $T_1$ executes a single write operation $w_1(x_1)$ and then commits on node $p_i$; (iii) $T_2$ executes a single write operation $w_2(y_2)$ and then commits on node $p_j$. In addition, there is a read-only transaction $T_3$ that executes a read operation $r_3(x_0)$ on node $p_i$ (i.e. $x_0 \ll x_1$) before the commit of $T_1$, a read operation $r_3(y_2)$ on node $p_j$, and then commits. Therefore $T_3$ misses the version of $x$ committed by $T_1$ but is able to read the last committed version of $y$ because this version still belongs to a consistent snapshot for $T_3$. This execution is admitted by EUS because the resulting history $\mathcal{H}$ is serializable and it is equivalent to the serial history in which $T_3$ is executed after $T_2$ and before $T_1$. But the execution violates the real-time order between $T_1$ and $T_2$ according to the definition in Section 3.4.3, since $T_1 \prec_{\mathcal{H}} T_2$ and there is an oriented path from $T_2$ to $T_1$ in $DSG(\mathcal{H})$.

In the second execution there are two transactions $T_1$, $T_2$ such that (i)

$T_1 \prec_{\mathcal{H}} T_2$; (ii) $T_1$ executes a single write operation $w_1(x_1)$ and then commits the new version $x_1$ both on node $p_i$ and node $p_j$; (iii) $T_2$ executes a read operation $r_2(x_0)$ on node $p_j$ (i.e. $x_0 \ll x_1$) before the commit of $T_1$ has been finalized on $p_j$. This execution is admissible if $p_i$ is supposed to be the coordinator of $T_1$'s commit phase, which can reply the completion of $T_1$ as soon as it has finalized the commit of $T_1$ locally. In fact, the second phase of the 2PC for $T_1$ does not wait for all the participants to have finalized the commit of $T_1$, and transaction $T_2$ has a real-time dependency on $T_1$ since its begin is executed in the system after $p_i$ has notified the client for the commit of $T_1$. In this case the resulting history $\mathcal{H}$ violates the real-time order between $T_1$ and $T_2$, since $T_1 \prec_{\mathcal{H}} T_2$ and there is an oriented path, i.e. a direct anti-dependency edge, from $T_2$ to $T_1$ in $DSG(\mathcal{H})$.

Even if the issue related to the second execution can be avoided in some cases by enforcing the coordinator of a 2PC to wait for a reply from each participant also in the second phase of the 2PC, GMU is still not able to always guarantee the real-time due to the genuineness of the replication scheme.

Nevertheless GMU is able to cope with one relevant aspect included in the definition of the real-time order and that is very important for enhancing the freshness of data returned by read operations. Roughly speaking GMU ensures that the read operation $r(x)$ on $x$ of a transaction $T$ executed on a quiescent state of the system always returns the last committed version of $x$. The system is defined to have a *quiescent* state if it is not executing any transaction, except $T$, and all commit-pending transactions have finalized their commit on all nodes of the system.

The latter feature is provided because a transaction does not fix its visible snapshot at the beginning of its execution but it always tries to extend its view on the transactional state as soon as it touches for the first time a new (not yet observed) node. This is a powerful feature because it allows a transaction to observe the last committed snapshot for scenarios in which no other transactions are changing the system state concurrently and despite the genuineness of the replication protocol. Other existing GPR protocols that also guarantee abort-free read-only transactions either are not able to achieve this result [91] or provide the same data freshness guarantees by adopting a more expensive memory footprint [8], i.e. vector clocks having the size equal to the number of objects in the transactional state.

A formalization of such a behavior by GMU is given by the following theorem.

**Theorem 6.3.1.** *For any datum $x$ and any transaction $T_i$ that executes on a quiescent state of the system, if $T_i$ performs read operation $r_i(x_k)$ then $x_k$ is the most recent committed version of $x$.*

*Proof.* The proof follows by induction on the elements in the read-set of $T_i$. It is first proved that the thesis is verified when $r_i(x_k)$ is the first read operation of $T_i$ (base step). Then, it is proved that the thesis is verified when $r_i(x_k)$ is the $n$-th read operation of $T_i$, where the thesis is already verified for all the previous $n-1$ read operations of $T_i$ (inductive step). It is also assumed that $p_h \in \mathit{replicas}(\{x\})$ and $T_i$ executes the read operation $r_i(x_k)$ on $p_h$.

Base step: $r_i(x_k)$ is the first read operation of $T_i$. The proof of this step follows by contradiction by assuming that $x_k$ is not the most recent committed version of $x$. Therefore there is a version $x_j$ already committed on $p_h$ at the time $T_i$ executes $r_i(x_k)$ such that $x_k \ll x_j$. This can only happen if $T_i$ skips version $x_j$ committed by transaction $T_j$ on $p_h$ because there exists an index $m$ such that $p_m$ is the originating node of $T_i$, $T_i.\mathit{hasRead}[m] = \top$ (due to the initialization of $T_i.\mathit{hasRead}$) and $x_j.\mathit{vid}[m] > T_i.VC[m]$ (see line 9 of Algorithm 9). But since $T_i.VC[m]$ is equal to $\mathit{CLog.mostRecentVC}[m]$ on $p_m$ at the time $T_i$ began, and only a commit involving node $p_m$ can increase entry $m$ of every vector clock in the system, it would entail that the commit of $T_j$ transitively depends on a transaction committed on $p_m$ after $T_i$ began. This contradicts the hypothesis of quiescent system and therefore $x_k$ can only be the last committed version of $x$.

Inductive step: $r_i(x_k)$ is the $n$-th read operation of $T_i$ and the previous $n-1$ read operations of $T_i$ verify the thesis, namely for each datum $y$, if $r_i(y_q)$ is the $t$-th read operation of $T_i$, where $t = 1, \ldots, n-1$, then $y_q$ is the most recent committed version of $y$. The proof of this step follows by contradiction by assuming that $x_k$ is not the most recent committed version of $x$. Therefore there is a version $x_j$ already committed on $p_h$ at the time $T_i$ executes $r_i(x_k)$ such that $x_k \ll x_j$. This can only happen if $T_i$ skips version $x_j$ committed by transaction $T_j$ on $p_h$ because there exists an index $m$ such that $T_i.\mathit{hasRead}[m] = \top$ (due to a previous read operation on $p_m$) and $x_j.\mathit{vid}[m] > T_i.VC[m]$ (see line 9 of Algorithm 9). But since only a commit involving node $p_m$ can increase entry $m$ of every vector clock in the system, this entails that the commit of $T_j$ transitively depends on a snapshot $S$ committed on $p_m$ that has been skipped by one of the first $n-1$ read operations. Without loss of generality it is supposed that operation $r_i(y_q)$ has skipped $S$ on $p_m$. This is a contradiction for the following reasons:

– in case $S$ contains a version of $y$, $r_i(y_q)$ did not return the most recent committed version of $y$;

– in case $S$ does not contain a version of $y$, $S$ has been committed after $r_i(y_q)$ was executed on $p_m$.

Both cases are in contradiction with the quiescence of the system and therefore $x_k$ can only be the last committed version of $x$. □

## 6.4 Experimental Evaluation

GMU has been integrated in Infinispan$^2$, a mainstream open source in-memory distributed data platform. Analogously to many other contemporary distributed cache platforms, Infinispan [65] externalizes a simple key-value store interface. In order to maximize scalability, Infinispan relies on weak consistency models, and on a lightweight consistent hashing scheme [52] that allows partitioning data efficiently while ensuring good load balancing and minimum reshuffling of keys in presence of joins/departures of nodes from the platform. Further, Infinispan supports partial replication, allowing to store each key across a fixed, user-tunable number of replicas, thus achieving fault-tolerance without hampering scalability.

For what concerns consistency, the stronger consistency level ensured by Infinispan is Repeatable Read [11] (RR), an isolation level which ensures that no intermediate or aborted values are ever observed, and that no two consecutive reads on the same object within the same transaction can return different values. RR is significantly weaker than (E)US, as it allows the commit of (both read-only and update) transactions that observe non-serializable schedules [3].

Infinispan relies on an encounter based two phase locking scheme, which is applied only to write operations and that does not synchronize reads. Repeatability of read operations is instead guaranteed by storing the data items observed by read operations, and returning them upon subsequent reads. For what concerns the replication protocol of Infinispan, it relies on a classic 2PC-based distributed locking algorithm [38].

Designed to achieve high scalability and support weak consistency models, Infinispan represents an ideal baseline to evaluate the costs incurred by GMU to provide stronger consistency guarantees. In addition also a non-genuine multiversion-based replication scheme has been implemented, which, analogously to the one in [54], relies on a fully replicated, logically centralized, global scalar clock, used to totally order committing update transactions. This protocol is named NGM (Non-Genuine Multiversioning) in the following.

For experimental study two well-known benchmarks were used, namely TPC-C [93] and YCSB [23]. The workload generated by TPC-C is representative of OLTP environments and characterized by complex and heterogeneous transactions, with very skewed access patterns and high conflict probability. YCSB (Yahoo! Cloud Serving Benchmark) [23] is a framework specifically aimed at benchmarking NoSQL key-value data grids and cloud stores. The transactional profile of this benchmark is quite different from the one of TPC-C, with simpler, shorter transactions that rarely conflict.

The results presented in the following were obtained using two experimen-

$^2$The GMU prototype is publicly available at the URL `http://www.cloudtm.eu`.

tal platforms. The first one, FutureGrid [3], is a public distributed test-bed for parallel and cloud computing. This platform allows to evaluate GMU in environments representative of public cloud infrastructures, which are typically characterized by more competitive resource sharing, ample usage of virtualization technology, and relatively less powerful nodes. In the FutureGrid platform, experiments using up to 40 virtual machines (VM) were performed. Each VM was equipped with 7GB RAM, two 2.93GHz cores Intel Xeon CPU X5570, running CentOS 5.5 x86_64. All the VMs were deployed in the same physical data-center and interconnected via Gigabit Ethernet. In all experiments performed on FutureGrid a single thread per node was adopted to inject transactions (in closed loop), which guaranteed a high utilization of the machine's resources without overloading.

The second experimental platform, referred to as Cloud-TM, is a dedicated cluster of 20 homogeneous nodes, where each machine is equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 processors and 16 GB of RAM, running *Linux 2.6.32-33-server* and interconnected via a private Gigabit Ethernet. This platform is representative of small/medium private clouds or data-centers environments, with dedicated servers and a fairly large amount of available (computational and memory) resources per node. In order to maintain a similar ratio between threads and available cores with respect to the experiments performed in FutureGrid, in all experiments performed on Cloud-TM, four threads per node were used to inject transactions (in closed loop).

The analysis first starts with the results obtained by running YCSB using the Cloud-TM platform. The Workload A [23] of the benchmark was used, which is an update intensive workload (comprising 50% of update transactions) simulating a session store that records recent client actions. Figure 6.1 reports the maximum throughput (committed transactions per second) achievable by GMU and by Infinispan's partial replication protocol that ensures Repeatable Read (referred to as RR, in the following). The plot shows that the average reduction in throughput for GMU oscillates around 8%, and that it scales linearly at the same rate as RR, providing an evidence of the efficiency and scalability of the proposed solution.

Next, Figure 6.2 reports the results achieved using, on FutureGrid, the TPC-C benchmark configured with a read-dominated profile, composed at the 90% by read-only (Order-Status profiles) transactions and, for the remaining 10%, by update transactions (Payment and New-Order profiles) in equal parts. The plot confirms the efficiency and scalability of GMU. Surprisingly, in this scenario, despite providing consistency guarantees, GMU even outperforms RR by up to 10%. A profiling study has highlighted that these gains are

[3]www.futuregrid.org

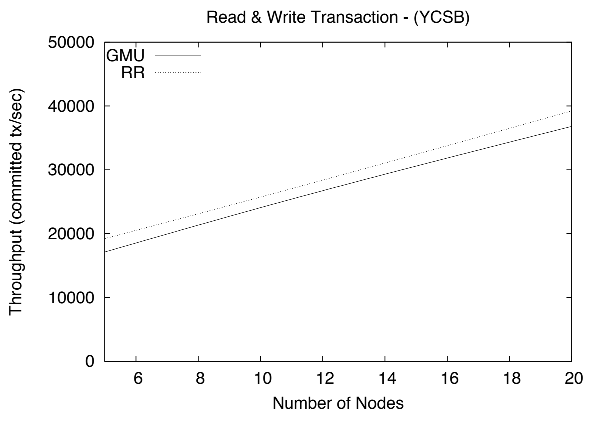


Figure 6.1: YCSB Benchmark (Cloud-TM).

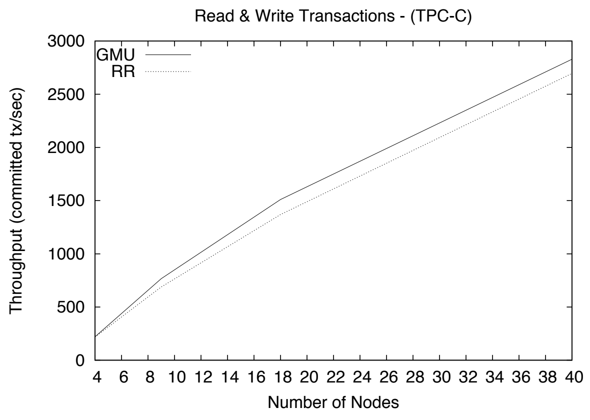


Figure 6.2: TPC-C Benchmark (FutureGrid).

imputable to the fact that GMU, unlike RR, avoids the overhead of storing previously read values to guarantee consistency. Read-only transactions in TPC-C, in fact, tend to perform a large number of operations, forcing RR to perform a large number of cloning operations to store read versions in the transactional context.

Finally, Figure 6.3 reports the results achieved by running TPC-C on the Cloud-TM platform. With 4 threads injecting transactions per node, the degree of concurrency is significantly higher than in the former experiment, leading to significant conflicts both at the logical (data) and at the physical (computing/network resources) level. The plots in this case report also the performance of the above described non-genuine multiversion partial replication protocol (NGM). The experimental data clearly demonstrate the detrimental

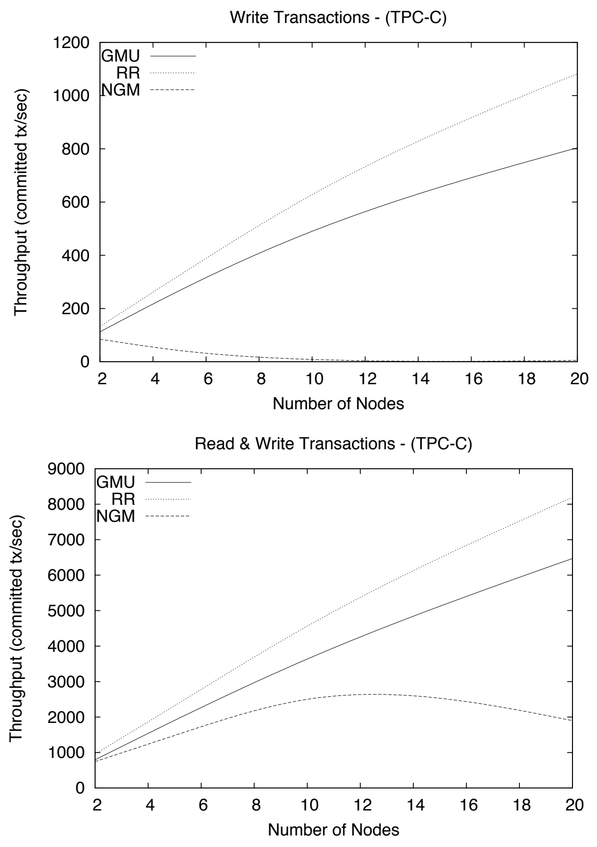


Figure 6.3: TPC-C Benchmark (Cloud-TM)

.

effect on system scalability due to the high logical contention on the fully replicated global clock, which leads to the drastic decay of the throughput (in particular of write transactions, see plot on top of Figure 6.3).

By comparing the performance of GMU with that of RR it emerges that the increase in the logical/physical contention level characterizing this configuration has a stronger impact on GMU. In fact, even though GMU shows an almost linear scalability trend, data reveal that it suffers of a higher abort rate than RR: for 20 nodes, the abort rate is on the order of the 15% for GMU, whereas it is around 8% for RR. This data shows that, in high contention scenarios, strong consistency semantics do pay a performance toll, which, in this specific configuration, corresponds to a throughput reduction ranging from 10% (at 4 nodes) up to 20% (at 20 nodes). On the other hand this is an unavoidable cost to pay in the context of applications whose correctness can be endangered by adopting non-serializable isolation levels. Note that TPC-C does belong to this class of applications, as in fact several of its transactional profiles might generate data corruptions in presence of concurrency anomalies such as those possible using Repeatable Read.

Chapter 7

# Additional Tradeoffs in the Design of Multi-Version GPR Protocols

With GMU it has been proved how to design a scalable multi-version solution for replicated transactional systems by guaranteeing highly desirable properties which no other protocol guaranteed before, i.e. genuine partial replication (GPR) and abort-free read-only transactions. This has been achieved by still ensuring a strong consistency level, i.e. EUS.

In other words, GMU does not require an expensive distributed validation for read-only transactions, and allows the abort of update transactions only if they encounter a conflict with other update transactions. The latter property, that as been defined as *multi-versioned permissiveness* in the context of in-memory transactional systems [80], entails in the context of replicated systems that read-only transactions are never forced to abort unless the coordinators of those transactions crash and clearly if there is at least a correct node storing every datum requested by the transaction.

Other than meeting this base requirements GMU also has the following two additional features for read operations:

– Every read operation is always provided with a consistent state of the transactional system, namely it is never allowed to observe a transactional state that would not be producible by some serializable execution of write transactions.

– The read operation $r_i(x)$ of a transaction $T_i$ never returns a version older than the one created by the latest transaction $T_k$ that has performed a write operation on $x$, if $T_i$ is executed in a quiescent system state.

The former feature is the reason why GMU guarantees abort-free read-only transactions and explains why they do not undergo distributed validation procedures after their execution: in fact, since a transaction always observes a consistent state, as soon as it completes its execution it can safely commit by appearing as serialized at a point in time that coincides with the execution of one of its read operations.

Data freshness is ensured thanks to the logic used in GMU to regulate the advance of the visible snapshots, which always tries to extend the view on the transactional state as soon as it touches for the first time a new (not yet observed) node.

These desirable properties, however, do come with some costs. Concerning consistency, GMU does not ensure serializability but only EUS. Regarding meta-data, GMU associates vector clocks, and not simple scalar, unlike for instance SPECULA. Indeed, vector clocks can have a non-negligible cost at high systems scale, as their usage leads to generate larger messages (i.e. higher network load), larger processing and storage overheads (i.e. higher CPU and memory consumption).

This Chapter seeks an answer to the following question: can we avoid the two above drawbacks of GMU, by relaxing exclusively its data freshness guarantees, and preserving its other desirable properties (i.e., GPR and abort-free read-only transaction)?

It is shown that the answer to the above question is yes, by presenting SCORe (SCalable One-copy serializable Replication protocol) [78]. Like GMU, SCORe employs a genuine partial replication scheme which guarantees that read-only transactions always observe a consistent snapshot of the data, hence avoiding to incur in expensive remote validation phases. This result is achieved by combining a local multiversion concurrency control algorithm with a highly scalable distributed logical-clock synchronization scheme that only requires the exchange of a scalar clock value among the nodes involved in the handling of a transaction. All the above features jointly allow SCORe to be performance effective, highly scalable, and able to provide support for a wider set of applications, including those that impose strict data consistency requirements.

The prize to pay is the possible lack in data freshness because SCORe is not able to dynamically advance the visible snapshot of a transaction through its execution. Therefore, unlike GMU, SCORe is not able to ensure that every read operation on datum $x$ executed in a quiescent system state always returns the last committed version of $x$. In particular SCORe only guarantees that a transaction is allowed to observe at least the snapshot committed on the transaction's originating node at the time the transaction began. This property is still important for scenarios in which a client submits a sequence of transactions by contacting always the same node, and it is guaranteed by other consistency criteria defined for replicated transactional systems, e.g.

Prefix-Consistent Snapshot Isolation [33].

As GMU, SCORe has been integrated in Infinispan [65]. This allows for performing a fair experimental comparison between the two protocols. Furthermore, this choice of integrating SCORe in a popular open-source data grid allowed enhancing the visibility of the proposed protocol also to the industrial community.

The effectiveness of SCORe has been assessed via an extensive experimental study based on both the TPC-C [93] and YCSB [23] benchmarks, using as experimental testbeds a private cluster with up to 20 nodes, and a public cloud (FutureGrid) with up to 100 nodes. Major outcomes from the study entail demonstrations of linear scalability by SCORe across a wide range of workloads, and analogous performance to that achievable by GMU, which however provides different guarantees for what concerns consistency and data freshness.

The remainder of this Chapter is organized as follows. The SCORe protocol is presented in Section 7.1 while its proof of correctness is provided in Section 7.2. The results of the experimental analysis are reported in Section 7.3.

## 7.1 The SCORe Protocol

### 7.1.1 Overview

SCORe is a genuine (hence highly scalable) partial replication protocol that implements a One-Copy Serializable distributed multi-version scheme. As in typical non-distributed multi-version algorithms [12], SCORe replicas store multiple versions of the data items that they maintain, each tagged with a scalar timestamp. Therefore SCORe adopts the data model as described in Chapter 3 and every version is associated a scalar version identifier *vid*. However, SCORe introduces a novel distributed timestamp management scheme that addresses two main issues: (i) establishing the snapshot visible by transactions, i.e. selecting which one, among the multiple versions of a datum (replicated across multiple nodes) should be observed by a transaction upon a read operation; (ii) determining the final global serialization order for update transactions via a distributed agreement protocol that takes place during the transactions' commit phase.

To this end SCORe maintains two scalar variables per node, namely *commitId* and *nextId*. The former one maintains the timestamp that was attributed to the last update transaction when committed on that node. *nextId*, on the other hand, keeps track of the next timestamp that the node will propose when it will receive a commit request for a transaction that accessed some of the data that it maintains.

Snapshot visibility for transactions is determined by associating with each transaction $T$ a scalar timestamp, called *snapshot identifier* or, more succinctly, *sid*. The *sid* of a transaction is established upon its first read operation. In this case the most recent version of the requested datum is returned, and the transaction's *sid* is set to the value of *commitId* at the transaction's originating node, if the read can be served locally. Otherwise, if the requested datum is not maintained locally, $T.sid$ is set equal to the maximum between *commitId* at the originating node and *commitId* at the remote node from which $T$ reads. From that moment on, any subsequent read operation is allowed to observe the most recent committed version of the requested datum having timestamp less than or equal to $T.sid$, as in classical multiversion concurrency control algorithms.

Therefore, unlike GMU, SCORe is not able to advance the observable snapshot on every read operation, and it only ensures that a transaction can observe at least all the writes that were committed on the transaction's originating node before the transaction begins.

To guarantee that the logical timestamps univocally identify committed snapshots of the transactional state (whether they are *commitId*, *nextId*, *sid* or *vid*), in SCORe they are represented in such a way for any pair of timestamps $id_i$ and $id_j$, if $id_i \leq id_j \wedge id_j \leq id_i$ then $i = j$, hence they are the same identifier associated to a unique commit. In addition if $i \neq j$, then either $id_i \leq id_j$ or $id_j \leq id_i$, and for any triple $id_i$, $id_j$ and $id_k$, if $id_j \leq id_j \wedge id_j \leq id_k$, then $id_i \leq id_k$. Therefore a set of identifiers in SCORe is always totally ordered under the binary relation $\leq$, and this property is ensured by implicitly supposing that a timestamp identifier can be represented as a pair of integer and node identifier (which can be a compact representation of its address that univocally identifies it in the system). In addition, throughout the following description, the notation $id_i < id_j$ is used to indicate that $id_i \leq id_j$ and $id_j \nleq id_i$, .

Analogously to GMU, SCORe relies on a genuine atomic commit protocol that can be seen as the fusion of the Two-Phase Commit algorithm (2PC) and the Skeen's total order multicast [44]. 2PC is used to validate update transactions and to guarantee the atomicity of the application of their post-images. Overlapped with 2PC, SCORe runs a distributed agreement protocol that allows to achieve a twofold goal: (i) totally ordering the commit events of transactions that update any data item in a partition $j$ among all the nodes that replicate $j$ (namely, $g_j$); (ii) tracking the serialization order between *update* transactions that exhibit (potentially transitive) data dependencies by totally ordering them via a scalar *commit timestamp* that is also used as version identifier of the post-images of committed transactions.

A key mechanism used in SCORe to correctly serialize transactions, and in particular to track write-after-read dependencies [12], is to update the *nextId* of a node upon the processing of a read operation. Specifically, if a node receives a read operation from a transaction $T$ having a *sid* larger than its local *nextId*, this is advanced to *T.sid*. This mechanism allows to guarantee that any update transaction $T^{up}$ that requests to commit on node $p_i$ at time $t$ is attributed a commit timestamp larger than the timestamp of any transaction $T$ that read a value from $p_i$ before time $t$, hence ensuring that $T^{up}$ is serialized after $T$.

Finally, since a transaction is attributed a snapshot identifier upon its first read, which is used throughout its execution, SCORe guarantees that the snapshot read by a transaction is always consistent with respect to a prefix of the equivalent serial history of committed transactions. As a consequence, in SCORe read-only transactions never abort and do not need to undergo any distributed validation.

The pseudocode of the SCORe protocol is reported in Algorithms 12, 13, 14, 15, 16 and 17, and discussed and analyzed in the following. For the sake of presentation, it is first assumed that the transaction's coordinator does not crash, and then Section 7.1.4 discusses how to relax this assumption.

### 7.1.2 Handling of Read and Write Operations

SCORe buffers write operations of transactions in a private write-set (denoted as *ws* in Algorithm 12), which is only made visible upon transaction's commit.

Read operations on a datum $d$ first check whether $d$ has already been updated by the transaction, returning in this case the value present in the transaction's writeset (lines 5-6 of Algorithm 12). Otherwise, it is necessary to establish which of the versions of $d$ is visible to the transaction. As already mentioned, transactions establish the *sid* that they use to determine version's visibility upon their first read. If this read operation is local, the transaction's *sid* is simply set equal to the originating node's *commitId* (lines 7-8 of Algorithm 12). Otherwise, it is set equal to the maximum between the *commitId* of the remote node from which the data is read and the *commitId* of the transaction's originating node (lines 17-18 of Algorithm 12 and lines 3-4 of Algorithm 14). Further, if the transaction's *sid* is higher than the node's *nextId*, the latter is set equal to *T.sid* (line 3 of Algorithm 13). This ensures that update transactions that subsequently issue a commit request on that node are serialized after $T$.

Next, the version visible by transaction $T$ is determined, as in conventional MVCC algorithms [12], by selecting the most recent version having commit timestamp less than $T$'s snapshot identifier (lines 6-11 of Algorithm 13). Before doing so, however, $T$ first waits for the completion of the commit phase of any transaction $T'$ that i) is updating $d$, and ii) is currently in its commit phase (line 5 of Algorithm 13). In fact, in case $T'$ is committed successfully, as it

**Algorithm 12** Write and Read operations (node $p_i$).

```
1:  void Write(Transaction T, Key k, Value val)
2:      $T.ws \leftarrow T.ws \setminus \{\langle k, - \rangle\} \cup \{\langle k, val \rangle\}$
3:
4:  Value Read(Transaction T, Key k)
5:      if $\exists < k, val > \in T.ws$ then
6:          return $val$
7:      if is first read of $T$ then
8:          $T.sid \leftarrow p_i.commitId$
9:      if $p_i \in replicas(\{k\})$ then
10:         $[val, last] \leftarrow doRead(T.sid, k)$
11:     else
12:         if (is first read of $T$) then
13:             send ReadRequest($[T.id, k, T.sid, \top]$) to all $p_j \in replicas(\{k\})$
14:         else
15:             send ReadRequest($[T.id, k, T.sid, \bot]$) to all $p_j \in replicas(\{k\})$
16:         wait receive ReadReturn($[tid, val, newRsid, lastCsid, last]$) from
    any $p_h \in replicas(\{k\})$
17:         if is first read of $T$ then
18:             $T.sid \leftarrow newRSid$
19:     if $last = \bot \wedge T.ws \neq \emptyset$ then
20:         throw ABORT
21:     $T.rs \leftarrow T.rs \cup \{k\}$
22:     return $val$
```

will be clearer in the following, it might be attributed a timestamp smaller than $T.sid$. Hence, $T'$ would be totally ordered before $T$ and the version of $d$ created by $T'$ would be visible to $T$. If $T'$ aborted, on the other hand, $T$ should not see its updates. In order to enforce the correct tracking of this read-after-write dependence, SCORe forces any transaction $T$ reading a data item $d$ to wait until there are no longer transaction commit events pending on $d$ and with a (either final or temporary) commit timestamp smaller than $T.sid$.

**Algorithm 13** Version visibility logic (node $p_i$).

```
1: [Value, bool] doRead(SnapshotId readSid, Key k)
2:     // Track write-after-read dependence
3:     $p_i.nextId \leftarrow max(p_i.nextId, readSid)$
4:     // Enforce read-after-write dependence
5:     wait until ($p_i.commitId \geq readSid \vee k.exclusiveUnlocked()$)
6:     $Version\ ver \leftarrow k.lastFinal$
7:     $bool\ last \leftarrow \top$
8:     while $ver.vid > sid$ do
9:         $ver \leftarrow ver.prev$
10:        $last \leftarrow \bot$
11:    return $[ver.val, last]$
```

The logic for handling remote read operations is defined by Algorithm 14. It is worthy to highlight that, even though transactions update their own *sid* only upon their first read operation, a node attempts to advance its local timestamps *commitId* and *nextId* whenever it receives a message (associated with the request or the response of a read operation) from another node in the system informing it that snapshots with higher timestamps have been already committed. This mechanism, which aims to maximize the freshness of visible snapshots, is encapsulated by the *updateNodeTimestamps* function (lines 12-15 of Algorithm 14). This function advances immediately the *nextId* timestamp, which is used to determine the timestamp proposed for future commit requests. However, additional care needs to be taken before advancing the node's *commitId* timestamp. As this timestamp determines the (minimum) snapshot visible by locally generated transactions, in fact, it can be increased to a new value, say *commitId'*, only if it is found that there are no committing transactions that may be given a timestamp less than or equal to *commitId'* (lines 17-18 of Algorithm 14).

Finally, SCORe includes a simple, yet effective, optimization that consists in immediately aborting update transactions which, based on their snapshot identifier, are forced to observe, upon a read operation, data item versions that have been already overwritten by more recently committed transactions (lines 19-20 of Algorithm 12).

**Algorithm 14** Handling of remote reads (node $p_i$).

```
1: upon receive ReadRequest([int tid, Key k, SnapshotId readSid, bool firstRead]) from $p_j$
2:     $SnapshotId\ newReadSid \leftarrow readSid$
3:     if $firstRead = \top \wedge p_i.commitId > newReadSid$ then
4:         $newReadSid \leftarrow p_i.commitId$
5:     $[val, last] \leftarrow doRead(newReadSid, k)$
6:     send ReadReturn($[tid, val, newReadSid, p_i.commitId, last]$)
7:     $updateNodeTimestamps(readSid)$
8:
9: upon receive ReadReturn([int tid, Value val, SnapshotId newRsid, SnapshotId lastCsid, bool last]) from $p_j$
10:    $updateNodeTimestamps(lastCsid)$
11:
12: void updateNodeTimestamps(SnapshotId lastCommittedSid)
13:    // Update global snapshot knowledge
14:    $p_i.nextId \leftarrow max(p_i.nextId, lastCommittedSid)$
15:    $p_i.maxSeenId \leftarrow max(p_i.maxSeenId, lastCommittedSid)$
16:
17: upon $(p_i.maxSeenId > p_i.commitId \wedge CommitQueue.isEmpty())$
18:    $p_i.commitId \leftarrow max(p_i.maxSeenId, p_i.commitId)$
```

### 7.1.3 Commit Phase

As already mentioned, with SCORe read-only transactions (lines 2-3 of Algorithm 15) can be committed without undergoing distributed validation phases (unlike, for instance, in [86]).

Update transactions, on the other hand, execute a Two-Phase Commit protocol, which is detailed in the following. To guarantee genuineness, SCORe involves in the commit phase of a transaction $T$ only the nodes that maintain replicas of the data items that $T$ accessed plus the coordinator of $T$, namely the node originating $T$. More in detail, when a node $p_i$ requests to commit transaction $T$, it broadcasts a PREPARE message to all nodes $p_j$ belonging to $replicas(T.rs \cup T.ws) \cup p_i$ (line 6 of Algorithm 15). Upon the receipt of this message, node $p_j$ verifies whether the transaction can be serialized after every transaction that has locally committed so far. To this end, it attempts to acquire exclusive, respectively shared, locks for the data in $T$'s write-set, respectively read-set, that it locally maintains. This lock acquisition is non-blocking since the node waits for a busy lock only for a certain amount of time, which is determined by means of a configurable timeout parameter (lines 2-3 of Algorithm 16). Next, if the acquisition of the locks succeeds, the node validates $T$'s read-set (line 4 of Algorithm 16), verifying that none of the items read by $T$ has been overwritten by a more recently committed transaction (in terms of timsestamp identifiers). If any of these operations fails, $T$ is simply rolled back, which will yield to the abort of the whole distributed transaction, as in classic 2PC (lines 6-7 of Algorithm 16).

**Algorithm 15** Commit phase (node $p_i$).

```
1:  bool Commit(Transaction T)
2:      if $T.ws = \emptyset$ then
3:          return $\top$;
4:      bool outcome $\leftarrow \top$;
5:      Set proposedSn $\leftarrow \emptyset$;
6:      send PREPARE($[T, T.sid, T.rs, T.ws]$) to all $p_j \in replicas(T.rs \cup T.ws) \cup \{p_i\}$
7:      for all $p_j \in replicas(T.rs \cup T.ws) \cup \{p_i\}$ do
8:          wait receive VOTE($[T, sn, res]$) from $p_j$ or timeout
9:          if $res = \bot \vee$ timeout then
10:             $outcome \leftarrow \bot$
11:             break
12:         else
13:             $proposedSn \leftarrow proposedSn \cup sn$
14:     $T.sid \leftarrow max(proposedSn)$
15:     send DECIDE($[T, T.sid, outcome]$) to all $p_j \in replicas(T.rs \cup T.ws) \cup \{p_i\}$
16:     wait until $T.completed = \top$
17:     return $T.outcome$
18:
19: bool validate(Set rs, SnapshotId sid)
20:     for all $k \in rs$ do
21:         if $k.lastFinal.vid > sid$ then
22:             return $\bot$
23:     return $\top$
```

If the transaction passes the validation phase, however, the VOTE message of 2PC is exploited to overlap a distributed agreement scheme similar in spirit to Skeen's total order multicast algorithm that aims to establish the final serialization order for the transaction, as in GMU. More in detail, $p_j$ increments the $nextId$ timestamp, inserts the triple $\langle T, p_j.nextId, pending\rangle$, defined on the domain $Transaction \times SnapshotId \times \{pending, ready\}$ in a queue of pending committing transactions (denoted as $CommitQueue$ like in GMU) ordered by $SnapshotId$, and sends back to the transaction coordinator the value of $p_j.nextId$ in piggyback to the VOTE message (lines 8-11 of Algorithm 16). The coordinator gathers the VOTE messages (aborting the transaction in case one of the contacted node does not respond within a predefined timeout), determines the final commit timestamp for $T$ as the maximum among the timestamps proposed by the transaction's participants, and broadcasts back a DECIDE message with the transaction's final commit timestamp (lines 7-15 of Algorithm 15).

Upon the receipt of the DECIDE message (lines 13-25 of Algorithm 16) with a positive outcome, unlike classical 2PC and like the GMU scheme, the transaction is not necessarily immediately committed. In fact, as each data item is replicated over more than one node, and since 1CS has to be ensured without requiring the validation of read-only transactions, SCORe guarantees that the commit events of all update transactions (even non-conflicting ones) are totally ordered across all the replicas of a same partition. To ensure this result, when a DECIDE message is received on $p_j$ for transaction $T$ with final commit timestamp $fsn$, $T$ is removed from *CommitQueue* and it is immediately committed (atomically increasing $p_j.nextId$) only if there are no other transactions in *CommitQueue* with snapshot id less than $fsn$. If this is not the case, the old entry of $T$ is updated in *CommitQueue* with the values $\langle T, fsn, ready\rangle$ and it is ordered accordingly, while the commit of $T$ is is delayed till it can be ensured that no other pending transaction will ever receive a final commit snapshot id less than $fsn$ (Algorithm 17).

**Algorithm 16** Prepare and Decide messages (node $p_i$).

```
1:  upon receive PREPARE([Transaction T]) from $p_j$
2:      $bool\ outcome \leftarrow getExclLocksWithTimeout(T.id, T.ws)$
3:      $outcome \leftarrow outcome \wedge getSharedLocksWithTimeout(T.id, T.rs)$
4:      $outcome \leftarrow outcome \wedge validate(T.rs,\ T.sid)$
5:      $SnapshotIdsn \leftarrow NULL\_SID$
6:      if $outcome = \perp$ then
7:          $releaseLocks(T.id, T.ws, T.rs)$
8:      else
9:          $sn \leftarrow p_i.nextId \leftarrow p_i.nextId + 1$
10:         $CommitQueue.put(\langle T, sn, pending\rangle)$
11:     send VOTE ($[T.id, sn, outcome]$) to $p_j$
12:
13: upon receive DECIDE([Transaction T, SnapshotId fsn, bool outcome]) from $p_j$
14:     if $outcome = \top$ then
15:         $p_i.nextId \leftarrow max(p_i.nextId, fsn)$
16:         if $p_i \in replicas(T.ws \cup T.rs)$ then
17:             $CommitQueue.update(\langle T, fsn, ready\rangle)$
18:         else
19:             $T.outcome \leftarrow \top$
20:             $T.completed \leftarrow \top$
21:     else
22:         $CommitQueue.remove(T)$
23:         $releaseLocks(T.id, T.ws, T.rs)$
24:         $T.outcome \leftarrow \perp$
25:         $T.completed \leftarrow \top$
```

**Algorithm 17** Finalizing the commit phase of transaction $T$ (node $p_i$).

```
1: upon $(\exists\langle T, fsn, s\rangle : \langle T, fsn, s\rangle = CommitQueue.head \wedge s = ready \wedge (\nexists\langle\bar{T}, \bar{sn}, \bar{s}\rangle \in CommitQueue : \bar{sn} < fsn))$
2:     $\forall\langle k, val\rangle \in T.ws : p_i \in replicas(\{k\})$ do $apply(k, val, fsn)$
3:     $p_i.commitId = fsn$
4:     $CommitQueue.remove(T)$
5:     $releaseLocks(T.id, T.ws, T.rs)$
6:     $T.outcome \leftarrow \top$
7:     $T.completed \leftarrow \top$
```

### 7.1.4 Garbage Collection and Fault-Tolerance

As in non-distributed MVCC algorithms, versions of a data item $d$ having timestamps less than the *sid* of any active transaction can be safely removed, provided that most recent versions of $d$ have already been committed. In a distributed platform, it is required to disseminate the information on the *sid* of the oldest active transaction at each node. This information can be spread by relying, e.g., on lazy approaches based on piggybacking or gossip [95], as described for GMU.

Like GMU, for simplicity SCORe has been presented as layered on top of 2PC protocol. Therefore the issues of how ensuring fault-tolerance in SCORe are the same of GMU and the solutions described in Section 6.1.4 apply to SCORe as well.

## 7.2 Correctness Proof

The correctness proof of SCORe is less complex than the one of GMU because it only needs to show that every multiversioned history of committed transactions executed by SCORe is One-Copy Serializable (1CS). In addition, this Section also proves that SCORe provides an isolation guarantee that is even stronger than 1CS, i.e. Executing 1CS (E1CS) [3] (Section 3.4.2), because it enforces that every executed history (without limiting to those of committed transactions) is 1CS. Therefore it is proved that for every history $\mathcal{H}$, both $DSG(\mathcal{H}^c)$ and $DSG(\mathcal{H})$ do not contain any oriented cycle, where $\mathcal{H}^c$ is the history obtained from $\mathcal{H}$ by removing all the aborted and executing transactions.

**Theorem 7.2.1.** *For each history $\mathcal{H}$, $DSG(\mathcal{H}^c)$ does not contain any oriented cycle, where $\mathcal{H}^c$ is the history obtained from $\mathcal{H}$ by removing all the aborted and executing transactions.*

*Proof.* The proof is based on establishing a mapping between each vertex $V_{T_i}$ in $DSG(\mathcal{H}^c)$ and the value of the commit timestamp of $T_i$, denoted as

$commitSId(T_i)$. The acyclicity of the $DSG(\mathcal{H}^c)$ is proved by showing that for each edge $V_{T_i} \xrightarrow{E} V_{T_j} \in DSG(\mathcal{H}^c)$ SCORe guarantees that $commitSId(T_i) \leq commitSId(T_j)$.

Note that, if $T_i$ is a read-only transaction, $commitSId(T_i)$ is equal to the *sid* assigned to $T_i$ upon its first read operation. On the other hand, in case $T_i$ is an update transaction, $commitSId(T_i)$ is computed during $T_i$'s commit phase and is equal to the maximum identifier among the ones proposed by the nodes involved in the commit of $T_i$ .

$E$ is first assumed to be a direct write-dependence edge, and in this case it is shown that SCORe ensures $commitSId(T_i) < commitSId(T_j)$. This is because $T_i$ and $T_j$ are both update transactions and they commit on a common subset $S$ of the nodes in the system (at least the nodes storing the data item on which the write-dependence is materialized). In fact, in accordance with the design of the commit phase, it is ensured that: (i) $T_j$ cannot enter the commit phase of the protocol before $T_i$ has committed, since $T_j$ has to wait for the release of some exclusive lock owned by $T_i$ at least on the nodes in $S$; (ii) $T_i$ updates the *nextId* on the nodes in $S$ to a value at least equal to $commitSId(T_i)$ before finalizing its commit; (iii) the $commitSId(T_j)$ is chosen as the maximum among the *nextId* values, incremented by one, of the nodes involved in the commit of $T_j$.

Now assume that $E$ is a direct read-dependence edge. This means that $T_j$ has read a version committed by $T_i$. Therefore the snapshot identifier used by $T_j$ to perform read operations, i.e. $T_j.sid$, is greater than or equal to the $T_i$'s commit snapshot identifier due to the reading rule defined by the protocol. So, if $T_j$ is a read-only transaction, this entails that $commitSId(T_i) \leq T_j.sid = commitSId(T_j)$; otherwise, if $T_j$ is an update transaction its commit snapshot identifier will be always greater than its reading snapshot identifier, since the value proposed by each node involved in the commit of $T_j$ (i.e. the incremented *nextId*) is greater than every snapshot seen by $T_j$. As a consequence, $commitSId(T_i) < commitSId(T_j)$ holds.

Finally, if $E$ is a direct anti-dependence edge, then two scenarios have to be distinguished. In the former, if $T_i$ is a read-only transaction, then the $commitSId(T_j)$ is greater than $commitSId(T_i)$ since (i) the $T_j$'s commit snapshot identifier is at least equal to all the values proposed for its commit and (ii) there exists a value among the one proposed that is guaranteed to be greater than $T_i$'s reading snapshot identifier (i.e. $commitSId(T_i)$ in this scenario) due to the visibility rule adopted on each read operation of $T_i$. In particular, $T_i$ performs a read operation on a data item $x$ of a node $p$ only after it has ensured that (i) the *nextId* value on $p$ will be greater than or equal to its reading snapshot identifier and (ii) no transaction will commit an update on $x$ using a snapshot id not greater than $commitSId(T_i)$. Otherwise, if $T_i$ is an update transaction, it is guaranteed that at the time $T_j$ commits, $T_i$ has been already successfully committed otherwise $T_i$'s read-set would have been invalidated by $T_j$. This case is analogous to the one in which $E$ is a write-dependence edge since there are two update transactions, $T_i$ and $T_j$, that commit on a common subset of nodes $S$, and $T_i$ commits before $T_j$; therefore $commitSId(T_i) < commitSId(T_j)$ holds.

Therefore if $DSG(\mathcal{H}^c)$ contained an oriented cycle $C$, it would follow that for every transaction $T_i$ involved in $C$, $commitSId(T_i) < commitSId(T_i)$, which is impossible. Hence the claim follows. □

Theorem 7.2.1 have proved that SCORe guarantees 1CS. Now, the following Theorem proves that SCORe guarantees a consistency criterion stronger than 1CS, i.e. E1CS, because the protocol ensures that the read operations issued by every transaction $T \in \mathcal{H}$, even those that eventually abort, observe the state generated by a prefix of a sequential history equivalent to $\mathcal{H}^{upc}$, where $\mathcal{H}^{upc}$ is the history of committed update transactions in $\mathcal{H}$. In addition, unlike EUS, for any pair of transactions $T_i$ and $T_j$, the states observed by $T_i$ and $T_j$ are generated by two prefixes of the same sequential history equivalent to $\mathcal{H}^{upc}$.

**Theorem 7.2.2.** *For each history $\mathcal{H}$, $DSG(\mathcal{H})$ does not contain any oriented cycle.*

*Proof.* The proof automatically follows by the proof of Theorem 7.2.1 and by considering that: (i) since a write operation is externalized only upon a successful commit, a live or an aborted transaction at time $t$ can be considered as a committed read-only transaction that contains its read prefix performed until $t$, except the operation which has triggered an abort (if any); (ii) the $DSG(\mathcal{H})$ graph is built as $DSG(\mathcal{H}^c)$ (so it has a node for each committed transaction) and it also has a node for each aborted/live transaction reduced to its read prefix. □

The property E1CS, is also implied by the more recent Opacity property defined as the reference correctness criterion for in-memory transactional systems [40] (Section 3.4.3). However, it is easy to show that, since SCORe fixes the timestamp of a transaction upon its first read operation, it neither guarantees real-time order, as required by Opacity, nor tries to maximize the freshness of data read, like GMU.

## 7.3 Experimental Evaluation

This Section reports the results of an experimental study aimed at evaluating the performance and scalability of SCORe. This study is based on a prototype implementation of SCORe[1] that has been integrated within the Infinispan data grid system.

The strongest consistency level ensured by Infinispan is Repeatable Read [11] (RR), which guarantees that no intermediate or aborted values are ever observed, and that no two reads on the same key within the same transaction can return different values. RR is definitely weaker than Serializability, as it allows the commit of (both read-only and update) transactions that observe non-serializable schedules [3].

Being Infinispan designed to achieve high scalability in the context of weak data consistency models, it represents an ideal baseline to evaluate the costs incurred in by the SCORe protocol in order to provide 1CS (i.e. strong consistency) guarantees.

SCORe has been evaluated by using the same benchmarks adopted for the experimental evaluation of GMU (Section 6.4): the TPC-C [93] benchmark adapted to execute on a NoSQL platform such as Infinispan, and YCSB (Yahoo! Cloud Serving Benchmark) [23], which is specifically targeted at the assessment of key-value data grids and cloud stores.

The test-bed platforms adopted are the ones described in Section 6.4 for GMU, i.e. the Cloud-TM platform of 20 homogeneous nodes, and FutureGrid, which is a public distributed test-bed for parallel and cloud computing. Unlike the experimental evaluation of GMU, the experiments on top of the FutureGrid platform have been performed by using up to 100 virtual machines, equipped with 4GB RAM, two 2.93GHz cores Intel Xeon CPU X5570, running CentOS 5.7 x86_64.

Finally, for both deploys on the above described platforms, the replication degree of each data item is set to the value 2.

Figure 7.1 shows the achieved throughput values for TPC-C on top of the Cloud-TM platform while varying the number of involved nodes between 2 and 20. The plots in the top row refer to the workload composed at the 90% by read-only transactions, denoted as Workload A. The left plot reports the throughput for write transactions, whereas the right plot reports the throughput for read-only transactions. The performance of SCORe are contrasted with that of the native RR scheme supported by Infinispan, and with that of the GMU protocol. As already discussed, GMU ensures a consistency criterion (namely EUS) weaker than 1CS, but stronger than RR. In other words, GMU exhibits intermediate consistency semantics with respect to the other two analyzed protocols.

The plots highlight that SCORe attains throughput values that are even slightly better than those achieved by GMU. This phenomenon is explainable by considering that, while SCORe relies on a timestamping mechanism based

[1]The SCORe prototype is publicly available at the URL `http://www.cloudtm.eu`.

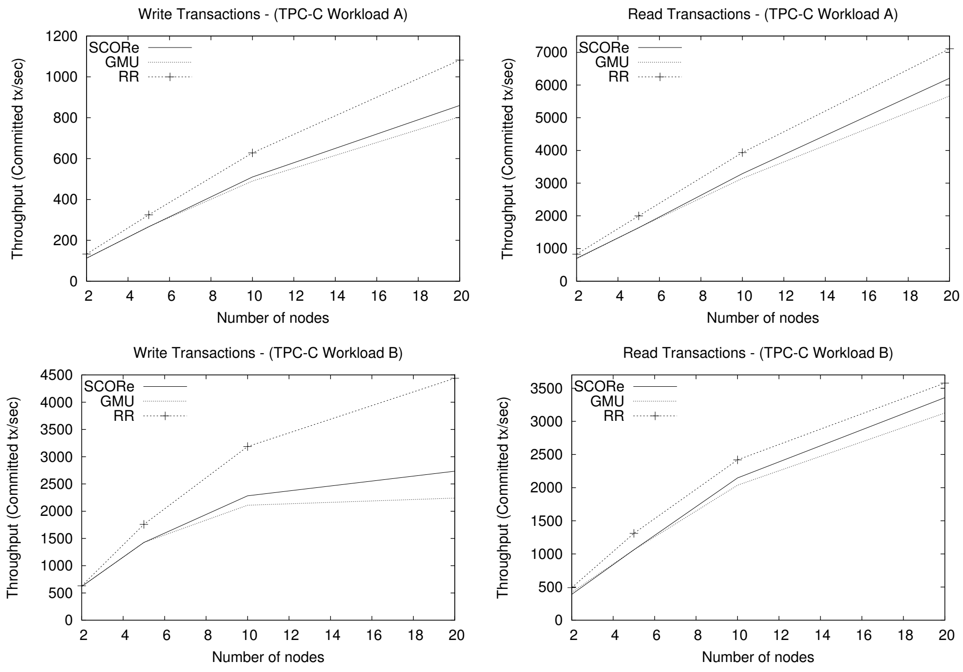


Figure 7.1: TPC-C Benchmark (Cloud-TM).

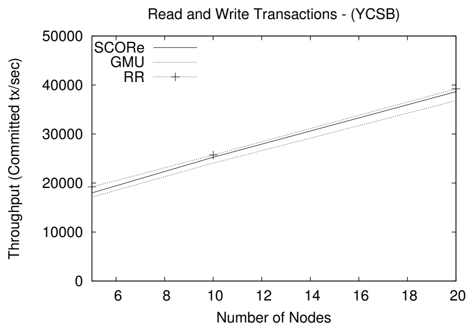


Figure 7.2: YCSB Benchmark (Cloud-TM).

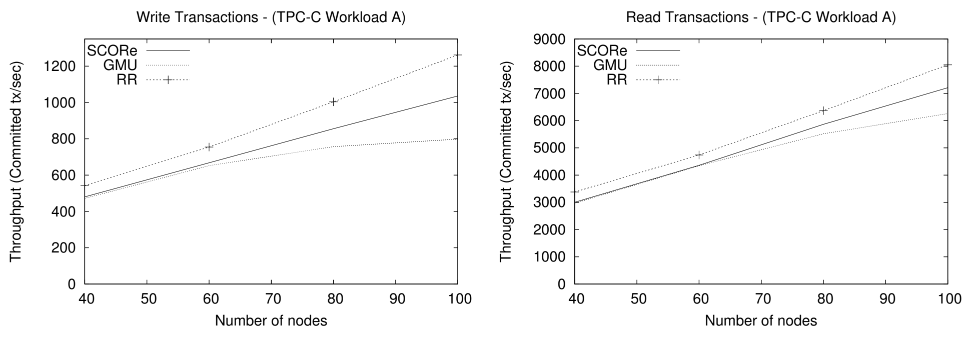


Figure 7.3: TPC-C Benchmark (FutureGrid).

on scalar clock values, GMU uses vector clocks, which introduce higher overheads with respect to scalar clocks as the number of nodes in system grows.

The plot in the bottom row of Figure 7.1 reports the results for TPC-C, obtained on top of the Cloud-TM platform, for the scenario encompassing 50% of read-only transactions, denoted as Workload B. While the comparative behavior of SCORe vs GMU follows trends similar to those observed for 90% read-only transactions, this time the performance loss of SCORe vs RR for update transactions grows significantly. This is essentially due to the fact that the increased volume of update transactions leads to an increased abort rate caused predominantly by aborts during the validation phase of the transaction's read-set (interestingly, the aborts due to failures in the lock acquisition phase turned out to be statistically marginal). In other words, as the update rate grows, the probability for an update transaction to access a stale snapshot accordingly grows. In particular, for the case of 20 nodes, the abort probability for update transactions with SCORe is on the order of 43%, while RR only exhibits around 8% abort rate for update transactions, with aborts exclusively caused by deadlocks. However, when considering the total throughput for Workload B (including both read-only plus update transactions), SCORe exhibits similar scalability trend when compared to RR. Overall, the data show that, for increased contention scenarios, strong consistency semantics do pay a performance toll, which, in this specific configuration, corresponds to a throughput reduction up to 22% (at 20 nodes). On the other hand this is an unavoidable cost to pay in applications whose correctness can be endangered by adopting non-serializable isolation levels.

Figure 7.2 shows the results obtained by running YCSB on the Cloud-TM platform. Workload A [23] of the benchmark was used, which is an update intensive workload (comprising 50% of update transactions) simulating a session store that records recent client actions. The maximum throughput (committed transactions per second) achievable by the three considered protocols has been

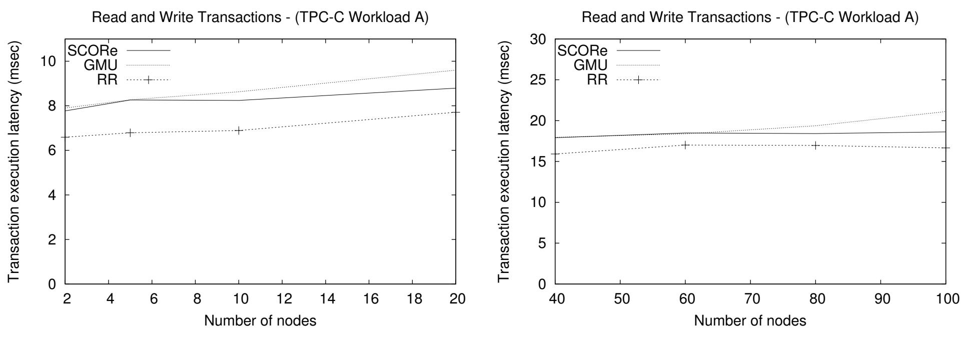


Figure 7.4: Average transaction execution latency (TPC-C Workload A) for both Cloud-TM (left) and FutureGrid (right).

reported. The plot shows that the average reduction in throughput for both SCORe and GMU, compared to RR, oscillates around 8%, and that, again, the throughput scales linearly at the same rate as RR, providing an evidence of the efficiency and scalability of the proposed solution when considering transaction profiles featuring applications natively tailored for key-value data stores.

Figure 7.3 presents the results obtained by running Workload A of TPC-C on FutureGrid. The data confirm the general trends already observed on the Cloud-TM platform, highlighting both the high scalability of the proposed solution and its high efficiency when compared to vector-clock-based solutions, such as GMU, whose overheads grow linearly with the scale of the platform.

Finally, for completeness of the analysis, Figure 7.4 reports the average transaction execution latency for the case of TPC-C (Workload A) run on both Cloud-TM and FutureGrid. By the data it is shown that, for all the protocols, latency values stay almost flat while increasing the size on the underlying platform (and consequently of the total workload sustained), which again supports the claim of good scalability of SCORe. Further, the relevance of this result is supported by the fact that all the reported values were related to scenarios where the utilization of infrastructural resources was high (as an example, for the tests with TPC-C on top of FutureGrid the CPU utilization was constantly observed to be over the 80%). Hence, the data refer to scenarios where the throughput was relatively close to the maximum sustainable one.

# Chapter 8

# Concluding Remarks

This dissertation have presented innovative research results in the context of distributed transactional systems, with the common objective of enhancing the efficiency in the handling of data replication. This has been done to effectively cope with recent technological trends, which have significantly increased the relative impact that the inter-replica synchronization costs have on the performance of the systems. In particular, three main recent architectural evolutions have exacerbated the ratio between replicas coordination time and transaction execution time, raising the need for revisiting existing approaches to transactional replication. The first one is the Transactional Memory (TM) programming paradigm, which has changed the structure of the transactional workloads: while relational databases are optimized to support complex SQL queries on data possibly stored on disk, TM's workloads are often dominated by transactions that perform very few read/write access to in-memory variables. The second one is the Solid-State Drive technology that has allowed the implementation of faster storage components, and has changed non-functional (e.g. performance/response time) characteristics of transactional processing. The third one is the trend towards ever growing levels of scale, which, by increasing the number of replicas that need to agree on the transactions' outcome, generally leads to an increase of per-transaction replica coordination cost.

Four replication protocols have been proposed in order to face the aforementioned issue via the exploitation of differentiated approaches tailoring either full or partial replication scenarios. In the following the key aspects of each proposed solution are summarized.

– SPECULA is a transactional replication protocol for fully replicated environments that aims at minimizing the overhead of the replica coordination phase by speculating on its eventual success. It lets application level threads speculatively commit transactions and optimistically pipeline the execution of subsequent transactional/non-transactional code blocks. Therefore it is able to prevent threads from blocking till the completion of a distributed coordination for a transaction executed locally, by seeking the complete overlap between transactions execution and replicas synchronization. SPECULA relies on three key building blocks: an innovative multi-version concurrency control, which manages the coexistence of speculatively and finally committed data versions while ensuring serializability of the snapshots observed by transactions throughout their execution; a novel distributed certification protocol, which ensures that the history of finally committed transactions is One-Copy Serializable; a set of mechanisms (including continuations, undo-logging of updates on non-transactional heap variables, automatic detection of non-revocable operations) aimed at ensuring total transparency of the management of speculative executions for the user level application.

– Lilac-TM is a fully decentralized, LocalIty-aware LeAse-based repliCated TM protocol for fully replicated environments. Its core design is based on the idea of asynchronous leases such that a node needs to acquire exclusive leases on a subset $S$ of the transactional state to commit transactions on $S$. It exploits a novel, self-optimizing lease circulation scheme that provides two key benefits: (i) limiting the frequency of lease circulation by dynamically deciding whether to circulate leases or migrate transactions, and (ii) enhancing the contention management efficiency, by increasing the probability that conflicting transactions are executed on the same node. Lilac-TM relies on three main building blocks: a fine-grained lease manager, which facilitates the exploitation of locality and consequently reduces lease circulation, by decoupling lease requests from the requesting transaction's data-set via the fine-grained Lease Ownership Records; the Transaction Forwarder, which is responsible for managing the forwarding of a transaction to a different node, in order to achieve the goal of minimizing the execution rate of expensive Atomic Broadcast-based consensus protocols; the Distributed Transaction Dispatching module, which encapsulates the logic for determining whether to process the commit of a transaction locally, by issuing lease requests if required, or to migrate its execution to a remote node.

– GMU (Genuine Multi-version Update serializability) protocol is an innovative partial replication protocol for transactional systems. The core of GMU is a distributed multi-version concurrency control scheme, which relies on a novel vector clock based synchronization algorithm to track, in a totally decentralized (and consequently scalable) way, both data and causal dependency relations. In order to maximize scalability, GMU adopts a genuine partial replication mechanism that ensures that transactions only contact replicas storing the data that they accessed. Further, GMU never aborts read-only transactions and spares them from expensive distributed validation schemes. In addition, despite the genuineness of its replication mechanism, GMU always tries to maximize the freshness of data returned upon a read operation. This is achieved by adopting a weaker consistency criterion than classic One-Copy Serializability, namely EUS. Informally, EUS guarantees One-Copy Serializability for the committed update transactions and disallows all transactions (also aborted ones) to observe non-consistent snapshots. On the other hand it allows both read-only and executing/aborted transactions two observe different non-compatible histories of committed write transactions.

– SCORe (SCalable One-copy serializable Replication) protocol is an innovative genuine partial replication protocol for transactional systems that, like GMU, maximizes system scalability by demanding that only the replicas that maintain data accessed by a transaction are involved in its processing, and read-only transactions never abort or undergo an expensive distributed validation. Its novelty is given by the consistency level it guarantees, since SCORe is the first genuine partial replication protocol with abort-free read-only transactions that also ensures Executing One-Copy Serializability (E1CS). E1CS is stronger then One-Copy Serializability because it guarantees that the whole history of executed transactions is One-Copy Serializable, by also including aborted and executing transactions. This is achieved by reaching a compromise between data freshness and consistency since, SCORe achieves E1CS by sacrificing data freshness, and achieving a weaker property such that transactions can observe at least the write operations committed on the originating nodes when they begin. Further, unlike GMU, SCORe avoids relying on vector clocks, and uses simpler and more efficient scalar clocks to establish transactions' serialization order.

All the theoretical achievements have also been experimentally evaluated by integrating the designed protocols into state of the art academic and industrial transactional platforms. Additionally, the experimental results presented in this dissertation were obtained by relying on both traditional and cloud infrastructures, which further strengths the relevance of the experimental outcomes.

# Bibliography

[1] Daniel J. Abadi. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer*, 45(2):37–42, February 2012.

[2] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Unlocking Concurrency. *Queue*, 4(10):24–33, December 2006.

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

[4] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP, 2007.

[5] Sérgio Almeida, João Leitão, and Luís Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[6] Cristiana Amza, Alan Cox, Karthick Rajamani, and Willy Zwaenepoel. Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, 1997.

[7] Ivo Anjo and João Cachopo. JaSPEx: Speculative Parallel Execution of Java Applications. In *Proceedings of the Simpósio de Informática*, INFORUM, 2009.

[8] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-monotonic Snapshot Isolation: Scalable and Strong Consistency for Geo-replicated Transactional Systems. In *Proceedings of the IEEE 32nd International Symposium on Reliable Distributed Systems*, SRDS, 2013.

[9] J. E. Armendáriz-Iñigo, A. Mauch-Goya, J. R. González de Mendívil, and F. D. Muñoz Escoí. SIPRe: A Partial Database Replication Protocol with SI Replicas. In *Proceedings of the ACM Symposium on Applied Computing*, SAC, 2008.

[10] Alberto Bartoli and Ozalp Babaoglu. Selecting a "Primary Partition" in Partitionable Asynchronous Distributed Systems. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, SRDS, 1997.

[11] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD, 1995.

[12] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., 1987.

[13] Eric A. Brewer. Towards Robust Distributed Systems (Abstract). In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, PODC, 2000.

[14] Andrey Brito, Christof Fetzer, and Pascal Felber. Multithreading-Enabled Active Replication for Event Stream Processing Operators. In *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems*, SRDS, 2009.

[15] João Cachopo and António Rito-Silva. Versioned Boxes As the Basis for Memory Transactions. *Science of Computer Programming*, 63(2):172–185, December 2006.

[16] Lasaro Camargos, Fernando Pedone, and Rodrigo Schmidt. A Primary-Backup Protocol for In-Memory Database Replication. In *Proceedings of the 5th IEEE International Symposium on Network Computing and Applications*, NCA, 2006.

[17] Nuno Carvalho, João Cachopo, Luís Rodrigues, and Antonio Rito Silva. Versioned transactional shared memory for the FenixEDU web application. In *Proceedings of the 2nd workshop on Dependable distributed data management*, SDDDM, 2008.

[18] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. Asynchronous Lease-based Replication of Software Transactional Memory. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware, 2010.

[19] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. A Generic Framework for Replicated Software Transactional Memories. In *Proceedings of the IEEE 10th International Symposium on Network Computing and Applications*, NCA, 2011.

[20] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. SCert: Speculative Certification in Replicated Software Transactional Memories. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR, 2011.

[21] Bernadette Charron-Bost and André Schiper. Uniform Consensus is Harder Than Consensus. *Journal of Algorithms*, 51(1):15–37, April 2004.

[22] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 33(4):427–469, December 2001.

[23] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC, 2010.

[24] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luís Rodrigues. $D^2$STM: Dependable Distributed Software Transactional Memory. In *Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC, 2009.

[25] Maria Couceiro, Paolo Romano, and Luis Rodrigues. PolyCert: Polymorphic Self-optimizing Replication for In-memory Transactional Grids. In *Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware*, Middleware, 2011.

[26] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proceedings of the VLDB Endowment*, 4(8):494–505, May 2011.

[27] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP, 2007.

[28] Xavier Défago, André Schiper, and Péter Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys*, 36(4):372–421, December 2004.

[29] Aditya Dhoke, Binoy Ravindran, and Bo Zhang. On Closed Nesting and Checkpointing in Fault-Tolerant Distributed Transactional Memory. In *Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS, 2013.

[30] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC, 2006.

[31] Nuno Lourenço Diegues and Paolo Romano. Bumper: Sheltering Transactions from Conflicts. In *Proceedings of the IEEE 32nd International Symposium on Reliable Distributed Systems*, SRDS, 2013.

[32] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM Can Be More Than a Research Toy. *Communications of the ACM*, 54(4):70–77, April 2011.

[33] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database Replication Using Generalized Snapshot Isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, SRDS, 2005.

[34] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[35] Svend Frølund and Rachid Guerraoui. Implementing E-Transactions with Asynchronous Replication. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):133–146, February 2001.

[36] Hector Garcia-Molina. Elections in a Distributed Computing System. *IEEE Transactions on Computers*, 31(1):48–59, January 1982.

[37] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *ACM SIGACT News*, 33(2):51–59, June 2002.

[38] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The Dangers of Replication and a Solution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD, 1996.

[39] Jim Gray and Leslie Lamport. Consensus on Transaction Commit. *ACM Transactions on Database Systems*, 31(1):133–160, March 2006.

[40] Rachid Guerraoui and Michal Kapalka. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2008.

[41] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys, 2007.

[42] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. Throughput Optimal Total Order Broadcast for Cluster Environments. *ACM Transactions on Computer Systems*, 28(2):5:1–5:32, July 2010.

[43] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming.* Springer-Verlag New York, Inc., 2006.

[44] Rachid Guerraoui and André Schiper. Genuine Atomic Multicast in Asynchronous Distributed Systems. *Theoretical Computer Science*, 254(1-2):297–316, March 2001.

[45] R. C. Hansdah and Lalit M. Patnaik. Update Serializability in Locking. In *Proceedings on International Conference on Database Theory*, ICDT, 1986.

[46] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable Memory Transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2005.

[47] Danny Hendler, Alex Naiman, Sebastiano Peluso, Francesco Quaglia, Paolo Romano, and Adi Suissa. Exploiting Locality in Lease-Based Replicated Transactional Memory via Task Migration. In *Proceedings of the 27th International Conference on Distributed Computing*, DISC, 2013.

[48] Maurice Herlihy and Victor Luchangco. Distributed Computing and the Multicore Revolution. *SIGACT News*, 39(1):62–72, March 2008.

[49] Maurice Herlihy, Victor Luchangco, and Mark Moir. A Flexible Framework for Implementing Software Transactional Memory. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA, 2006.

[50] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA, 1993.

[51] Ricardo Jiménez-Peris, Marta Patiño Martínez, and Gustavo Alonso. Non-Intrusive, Parallel Recovery of Replicated Data. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, SRDS, 2002.

[52] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, STOC, 1997.

[53] Bettina Kemme and Gustavo Alonso. A Suite of Database Replication Protocols Based on Group Communication Primitives. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, ICDCS, 1998.

[54] 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, 2000.

[55] Bettina Kemme, Fernando Pedone, Gustavo Alonso, and André Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, ICDCS, 1999.

[56] Bettina Kemme, Fernando Pedone, Gustavo Alonso, Andre Schiper, and Matthias Wiesmann. Using Optimistic Atomic Broadcast in Transaction Processing Systems. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):1018–1032, July 2003.

[57] Eric Koskinen and Maurice Herlihy. Checkpoints and Continuations Instead of Nested Transactions. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA, 2008.

[58] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[59] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[60] Leslie Lamport. Generalized Consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft, `http://research.microsoft.com/apps/pubs/default.aspx?id=64631`, March 2005.

[61] Tobias Landes. Dynamic Vector Clocks for Consistent Ordering of Events in Dynamic Distributed Applications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems and Applications*, PDPTA, 2006.

[62] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don'T Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP, 2011.

[63] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI, 2013.

[64] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. High Performance State-machine Replication. In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN, 2011.

[65] Francesco Marchioni and Manik Surtani. *Infinispan Data Grid Platform.* PACKT Publishing, 2012.

[66] Hugo Miranda, Alexandre Pinto, and Rodrigues Luís. Appia: A Flexible Protocol Kernel Supporting Multiple Coordinated Channels. In *Proceedings of the the 21st International Conference on Distributed Computing Systems*, ICDCS, 2001.

[67] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD Thesis, Massachusetts Institute of Technology, 1985.

[68] Roberto Palmieri, Francesco Quaglia, and Paolo Romano. AGGRO: Boosting STM Replication via Aggressively Optimistic Transaction Processing. In *Proceedings of the 9th IEEE International Symposium on Network Computing and Applications*, NCA, 2010.

[69] Roberto Palmieri, Francesco Quaglia, and Paolo Romano. OSARE: Opportunistic Speculation in Actively REplicated Transactional Systems. In *Proceedings of the IEEE 30th International Symposium on Reliable Distributed Systems*, SRDS, 2011.

[70] Roberto Palmieri, Francesco Quaglia, and Paolo Romano. ASAP: An Aggressive SpeculAtive Protocol for Actively Replicated Transactional

Systems. In *Proceedings of the IEEE 11th International Symposium on Network Computing and Applications*, NCA, 2012.

[71] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[72] 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, 2000.

[73] Stacy Patterson, Aaron J. Elmore, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Serializability, Not Serial: Concurrency Control and Availability in Multi-datacenter Datastores. *Proceedings of the VLDB Endowment*, 5(11):1459–1470, July 2012.

[74] Fernando Pedone, Rachid Guerraoui, and André Schiper. The Database State Machine Approach. *Distributed and Parallel Databases*, 14(1):71–98, July 2003.

[75] Fernando Pedone and André Schiper. Optimistic Atomic Broadcast: A Pragmatic Viewpoint. *Theoretical Computer Science*, 291(1):79–101, January 2003.

[76] Sebastiano Peluso, João Fernandes, Paolo Romano, Francesco Quaglia, and Luís Rodrigues. SPECULA: Speculative Replication of Software Transactional Memory. In *Proceedings of the IEEE 31st Symposium on Reliable Distributed Systems*, SRDS, 2012.

[77] Sebastiano Peluso, Roberto Palmieri, Francesco Quaglia, and Binoy Ravindran. On the Viability of Speculative Transactional Replication in Database Systems: A Case Study with PostgreSQL. In *Proceedings of the 12th IEEE International Symposium on Network Computing and Applications*, NCA, 2013.

[78] Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. SCORe: A Scalable One-copy Serializable Partial Replication Protocol. In *Proceedings of the ACM/IFIP/USENIX 13th International Middleware Conference*, Middleware, 2012.

[79] Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luís Rodrigues. When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication. In *Proceedings of the IEEE 32Nd International Conference on Distributed Computing Systems*, ICDCS, 2012.

[80] Dmitri Perelman, Rui Fan, and Idit Keidar. On Maintaining Multiple Versions in STM. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC, 2010.

[81] Francisco Perez-Sorrosal, Marta Patiño Martínez, Ricardo Jiménez-Peris, and Bettina Kemme. Consistent and Scalable Cache Replication for Multi-tier J2EE Applications. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Middleware, 2007.

[82] Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC, 2006.

[83] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based Transactional Memory with Scalable Time Bases. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA, 2007.

[84] Paolo Romano, Roberto Palmieri, Francesco Quaglia, Nuno Carvalho, and Luís Rodrigues. Brief Announcement: On Speculative Replication of Transactional Systems. In *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, 2010.

[85] O. T. Satyanarayanan and D. Agrawal. Efficient Execution of Read-Only Transactions in Replicated Multiversion Databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):859–871, October 1993.

[86] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-Store: Genuine Partial Replication in Wide Area Networks. In *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems*, SRDS, 2010.

[87] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[88] Daniele Sciascia, Fernando Pedone, and Flavio Junqueira. Scalable Deferred Update Replication. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN, 2012.

[89] D. 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 Pacific Rim International Symposium on Dependable Computing*, PRDC, 2007.

[90] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, PODC, 1995.

[91] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP, 2011.

[92] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2012.

[93] TPC Council. TPC Benchmark C, Revision 5.11. February 2010.

[94] Alexandru Turcu and Binoy Ravindran. On Open Nesting in Distributed Transactional Memory. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR, 2012.

[95] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(2):164–206, May 2003.

[96] Li Wang and Wanlei Zhou. Primary-Backup Object Replications in Java. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, TOOLS, 1998.

[97] Xinli Wang, Jean Mayo, Wei Gao, and James Slusser. An Efficient Implementation of Vector Clocks in Dynamic Systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems and Applications*, PDPTA, 2006.

[98] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding Replication in Databases and Distributed Systems. In *Proceedings of the the 20th International Conference on Distributed Computing Systems*, ICDCS, 2000.

[99] 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, 2000.

[100] Pawel T. Wojciechowski, Tadeusz Kobus, and Maciej Kokocinski. Model-Driven Comparison of State-Machine-Based and Deferred-Update Replication Schemes. In *Proceedings of the IEEE 31st Symposium on Reliable Distributed Systems*, SRDS, 2012.

[101] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the javaTM System. In *Proceedings of the 2nd USENIX Conference on Object-Oriented Technologies - Volume 2*, COOTS, 1996.

[102] Shuqing Wu and Bettina Kemme. Postgres-R(SI): Combining Replica Control with Concurrency Control Based on Snapshot Isolation. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE, 2005.

[103] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP, 2013.